Fakultät für Luftfahrt, Raumfahrt und Geodäsie

Lehrstuhl für Flugsystemdynamik

Modular model-based development of safety-critical flight control software

Dipl.-Ing. Univ. Markus Tobias Hochstrasser

Vollständiger Abdruck der von der Fakultät für Luftfahrt, Raumfahrt und Geodäsie

der Technischen Universität München

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.) genehmigten Dissertation.

Vorsitzender: Prof. Dr.-Ing. Oskar J. Haidn

Prüfer der Dissertation: 1. Prof. Dr.-Ing. Florian Holzapfel

2. Prof. Dr.-Ing. Stephan Myschik

Die Dissertation wurde am 12.06.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Luftfahrt, Raumfahrt und Geodäsie am 27.10.2020 angenommen.

Acknowledgments

Accomplishing the work for this thesis and writing it down would not have been possible without the support of many people throughout all the years.

The five years at the Institute of Flight Dynamics had become richer of experience than I expected the day, when I joined Prof. Holzapfel's team in 2013. In an enormous team effort, we brought various manned and unmanned aircraft into the sky, with complex and large, but also, and that is even more important, safe software, which did what it was intended to do.

I want to thank all the colleagues I worked with during these years. Our achievements, the long nights, our regular coffee breaks and the conferences spread all over the world, which we attended together, will remain a unique and unforgettable experience in my life.

My special acknowledgments go to Prof. Florian Holzapfel. He gave me the possibility to work with the Institute and organized the funding for this work. He allowed me finding my direction from mechanics to software. Throughout the years, he stood by my side with loyalty and was always available for questions and discussions. With his large network of contacts, he initiated discussions and workshops with software and aerospace experts around the world, which form the solid basis for this thesis.

I also want to sincerely thank Prof. Stephan Myschik. He was a valuable advisor over the years, both technically and psychologically. Without the encouraging and constructive meetings in his office, this thesis probably would have never been finished. Thank you for all the lunches at our Indian restaurant, which we will hopefully continue.

In addition, there were a lot of people from The MathWorks Inc., who supported me with their knowledge during this thesis. Thank you very much for your responsiveness and the opportunity to work together.

Last but not least, I want to thank my family. My wife, who accepted the countless hours in the office and the numerous weekends, which I spent in front of my computer. Thank you for bearing the gray hairs, that literally began to grow during this period of time. And my parents, who always supported me on the path I chose.

Kurzfassung

Die Entwicklung sicherheitskritischer Software befindet sich in einem stetig wachsenden Spannungsfeld. Auf der einen Seite wird eine aufwändige und akribische Nachweisführung gefordert, auf der anderen Seite wächst die Softwaregröße sowie die Komplexität der Software rasant. Hinzu kommen immer kürzer werdende Entwicklungszyklen. Für klein- und mittelständische Unternehmen ist ein Einstieg in die sicherheitskritische Softwareentwicklung mit enormem Ressourcenaufwand verbunden. Modellbasierte Softwareentwicklung bildet hier seit jeher eine gute Lösung, stößt aber hinsichtlich der Skalierungsanforderungen zunehmend an ihre Grenzen.

Diese Arbeit stellt einen neuen, durchgängigen Ansatz für modulare, hochautomatisierte modellbasierte Softwareentwicklung mit Simulink® und Stateflow® der Firma MathWorks vor. Er basiert auf aktuellen Luftfahrtstandards und bietet eine Lösung für die gegensätzlichen Anforderungen in der Softwareentwicklung.

Im Rahmen der Arbeit wurde ein modularer Entwicklungszyklus und -prozess entwickelt, der auf einem modularen Codegenerierungsansatz basiert und sich damit maßgeblich von existierenden Lösungen mit integraler Codegenerierung unterscheidet. Für den Prozess wurden neue Modellierungsrichtlinien definiert sowie eine Vielzahl an Artefakten implementiert, die beim Aufsetzten einer konsistenten Entwicklungs- und Simulationsumgebung, modularer Codegenerierung sowie einer weitgehend automatisierten Erstellung der Zertifizierungsnachweise unterstützen.

Das neu entwickelte Softwarewerkzeug *SimPol* unterstützt bei der Umsetzung der durch Standards geforderten bidirektionalen Nachverfolgbarkeitsanforderungen mit Siemens Polarion® REQUIREMENTS™. Das Softwarewerkzeug hebt sich in Bezug auf Datenmodell, Arbeitsablauf, und Konfigurationsmanagement von bestehenden Lösungen ab und eröffnet dadurch neue Möglichkeiten der automatisierten Verifikation, Reparatur und Bearbeitung von Verweisen.

Das neuartige, auf Prozessaufgaben maßgeschneiderte Automatisierungswerkzeug *mrails* kombiniert Informationen aus der Automatisierung und statischen Nachverfolgbarkeitsverweisen und erlaubt dadurch die automatisierte Überprüfung der Aktualität, Konsistenz und Vollständigkeit von Entwicklungs- und Verifikationsartefakten. Zudem stellt es hochintegrierte Arbeitsabläufe für Review-Aufgaben bereit. Die Anwendbarkeit wird mit einer exemplarischen, weitgehend vollständigen Implementierung des modularen Entwicklungsprozesses demonstriert.

Abstract

Development of safety-critical software emerges in a field of tension between implementation rigor and extensive verification evidence on the one hand as well as steadily growing software complexity and size combined with shorter development cycles on the other hand. For small and medium size companies, entering the market of safety-critical software development is connected to enormous efforts. Model-based software development has been a viable solution to reduce adoption risks for years, but faces more and more scalability limitations.

This thesis presents a new, consistent approach for modular, highly automated model-based software development based on Simulink® and Stateflow® of The MathWorks. It respects the currently accepted software development standards for airborne software and has the objective to eliminate the opposing requirements of software development.

In the scope of this work, a modular development life cycle and process has been developed with generation of modular code at its core. Thereby, the approach significantly differentiates from existing solutions, which apply integral code generation. For this process, new modeling guidelines have been defined as well as a variety of artifacts been implemented, which support the setup of a consistent development and simulation environment, generation of modular code, and a broadly automated creation of certification evidence.

The novel software tool *SimPol* helps to implement standard-compliant, bidirectional traceability between Siemens Polarion® REQUIREMENTS™ and artifacts from the model-based environment. The tool sets itself apart from existing solutions with respect to its data model, workflows, and configuration management capabilities. It allows automated verification, repair, and editing of traces.

The new, process-oriented build tool *mrails* combines data obtained from automation with static traceability and derives information about up-to-dateness, consistency, and completeness of development and verification artifacts. In addition, it provides deeply integrated workflows for review tasks. Its applicability is shown with an exemplarily, almost complete implementation of the modular development process.

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Description
ARP-4754A / ARP	Refers to SAE ARP-4754A “Guidelines for Development of Civil Aircraft and Systems” [1]
AS	Assumption
COTS	Commercial-off-the-self
CP	Coding Process
DAL	Design Assurance Level
DO-178C	Refers to RTCA DO-178C “Software Considerations in Airborne Systems and Equipment Certification” [2]
DO-248C	Refers to RTCA DO-248C “Supporting Information for DO-178C and DO-278A” [3]
DO-330	Refers to RTCA DO-330 “Software Tool Qualification Considerations” [4]
DO-331	Refers to RTCA “Model-Based Development and Verification Supplement to DO-178C and DO-278A” [5]
DO-333	Refers to RTCA DO-33 “DO-333 Formal Methods Supplement to DO-178C and DO-278A” [6]
DP	Discussion Paper (in DO-331)
DP	Design Process
DR	Design rule
FAQ	Frequently Asked Questions
FCC	Flight control computer
FPU	Floating-point unit
HLR	Software high-level requirement according to glossary of DO-178C
HW	Hardware
ICD	Interface Control Document
IDAL	Item Development Assurance Level
JTAG	Joint Test Action Group (synonym for a standardized debugging interface)
LLR	Software low-level requirement according to glossary of DO-178C
MB	Model-based
MBSwD	Model-based Software Development
MR	Module design rule
PDI	Parameter Data Item (DO-178C)
PIL	Processor-in-the-loop
PPC	PowerPC
PSAC	Plan for Software Aspects for Certification
SIL	Software-in-the-loop
SL/SF	Simulink and Stateflow

SLCI	Simulink Code Inspector
SRATS	System requirements allocated to software
SW	Software
SwDP	Software Development Plan
SwVP	Software Verification Plan
TQL	Tool Qualification Level
TR	Traceability rule
TUM-FSD	Institute of Flight System Dynamics, Technische Universität München
WCET	Worst-case-execution-time

1 Introduction

1.1 Background

For safety-critical software applications, development standards have evolved over the last decades. In aerospace, so-called “heavy” development processes are state-of-the-art and explicitly demanded by authorities. They have been proving their applicability for years as framework for safety-critical software development. The commonly applied process standard is RTCA DO-178C [2]. The corresponding standard in the automotive domain is ISO 26262-6:2018 [7] (ISO 26262-6), which overlaps with DO-178C in large parts. However, the last couple of years revealed a discrepancy between software development needs of the market and the requirements imposed by the processes.

Software steadily enters new fields of application and conquers new domains. Especially controller tasks evolve from automatic functionality to autonomous behavior with new requirements on development and verification activities. In consequence, software development faces new challenges: Firstly, software complexity and size increases rapidly due to the new functionalities [8]. Secondly, software criticality raises. Software takes over more and more safety-critical tasks with growing authority. Thirdly, software development cycles accelerate [9, p. 17]. Software vulnerabilities are of higher public interest, easier to access and exploit, and can cause significantly larger damage due higher software criticality.

In non-safety-critical applications, these challenges are reflected by new methods and strategies for software development, like model-based and agile software development or DevOps (practices combining software development “Dev” and IT operations “Ops”). In model-based software development (MBSwD), the software specification is made graphically. The resulting models are formal and can directly be used for code generation or enhanced analysis methods. Agile software development comprises agile methods and processes following the principles of the agile manifesto [10, p. 59], promoting less planning and more interaction as well as communication between developers. DevOps aims at bringing development and operations closer together by providing the IT infrastructure and processes for continuous delivery of software in an agile environment (cf. [11]).

In safety-critical applications, the adoption of these solutions still faces significant challenges. The core problem is known under the term “big-freeze problem” as formulated by the Open-DO initiative some years ago[1], saying that it is extremely difficult to introduce changes to a software once it is certified. Reasons for the “big-freeze” are the large number of verification activities, the strict process, the lack of automation, or the difficulty to safely identify and isolate the impact of a change. In consequence, changes are accumulated over some period of time and implemented, verified, and released together, to reduce the effort.

[1] “Open-DO – Towards a cooperative and open framework for the development of certifiable software”, http://www.open-do.org/ [Accessed on: Sep. 07 2019]

Furthermore, many agile commandments conflict with the requirements of “heavy” development processes (e.g., “Responding to a change over following a plan” [10, p. 59]). Strategies to unite these concepts are still subject to research and pilot projects (cf. [12, 13]). A key principle of DevOps is to “automate almost everything” [11]. This goal is much harder to achieve in a safety-critical process due to the larger number of manual activities like reviews or the necessity for traceability.

And especially in combination with model-based design, this becomes a challenge. The majority of automation solutions for software development are designed for written code and have difficulties to work with binary model files.

As a consequence, the effort and uncertainty to setup such a process is immense. At the same time, the landscape of companies changes. With unmanned aerial vehicles (UAVs), many small- and medium-size companies enter the aerospace market and have ambitions to certify their software. In contrast to large aircraft manufacturers, they often have less experience and a smaller budget.

It remains a challenge to economically apply these “heavy” development standards side by side with scalability in size, complexity, and short release cycles. Kennedy states that “support for innovation in software engineering is crucial to the continued success of the aviation industry” [8, p. 3D3-13].

1.2 Objective and motivation

Main objective of this thesis is to provide a consistent model-based software development process with tooling that satisfies applicable standards and is scalable in the sense of agile methods and DevOps. In particular, the thesis shall

1. Lower the adoption risk of a MBSwD for smaller companies by providing a MBSwD process and tooling, which is out-of-the box applicable, bridges tool gaps, and is consistent.

2. Improve development efficiency, ensure scalability, and support for agile development by tool usage and the introduction of a high degree of software modularization and reuse whilst adhering to process requirements and safety.

3. Provide a solution for exhaustive process automation and keeping a “ready-to-certify” state of software artifacts.

Process modularization means that not only the software is broken down into modules, but also the process. Objective is to maximize concurrent development and work out integration solutions, necessary tool enhancements, and processes that guarantee a consistent set of certification evidence in the end.

To overcome the “big-freeze” problem, the goal of the Open-DO initiative was “to develop techniques and tools that will allow all software to be constantly maintained in a ‘ready-to-certify’ state” [14]. That means, it is admirable to keep software in a valid state after each change, so that only some finalizing reviews are required for a new release. Performing additional verification, like adding test cases, should not be necessary at this point of time anymore. Objective was to setup a process automation and tooling that keeps this ‘ready-to-certify’ state covering a maximum of the defined process activities whilst minimizing the additional effort for developers.

1.3 Scope

The three objectives are addressed in the scope of a DO-178C process under certain assumptions:

- The whole process bases DO-178C “Software Considerations in Airborne Systems and Equipment Certification” [2] including DO-331, the “Model-Based Development and Verification Supplement” [5].
- The software under development is a flight control algorithm for an existing flight control computer (FCC), which has been developed at the Institute of Flight System Dynamics (Technical University of Munich, TUM-FSD).
- The process respects the tool landscape and previously performed projects at TUM-FSD.
- Certification under EASA is the goal.
- MBSwD is done with MATLAB®, Simulink®, and Stateflow® (SL/SF)[2] as central modeling tool and Embedded Coder[3] for automatic code generation, both in release 2017b with update 9 (R2017b).
- Requirements are developed and managed with Siemens Polarion® REQUIREMENTS™ (Polarion)[4].

The thesis describes the MBSwD to the best of the author’s knowledge and belief. The process setup bases on discussions with experts and on extensive experience collected during the development of flight control algorithms for the DA-42 research aircraft [15]. The algorithms span functionality of unmanned and piloted aircraft, from system automation [16, 17], over low-level control [18–21] and autopilot features [22–25], up to enhanced flight management [26–30] or standardized communication protocols (like STANAG4586 [31]). Parts of the process have also been presented to the “Wehrtechnische Dienststelle für Luftfahrzeuge und das Musterprüfwesen für Luftfahrtgerät der Bundeswehr” (WTD61/ML) in the project MezA („Model-based development of certifiable avionic systems”).

The process has not been evaluated in an approved or active certification project involving authorities. Different auditors may have different opinions on aspects of the model-based development process and may demand adaptions.

[2] Product of The MathWorks Inc., https://de.mathworks.com/products/simulink.htm [Accessed on: Jan. 04 2019], Release R2017b

[3] Product of The MathWorks Inc., https://de.mathworks.com/products/embedded-coder.html [Accessed on: Jan. 04 2019], Release R2017b

[4] Product of Siemens AG, https://polarion.plm.automation.siemens.com/products/polarion-requirements [Accessed on: Sep. 07 2019]

The presented, novel process bases on SL/SF Release 2017b. The release had to be frozen to a certain point of time in order to provide a consistent documentation. It is valid for later releases to a large part, however, newer releases may have additional features or fixes so that some workarounds or constraints can be avoided. Whenever the author is aware of such changes, a respective foot note gives indication.

DO-178C and DO-331, which is the supplement for model-based development of DO-178C, describe a Planning Process, four Development Processes and four supporting, integral processes. The thesis mainly focuses on the Design and Coding process as illustrated in Figure 1 and respective verification activities. Both the Design and Coding process are addressed from high-level planning to rules in standards. The Requirements and Integration processes are only discussed selectively as well as the related verification activities.

Some tools use formal methods and are thus subject to DO-333, the DO-178C supplement for formal methods [6]. The additional objectives are not covered by the thesis, but can easily be applied on top.

Configuration management plays a minor role in this thesis, but is considered from time to time for the processes in focus. Out of scope in this thesis are the quality assurance and certification liaison processes.

The more detailed a process setup is described, the higher is the degree of tailoring to a given project, company, or tool landscape. The work presented in this thesis thus is not generic in many parts and strongly relies on a clearly defined context.

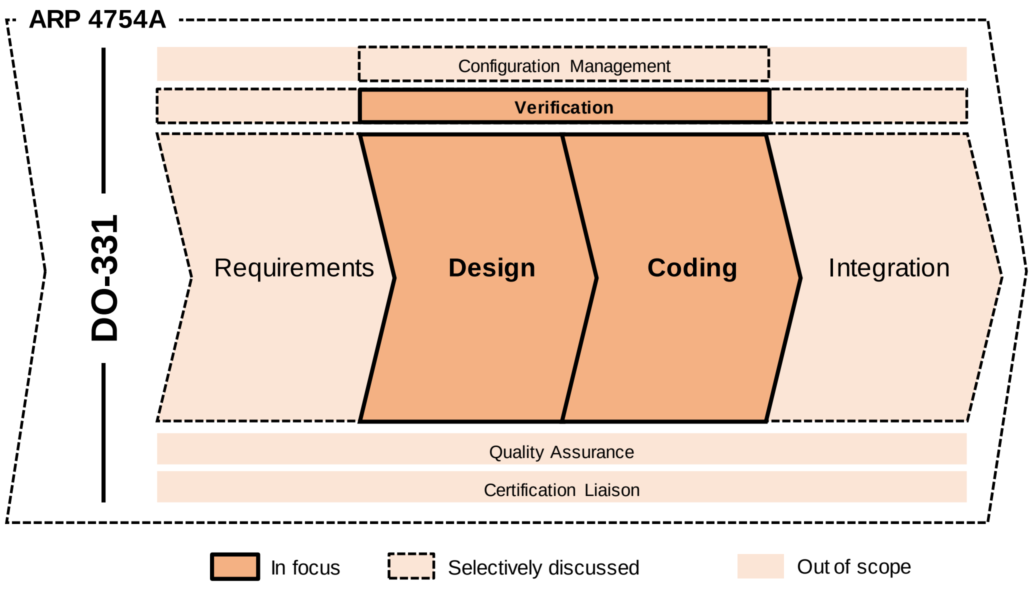


Figure 1: Considered development and integral processes in the scope of the thesis

Finally, DO-178C requires tool qualification, i.e., the tools used to satisfy objectives in the process have to be verified as well. Tool qualification is described by DO-330 [4]. This thesis addresses tool qualification just superficially. The process has been defined with tool qualification in mind, but it often goes beyond the scope of the tool qualification kit provided by MathWorks [32]. Due to the limited timeframe, it was not possible to perform any tool qualification or assess the gap in detail.

1.4 Structure of the thesis

After the introduction, some fundamentals, like the structure of DO-178C/DO-331 and its relation to other process frameworks, are discussed in section 2. In addition, an overview of MBSwD is given.

The project context and the necessary assumptions for system, hardware, and surrounding software aside from the software developed with MBSwD, are discussed in section 3. Most of the work in this section has been collective effort at TUM-FSD, to which the author has contributed.

The main content of the thesis has been split into four chapters. Due to the difference of the addressed topics, each part introduces the state-of-the art separately and provides a short summary with an outlook on future work. The four parts are:

- modular development process (part 1)
- modeling framework for safety critical MBSwD in Simulink
- traceability tooling and rules
- process-oriented build tool
- modular development process (part 2)

For didactical reasons, the presentation of the modular development process has been separated into two parts. Part 1 in section 4 introduces the process assembled by the author from a high-level view. At first, process breakdown and model usage are presented. After that, tasks for development and verification are defined that fulfill DO-178C/DO-331 objectives. The development tasks and the used tools are discussed in section 4.5. The verification tasks for review and analysis as well as for testing are provided in section 4.6.

Section 5 describes the new modeling framework for modular model-based software development in SL consisting of a holistic set of rules and a composed modeling environment.

Section 6 introduces the new developed traceability tooling to bridge the gap between Polarion and SL as well as a set of traceability rules.

Section 7 presents a new implemented process-oriented build tool *mrails*, which serves as foundation and binding framework to automate and visualize the tasks of the MBSwD process.

As part 2 of the modular development process, the detailed implementation of the superficially defined tasks of section 4 in the build tool framework are presented in section 8. Central aspect is the realization of generation of modular code in section 8.1 and the automation of verification tasks in section 8.2.

Finally, section 9 evaluates the achievements of this thesis with respect to the overall objectives, summarizes limitations and offers an outlook on future work.

1.5 Contributions

This thesis provides solutions beyond the current state-of-the-art for:

- an efficient and consistent modular model-based software development process for flight control algorithm development.
- a novel and consistent modeling framework for safety-critical MBSwD in SL/SF that supports modularization in the scope of DO-178C by design and coding rules, a safe modeling subset supporting the modular process, and a consistent modeling environment.
- traceability rules and a new, process-oriented tooling efficiently bridging the traceability gap between requirements in Polarion and implementation as well as testing in the SL/SF environment.
- a new kind of build tool with process-oriented features supporting a degree of automation beyond today's tool capabilities.
- detailed process task procedures to efficiently, consistently, and safely apply the selected tools of the modular process, as well as an organized way of assessing, reviewing, and documenting the results.

Detailed descriptions of the contribution are highlighted throughout the thesis in blue boxes as displayed below. The following sub-sections summarize the contributions.

Contribution X: Blue boxes of this kind describe the contribution of the section(s) at hand in detail throughout the thesis.

1.5.1 Modular development process (part 1)

A new modular MBSwD process has been implemented based on the applicable process frameworks. It breaks down the activities, which are required to fulfill objectives of the standard, into modular sub-processes and thus unveils the potential of agile, parallel development. This breakdown goes beyond industry standards, since various challenges had to be solved like enhanced data and control coupling considerations or tool usage outside their normal use case.

In detail, the contributions of the modular MBSwD are:

Contribution 1: In order to ensure scalability and maximize concurrent, team-based, agile development, a new modular software development life cycle has been specified. It not only introduces an architectural breakdown of the software, but also distributes DO-178C process activities onto module- and integration-levels. Process activities from design to code verification can thus be executed earlier and on encapsulated entities. Evidence for certification is independent and reusable, which saves significant effort. A novel approach for generation of modular code, which differentiates from the broadly applied integral code generation, amplifies this effect.

Contribution 2: The abstract DO-178C development objectives have been concretized in specific tasks adapted for the modular development life cycle. Different model usage workflows, especially relevant for flight control development, have been considered. Defined integration-level tasks focus on the distribution of work (allocation of requirements) and the integration later on. Module-level tasks cover the actual implementation and traceability. This new distinction and the consistent consideration throughout task definition is one of the key concepts guaranteeing DO-178C compliance in a modular process.

Contribution 3: The abstract DO-178C verification objectives have been concretized in specific tasks adapted for the modular development life cycle. The unique separation of tasks into module- and integration-level tasks leverages full modular development. Integration-level tasks focus on verification of the interfaces and overall completeness. Module-level tasks allow early verification and detection of design flaws. This, and the low complexity of the design on this level, leverage significant time and cost savings.

Contribution 4: An economic modular simulation and testing strategy has been specified. The testing strategy spans simulation, software-in-the-loop testing, reuse of test cases, model coverage, structural coverage, and data and control coupling. It improves reusability of test cases and decreases rework by front-loading and modularizing verification activities. Rework cycles are also significantly shortened, which results in cost reduction and faster development.

The work has partially been published in [33] and presented on DGLR workshops [34, 35].

1.5.2 Modeling framework for safety-critical MBSwD in SL/SF

With the modeling framework, a package of rules and tools has been developed that constrains SL/SF for modular, safety-critical software development. In detail, the following contributions are claimed:

Contribution 5: A new set of design rules for modular development has been created. The rules differ from existing modeling rules by describing overall concepts of the software design in a development tool-independent manner, but with MBSwD in mind. The new design rules help developers to understand overall principles and provide reasoning for modeling rules. They cover topics, which are essential in modular workflows but fairly unaddressed in existing rule sets, i.e., they formulate requirements for high-level architectures, data encapsulation, and interface contracting and they specify the way to handle DO-178C Parameter Data Items or deactivated and noncovered design.

Contribution 6: A set of coding rules specifically tailored to auto-generated code of Embedded Coder has been written. The coding rules have been specifically defined with respect to the used code generator, compiler, and hardware.

Contribution 7: A large set of naming conventions has been established ensuring consistent naming of model elements throughout the design. The rules are important to avoid identifier clashes or identification of responsibilities for model elements in a team-based development process.

Contribution 8: Rules for the high-level architectural design have been defined, which map the generic Design Rules to the SL/SF development environment. They dictate a consistent solution to specify interfaces of architectural entities, including rarely applied concepts of contracting and encapsulation. To the authors knowledge, the provided rule set is the only rule set addressing modular design / code generation from this perspective and thus significantly reduces the adoption effort for any reader.

Contribution 9: A safe modeling subset for SL/SF has been assembled. It is a set of well-defined and justified rules limiting SL/SF features and adding conventions. They cumulate best practices collected by the author in the various accompanied projects. In contrast to many other existing guideline sets, which blacklist prohibited features, a whitelisting approach is followed. Only features from a permitted subset shall be used. This safe modeling subset targets compatibility of the process, tools, and tasks. It is usable out-of-the-box and significantly lowers the adoption risk, with which large and small companies struggle. Incremental try-and-error to reach a compliant guideline is significantly reduced. The rules are also an important pillar for generation of modular code.

Contribution 10: A novel foundation block library limiting the usable model elements has been created. This library deviates from block libraries shipped by MathWorks in R2017b, since it is compliant with the rules at hand (e.g., naming conventions)

Contribution 11: DO-178C/DO-331 concepts relevant for model-based design have been interpreted for SL/SF and constituted in rules. Consistent concepts for the definition of model elements contributing to the design, noncovered and deactivated design, algorithm correctness and the usage of model element libraries have been prepared. These concepts significantly determine, how SL/SF is used and are not covered by documentation offered by MathWorks. They lower the adoption risk of model-based design and ease discussions with authorities in a certification project.

Contribution 13: A consistent modeling environment has been assembled. The modeling environment comprises all resources to setup SL/SF in the light of the given design, modeling, traceability, and coding rules. As distributable package, they allow any developer to quickly turn SL/SF in the controlled environment necessary to implement Design Models.

Aspects for safety-critical software development with SL/SF and especially the modeling environment have published by the author in [36] and as co-author in [37].

1.5.3 Traceability rules and tooling

Contribution 14: The author has developed a new, publically available tool called *SimPol* to manage traceability between Polarion and SL artifacts. Compared to existing solutions, *SimPol* supports all artifacts in SL relevant for the process (SL models, SL tests, SL data dictionaries) by an extendable, pluggable software architecture. In addition, the management effort is reduced by loading bidirectional traces into an integrated, abstract data model, which leverages automatic identification of missing, corrupted, or outdated traces and their resolution, bending of trace links to other artifact revisions, as well as impact analysis.

Contribution 15: A new set of traceability rules supports the adoption and consistent usage of *SimPol*, for example by clarifying, what kind of artifacts have to be traced to requirements and to which granularity or how derived low-level requirements are handled. The rules outline a directly usable traceability solution, which covers many existing use cases and lowers the adoption effort for small companies, which are not familiar with traceability so far.

SimPol workflows have been published in [38] and presented at [34].

1.5.4 Process-oriented build tool and process automation

In order to keep the process in a “ready-to-certify” state, the process-oriented build tool has been developed.

Contribution 16: A new type of build tool specifically designed for model-based development in a process has been implemented. The tool provides a common framework to bundle the implementation of process tasks and automate their execution based on dependencies. The tool specifically addresses the needs of a safety-critical process by providing solutions for review workflows as well. Especially review workflows are not well covered by traditional build tools.

Contribution 17: An innovative approach to couple traceability with build dependencies collected during the build has been implemented. The new symbiosis generates fine-granular traceability of sub-file level in local and CI workflows and facilitates enhanced process analysis like checking of up-to-dateness, completeness, or cleanliness of development and verification artifacts.

Contribution 18: The process-oriented build tool leverages novel automated completeness assessment of the certification artifacts, i.e., whether all activities have been performed on all artifacts and the necessary certification evidence exists. This allows to keep the process in a certification-ready state and helps developers to identify upcoming work.

Contribution 19: The process-oriented build tool leverages novel automated consistency assessment to check whether artifacts are outdated and have to be generated or reviewed again. This allows to keep the process in a certification-ready state at all times.

Contribution 20: The process-oriented build tool has been equipped with easy-to-use web-based UIs. They summarize the status of the project at a central point, provide status information for each task and direct links to open the relevant artifacts. This overview helps developers keeping track of the project status

Contribution 21: A novel framework for standardizing task execution, evaluation, review, and report generation has been established. Generalized status types with consistent consequences or actions for all tool outputs have recognition value for developers.

Contribution 22: Functionality for the generation of dynamic review lists has been integrated into the process-oriented build tool. They can be auto-generated based on the context. Thereby, review work is significantly reduced and a higher level of consistency is achieved, since review lists are managed by the tool at a central place and the review status is directly indicated as for any other automated task.

The author has published the process-oriented build tool and shown its application in [33, 39] and has presented the work at [34].

1.5.5 Modular development process (part 2)

Finally, the development and verification tasks of the modular MBSwD have been detailed in application procedures. Tools have been configured and tool usage has been carefully selected. Furthermore, result assessment and the handling of deviations has been chosen. All tasks have been automated and the result assessment been standardized, so that each task can be plugged into the process-oriented build tool.

Contribution 23: A process to generate modular code with Embedded Coder has been realized, which fulfills configuration management requirements. Additional scripts and resources have been created to allow generation of modular code and safe integration into higher-level SW modules afterwards. Such a workflow is not supported by the MathWorks tool chain natively. Central advance is the handling of shared code. Generation of modular code enables reusability of code as well as early and separate verification.

Contribution 24: A task for static model analysis has been defined and implemented. Resources have been created that allow the automated execution and result assessment of SL Model Advisor checks for models. Checks have been implemented for new and adapted modeling rules. A new categorization for the criticality of check violations has been introduced. The SL Model Advisor results have been integrated into the process-oriented build tool.

Contribution 25: A task for static module analysis has been defined and implemented. Resources have been created that allow the automated execution and result assessment of SL Model Advisor checks for whole modules. Checks on module level are not part of any check set shipped by MathWorks and therefore represent an innovation. They are special for the modular process at hand.

Contribution 26: Dynamic check lists have been implemented for model review tightly integrated in the process-oriented build tool. The dynamic check lists are auto-generated and inherit the full feature set of the process-oriented build tool, like automated evaluation of up-to-dateness. This significantly reduces the review effort.

Contribution 27: Dynamic check lists have been implemented for traceability review. They are tightly integrated in the process-oriented build tool. The dynamic check lists are auto-generated and inherit the full feature set of the process-oriented build tool, like automated evaluation of up-to-dateness. This significantly reduces the review effort.

Contribution 28: Simulink Design Verifier has been adapted to perform assurance-guarantee analysis, run-time error analysis, and dead-logic detection. Resources have been implemented that allow the automated execution and result assessment. Additional functionality has been added to support the modular approach of the process at hand.

Contribution 29: Rules for simulation test procedure and case development with Simulink Test have been developed. These rules help organizing simulation test in a module, ensure efficient reusability for simulation and tests, and support the modular test and coverage collection system.

Contribution 30: The jobs for dynamic checklists to review simulation/test cases and procedures have been set up. The dynamic check lists are auto-generated and inherit the full feature set of the process-oriented build tool, like automated evaluation of up-to-dateness. This significantly reduces the review effort.

Contribution 31: Resources have been created to automatically execute simulation tests and assess the results. Dynamic check lists have been implemented for result review. They are tightly integrated in the process-oriented build tool. The dynamic check lists are auto-generated and inherit the full feature set of the process-oriented build tool, like automated evaluation of up-to-dateness. This significantly reduces the review effort.

Contribution 32: Criteria for model coverage assessment have been constituted. Resources have been created to aggregate model coverage per module and across modules. Such an automated coverage aggregation and assessment is not natively supported by MathWorks, but inevitable in a modular process. Model execution coverage is also applied in a new way to asses coupling between models.

Contribution 33: A task has been defined and resources have been created to automatically execute automated code review with Simulink Code Inspector and assess the returned results.

Contribution 34: Resources have been created to configure Polyspace BugFinder, so that it checks the selected coding rules and design errors. The analysis and result evaluation has been fully automated. Different configurations have been developed for module- and component-level analysis in the modular process.

Contribution 35: Resources have been implemented that allow the automated execution and result assessment of simulation/test cases in software-in-the-loop mode and perform equivalence comparison with previously recorded model simulation results.

Contribution 36: Criteria for code coverage assessment have been constituted. Resources have been created to aggregate code coverage per module and across modules. Such an automated coverage aggregation and assessment is not natively supported by MathWorks, but inevitable in a modular process.

2 Fundamentals

DO-178C 10.0 states that “certification authorities consider the software as part of the airborne system or equipment installed on a certified product; that is the certification authorities do not certify the software as unique, stand-alone product.” The software, which is developed with model-based techniques in the scope of this work, is just a small piece in a large project. Thus it is very important to have a clear picture, how the process and product are embedded.

The cascade of process layers is depicted in Figure 2. Any development company needs an approval by the authority. To obtain the approval, compliance to common quality standards must be shown, for example EN ISO 9000ff.

In a conducted project, a certification basis is agreed with the authorities depending on the type of the developed product. The development of the whole product is then performed under ARP-4754A “Guidelines for Development of Civil Aircraft and Systems” [1]. ARP-4754A is strongly coupled with ARP-4761 for conducting the safety assessment. Systems and items are standardized breakdown levels of ARP-4754A. A system (e.g., an aircraft) consists of multiple subsystems (e.g., flight control system, landing gear,…), which themselves are decomposed into multiple items.

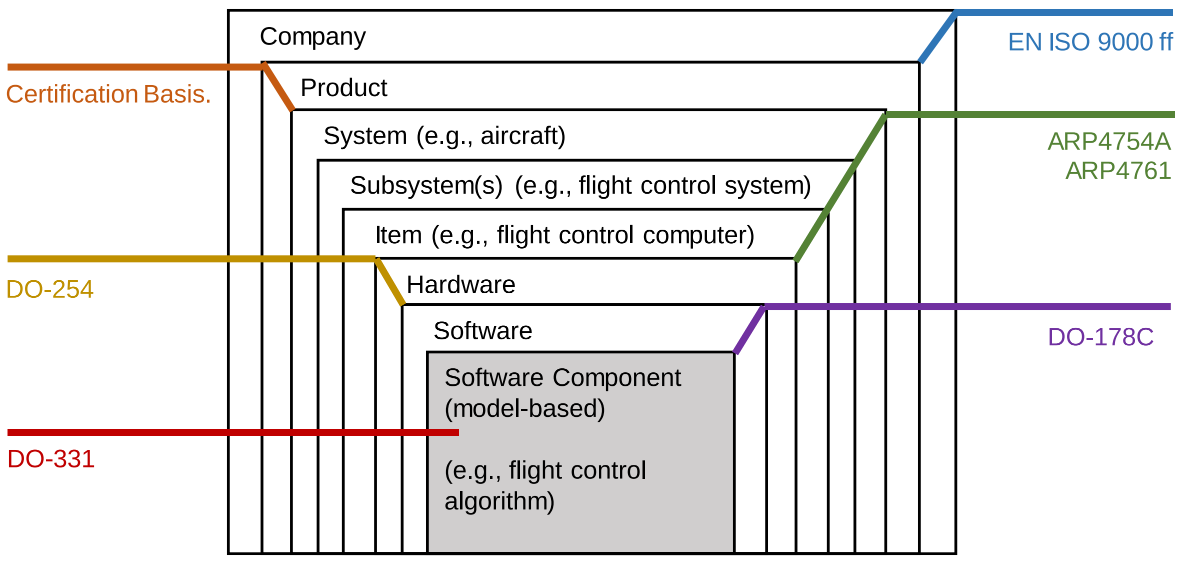

Company
Product
System (e.g., aircraft)
Subsystem(s) (e.g., flight control system)
Item (e.g., flight control computer)
Hardware
Software
Software Component (model-based)
(e.g., flight control algorithm)
Certification Basis.
DO-254
DO-331
EN ISO 9000 ff
ARP4754A
ARP4761
DO-178C

Figure 2: Software component in context

Items, the smallest units on system level, are, for example, components like a flight control computer with all its physical interfaces, cooling, case, or mounting. For electronic hardware (HW), DO-254 is the most common development standard being followed.

The software (SW) itself is programmed and compiled for a specific HW computation target. The SW is typically divided into components. Operating system, drivers or interface code are typical non-model-based components developed under DO-178C. Only specific components are developed under DO-331 and MBSwD, like the flight control algorithms, since MBSwD is not beneficial for all types of software.

2.1 Certification basis

Certification of airborne software in the European Union is controlled by EASA. It is typically obtained within the scope of a whole aircraft Type Certificate (TC) or a European Technical Standard Order (ETSO) for installable equipment.

In case of a “Normal Category” aircraft, the basic certification specification (CS) is CS-23. At the time this thesis was written, the current version was Amendment 5. The relevant paragraph of the CS for digital flight control systems §23.2500 and appended Acceptable Means of Compliance (AMCs) do not provide further guidance. AMC 20-115D [40] (EASA) and AC 20-115D [41] (FAA) recognize RTCA DO-178(B/C) / ED-12(B/C) and supplements as “acceptable means, but not the only means, for showing compliance […] with regard to software aspects of airborne systems and equipment [40]”. AMC 20-115D repeals Certification Memorandum CM-SWCEH-002 Issue 1 “Software Aspects of Certification” [42] and the proposed CM-SWAEH-002 Issue 2 of October 2013 [43].

2.2 ARP-4754A system development process

ARP-4754A is the recognized development standard for aircraft and equipment systems that implement aircraft functions. It describes a top-down V-development cycle as illustrated in Figure 3 (cf. ARP-4754A 4.1.3).

The cycle begins with the identification of aircraft-level requirements and functions. Aircraft functions are allocated to derived systems in the “System Requirements Identification”. From these systems, a system architecture with details up to the item-level is created, to which system functions and safety requirements are allocated. Items are further split into HW and SW. Output of the “Item Requirements Identification” process are system-level requirements allocated to HW or SW (cf. ARP-4754A 4.5).

The item design process itself is not detailed in ARP-4754A, but in other standards like DO-254 for hardware and DO-178C for software development. ARP-4754A is explicitly written to support these other standards.

The item is validated against the allocated system requirements and then integrated into the system. The system is validated against higher-level system requirements and afterwards integrated into the aircraft and validated again.

The whole system development process is typically spread over multiple teams and companies and requires a more granular hierarchy, e.g., with subsystems to cover large projects.

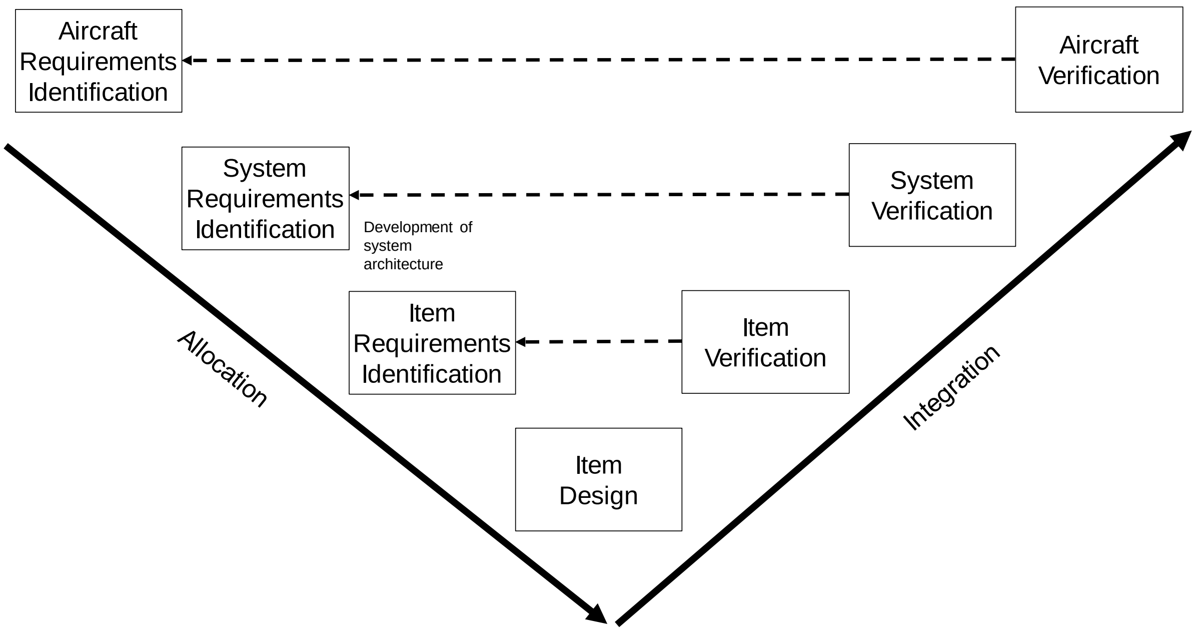

Aircraft Requirements Identification
Aircraft Verification
System Requirements Identification
System Verification
Development of system architecture
Item Requirements Identification
Item Verification
Item Design
Allocation
Integration

Figure 3: Simplified ARP-4754A process

During each step on the left side of the V in Figure 3, ARP-4754A requires a safety assessment with specific methods, which are detailed in ARP-4761. The resulting Function Development Assurance Level (FDAL) determines the rigor of the requirement identification phases and the respective verification. To the item, an *Item Development Assurance Level* (IDAL) is assigned. The IDAL determines the rigor of the software and hardware development processes.

Primary input for a software development process (item design) are *system requirements allocated to software* (SRATS), and the IDAL. However, a constant information flow between system and the different item design processes is emphasized in ARP-4754A 4.6.1.

2.3 DO-178C

DO-178C spans a commonly accepted, abstract process framework to develop software for airborne systems and equipment. DO-178C is considered as "heavy" development process due to its extensive process requirements. The standard assumes that the software process is embedded in a ARP-4754A system development process.

Additional information on DO-178C is provided by DO-248C "Supporting Information for DO-178C and DO-278A" [3]. Furthermore, DO-178C has been published with three supplements:

- DO-331 "Model-based development and verification supplement to DO-178C and DO-278A" [5]
- DO-332 "Object-oriented technology and related techniques supplement to DO-178C and DO-278A" [44]
- DO-333 "Formal methods supplement to DO-178C and DO-278A" [6]

The basic DO-178C standard defines three *software life cycle processes*, the software planning process, the development process, and an integral process accompanying them (cf. DO-178C 3.1). The three processes are schematized in Figure 4. Development and integral processes are both divided into four sub-processes.

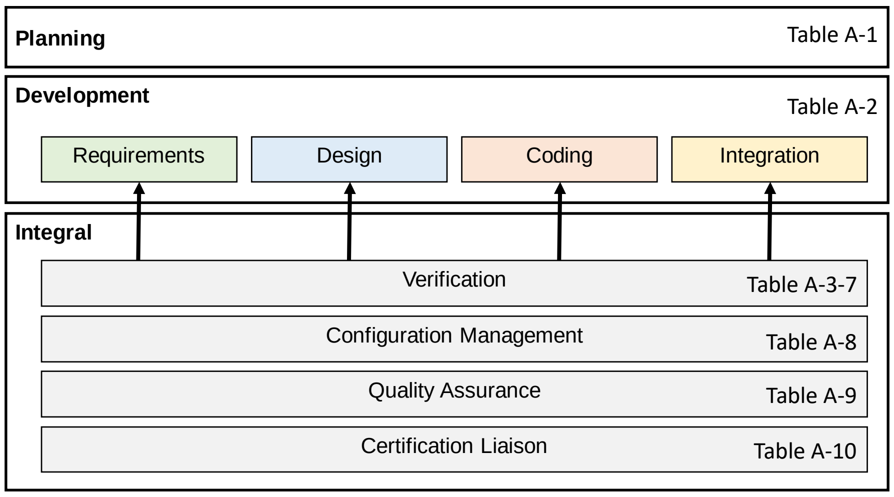


Figure 4: Processes and objectives summarized in DO-178C

Tailoring, interpretation, and implementation of the DO-178C processes is subject to the process applicant. For most of the processes, DO-178C dictates a set of objectives, to which compliance must be shown. Tables A-1, A-2, and A-8 to A-10 of DO-178C Appendix A list objectives for the process implementation, which are observed by the software quality assurance process (cf. DO-178C 8.1a). Tables A-3 to A-7 address verification of outputs of the development processes and are handled by the verification process.

Which objectives apply and how rigorous the processes are, is mainly driven by the allocated IDAL of the item or, in the terms of DO-178C, the *software level*. The software is ranked from A to D, with A indicating the highest criticality. Depending on the software level, DO-178C may require independent verification of objectives. Independence requirements are not further addressed in the scope of this thesis.

Although these processes provide a clear framework for software development, the actual software life cycle, i.e., the partitioning into nested life cycles and the sequencing of these sub-processes, is explicitly not prescribed by the standard (cf. DO-178C 3.2).

Planning process

The process implementation and the activities, how the objectives shall be fulfilled, are documented in five plans:

- Plan for Software Aspects of Certification (PSAC)
- Software Development Plan (SwDP)
- Software Verification Plan (SwVP)
- Software Configuration Plan
- Software Quality Assurance Plan

The PSAC is the plan submitted to the certification authorities in an early stage and provides an overview of the process. The remaining four plans are written for the developers to provide guidance.

In addition to the plans, three development standards have to be written, providing detailed information for the developers in the requirement, design, and coding phase:

- Software Requirement Standard
- Software Design Standard
- Software Code Standard

Development process (SwDP)

As depicted in Figure 4, the development process has the following four sub-processes:

- Requirements Process
- Design Process (DP)
- Coding Process (CP)
- Integration Process

Figure 5 illustrates the development processes in a V-model. Although the idea of the V-model, which is componentization and top-down development on the left side, and the integration on the right side of the V, slightly differs from DO-178C, it provides a simple illustration. Software is more and more detailed from SRATS to Executable Object Code during development on the left side.

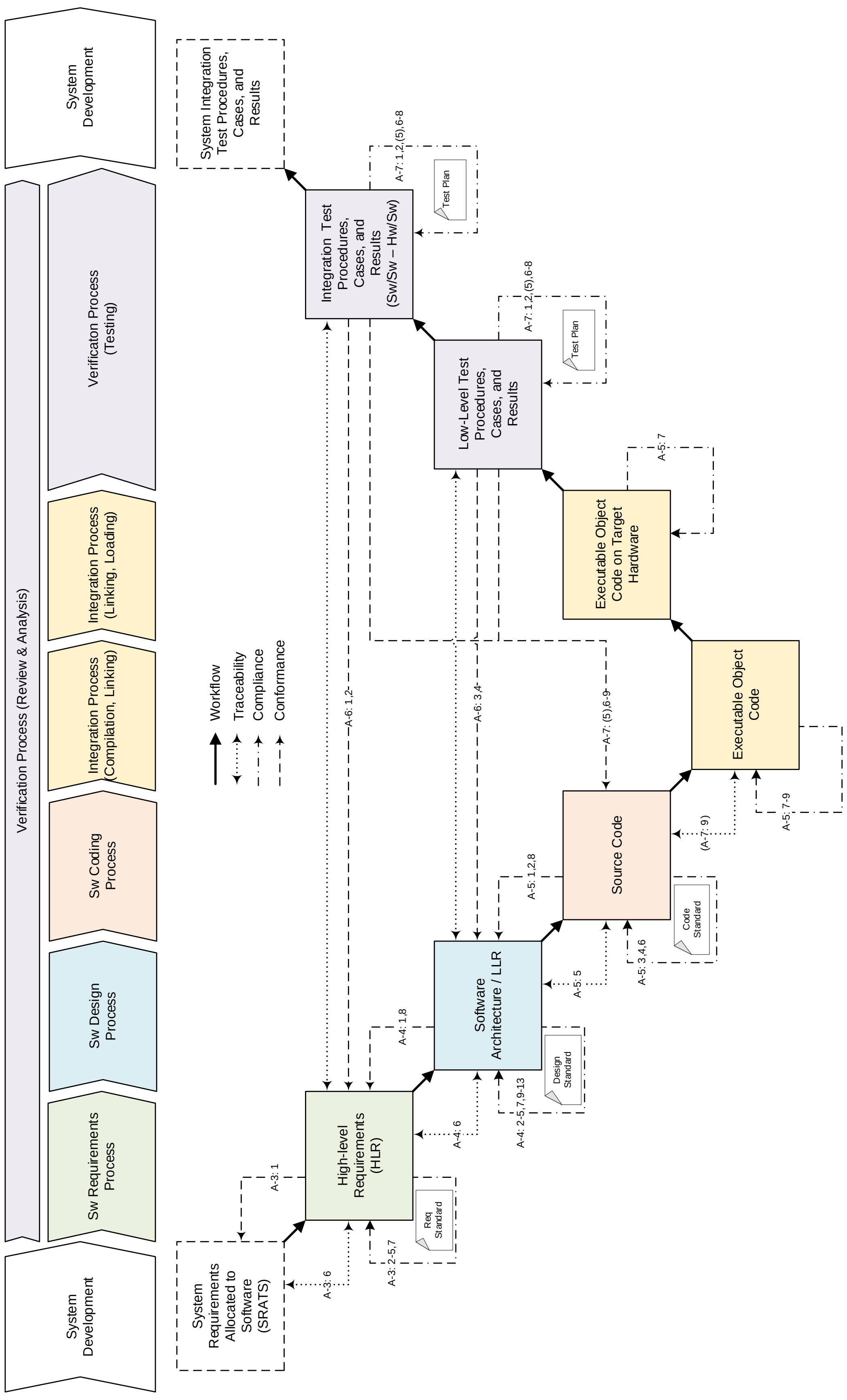

System Development
Verification Process (Review & Analysis)
Sw Requirements Process
Sw Design Process
Sw Coding Process
Integration Process (Compilation, Linking)
Integration Process (Linking, Loading)
Verificaton Process (Testing)
System Development
System Requirements Allocated to Software (SRATS)
System Integration Test Procedures, Cases, and Results
Workflow
Traceability
Compliance
Conformance
High-level Requirements (HLR)
Req Standard
Software Architecture / LLR
Design Standard
Source Code
Code Standard
Executable Object Code
Executable Object Code on Target Hardware
Low-Level Test Procedures, Cases, and Results
Test Plan
Integration Test Procedures, Cases, and Results (Sw/Sw – Hw/Sw)
Test Plan
A-3: 1
A-3: 6
A-3: 2-5,7
A-4: 6
A-4: 1,8
A-4: 2-5,7,9-13
A-5: 5
A-5: 1,2,8
A-5: 3,4,6
(A-7: 9)
A-5: 7-9
A-5: 7
A-6: 1,2
A-6: 3,4
A-7: (5),6-9
A-7: 1,2,(5),6-8
A-7: 1,2,(5),6-8

Figure 5: DO-178C process in a V-model software life cycle

Main target of the Requirements Process, as described in DO-178C 5.1, is to derive *Software High-Level Requirements* (HLRs) from SRATS*.* HLRs describe, *what* the software shall do. The development of HLRs must conform to rules of the Software Requirement Standard.

In the subsequent design process, HLRs are developed into *Software Low-Level Requirements* (LLRs) and a *Software Architecture* (cf. DO-178C 5.2). LLRs describe, *how* the software shall be implemented. The outputs must conform to the rules defined in the Software Design Standard. LLRs and software architecture form the *SW Design*.

In the coding process, LLRs and software architecture are used to program the *Source Code* (cf. DO-178C 5.3). The code must conform to the rules defined in the Software Code Standard.

According to DO-178C 5.4.1 a., the objective of the integration process is to produce *Executable Object Code* as well as associated Parameter Data Item files and load them into the target. Input therefore is the Source Code. Whether the integration process of DO-178C should be seen on the left or right side of the V-model is controversial, since it is not the classical component integration as intended by the V-model. Figure 5 splits the integration process in two steps. The argumentation is that compilation from Source to Executable Object Code is a development step, since it transforms Source Code independently of other software libraries, and linking as well as loading integrates software with software or software with hardware, respectively.

Verification process (SwVP)

The verification process is executed along with the development process. Its obligation is to verify the outputs of the development process.

DO-178C distinguishes *review and analyses* from *testing*. In the V-model of Figure 5, the review and analysis process accompanies both sides of the V as part of the verification process. The testing processes can only be found on the right-hand side of the V. The verification objectives are illustrated by arrows grouped concerning their subject, i.e., traceability, compliance to other artifacts, and conformance to standards.

Review and analysis can be seen as parallel activity to the processes directly verifying the outputs after each phase. According to DO-178C 6.3, “analyses provide repeatable evidence of correctness and reviews provide a qualitative assessment of correctness”. Analyses are also called *static software verification* typically not requiring an execution of the software. Which objectives are verified by reviews, and which by analysis, depends on the process implementation.

Testing is considered as *dynamic software verification*, which requires execution of the software. DO-178C explicitly claims that only executable object code is considered as “tested” (cf. DO-178C 6.4).

Formally, tests consist of test procedures, test cases, and (after execution) of test results. All tests for certification credit must be derived from requirements (cf. Figure 6). This is called *requirements-based testing* (cf. DO-178C 6.4.2). DO-178C distinguishes between low-level testing and software as well as hardware-software integration testing.

Testing itself requires review and analysis activities. This comprises analysis to assess the achieved requirements-based and structural coverage, but also reviews for test procedures and test results. These actions are referred to as “verification of verification” and documented in Table A-7. The mentioned test plan in Figure 5 is not required by DO-178C, but follows the recommendations of Rierson [45] in 9.6.4 to provide guidelines for test case development, review checklists, or a target environment description.

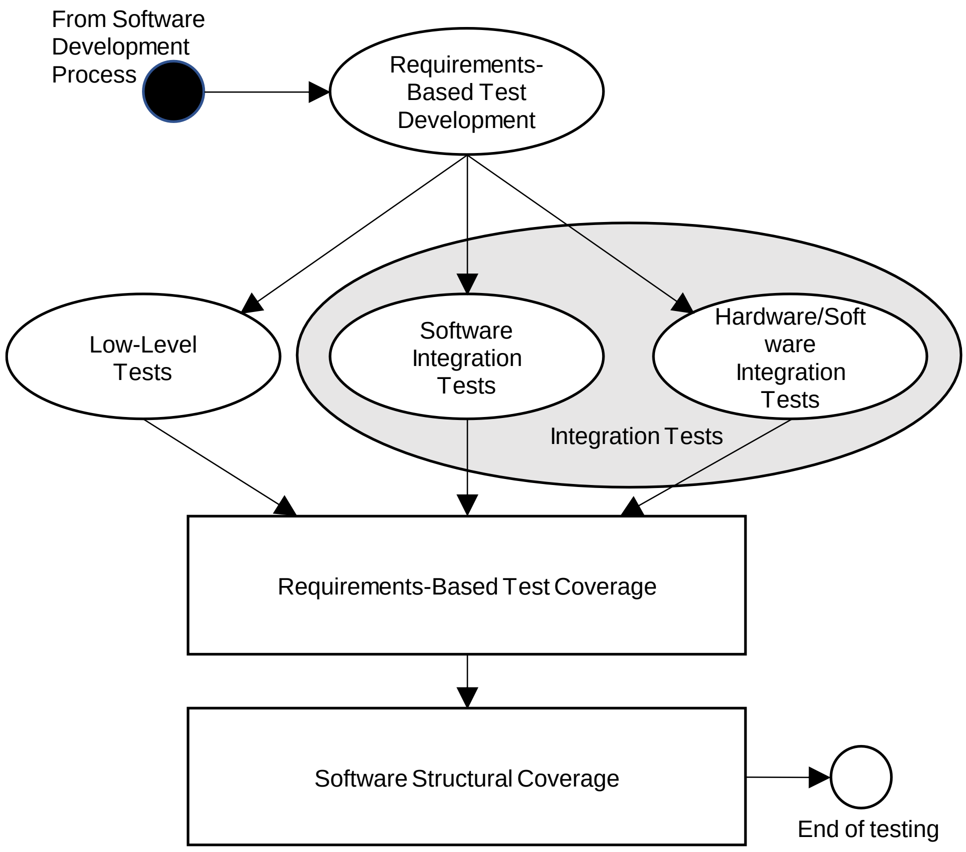

From Software Development Process
Requirements-Based Test Development
Low-Level Tests
Software Integration Tests
Hardware/Software Integration Tests
Integration Tests
Requirements-Based Test Coverage
Software Structural Coverage
End of testing

Figure 6: Testing activities from DO-178C Figure 6-1

Other integral processes

Besides the verification process, software configuration management, software quality assurance, and certification liaison processes support software development.

Configuration management is one of the most important processes to manage the complexity of development. In DO-178C, it basically comprises of:

- Configuration identification (cf. DO-178C 7.2.1)
- Baselines and traceability (cf. DO-178C 7.2.2)
- Problem reporting, tracking, and corrective action (cf. DO-178C 7.2.3)
- Change control (cf. DO-178C 7.2.4)
- Change review (cf. DO-178C 7.2.5)
- Configuration status accounting (cf. DO-178C 7.2.6)
- Archive, retrieval, and release (cf. DO-178C 7.2.7)

Often, the configuration management is strongly linked with the system development process, especially concerning problem reporting and change control. The Configuration Management Process classifies relevant data into Data Control Category 1, the highest category, and Data Control Category 2. The Data Control Category specifies the activities to be done in the process (cf. DO-178C Table 7-1).

Objective of the software quality assurance process is to verify that the process is implemented according to DO-178C and that the implementation is applied during software development. Process assurance activities may have an overlap with respective activities of system processes or company-wide quality management system standards like EN ISO 9000ff. The process finally requires a Conformity Review "for a software product submitted as part of a certification application" (DO-178C 8.3).

The Certification Liaison Process briefly describes the communication with the authority and deliverables. However, authority clarifications, like [43], provide a far more detailed structure, for example by introducing "stages of involvement".

Software Life Cycle Data

Software Life Cycle Data is the data produced and controlled during the software life cycle. Handling this data must fulfill the requirements defined in DO-178C 11.0 and is task of configuration management. All before-mentioned plans, standards, development outputs (i.e., HLRs, Design, Source Code, Executable Object Code), as well as verification cases, procedures, results, and trace data are part of this data.

Traceability

For safety-critical software, traceability is a mandatory component of approval and certification processes [46]. Also in DO-178C, the concept of traceability plays an important role. It is an important concept of software engineering to ensure completeness, consistency and the absence of undocumented functionality, but also to ease impact analysis and change management throughout the whole software lifecycle.

The standard distinguishes between configuration traceability (cf. DO-178C 7.2.2), development traceability (5.5) and verification traceability (cf. DO-178C 6.5). Figure 7 illustrates the required development and verification traces between artifacts. It also references relevant paragraphs of the standard. Traceability into object code is only required for DAL A (cf. DO-178C 6.4.4.2). All development and verification traces must be bidirectional as stated in the respective sections of DO-178C. DO-178C summarizes development and verification traceability information under the term *Trace Data* (cf. DO-178C 11.21).

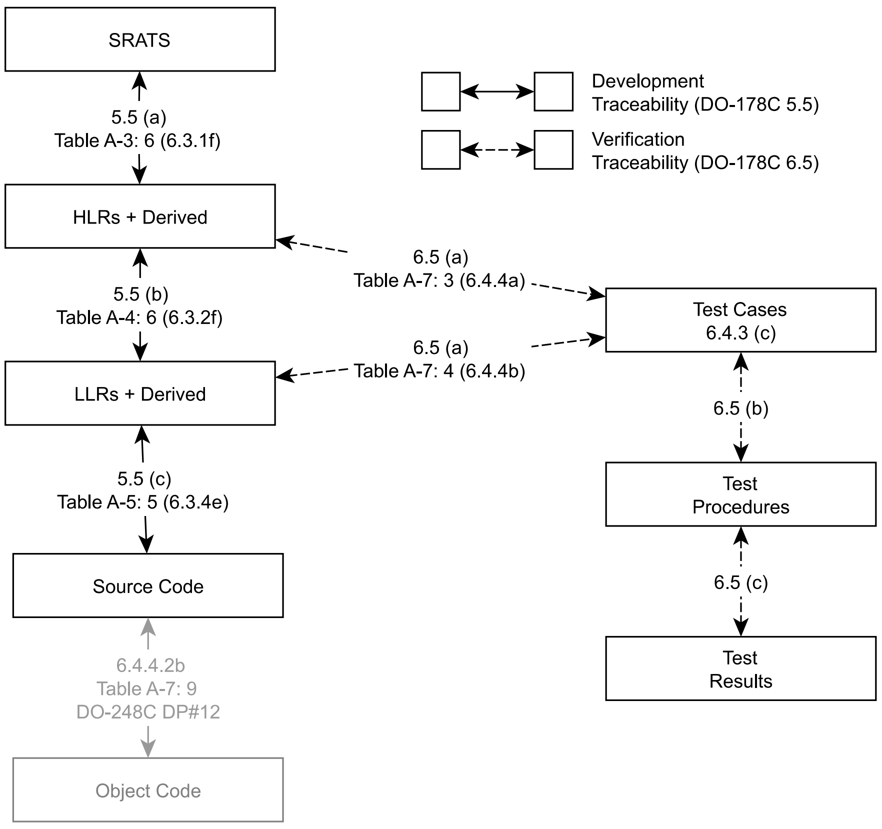


Figure 7: Development and verification traceability according to DO-178C

2.4 DO-331

The supplement DO-331 addresses the role of models in software development, modeling techniques, and how they can be utilized for DO-178C software life cycle activities and data. DO-331 is supplemental to DO-178C. Many parts remain untouched. For example, traceability is handled as in DO-178C and the respective objectives are fully applicable (cf. DO-178C MB.5.5).

In terms of DO-331, a model is “an abstract representation of a given set of aspects of a system that is used for analysis, verification, simulation, code generation, or any combination thereof. A model should be non-ambiguous regardless of its level of abstraction” (DO-331 p. 82). For example, an illustration used as additional information for an HLR is not subject to DO-331, since its formality is not leveraged for any of the aforementioned tasks.

The supplement clearly distinguishes between *Design Models* and *Specification Models*. According to DO-331, a Specification Model replaces “high-level requirements that provide an abstract representation of function, performance, interface, or safety characteristics of software components” (DO-331 MB.1.6.2), whereas “a Design Model includes LLRs and/or software architecture” (DO-331 MB.1.6.2). These are the only types of models possible to replace Software Life Cycle Data.

For each model type, DO-331 introduces additional objectives as well as verification requirements. Central is the introduction of a fourth standard, the Software Model Standard.

DO-331 also promotes model simulation as new verification method to satisfy dedicated objectives. Figure 8 shows the objectives, which may be fulfilled using model simulation on the Design Model. Bold are the most commonly satisfied objectives. Model simulation can be applied on both the Specification and Design Models. The standard distinguishes between model simulation for verification of the model (cf. DO-331 MB.6.8.1) and model simulation for verification of the executable object code (cf. DO-331 MB.6.8.2). The first may be used to satisfy review and analysis objectives and latter to replace or support testing on the Executable Object Code. However, independent of how simulation is used, the guidance of DO-178C for review and analysis (cf. DO-178C 6.3) and testing (cf. DO-178C 6.4) must be satisfied.

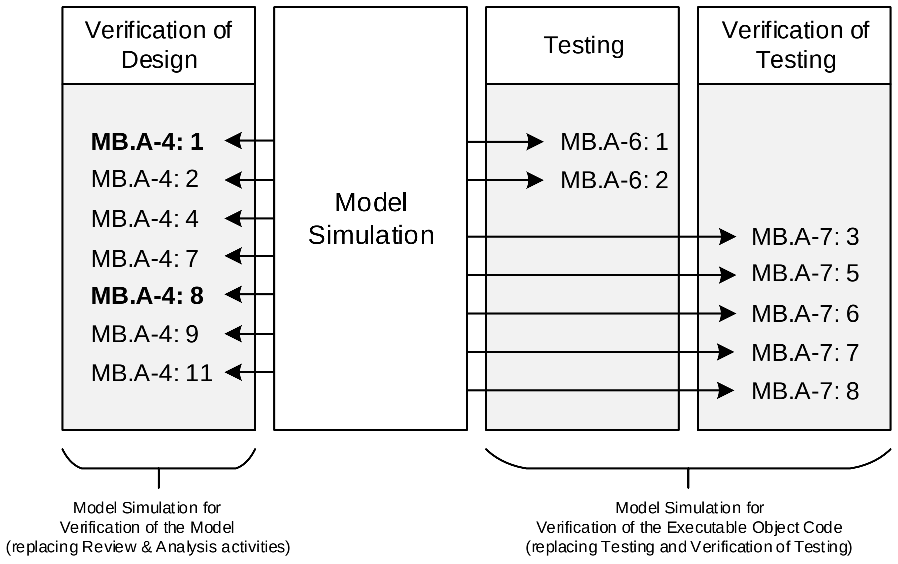

Verification of Design
Model Simulation
Testing
Verification of Testing
MB.A-4: 1
MB.A-4: 2
MB.A-4: 4
MB.A-4: 7
MB.A-4: 8
MB.A-4: 9
MB.A-4: 11
MB.A-6: 1
MB.A-6: 2
MB.A-7: 3
MB.A-7: 5
MB.A-7: 6
MB.A-7: 7
MB.A-7: 8
Model Simulation for Verification of the Model (replacing Review & Analysis activities)
Model Simulation for Verification of the Executable Object Code (replacing Testing and Verification of Testing)

Figure 8: Model simulation usage possibilities according to DO-331 MB.6.8

2.5 DO-333

DO-333 is the DO-178C supplement handling the application of formal methods. Formal methods base on mathematical methods of theoretical computer science to rigorously verify software. DO-333 applies, if a *formal analysis* is applied on a *formal model* to meet verification objectives.

A formal model is "an abstract representation of a given set of aspects of the software that is used for analysis, simulation, and/or code generation" with "an unambiguous, mathematically defined syntax and semantics" (DO-333 FM.1.6.1). Formal models may either be a direct development output or may be derived from software artifacts. For formal analysis on a Design Model, both is often the case, since further intermediate representations are derived from a formal model.

Formal analysis “implies that all execution cases are taken into account, achieving exhaustive verification” (DO-333 FM.1.6.2). Exhaustiveness is the main difference to any other verification method. In addition, DO-331 FM.1.6.2 requires that formal analyses are sound, i.e., it “never asserts that a property is true when it is not true” (DO-331 FM.1.6.2). But it may raise false positive (false alarms) or undecidable results.

Asserted properties are either provided by the user or automatically embedded in the tool. For example, a formal method for code analysis may have embedded properties for division by zero. But a user could also implement own properties by adding `assert` statements to the code, which are picked by the formal method.

Depending on the verification objective of DO-178C, for which a formal method is applied, additional DO-333 objectives have to be fulfilled. In the scope of this work, model checking and abstract interpretation tools are used across the process for various verification objectives. For their use cases, DO-333 definitely applies, but the additional objectives are not addressed in the scope of this thesis. Since they are mainly supplemental, they can be easily added if needed.

2.6 DO-330 Tool qualification

Any tool fully or partially eliminating, automating, or reducing a process required by DO-178C is subject to *tool qualification*, if the output of the tool is not verified in the Verification Process (cf. DO-178C 12.2.1). The tool qualification process provides confidence in the tool and is described in DO-330 [4].

DO-330 is independent of DO-178C. DO-330 distinguishes five *tool qualification levels* (TQL), determining, how rigorous the tool qualification process is. The TQL reaches from 1 (the highest criticality) to 5 (the lowest criticality).

Defining a tool evaluation strategy, which defines the necessary TQL, is up to the respective development standard. DO-178C bases the TQL on the tool categorization and the software level of the developed software. It distinguishes criteria 1, 2, and 3 tools (cf. DO-178C 12.2.2). If the tool output is part of the resulting software, it is considered as criteria 1 tool (development tool). Criteria 3 tools are typical verification tools. If the output of a verification tool is additionally used to justify the elimination of a verification or development process, it is considered as criteria 2 tool. For example, the SCADE code generator is used as criteria 1 tool in [47], Polyspace Code Prover as criteria 2 tool [48, p. 4-1] and Simulink Test as criteria 3 tool [49, p. 4-1].

DO-178C Table 12-1 shows how the TQL is finally obtained as a function of software level and tool criteria. For a DAL B software, a criteria 1 tool has TQL-2, a criteria 2 tool TQL-4, and a criteria 3 tool TQL-5.

The dimension of tool qualification is just superficially covered by this thesis.

2.7 Model-based design and software development

Models can be used along the full development life cycle, from system to software development. The first use case is for plant modeling, like aircraft dynamics or mechanical systems. *Plant models* are simulation models of physical systems abstracted by mathematical equations. Simulation of a physical system supports controller development and validation of requirements on system level. For example, aircraft dynamics are approximated with simplified equations of motion. Input of the model are control commands, output is the state of the aircraft. This model can directly be coupled with a controller. Such a setup is called model-in-the-Loop.

The controller is designed in *algorithm design models*. Goal is rapid prototyping, model-in-the-loop validation, but also linearization and the application of linear system theory methods. These models are mainly created in in the system design processes.

When it comes to models used for code generation for a controller, specific *software design models* are created. These models respect code generation limitations and have to be discrete and with a fix step size. Algorithm design models can be refactored into software design models under certain conditions, but could also be used as executable specification model.

Another use case are *requirement models*, which formalize textual requirements. In combination with algorithm or software design models, these can be used as requirement observers or for formal proofs.

Finally, there are *verification and validation models*, or also called “test harness models”. They implement test cases and procedures and can be plugged to algorithm or software design models. They can specify the input, but also evaluate the output. These models can also be test beds, e.g., for testing platform interfaces.

3 Project context

This section introduces the project context, for which the MBSwD process has been set up in this thesis.

As depicted in Figure 9, the considered item is a FCC, which is part of a flight control system. The flight control system spans all electro-mechanical components from control surfaces, control rods, actuators to onboard network, sensors, and computers. The FCC is one of many redundant computers. With the system architecture, the criticality of the item has been mitigated to IDAL B. Allocated system requirements to SW or HW, interface control documents (ICDs) as well as the IDAL are inputs to the item design processes.

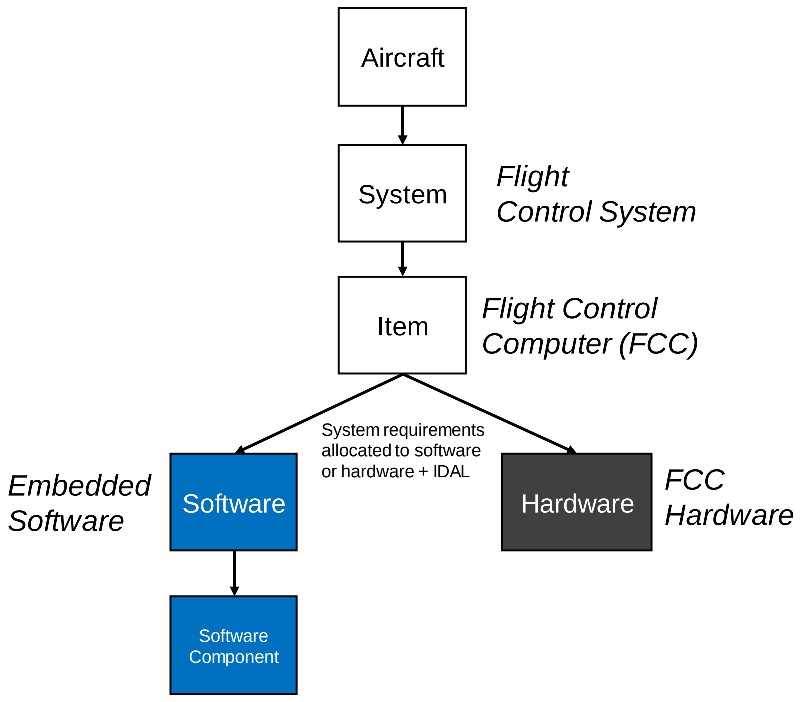


Figure 9: System breakdown

3.1 MBSwD as embedded process

The MBSwD process is not a standalone process. As part of the item design, it has a strong linking to the system processes, but also to the development of the hardware. This interplay often fades into obscurity, since in both industry and research, system, software, and hardware are isolated topics.

However, especially for flight control software, the system and software levels blur. For example, performance requirements for a stability augmentation system or handling qualities are formulated for the aircraft, but directly impact the control law design in the software. Model-based development even promotes a merge between system and software development as described later on.

In addition, it is fairly unrealistic that the whole embedded software is developed using model-based techniques under DO-331. On the one hand, there are technical limitations of the modeling tools and auto-code generators making traditional programming more efficient in some use cases. For example, handling and transformation of incoming and outgoing byte streams of component interfaces is a problem, which is hard to implement in SL (cf. [31]). On the other hand, a significant number of objectives remains untouched by DO-331 and is covered by traditional DO-178C development.

Figure 10 illustrates the chosen integration of system development and item design processes. Hardware and software development are both driven by allocated system requirements (as part of “Item Requirement Identification”, ARP 4.1.7) and the assigned IDAL. The system process accompanies the item design and manages the information exchange (ARP 4.6.1.3).

The DO-254 hardware process is not of special relevance in this thesis. It is assumed that the FCC has already been developed, satisfies the project target, and is available as commercial-off-the-shelf (COTS) product with the necessary documentation. The FCC hardware is briefly described in section 3.1.

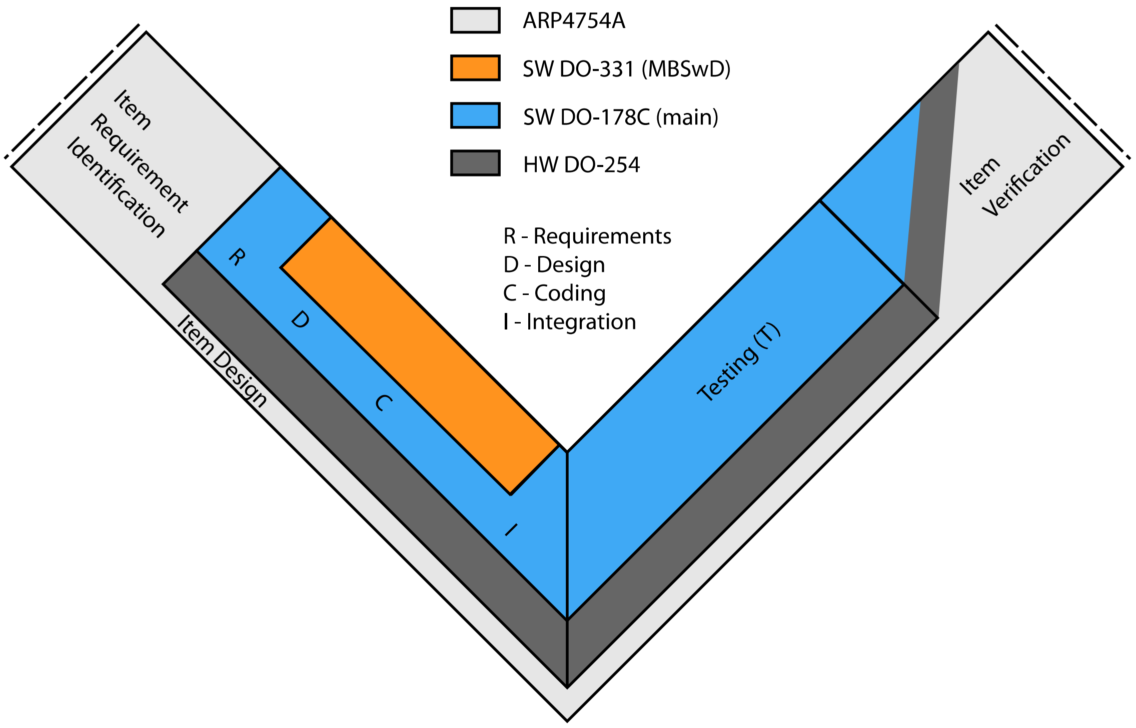


Figure 10: Integration of system, hardware, and software processes

On software side, it is distinguished between a traditional DO-178C software development process (blue) and a MBSwD process (orange). The main DO-178C process starts before the MBSwD process as illustrated in Figure 10 and defines, at first, a software architecture, decomposing the whole software application into *software components* (cf. section 3.3) and the respective HLRs. They respect considerations closely connected to the hardware, for example partitioning, execution rate, memory sections, or interface handling of the software.

Here is also decided, which software component shall be developed in the traditional DO-178C and which in the MBSwD process. Latter start a sub-process. Traditionally developed software components remain in the scope of the main DO-178C process.

Not all system requirements are directly refined in HLRs. Some of them are directly passed to the MBSwD process.

3.2 Hardware

Target environment for the software is the FCC hardware (Figure 11), which has been specified by TUM-FSD and developed by the Aircraft Electronic Engineering GmbH (AEE). It has been presented by Hornauer [50] and Nürnberger [37].

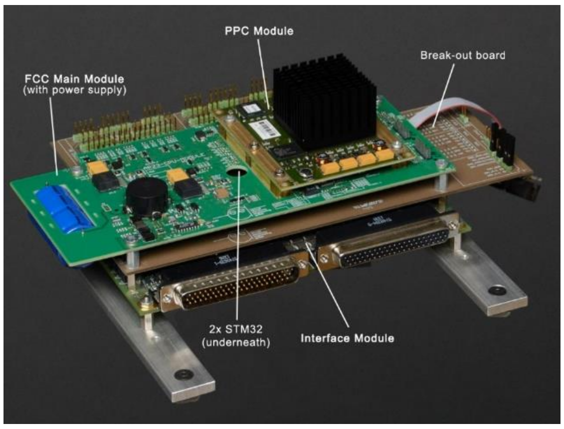


Figure 11: FCC hardware

For design of airborne hardware and environmental conditions, DO-254 and DO-160G [51], respectively, have been considered during the development. The FCC hardware can be segregated into three hardware modules (Figure 12). There is the *FCC main module* including power supply and two STM32 microcontrollers with ARM Cortext-M3 I/O processors. It carries a PowerPC (PPC) with a MPC8349 processor (further on called *main processor*) and the interface hardware module with the physical interfaces.

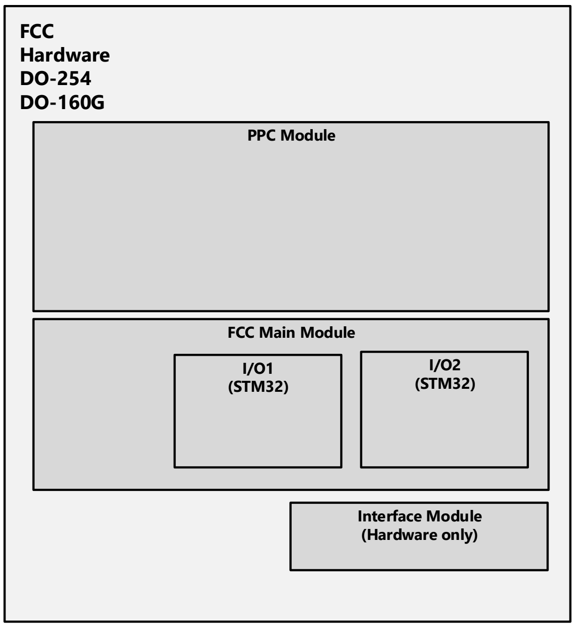


Figure 12: FCC hardware modules

The I/O processors handle the data exchange with the external interfaces and transfer it to the PPC on request via point-to-point dual duplex Ethernet interfaces as illustrated in Figure 13. The application software is allocated to the PPC.

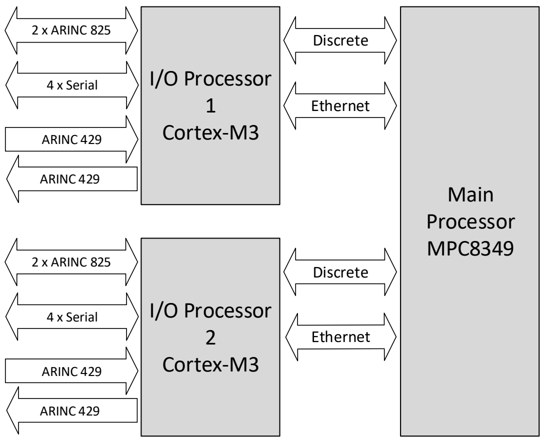


Figure 13: FCC processor communication (from [37])

Of special interest for the software process are the tracing capabilities of the FCC hardware. The Cortex-M3 processors of the STM32 provide a so-called “Hardware Trace” via the ARM Embedded Macrocell. Coverage can be collected with hardware probes. The MPC of the PPC only allows a “Software Trace”, for which handlers in the Source Code or Executable Object code must be registered (so-called “instrumentation” of the software).

3.3 Software components

In the scope of a traditional DO-178C process, a software architecture has already been established distinguishing different software components (cf. [50, 52] and Figure 14).

The main flight control algorithm is contained in the single-rate, cyclically called *software application*, which shall be implemented with MBSwD according to DO-331. It is embedded in the conventionally developed, handwritten *software application framework*. Both software components are executed on the PPC module.

Additional software components, like the Board Support Package (BSP), device drivers, support libraries, or the Real-Time Operating System (RTOS), are considered COTS products, providing the required documentation and evidence or qualification kits to apply them in a safety-critical application. The handling of these COTS products is not further detailed in this thesis.

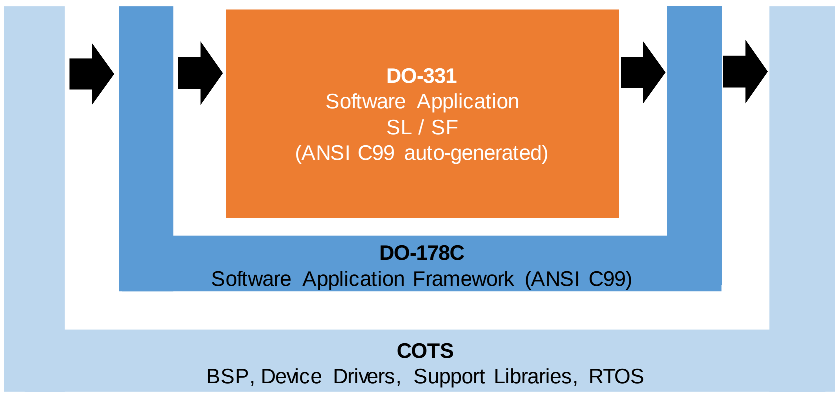

DO-331
Software Application
SL / SF
(ANSI C99 auto-generated)
DO-178C
Software Application Framework (ANSI C99)
COTS
BSP, Device Drivers, Support Libraries, RTOS

Figure 14: Software components on PowerPC based on considerations from [50] and [52]

Implementation details of the software application framework have been published by Nürnberger and Hochstrasser [37] and are only shortly summarized here. As depicted in Figure 15, the main processor executes a foreground as well as a background task and is triggered by a periodic interval timer. In each cycle, the first action is to request data from the I/O processors and perform a busy wait for a fix time interval. This interval has been chosen based on worst-case considerations. Subsequent to the waiting interval, the foreground task extracts the received data and after that calls the main cycle of the software application ("execute controller"). After finishing the cycle, the output data is packed and forwarded to the I/O processor, which itself transmits them to external interfaces. Finally, hardware check functions are called, before the periodic interval timer is reset again. The whole software on the main processor is implemented without interrupt service routines.

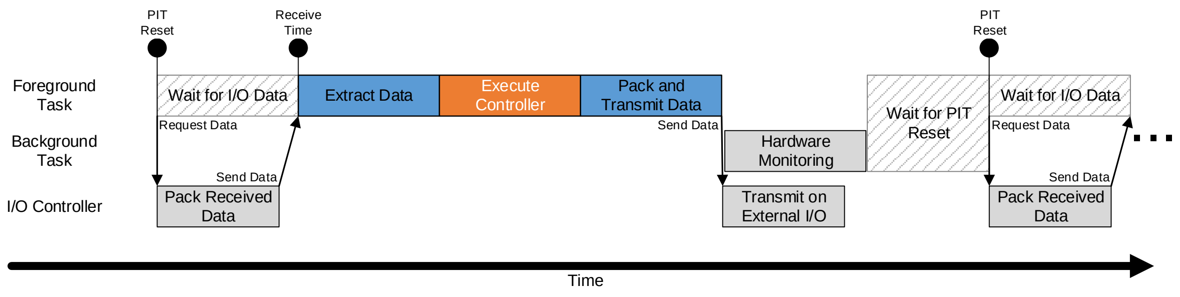


Figure 15: FCC sequence of tasks from [37]

The data transmitted through the interfaces of the FCC hardware are defined in the system architecture and ICDs for the respective network protocols. All interfaces of the system architecture are managed in central ICD databases realized in Excel. The ICD database is used to generate XML ICD data files for each interface. The XML language has been chosen due to its simple transformability and wide-spread tool support. All XML interface specifications can be transformed to PDF reports for documentation. The generic ICD definition process has been implemented by Markus Hochstrasser, Andreas Schwierz (Technische Hochschule Ingolstadt) and Lukas Steinert (TUM-FSD). The specification was supported by Stanislav Braun (TUM-FSD) and Markus Geiser (TUM-FSD) for the Sagitta project and Lars Peter (TUM-FSD), Lukas Steinert (TUM-FSD), Patrick Lauffs (TUM-FSD) and Christoph Dörhöfer (TUM-FSD) for the DA-42 research aircraft [15].

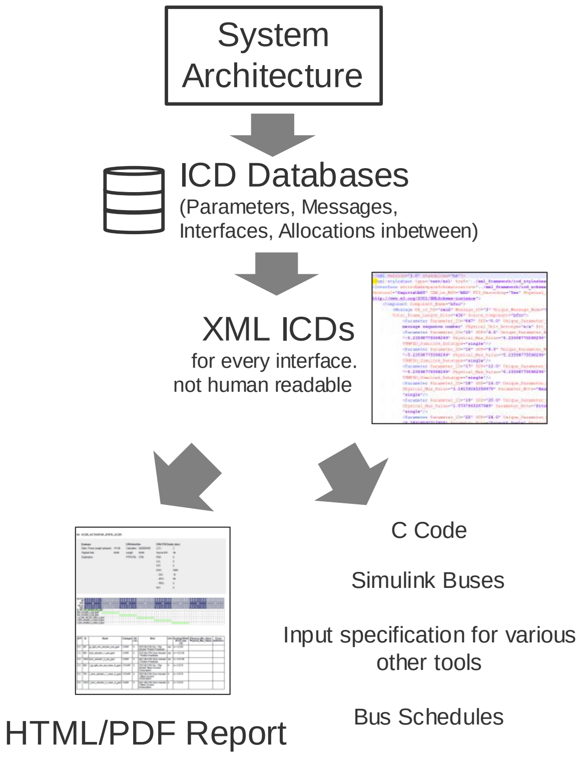


Figure 16: Interface control and generation process

The XML interface specifications are used to generate large parts of the software application framework, i.e., the “Extract Data” and “Pack and Transmit Data” functions of Figure 15. The code performs:

- data type conversion.
- copying the fields of the messages into the pre-defined location in the input structure of the main model-based software application, if a new message was received.
- packaging of the output structure fields into the message byte stream, if the application code requests sending the message.

The code generator has been implemented by Lukas Steinert, Markus Hochstrasser, and Kajetan Nürnberger of TUM-FSD. The framework code is not in the scope of this thesis.

3.4 Summary of assumptions

In this section, a couple of additional assumptions relevant for the MBSwD imposed by the system, hardware, and software processes are presented.

AS 1 – System process is defined and interfaces between system and software process are established.

For the system development process, this thesis relies on various achievements from projects at TUM-FSD, which planned a system process for flight controls following ARP-4754A. Therein, the application of modeling and simulation is a central part of the validation and verification strategy. Especially model-in-the-loop simulations are used to validate requirements.

AS 2 – FCC hardware is defined and available, and debugging capabilities are understood.

Cf. section 3.1.

AS 3 – System and software requirements are documented in Polarion.

System and software requirements are documented in Polarion. The assumption is that every requirement is a separate, so-called *work item* with a unique ID.

AS 4 – Process to handle derived software requirements in Polarion exists.

A process exists to handle derived software requirements in Polarion, identify them, trace them, and forward them to the system safety assessment.

AS 5 – Process to allocate system requirements to the software components in Polarion exists.

A process to allocate requirements to software components exists, for example by setting an ownership in Polarion.

AS 6 – Review and analysis for DO-178C software requirements exist.

As part of the integral verification process, review and analysis is used to verify the objectives of DO-178C Table A-3.

AS 7 – Design process breaks down the software into components.

Cf. section 3.3.

AS 8 – Fundamental functional and process-relevant DO-178C architectural and software design decisions have been made for the MBSwD component.

The MBSwD software component

- only interacts with the HW through the software application framework
- has a separate set of allocated requirements
- is cyclically called with a single step function. The execution of a step is not interrupted.

Cf. section 3.3.

DO-178C 2.4 and 2.5 give an overview of architectural and software considerations, which should be made on system level. Table 1 maps these considerations to the regarded software application.

Consideration	Design Decision and Rationale
Software Level	DAL B
Partitioning requirements	Not relevant for the nested MBSwD sub-process, but may play a role for the main DO-178C process.
Independence requirements	Not further considered in this thesis.
Multi-versions dissimilar software	The system process assesses the degree of dissimilarity. If multiple-version dissimilar software shall be used, the presented approach can be seen as one version.
Safety monitoring	Not relevant for the nested MBSwD sub-process, but may play a role for the main DO-178C process.
Parameter Data Items (PDIs)	For MBSwD, as described in AS 9.
User-modifiable software	Flight control software is not user-modifiable.
Commerical-Off-The-Self Software	COTS software is part of the whole software, but the focus is on the MBSwD part.
Option-selectable Software	No option-selectable software component is assumed, only a single configuration.
Field-Loadable Software	Not considered for the application software, but may play a role for the main DO-178C process. Field-loadable software mainly enforces requirements concerning the integrity of the loading process.

Table 1: Architectural and software considerations

AS 9 – Use cases for Parameter Data Items exist.

DO-178C introduces the formal construct *Parameter Data Item* (PDI), which describes “a data set that influences the behavior of the software without modifying the Executable Object Code and is managed as separate configuration item” (DO-178C 2.5.1). A PDI “contains only data (and no Executable Object Code)” (DO-248C 4.20.2). It describes both structure and values of the data set. The representation of PDI values in a target-readable format are *Parameter Data Item Files (PDI Files).* If certain conditions are met, the parameter data item values can be managed in a separate life cycle and a change does not require reverification of the Executable Object Code of the software they are embedded in (DO-178C 6.6).

In the considered application software, PDI Files are only introduced for the software manufacturer to accelerate adaption steps. Typical example for PDIs are lookup table values or controller gains. PDI Files, software application, and software application framework are approved as part of the type design data. The software is not field-loadable, but fully loaded in the factory. To reload the data, the FCC has to be opened and certain verification steps have to be repeated. The considerations for user-modifiable software do not apply, since it is not intended by the use case and not possible by implementation to be changed by the user.

No parts of the software shall be option-selectable by PDIs, i.e., PDI configurations shall not be used to activate and deactivate code.

PDIs can contain any data, but are not subject to aeronautical data, which must show compliance to RTCA/DO-200B [53] to obtain an airworthiness approval [54].

AS 10 – Source Code language is C99 and a Software Code Standard exists.

The Source Code language is C99. The Software Code Standard must be applied during implementation.

AS 11 – Tool chain for compilation, linking, and loading of Source Code is defined.

Figure 17 briefly outlines the process and tools to convert Source Code to Executable Object Code as well as to load it into the target. Preprocessor and compiler generate relocatable object files from the source files. Relocatable object files, for example, use relative memory addresses. The linker merges relocatable object files into a single executable object file and resolves symbols. The loader copies the program into memory and may also perform a relocation depending on the start address [55].

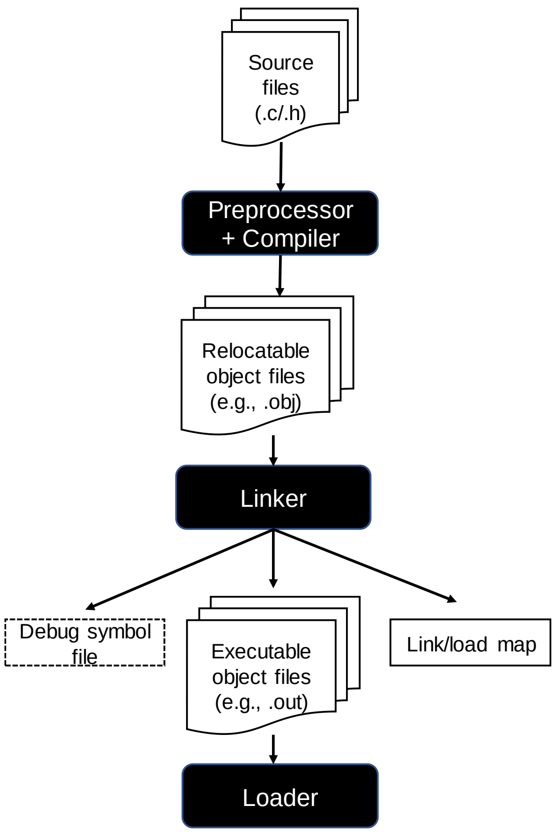


Figure 17: Compilation, linking, and loading

TUM-FSD has an established tool chain for the steps in Figure 17. As preprocessor, the GCC Preprocessor is used [56]. Cross-compiler for the PowerPC is CompCert, which "is formally verified, using machine-assisted mathematical proofs, to be exempt from miscompilation issues" [57, p. 7]. The formal verification increases trust in the compilation and leverages the relevance of Source Code analysis techniques, but it is important to mention, that no certification credit is sought for it. CompCert is an optimizing compiler. Since DAL B software is developed (cf. AS 8), traceability analysis into object code is not required and optimization does not negatively impact verification activities. For linking, the GCC linker is used [58]. A loader is not used. The addresses are relocated to a given entry address by the linker.

AS 12 – Processes and tools for the review and analysis of the integration process are defined.

Review and analysis happens along the objectives of DO-178C Table A-5. Stack analysis is assumed to be executed with VerOStack[5] and worst case execution time (WCET) analysis with AbsInt aiT [37]. Both analyses are not further detailed in the scope of this thesis.

[5] https://www.verocel.com/tools/stack-analysis/ [Accessed on: Jun. 24 2019]

AS 13 – Processes and tools for testing of the fully integrated Executable Object Code exist.

Tools to execute test cases on the fully integrated HW exist. Data and control coupling coverage can be measured.

AS 14 – Processes and tools for change management are defined.

Any software component is considered as separate Configuration Item in the sense of DO-178C and treated as a unit in the software configuration management process.

Chosen tool for version control is GIT. Every configuration item is in a separate GIT repository. Baselines are tagged revisions in the version control system as described in the configuration management plan. By committing data to the version control system, the configuration of the configuration item is unambiguously identified and traceability between configuration items is automatically established.

4 Modular development process (part 1)

4.1 Objective

Objective is to derive the new modular, model-based development process from the abstract DO-178C/DO-331 process framework. This includes the definition of a software life cycle, which is compliant to DO-178C, transformation activities of the standard into actual tasks, and the setup of a verification strategy.

4.2 State-of-the-art

Adoption risks of model-based-design

A recent survey funded by NASA revealed a strong increase in the number of companies using model-based design with auto-code generation in a safety-critical context over the last years [59, 60]. The companies participating in [60] clearly identified an improvement in productivity and software quality coming along with MBSwD process, but also emphasized difficulties.

The adoption of MBSwD processes still raises challenges. In [60], an overwhelming majority of participating companies emphasized the difficulties they had transitioning from traditional software engineering to MBSwD and learning the effective use of tools. It aligns with a survey performed by MathWorks [61], which rates tool complexity as one of the most common adoption challenges, which their customers have. Also companies interviewed by Broy [62] saw “extremely high process redesign costs” as main negative aspect of MBSwD.

In the aerospace domain, only a couple of large companies have elaborate processes based on SL/SF for DO-178C/DO-331. Bhatt presents an in-house tool-chain for development and verification of DO-178B software based on SL/SF [63]. Smaller companies face a significant adoption risk due to missing initial tool knowledge and unknown dependencies. Some research projects help to lower this entrance barrier, e.g., Reke [64] presents a light-weight process for MBSwD in the automotive domain.

Main challenge is that DO-178C and DO-331 are process frameworks defining what has to be done, but not how. They have to be tailored to and implemented for a specific company or project. Tools must be selected, review checklists must be written, and the activities must be documented in procedures. Especially in MBSwD, with automatically generated code, this requires over- and foresight.

Tool vendors providing a “chain” of seamlessly integrated tools along the development process publish workflows and handbooks for different software development standards, mapping the objectives to their tools. For example, Esterel Technologies provides a “Methodology Handbook” for SCADE Suite [47] and MathWorks ships a workflow documentation with the tool qualification kit [32]. The workflow of MathWorks is also outlined in a set of publications for DO-178B/C and DO-331 [65, 66] and for software development standards of other domains in [67, 68]. However, most of these documentations remain abstract. They hardly lower the challenge of project adoption. For example, they do not define activities, procedures, review lists, or the required tool setup including rationales. This can be ascribed to the difficulty of formulating a universal solution. The planning and setup requires broad experience and always depends on existing processes in the company, available tools (for example for configuration management, requirement management, compiler,…), or the project itself (software level, target requirements, integration requirements,…).

A collection of dependencies is given by Paz [69], who assesses model-based approaches for the applicability in DO-178C projects and the reachable process coverage. The proposed characteristics clearly show the complexity of the interfaces and the evaluation concludes that today’s solutions have weaknesses especially concerning traceability and process coverage.

Various publications discuss certain aspects. Marques discusses impacted DO-178B objectives by MBSwD and the advantages of model simulation and model coverage analysis [70]. Wu [71] explains seamless model-based design and verification of controllers in SL/SF, but does not make any connection to applicable standards like DO-331. Basagiannis discusses, how co-simulation of cyber-physical systems in combination with formal methods can support the software certification process [72].

Several authors [73–75] point out important topics for the adoption of model-based design, but mainly focus on organizational strategies, which are independent of the selected process standard.

Existing examples for model-based development along DO-178C/DO-331 are illustrative, but have a limited scope. For example, Potter [76] provides a small, but holistic demonstration for helicopter flight control under ARP-4754 and DO-331. Available DO-331 workflow descriptions for SL/SF do not address team-based aspects and scalability [32, 65, 77–79]. In contrast, publications for scalability and team-based work in SL/SF remain general and do not address safety-critical software development [80, 81].

To sum up, significant process adoption challenges still exist. Published processes remain vague, are general, and lack of full process coverage. Provided examples are often small and do not address scalability issues. Setting up a reference process is most often an iterative process requiring significant resources, which only large companies can afford.

Modularization and scaling challenges

Modular software development is attractive from many viewpoints. It leverages independent development and verification, but is also the key factor for software reuse and thus an economic factor. The later the integration happens, the more work can be shifted to modular subprocesses, and the higher is the benefit. [8] outlines software reuse and agile workflow as methodology to counteract software complexity and costs, but also recognizes that many of these technologies are still difficult to apply for safety-critical aviation software.

Modularizing the software itself is a well-known software design concept. Sommerville calls it “component-based software engineering” [10] and Saleh treats modularity as one of the top reccuring software design concepts ([82] p.147).

The growing importance of modularization can be observed across all engineering domains by arising interface standards for hard- and software componentization. In the automotive domain, the AUTOSAR (AUTomotive Open System ARchitecture) is the most popular standard[6]. AUTOSAR describes a standardized software architecture for electronic control units (ECUs) and their communication. A similar, standardized architecture concept in aerospace is known under the keyword “Integrated Modular Avionic” (IMA), which is ascribed to an avionic hardware component with standardized hardware and software interfaces. For software-software interfaces, ARINC 653 “Avionics Application Software Standard Interface” [83] describes a service-based approach.

AUTOSAR and ARINC 653 are platform- and service-based concepts with an extensive runtime environment layer responsible for resource allocation/partitioning and scheduling. In aerospace, such architectures are less common than in the automotive domain. One reason therefore may be that aerospace software must always be certified together with the hardware component as a whole. The effort to certify a runtime environment is significant and certifiable runtime environments are thus just affordable and economical for large manufacturers. In addition, the service-based concept applied on flight control algorithms is not as beneficial as for other software, since a majority of the software is application-specific.

For many smaller applications, a more lightweight approach without a runtime environment is more feasible. One single runtime process without scheduling or memory partitioning requirements drastically reduces the certification effort. This was also the approach taken for the flight control algorithms in the projects at TUM-FSD. Anyway, distributed and concurrent development of the software is desirable, especially to create the necessary certification evidence in parallel with the module development.

[6] https://www.autosar.org/ [Accessed on: Sep. 07 2019]

The above mentioned standards only address software architecture, but not how the software life cycle looks like. Componentization is addressed in DO-178C just superficially, although it raises new questions concerning interface handling, or data and control coupling. A few publications addressing agile development life cycles in a DO-178C development exist, which are modular. [12, 13, 84, 85] superficially discuss and evaluate different variants of DO-178C software life cycles, including incremental and functional workflows, but do address the process breakdown on activity-level.

Also SL/SF, Embedded Coder, and the surrounding verification tool environment just partially leverage distributed, component-based workflows. Many features exist, like Simulink Projects, which ease component-based work [80], but a consistent workflow is not provided. Modeling guidelines for safety-critical software models do not address componentization. Also the DO Qualification Kit does not address a modular process [32].

In the work of this thesis, an efficient, modular MBSwD process has been developed, which avoids sophisticated architectural concepts and is compliant to DO-178C. Various tool extensions and modeling rules have been developed closing the gap, which the SL/SF tool chain leaves.

4.3 Structure

First step is the definition of a software life cycle. DO-331 knows different kinds of model usage. Section 4.4 addresses the selection of model usage for control algorithms, which already have very concrete requirements and models at a higher level. Heart of the life cycle is the new modular MBSwD, which is presented in section 4.4.2. Section 4.4.3 introduces the novel modular code generation approach, which is an essential part of the process.

Section 4.5 address the selected development tasks for the design and coding process, which satisfy both DO-331 objectives and modularization requirements.

In section 4.6, verification tasks for the design and coding process are defined. The verification tasks are separated into review and analysis (section 4.6.1 and 4.6.2) and model simulation with testing (section 4.6.3). Data and control coupling is separately discussed in section 4.6.4 as overarching concept.

Part 2 in section 8 presents the detailed task procedures for both development and verification tasks. They have been moved to a separate, downstream section, since knowledge of model-based design in SL/SF is required.

4.4 Software Life Cycle

4.4.1 Model usage

The first question to be answered is, how models are used in the development process. As already discussed, DO-331 distinguishes Specification and Design Models. In both cases, the models replace requirements. Furthermore, DO-331 outlines five scenarios as in Table 2 and discussed in detail in DO-331 MB.B.17 Discussion Paper (DP) #1.

Example 1,4, and 5 only leverage the Design Model approach. In example 1, HLRs are derived from system requirements and the Design Model solely replaces the software design. In example 4, the Design Model replaces both HLRs and the software design. Example 5 goes even further and promotes a case, in which the Design Model is already developed as part of the system development. Example 2 and 3 introduce a Specification Model representing HLRs, from which either a Design Model is derived or textual LLRs and software architecture.

Process that generates the Life Cycle Data	**MB Example 1**	**MB Example 2**	**MB Example 3**	**MB Example 4**	**MB Example 5**
System Requirement and System Design Processes	Requirements allocated to software	Requirements from which the Model is developed	Requirements from which the Model is developed	Requirements from which the Model is developed	Requirements from which the Model is developed
					Design Model
Software Requirement and Software Design Processes	Requirements from which the Model is developed	Specification Model	Specification Model	Design Model	
	Design Model	Design Model	Textual Description		
Software Coding Process	Source Code	Source Code	Source Code	Source Code	Source Code

Table 2: Model usage examples from DO-331 Table MB.1-1

Information on acceptance of the different workflows by both industry and authorities is barely publically available. In the DO-178C handbook [47], Esterel follows example 1, similar to Erkinnen [65] and Marques [70]. Potter [66] and Weber [86] promote example 5 and Eisemann [79] outlines example 2. The DO Workflow document provided by MathWorks [32] is not written for a specific approach. It addresses SL/SF for both Specification and Design Models.

The chosen approach in this thesis supports example 1, 4, and 5. For all three scenarios, strong use cases could be found. Examples with specification models have not been considered, since they either exclude automatic code generation (example 3) or require two models (example 2).

Workflow 1

Example 1 is the most common approach. In many cases, system requirements have to be detailed for software purposes, e.g., robustness requirements with respect to external inputs, data types or interfaces. This scenario is inevitable and important.

Workflow 4

Example 4 is reasoned with the higher abstraction of Design Models compared to LLRs. For example, traditional HLRs often contain figures of state diagrams or truth tables. These diagrams are very close to the implementation in SF. In such cases, separate HLRs are just an artificially introduced level leading to “copy-and-paste development”. Important to mention is that this workflow does not merge HLRs and LLRs, since the system requirements take over the role of HLRs including all necessary activities and objectives (cf. DO-331 MB.5.0). The concerns of Position Paper CAST-15 “Merging High-Level and Low-Level Requirements” [87] are not applicable.

However, actions must be taken to ensure that the system requirements, which have been developed according to ARP objectives, serve as appropriate DO-178C HLRs. This has been addressed with an additional activity called “system requirement buy-off”, which is explained in section 4.5.1.

For SRATS, which become HLRs under workflow 4, it must be clarified, whether they are tested in the software or system process. This is a case by case decision. DO-178C 2.2.1h and 2.2.2i indicate that verification activities can be shared between system and software processes. Here, since software testing has stricter requirements (e.g., test coverage), all HLRs are at least tested under the MBSwD process. In consequence, testability is an important requirement for categorization into the workflows.

Workflow 5

Example 5 supports the special use case of flight control laws in software development. Flight control laws have to be designed on aircraft-level, since they involve knowledge of the whole system and are derived from aircraft-level handling qualities. Model-based development is commonly used to simulate and optimize the laws. Flight control law design models exist, but in a form not ready for software development. As Weber [86] points out, the traditional way would be defining software requirements “often representing written formulas practically ready to be pasted into FCS [flight control system, author’s note] software source code.” Essentially, existing models are artificially broken down into textual requirements.

Example 5 leverages an approach, which allows sharing the model between system and software process. However, a couple of arguments must be considered when planning this process.

Software architecture considerations or hardware considerations are barely known when the system design is developed. The approach taken in [86] is to develop a system-level model by system engineers and let software engineers transform and detail it to a Design Model later on. The system-level model *is discarded after that* and any further updates are made on the Design Model.

Additional effort is required to manage changes. Changes have to be coordinated between system and software engineering, if both are separate departments. A detailed change management process is given in [86]. Changes are implemented by the software engineering.

Finally, the modeling technique must be flexible enough to support both the use cases in system and software development as listed in Table 3. For controllers, system-level models often base on performance requirements, which need model-in-the-loop simulation and additional verification with hardware-in-the-loop or flight tests. SL/SF in combination with Embedded Coder can satisfy this dual role in general.

Since this workflow omits several requirement levels, it is applied in a very limited scope, i.e., in those cases, where additional requirement levels are just artificial. This is mainly the case for mathematical control algorithms. However, the transformation from system-level model to Design Model typically leads to new, additional software requirements like robustness requirements.

Except from the dual role and the enhanced change control, example 5 aligns with example 4. In workflow 5, the requirements are just selected from a higher level. The requirements, from which the system-level model has been developed, become HLRs. However, as in workflow 4, a buy-off is required.

System-level Model	Software Design Model
• Linearization • Application of tools of linear system theory • Parameter optimization • High performance for massive simulation (e.g., Monte-Carlo) • Model-in-the-loop simulation • Fulfill process of ARP-4754A	• Reflect, how the software works (LLR) • Robust against all inputs, robust outputs • Hold all information to generate standard-compliant code for a specific target and interfaces • Fulfill process of DO-178C • Verifiability on software level

Table 3: Application purposes of system-level models and Design Model

4.4.2 MBSwD process breakdown

Contribution 1: In order to ensure scalability and maximize concurrent, team-based, agile development, a new modular software development life cycle has been specified. It not only introduces an architectural breakdown of the software, but also distributes DO-178C process activities onto module- and integration-levels. Process activities from design to code verification can thus be executed earlier and on encapsulated entities. Evidence for certification is independent and reusable, which saves significant effort. A novel approach for generation of modular code, which differentiates from the broadly applied integral code generation, amplifies this effect.

In large projects, multiple developers join MBSwD. It is thus necessary to have a scalable strategy to work concurrently and contribute to MBSwD without causing conflicts. This thesis solves these aspects with the modular process introduced in the following, which allows concurrent team work and makes sure that the objectives of DO-178C/DO-331 are met. The first and obvious step is to break down the MBSwD software component into *software modules* as illustrated in Figure 18. One component can consist of multiple, nested modules, however there is a single *component module*. A component module defines the component interfaces and interacts with other components. All other modules are nested inside the component module. Each module can contain *units*. Units are the smallest, independently verifiable entities. They are further explained later on.

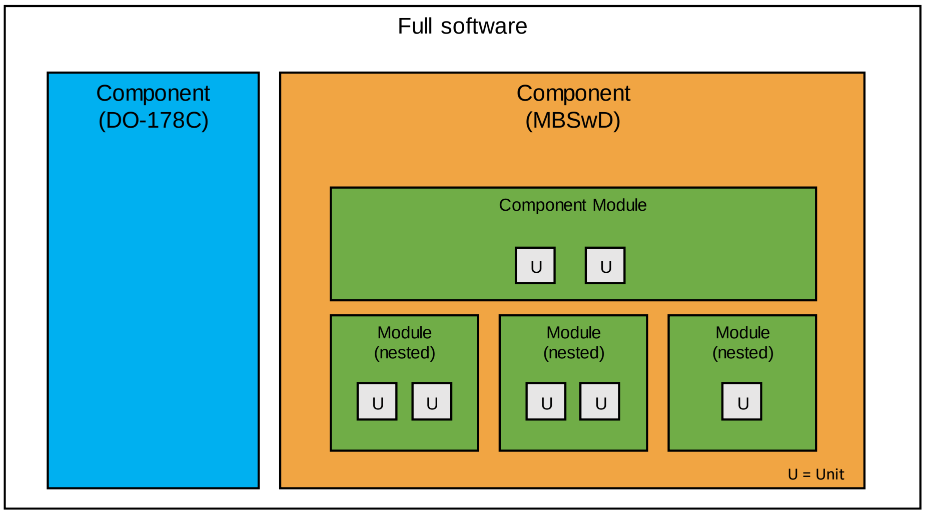


Figure 18: Architectural breakdown of software

Some process activities can be independently and concurrently performed in modules, others need the view of the whole component (for example, activities with regard to completeness).

The presented approach thus distinguishes two kinds of MBSwD processes as depicted in Figure 19, the *MBSwD component process (C)*, and the *MBSwD module process (M).* The component process covers tasks, which can *only* be done on component-level and need the full software application. One can say that they have *component scope*. The module processes are executed in isolation on module-level, they have *module scope*. A component module must follow both the component and module process.

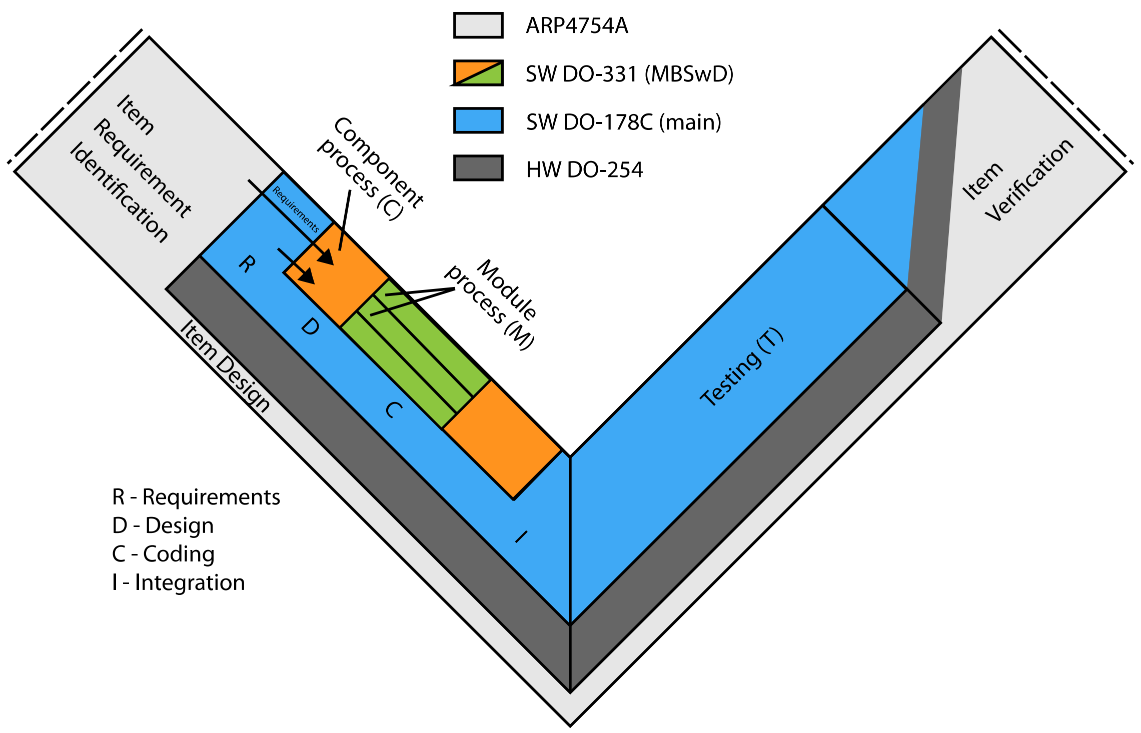


Figure 19: MBSwD process breakdown as V-model

A breakdown into modules is not straight-forward from a process point of view. It raises questions in the top-down workflow, e.g., how requirement allocation to modules is performed and how traceability is handled, or how interfaces are communicated and how modular code is generated. But also bottom-up, new problems have to be solved. For example, how coverage is aggregated or how the coupling between modules is assessed.

Figure 20 details the breakdown of Figure 19 with the DO-178C processes Requirements (R), Design (D), and Coding (C). Superscripts ^{C} and ^{M} indicate, to which MBSwD process they belong.

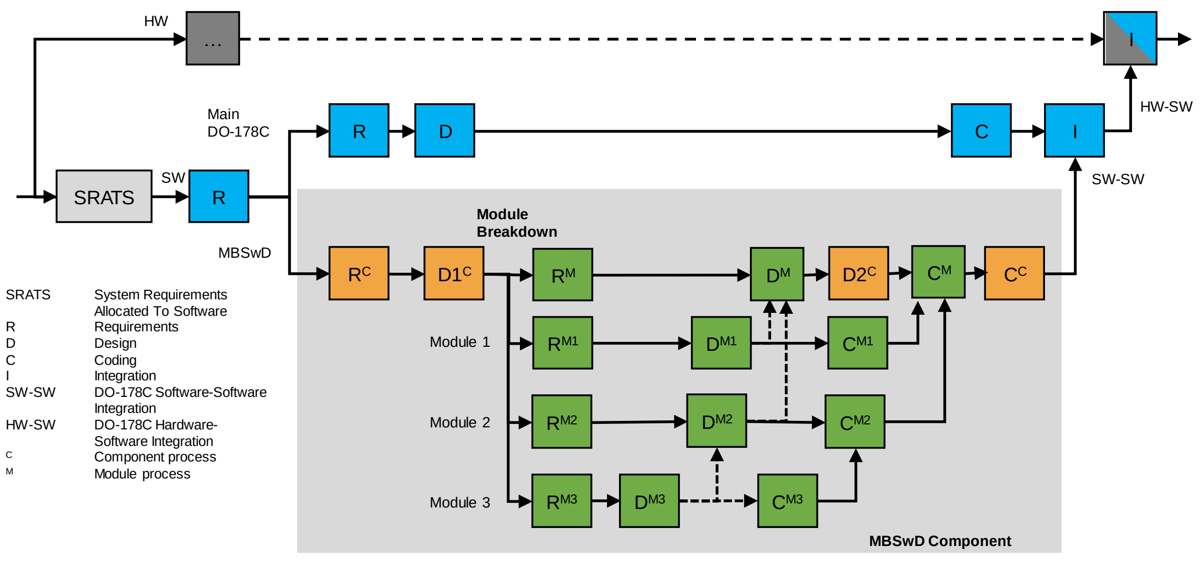


Figure 20: MBSwD process breakdown as flow diagram with three exemplary sub-processes

The main inputs to the MBSwD process are:

- system requirements allocated to software (SRATS)
- HLR allocated to the MBSwD component
- system-level model in case of a type 5 workflow
- IDAL (here IDAL of the FCC item)

The MBSwD component process begins with a common requirements process (R^C) as shown in Figure 20. Target of the process is to formally accept and categorize system requirements either into workflow 1, 4, or 5, and, if necessary, add or derive additional HLRs. For workflow 4 and 5, acceptance criteria are checked to make sure that the SRATS are compliant with the Software Requirement Standard.

The component design process has been split into two phases ($D1^C$ and $D2^C$). $D1^C$ is called the architecture phase, $D2^C$ the integration phase. From the allocated requirements of the MBSwD component, a module breakdown and architecture is established in $D1^C$. The module architecture is part of the Design Model, since it serves as integration model later on, too. The concept is further detailed in section 5.4.4. After $D1^C$, requirements are allocated to the nested modules and each module starts a module process.

In the module processes, developers concurrently concretize the requirements, document derived requirements, and establish traceability (R^M).

The requirements are implemented in Design Models (D^M). All Design Models are fully developed in SL/SF and traceability is established to the HLRs.

In each sub-level process, modular C code is generated for the Design Model with Embedded Coder (C^M).

Both design and code are integrated into dependent modules and finally into the component module as illustrated in Figure 20. After D^M of the component module, a single, fully integrated Design Model exists. The subsequent $D2^C$ mainly reflects verification on component scope of the fully integrated design. After C^M, all Source Code has been generated and verified on module-level. C^C specifies additional component-level verification activities on the full Source Code.

After that, the integration in the sense of DO-178C begins. The generated code is integrated with the code developed in the main DO-178C process, e.g., with framework code, converted to executable object code, and loaded onto the target (I).

4.4.3 Modular code

The intended workflow of Embedded Coder follows the approach that all Design Models are integrated into a single design. Code is then only generated once in the component process from the fully integrated Design Model (here called *integral code generation*). Figure 21 shows this alternative approach. Code would be generated after D^M of the component module. Consequently, all code review and analysis activities would have to be done in the component process.

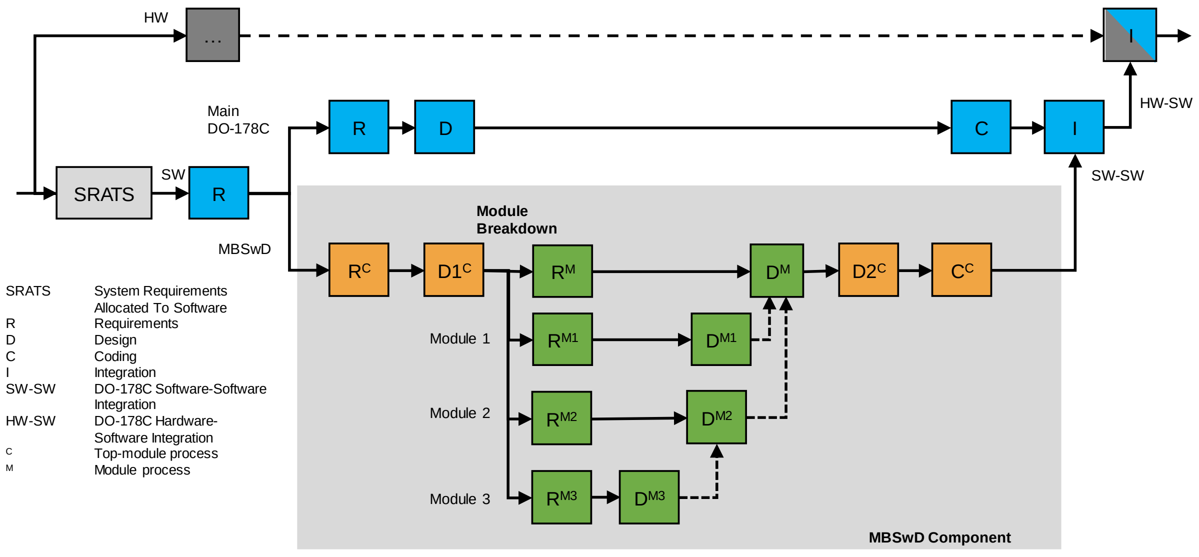


Figure 21: Traditional integral code generation approach

Integral code generation scores with high consistency of the code, but raises challenges concerning configuration management, limits reusability, and may produce significant, late rework. Latter is especially critical, since a code change always requires a model change connected with design verification.

In order to overcome these disadvantages, *generation of modular code* is focused in this work. In contrast to the MathWorks process, the software design is transformed to code in each module process separately and integrated *after* the coding process. Verification activities are executed in the module processes to a large degree.

Table 4 summarizes the improvements and new features, which have been added in order to achieve and solidify generation of modular code with Embedded Coder. Realization required an enhanced code generation workflow, which supports integration of existing code and handling of shared code (cf. section 8.1.2), but also design, coding, and modeling subsets optimized for modular code (cf. section 4.6.4).

Improvements	New features to be added
• Reuse of verified code possible (modular verified code can be used in other applications as is) • Limited impact of design changes on code (code of other modules remains untouched) • Higher scalability due to shorter code generation times and smaller cohesive entities with less complexity • Enables concurrent and modular workflows from design to code verification • Earlier detection of code verification issues and shorter rework cycles compared to integral code generation, since not all modules must be integrated, before code is generated and verified • Modular code is also the prerequisite for modular compilation, which may leverage even more possibilities for concurrent and incremental work • Less configuration management effort due to the limited impact of design changes on code and the possibility to manage code in independent version control repositories	• Build process adaptions, since Embedded Coder is made for top-level builds (e.g., handling of shared code, reuse of code,…) • Constraints the Design Model for generation of modular code and higher modeling discipline • Implementation of different verification tasks for component and module processes, since not all tasks can be fully executed on module level • Implementation of verification activities to guarantee interface compatibility, code consistency on code level as well as coupling

Table 4: Improvements and challenges of generation of modular code

4.5 Development Processes

Contribution 2: The abstract DO-178C development objectives have been concretized in specific tasks adapted for the modular development life cycle. Different model usage workflows, especially relevant for flight control development, have been considered. Defined integration-level tasks focus on the distribution of work (allocation of requirements) and the integration later on. Module-level tasks cover the actual implementation and traceability. This new distinction and the consistent consideration throughout task definition is one of the key concepts guaranteeing DO-178C compliance in a modular process.

4.5.1 Requirements Process (R)

This section briefly describes the requirements processes R^C and R^M with focus on a MBSwD process. It will not give detailed process guidance, since requirement engineering is similar to traditional DO-178C development.

According to DO-178C, target of the Requirements Process is to derive HLRs from SRATS. For a modular DO-331 process, this requires some additional considerations. As introduced in section 4.4, three workflows are possible. In workflow 1, the SRATS are refined with HLRs. In workflow 4, the SRATS represent the HLRs and the Design Model is directly developed from the HLRs. And finally, workflow 5 shares a Design Model with the system development process, which is directly developed from SRATS. This is equivalent to workflow 4, the requirements, which trace to the system-level model become SRATS.

In R^C, the SRATS are systematically categorized into the workflow they belong. This step is called the “system requirements buy-off” and is no activity, which is explicitly required by DO-331.

In $D1^C$, the software architecture is established and all requirements are allocated to software modules. After that, in the module process R^M, SRATS assigned to workflow 1 must necessarily be refined into HLRs, whereas workflow 4 requirements can directly be implemented. The whole process is sketched in Figure 22.

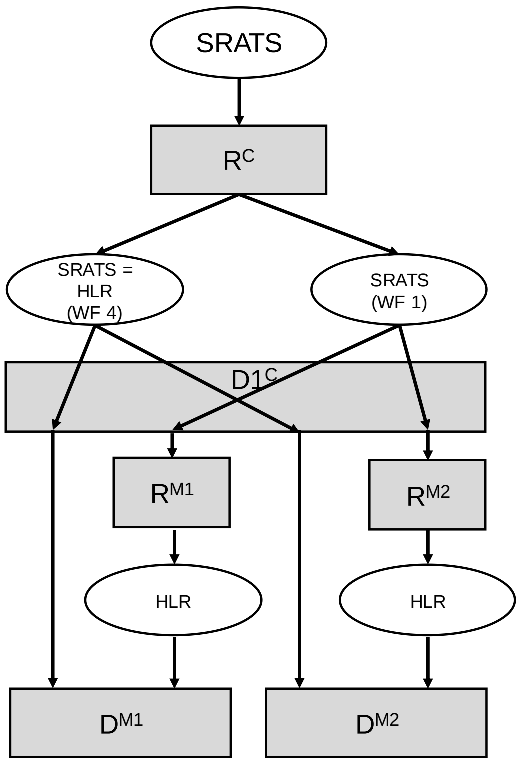


Figure 22: SRATS categorization and refinement

System requirements buy-off

Whereas the derivation of HLRs from SRATS is well understood and similar to traditional DO-178C development, the “system requirements buy-off” is a new concept and shall be elaborated.

According to DO-331 MB.B.17.4, the part of system requirements allocated to software “from which the Design Model was developed is considered to be the software HLRs, and therefore will need to comply with objectives of Annex MB.A Table MB.A-3”. The objectives of Table MB.A-3 are addressed by the Software Requirement Standard.

Workflow 1 derives HLRs from SRATS. The conformance of HLRs is checked against the Software Requirement Standard and, in consequence, they are full-value HLRs by definition. The SRATS used as HLRs in workflow 4 and 5 have been developed according to ARP-4754A 5.4.3 and thus only fulfill the requirements stated therein. However, they must comply to the Software Requirement Standard, too.

Instead of a full reverification against the Software Requirement Standard, a gap analysis is done. This avoids duplicate verification. The criteria are quite sensitive and are checked by a review board with specialists from both the software and hardware process. Otherwise, the risk of unintendedly skipping a necessary requirement level would be high. The chosen workflow is documented in the requirements.

4.5.2 Design Process (D)

DO-178C and DO-331 address necessary activities for the Design Process in MB.5.2.2 and the objectives MB.A-2:3-5,9,10 of Table MB.A. Output is the Design Description as described in MB.11.10 and Trace Data as in MB.11.21.

From the objectives and paragraphs, *tasks* have been derived. Tasks concretize activities with tool selection, configuration, or checklists. All tasks are summarized in the so-called “task table”, for the Design Process in Table 5. Each row in the table describes one task. The first column titled “Proc” indicates the process, in which the respective task has to be executed. If two processes are given, the task has two variants. The second column gives the name and ID of the task. The third and fourth column list required inputs and outputs to perform the respective task. The fifth column indicates, which DO objectives this task satisfies. The last column refers to the relevant section in this thesis, which covers the task.

Entry criteria for D1^C is that the system-requirement buy-off has been performed in R^C, i.e., requirements are categorized according to their workflow.

Exit criteria for D1^C is that the software architecture has been defined, requirements are allocated to the module.

Entry criteria for D^M is the exit criteria of D1^C and that workflow 1 SRATS have been refined (R^M).

Exit criteria for D^M is that allocated HLRs have been implemented, derived LLRs have been documented, traceability is established, and the design has been documented. Dependent modules must be available. Related verification tasks have been performed.

Entry criteria for D2^C is the exit criteria of D^M for the component module.

Exit criteria for D2^C is that related verification tasks with component scope have been performed.

Proc	Task ID - Name	Inputs	Outputs	Mapped DO-331 Objectives of Table MB.A-2	Section
D1^{C}	**SwDP-DP-MB 1 –** Development of the software architecture in SL/SF	Higher-level requirements allocated to software component	Software architecture (as part of the Design Model)	N/A	5.4.4 5.6.3
D1^{C}	**SwDP-DP-MB 2 –** Allocation of requirements to modules	Higher-level requirements allocated to software component	Allocated requirements to module	N/A	6.5.3
D^{M}	**SwDP-DP-MB 3 –** Design Model implementation in SL	Self-contained, allocated set of HLRs or system-level model Software architecture Dependent software modules	Design Model (of software module)	3 – Software architecture is developed 4 – Low-level requirements are developed 9 – Design Model elements that do not contribute to implementation or realization of any software architecture are identified 10 – Design Model elements that do not contribute to implementation or realization of any low-level requirement are identified	4.6.4
D^{M}	**SwDP-DP-MB 4 –** Tracing to higher-level requirements	Design Model for a self-contained set of requirements Higher-level requirements	Trace Data (of software module)	4 – Low-level requirements are developed	6
D^{M}	**SwDP-DP-MB 5 –** Documentation and assessment of derived low-level requirements	Design Model for a self-contained set of requirements	Identified and documented derived low-level requirements with assessed safety impact (of software module)	5 – Derived low-level requirements are defined and provided to the system processes, including the system safety assessment process	5.4.3 6.5.5
D^{M}	**SwDP-DP-MB 6 –** Assembly of Design Description	Design Model for a self-contained set of requirements Trace Data Documented derived LLRs	Design Description (of software module)	4 – Low-level requirements are developed	8.1.1
+ Review and analysis (cf. section 4.6.1)					

Table 5: Tasks for Design Process

Purpose of SwDP-DP-MB 1 and SwDP-DP-MB 2 is to develop the software architecture in SL according to the Software Design and Model Standard by defining:

- modules and necessary library modules
- units and module data
- interfaces between units
- dependencies of modules

and allocating requirements of the software component to the modules.

The actual allocation of requirements is done in Polarion. The developers of the sub-level process can then refine (R^M) or directly implement the requirements (D^M).

In SwDP-DP-MB 3, the detailed design (deeper software architecture and LLRs) is implemented following the Software Design and Model Standard. The Software Model Standard also identifies model elements not contributing to LLRs and software architecture, as requested by the objectives MB.A-2: 9 and 10. SwDP-DP-MB 3 does also cover the transformation of system-level models to shared Design Models for workflow 5.

The implementation of the Design Model is not a one-time task, but strongly coupled with the change management process (as part of the configuration management process). Two scenarios exist: Either a new set of HLRs is given and no implementation exists yet. In this case, all the requirements can be implemented and all downstream development and verification activities have to be performed. Or a baseline of the software already exists and the change management process is active. Every modification of the module content now requires a change request (CR) and the processes described in the Software Configuration Management Plan must be followed. Reverification is required for impacted artifacts and activities of the change.

Along with the implementation, bidirectional traces to higher-level requirements are established (SwDP-DP-MB 4). This potentially identifies LLRs, which cannot be traced to HLRs. Some are documented, verified, and justified as derived requirements in SwDP-DP-MB 5.

The final step is the assembly of a Design Description in SwDP-DP-MB 6. The Design Description is the documentation of the Design Model and Trace Data.

4.5.3 Coding Process (C)

The requirements and objectives for the Coding Process are provided by MB.5.3 and Table MB.A-2:6. Intention is to transform the Design Model into Source Code and provide Trace Data. The derived tasks are listed in Table 6.

Entry criteria of C^M is the exit criteria of D^M or the exit criteria of $D2^C$ for the component module (cf. Figure 19).

Exit criteria of C^M is that Source Code and Trace Data for the Design Model of the module has been created. Related verification activities have been performed.

Entry criteria of C^C is the exit criteria of C^M.

Exit criteria of C^C is that the Source Code of the software application has been verified as whole.

Proc	Task ID - Name	Inputs	Outputs	Mapped DO-331 Objectives of Table MB.A-2	Section
C^M	**SwDP-CP-MB 1 –** Modular source code generation	Design Model (of software module)	Source Code (of software module)	6 – Source Code is developed	8.1.2 5.5
C^M	**SwDP-CP-MB 2 –** Trace data generation This task part of SwVP-CP-MB 1.	Design Model (of software module) Source Code (of software module)	Trace Data (of software module)	6 – Source Code is developed	8.2.10
+ Review and analysis (cf. section 4.6.2)					

Table 6: Tasks for Coding Process

Table 6 only contains development activities for the module process, thus C^C is a pure verification process.

In SwDP-CP-MB 1, the Design Model of the software module is translated to modular Source Code compliant with the Software Code Standard. Embedded Coder is used for automatic code generation. The whole tailored code generation workflow and code generator settings are discussed in section 8.1.2.

Trace Data generation SwDP-CP-MB 2 from Source Code to SL/SF and vice versa is a second step with SLCI. Embedded Coder generates Trace Data during code generation, too. This data is considered as navigation help only, since it cannot be automatically verified.

4.6 Verification Processes

Contribution 3: The abstract DO-178C verification objectives have been concretized in specific tasks adapted for the modular development life cycle. The unique separation of tasks into module- and integration-level tasks leverages full modular development. Integration-level tasks focus on verification of the interfaces and overall completeness. Module-level tasks allow early verification and detection of design flaws. This, and the low complexity of the design on this level, leverage significant time and cost savings.

The Verification Process is integral and goes along with the respective Development Process. As already introduced in section 2.3, DO-178C distinguishes between review and analysis, and testing.

All review and analysis tasks are executed after the respective part of the Development Process. The provided breakdown is unique, since the tasks are particularly defined for the modular MBSwD. They are allocated to the respective modular processes. Everything that can be done on module-level, is done on module-level. Sections 4.6.1 and 4.6.2 address review and analysis.

For testing, a tailored simulation and testing strategy has been developed. It enhances traditional testing, which is normally performed after integration, with new concepts leveraged by MBSwD. Traditionally, software testing is performed after full integration. With the new techniques, parts of the process can be executed earlier and faster, which leads to shorter design cycles. The strategy is outlined in section 4.6.3. The tasks performed as part of the MBSwD process are elaborated in detail. Testing of the Executable Object Code is not explicitly addressed in the scope of this thesis.

4.6.1 Review and analysis of Design Process

The objectives for review and analysis of outputs of the Design Process (D^M and $D2^C$) are listed in DO-331 Table MB.A-4 and concretized in the tasks provided in Table 7.

Proc	Task ID - Name	Inputs	Outputs (Software Verification Results)	Mapped DO-331 Objectives of Table MB.A	Section
D^M	**SwVP-DP-MB 1 –** Static model analysis	Design Model (of software module)	Report and Justification	4-4: Low-level requirements are verifiable 4-5: Low-level requirements conform to standards 4-11: Software architecture are verifiable 4-12: Software architecture conforms to standards	8.2.1
D^M $D2^C$	**SwVP-DP-MB 2 –** Static module analysis	Design Model (of software module and dependent modules)	Report and Justification	4-5: Low-level requirements conform to standards 4-12: Software architecture conforms to standards	8.2.2
D^M $D2^C$	**SwVP-DP-MB 3 –** Model review	Design Model (of software module)	Checklist	4-3: Low-level requirements are compatible with target computer 4-5: Low-level requirements conform to standards 4-10: Software architecture is compatible with target computer 4-12: Software architecture conforms to standards	8.2.3
D^M $D2^C$	**SwVP-DP-MB 4 –** Traceability review and analysis	Design Model (of software module) Trace Data HLRs	Checklist	4-1: Low-level requirements comply with high-level requirements 4-6: Low-level requirements are traceable to high-level requirements	8.2.4
D^M	**SwVP-DP-MB 5 –** Design error detection	Design Model (of software module)	Report and Justification	4-2: Low-level requirements are accurate and consistent 4-4: Low-level requirements are verifiable 4-7: Algorithms are accurate 4-9: Software architecture is consistent 4-11: Software architecture is verifiable	8.2.5
Notes: Objective 4-13 addresses partitioning integrity and is not referenced. Partitioning, which is a way to isolate software components, has not been considered, since it is not an applied strategy in the MBSwD process (cf. assumption AS 8).					

Table 7: Tasks for review and analysis of Design Process

SwVP-DP-MB 1 uses *Simulink Model Advisor (SL Model Advisor)* [7] to verify the conformance with the rules defined in the Software Model Standard by static analysis. SL Model Advisor is an infrastructure, which is part of SL, to run static checks on a single SL model. SwVP-DP-MB 1 mainly focuses on the detailed design. In contrast, SwVP-DP-MB 2 performs module-wide and inter-module analysis. For example, it checks model data used across SL models or verifies if all requirements have been implemented.

Design and modeling rules, which cannot be verified are subject to manual model review (SwVP-DP-MB 3). Apart from that, SwVP-DP-MB 4 separately addresses the review of manually established traceability between the Design Model and LLRs. This also covers derived LLRs.

SwVP-DP-MB 5 proves the absence of certain design errors in the Design Model and interface compliance. The task leverages formal methods, for which DO-333 applies, and relies on *Simulink Design Verifier* [8]. The tool is a product of MathWorks for design error detection, dead logic detection, property proving, and test cases generation based on SL and SF models.

[7] Product of The MathWorks Inc., https://de.mathworks.com/help/simulink/examples/model-advisor.html [Accessed on: Jan. 04 2020] Release 2017b

[8] Product of The MathWorks Inc., https://de.mathworks.com/products/sldesignverifier.html [Accessed on: Jan. 04 2020], Release 2017b

4.6.2 Review and analysis of Coding Process

This section references the coding part of DO-331 Table MB.A-5 (C^M and C^C). The respective review and analysis tasks are listed in Table 8.

Proc	Task ID - Name	Inputs	Outputs (Software Verification Results)	Mapped DO-331 Objectives of Table MB.A	Section
C^M	**SwVP-CP-MB 1 –** Automatic code review	Design Model (of software module) Source Code (of software module)	Report	5-1: Source Code complies with low-level requirements 5-2: Source Code complies with software architecture 5-5: Source Code is traceable to low-level requirements 5-6: Source Code is accurate and consistent	8.2.10
C^M C^C	**SwVP-CP-MB 2 –** Static code analysis for standard compliance	Source Code (of software module)	Report and Justification	5-4: Source Code conforms to standards.	8.2.11
C^M C^C	**SwVP-CP-MB 3 –** Static code analysis for error detection	Source Code (of software module)	Report and Justification	5-4: Source Code conforms to standards. 5-6: Source Code is accurate and consistent.	8.2.12
C^M	**SwVP-CP-MB 4 –** Code review	Source Code (of software module)	Checklist	5-3: Source Code is verifiable. 5-4: Source Code conforms to standards. 5-6: Source Code is accurate and consistent	8.2.13
C^M	**SwVP-CP-MB 5 –** Code proving	Source Code (of software module)	Report and Justification	5-3: Source Code is verifiable. 5-6: Source Code is accurate and consistent	8.2.14
Notes: The process for Parameter Data Items is described in 5.4.4.6. Parameter Data Item Files are implemented and verified in the main DO-178C process, thus the objectives are not further considered (objectives MB.A-5:8,9). Objective MB.A-5:6, “Source Code is accurate and consistent”, requires additional tasks in the DO-178C main process, like worst-case execution timing analysis or memory analysis (cf. DO-178C 6.3.4f). The integration part of DO-331 Table MB.A-5, i.e., the compilation, linking, and loading of the software, which is expressed by objective MB.A-7, is not part of the MBSwD process.					

Table 8: Tasks for review and analysis of Code Process

Most important part is the backward verification of the code generation process with *Simulink Code Inspector (SLCI)*[9] in task SwVP-CP-MB 1, since Embedded Coder is not shipped with documentation for tool qualification under DO-330. Qualification of development tools (criteria 1 tool, cf. section 2.6), like code generators or compilers, is in most cases neither economically nor technically realizable [88].

SLCI verifies the code generation process of Embedded Coder afterwards [89]. It releases developers from extensive manual reviews of compliance and traceability between code and model, but also supports verification of accurateness and consistency of algorithms on the code level depending on the workflow [90]. In 2015, MathWorks announced the successful completion of a Stage of Involvement 4 (SOI#4) audit with Transport Canada (government department of Canada) for DO-330 TQL 4 [91]. SOI#4 is the final certification software review by the authority (cf. EASA CM SWAEH - 002 [43]).

SwVP-CP-MB 2 and SwVP-CP-MB 3 both use *Polyspace Bug Finder*[10] for code analysis. SwVP-CP-MB 2 searches violations against the Software Code Standard, whereas SwVP-CP-MB 3 focuses on the detection of implementation and runtime errors, like a division by zero. Code review (SwVP-CP-MB 4) is a manual review task for rules in the Software Code Standard, which are not covered by the preceding analyses.

SwVP-CP-MB 5 is formal analysis performed with *Polyspace Code Prover*[11] to prove the absence of certain runtime errors and interface compliance. Code Prover has less objectives than Bug Finder, but applies formal methods to prove the absence of errors. This task is subject to DO-333.

4.6.3 Model simulation and testing

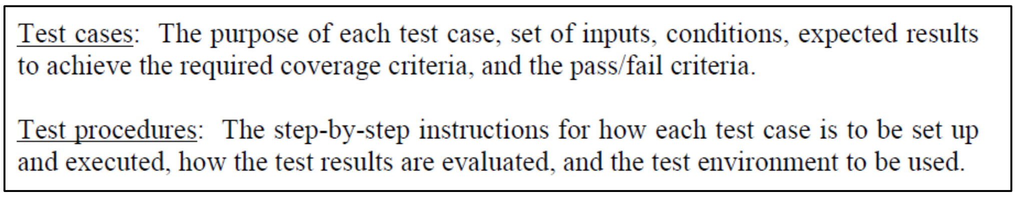
Contribution 4: An economic modular simulation and testing strategy has been specified. The testing strategy spans simulation, software-in-the-loop testing, reuse of test cases, model coverage, structural coverage, and data and control coupling. It improves reusability of test cases and decreases rework by front-loading and modularizing verification activities. Rework cycles are also significantly shortened, which results in cost reduction and faster development.

This section derives the complete simulation and testing strategy step by step. Testing is already known by DO-178C whereas model simulation has been introduced as new verification technique in DO-331 (cf. section 2.4). Testing executes *test cases* and *test procedures*, specified by DO-178C as in Figure 23, on the Executable Object Code.

[9] Product of The MathWorks Inc., https://de.mathworks.com/products/simulink-code-inspector.html [Accessed on: Jan. 04 2020], Release 2017b

[10] Product of The MathWorks Inc., https://de.mathworks.com/products/polyspace-bug-finder.html [Accessed on: Jan. 04 2020], R2017b

[11] Product of The MathWorks Inc., https://de.mathworks.com/products/polyspace-code-prover.html [Accessed on: Jan. 04 2020], R2017b



Test cases: The purpose of each test case, set of inputs, conditions, expected results to achieve the required coverage criteria, and the pass/fail criteria.

Test procedures: The step-by-step instructions for how each test case is to be set up and executed, how the test results are evaluated, and the test environment to be used.

Figure 23: Test cases and test procedures (DO-178C 11.13)

Test procedures specify, how to execute test cases and how to retrieve, archive and evaluate the results in a manner, that a person, who did not write the test cases, can reproduce the results [45, p. 208].

4.6.3.1 ***Model simulation***

Model simulation is the accepted technique by DO-331 to execute *simulation cases* with *simulation procedures* on Specification or Design Models in order to fulfill objectives of DO-178C/DO-331 (DO-331 MB.6.8). Concerning their definition, simulation and test cases/procedures are closely related (cf. DO-331 MB.11.13).

Figure 8 outlined objectives, which can be partially or fully satisfied by model simulation. In this MBSwD process, model simulation shall be used to show that the Design Model:

- complies to high-level requirements (DO-331 MB.A-4:1,8).
- is accurate, consistent, and verifiable (DO-331 MB.A-4:2,4,7,9,11).

In consequence, *simulation cases and procedures for model simulation* are derived from HLRs and executed against on the Design Model.

Creating simulation cases and procedures causes additional effort and makes only sense, if simulation cases are reused to test the Execrutable Object Code later on in the process, i.e., if simulation cases become test cases. Otherwise, duplicate work would be necessary to define separate simulation and test cases. The perspective of reuse is a key advantage of model-based design, since verification in an early design stage becomes possible without additional effort. Anyway, reusability imposes additional requirements on the simulation cases themselves, because all test case requirements of DO-178C 6.4 apply as well (cf. DO-331 MB.6.8). The requirements are addressed with the respective testing standards.

The chosen tool to author and run simulation and test cases is *Simulink Test (SL Test)*[12]. SL Test is an infrastructure to implement simulation cases and execute them on the Design Model or Executable Object Code. Appendix G provides some guidance on how to author test cases in Simulink Test.

Simulation and test procedures mostly differ due to a different testing environment and tools. The approach here is to keep them in separate high-level documents (cf. Appendix G).

[12] Product of The MathWorks Inc., https://de.mathworks.com/products/simulink-test.html, s[Accessed on: Jan. 04 2020], Release 2017b

4.6.3.2 *Review and analysis of model simulation / model coverage*

According to DO-331 MB.6.8.3.2, simulation cases, procedures, and results have to be reviewed and/or analyzed. For the chosen use case of model simulation, objectives MB14-16 of Table MB.A-4 apply:

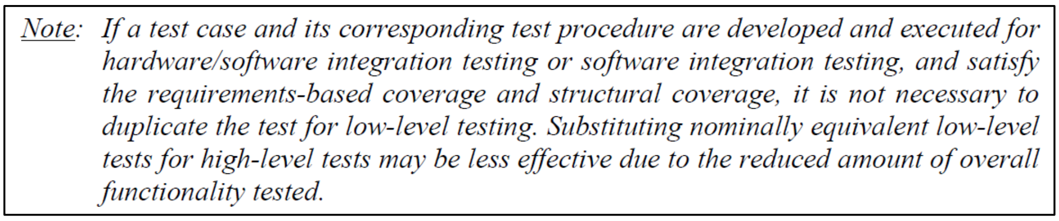
- Table MB.A-4:MB14 – Simulation cases are correct
    - + Table MB.A-7:3 – Test coverage of HLRs is achieved (cf. MB.6.8.3.2)
    - + Table MB.A-7:4 – Test coverage of LLRs is achieved (cf. MB.6.8.3.2)
- Table MB.A-4:MB15 – Simulation procedures are correct
- Table MB.A-4:MB16 – Simulation results are correct and discrepancies explained

Objectives MB.A-7:3 and 4 are named in MB.6.8.3.2 and refer to requirements-based coverage analysis (verifying that all requirements have test cases, in contrast to structural coverage) as defined in DO-178C 6.4.4.1.

MB.A-7:MB14-16 are mainly verified by review. Also requirements-based coverage of HLRs is verified with review tasks according to DO-178C 6.4.4.1. For requirements-based coverage of LLRs, Table MB.A-7 alternatively lists *model coverage* as means of compliance. According to DO-331 MB.6.7, “model coverage analysis determines which requirements expressed by the model were not exercised by verification based on the requirements from which the model was developed.” Model coverage data is collected during execution of simulation cases, i.e., it is a measure to evaluate, how extensive simulation cases derived from HLRs exercise the Design Model and thus the LLRs in this case.

However, the impact of model coverage goes further. Under the assumption that all simulation cases are reused as test cases for the Executable Object Code, model coverage implicitly shows the compliance to DO-331 Table MB.A-7:4 *for test cases* as well.

And further, the note in DO-178C 6.4 (Figure 24) holds out that low-level testing can be omitted, if requirements-based and structural coverage can already be satisfied without low-level tests.



Note: *If a test case and its corresponding test procedure are developed and executed for hardware/software integration testing or software integration testing, and satisfy the requirements-based coverage and structural coverage, it is not necessary to duplicate the test for low-level testing. Substituting nominally equivalent low-level tests for high-level tests may be less effective due to the reduced amount of overall functionality tested.*

Figure 24: Duplication of low-level testing (DO-178C 6.4)

If full model coverage is obtained and full structural code coverage is achieved with the simulation cases created from HLRs (and derived requirements), no additional low-level tests are required. Preservation of coverage from model to code (structural) is not guaranteed by Embedded Coder code generation, but a high degree of preservation can be achieved by modeling rules (cf. section 8.2.9). The chosen tool to record model coverage is Simulink Coverage[13].

4.6.3.3 ***Testing***

For model simulation, an executable representation of the Design Model could be used. Testing is performed on the product Executable Object Code. Test procedures and test cases are typically derived from HLRs and (if needed) from LLRs, but mainly simulation cases are reused as test cases.

4.6.3.4 ***Test coverage***

As already mentioned, testing is subject to test coverage analysis. On the one hand, this is requirements-based test coverage and on the other hand structural coverage. Since simulation cases are reused, requirements-based test coverage is already satisfied (if all simulation cases are reused as test cases, sufficient model coverage is achieved and respective reviews have been successfully performed).

Structural coverage is a measure for how good the code has been tested. Structural coverage is required by the following objectives (for DAL B):

- Table MB.A-7:6 – Test coverage of software structure (decision coverage) is achieved
- Table MB.A-7:7 – Test coverage of software structure (statement coverage) is achieved
- Table MB.A-7:8 – Test coverage of software structure (data coupling and control coupling) is achieved

The following discussion mainly refers to the objectives MB.A-7:6 and MB.A-7:7. Coupling coverage is discussed in section 4.6.4.

Structural coverage is recorded during testing and thus on Executable Object Code. However DO-178C leaves it open, whether coverage is mapped to Source Code, object code, or Executable Object Code, but the developer must take additional precautions for object code or Executable Object Code coverage (cf. DO-248C FAQ #42 or [92]). In consequence, coverage analysis is performed on Source Code in the presented MBSwD process.

[13] Product of The MathWorks Inc., https://de.mathworks.com/products/simulink-coverage.html [Accessed on: Jan. 04 2020], Release 2017b

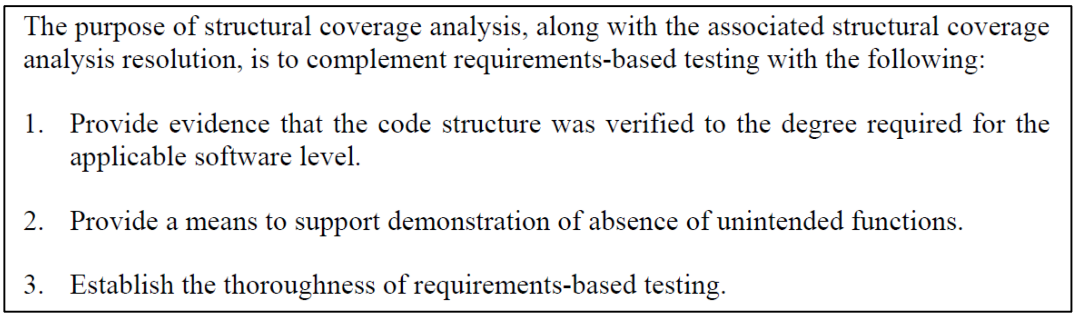
Typically, coverage is recorded on the target hardware. Bordin et al. [93] summarize that two categories of workflows for structural coverage recording in safety-critical processes exist. Some tools instrument the Source Code and measure coverage on the instrumented code. In this case, it must be shown that the instrumentation did not change the application behavior and the coverage is representative for the final code. This is typically achieved by additionally running the tests on the non-instrumented code and comparing the outputs of both runs for equivalence (*equivalence testing*). Other approaches use hardware probes and are non-intrusive [93]. The target environment (PPC) does not provide this capability (cf. assumption AS 2), so instrumented code with equivalence testing is necessary.

To record Source Code coverage, Simulink Coverage has been chosen. Simulink Coverage supports statement and decision coverage in the sense of DO-178C, although named differently. With SL Coverage, an uncommon workflow is applied promising faster iterations. In a first step, *software-in-the-loop* (SIL) testing is performed. Source Code is instrumented and compiled as executable for the *host computer* (cf. section 4.6.3.5). Test case execution and coverage collection is done on the host. In a second step, the non-instrumented Source Code is cross-compiled and loaded onto the target. The same test cases are executed on the target, again. This is called *Processor-in-the-loop* (PIL) testing (cf. section 4.6.3.5). Finally, the last step compares SIL and PIL outputs to confirm functional equivalence (equivalence testing).

The question, how valid this approach is and whether it conforms with DO-178C, is discussed in the following. The main difference compared to traditional approaches is that two different compilers are used, one for host compilation and another one for cross-compilation, and that the test environments differs. Structural coverage in SIL is collected on the instrumented Executable Object Code executed on the host computer, but not on the final target.

Concerning test environment, DO-178C 6.4.1 relaxes the requirements. It would even be possible to just run selected tests only on a “host computer simulator”, which is not even done here, since all tests are finally executed on the target environment. Just the structural coverage is collected on the host computer.

Structural coverage is not a measure to assess the object code. Structural coverage on object code is even not desired and requires further verification activities according to DO-248 FAQ#42. Only under DAL A, a traceability analysis is required to reveal object code that cannot be traced to Source Code (objective DO-178C A-7:9). DO-248C FAQ#43 states the purpose of structural coverage analysis as in Figure 25. (1) to (3) are measures for test cases and code structure, not for object code structure.


The purpose of structural coverage analysis, along with the associated structural coverage analysis resolution, is to complement requirements-based testing with the following:

1. Provide evidence that the code structure was verified to the degree required for the applicable software level.
2. Provide a means to support demonstration of absence of unintended functions.
3. Establish the thoroughness of requirements-based testing.

Figure 25: Purpose of structural coverage (according to DO-248C FAQ#43)

Thus it can be argued, that recording SIL coverage on the host computer fulfills the purpose of structural coverage. Anyway, it's worth to have a closer look at differences in the compilation process. The target compiler might:

- add a function, which the host compiler does not. However, this may not even be detected in a traditional approach, if coverage is mapped to Source Code and no object code traceability analysis is performed.
- remove or modify a function, which the host compiler does not. This would be detected in the traditional coverage approach, but not with SIL coverage. However, since the function has SIL coverage and equivalence testing is done, it is ensured that the code still behaves the same. For software below DAL A, there is neither a requirement for traceability nor structural equivalence of Source Code and object code, as long as it passes tests. Also note that in traditional approaches, instrumented code is also different Executable Object Code, for which the absence of such deviations cannot be proved beyond equivalence testing either.

Using SIL coverage the way described here, is a new approach. It should be discussed with authorities first. Even if SIL coverage cannot be used for certification evidence, since authorities insist on coverage measurements on the cross-compiled code in the target environment, it provides a valuable and fast way to assess the quality of code and tests in an early, prospective task. Coverage assessment can easily be coupled with PIL testing, but may require other coverage recording software. In this thesis, SIL coverage is assumed as valid certification evidence.

Finally, it must be noted that tool qualification is performed with the SIL host compiler [94]. The compiler can thus be considered as part of the structural coverage analysis tool chain.

4.6.3.5 *Test frameworks*

In a perfect world, all test cases are executed in the target environment with a fully integrated software image, i.e., a software image, which has all software components integrated and exactly matches the final software product. This approach is also followed by Verocel [95], a leading company providing tools and services for system and software development for safety-critical platforms.

Disadvantage of this approach is that intermediate variables must be exposed and identifiable throughout the full software image for controllability and observability to allow testing of parts of the software (specially to obtain full structural coverage). Making intermediate variables writable and identifiable in auto-generated code is challenging and must be considered in the Design Model.

Also DO-178C recognizes that “more precise control and monitoring of the test inputs and code execution than generally possible in a fully integrated environment” may be needed and thus “testing may need to be performed on a small software component that is functionally isolated from other software components (DO-178C 6.4.1)”.

In the defined testing strategy of the process at hand, three frameworks for testing are used:

- SIL testing
- PIL testing
- Full-image testing

For SIL, the SIL simulation framework included in Embedded Coder is used [96, pp. 64-47ff.]. During the build, it instruments a copy of the Source Code for coverage collection, couples it with interface code to enable communication with SL, and compiles it with the selected host compiler.

PIL testing shall allow testing without a full-image of the software on the target. PIL testing bases on a framework developed at TUM-FSD. Main advantage compared to other solutions is its independence from interface code. Key of the PIL framework is the communication with SL through a standardized JTAG interface[14]. The setup is schematized in Figure 26. Every execution step, the SL model transmits data to the target by calling the TRACE32 Remote API, which writes data via the JTAG debug interface. After writing, the target performs the calculation step. In the meanwhile, SL performs a busy wait, until the breaking point at the start of the step is reached again. Hornauer implemented this approach in cooperation with Lauterbach [52, 97]. Saulo O. D. Luiz, Markus Hochstrasser, and Kajetan Nürnberger introduced further improvements.

[14] JTAG (Joint Test Action Group) interface as synonym for IEEE-Standard 1149.1.

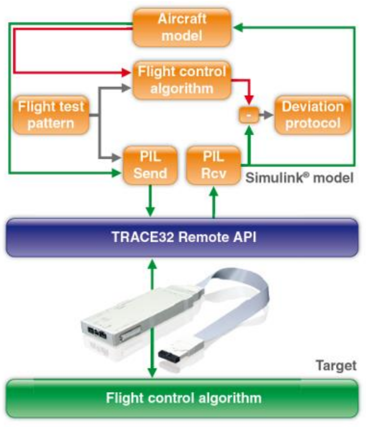


Figure 26: PIL setup for software integration and low-level testing using Trace32 (from [97])

Full-image testing uses the fully integrated software in the target environment for testing, i.e., not just the software application developed as part of the MBSwD process, but also the application framework. Not all PIL tests can be executed in the full image, but full-image testing is still the best approach and testing of subsets should just be a compromise. Here, *full-image* capable test cases are labeled accordingly and executed during hardware/software integration again. Finally, it must be verified that a sufficient number of full-image test cases exists.

4.6.3.6 ***Tasks***

For simulation procedure and case development, the objectives of DO-331 concerning both simulation and testing apply. The derived tasks are shown in Table 9.

In the first task (SwVP-DP-MB 6), simulation procedures and cases are derived from HLRs according to a test standard document (cf. Appendix G). They can be used for testing later on. Important part is establishing bidirectional traceability between HLRs, simulation procedures, and simulation cases. Under some circumstances, simulation cases may be derived from LLRs. However, this shall be avoided, since the more detailed the requirement is, the closer the testing is to “white-box” testing.

In SwVP-DP-MB 7, simulation cases and procedures are reviewed against the applied test standard. Review of simulation cases includes requirements-based test coverage assessment concerning HLRs according to DO-331 MB.6.4.4.a. Simulation and test procedures typically deviate from each other.

Proc	Task ID – Name	Inputs	Outputs (Software Verification Results)	Mapped DO-331 Objectives of Table MB.A	Section
D^M	**SwVP-DP-MB 6 –** Simulation / test procedure and case development	Design (of software module and dependent modules)	Simulation Procedures and Cases Trace Data	N/A	8.2.6
D^M	**SwVP-DP-MB 7 –** Simulation / test case and procedure review	Simulation Procedures and Cases Trace Data	Checklist	4-14: Simulation Cases are correct. 4-15: Simulation Procedures are correct. 7-3: Test coverage of high-level requirements is achieved.*	8.2.7
* Under the assumption that all simulation cases are used as test cases.					

Table 9: Tasks for simulation / test procedure and case development

The developed and verified simulation/test cases are executed for simulation and SIL testing separately. The results are also verified separately. Thus the derived tasks have been split into two tables, Table 10 and Table 11.

In Table 10, SwVP-DP-MB 8 describes the execution of the simulation cases and the review of the simulation results. SwVP-DP-MB 9 assesses the correctness of the simulation cases using model coverage, which is a by-product of the simulation execution. In D2^C, the task accounts for coupling coverage.

Proc	Task ID - Name	Inputs	Outputs (Software Verification Results)	Mapped DO-331 Objectives of Table MB.A	Section
D^M	**SwVP-DP-MB 8 –** Simulation testing & result review	Simulation Procedures and Cases	Simulation Result Report	4-16: Simulation Results are correct and discrepancies are explained. 4-1: Low-level requirements comply with high-level requirements 4-2: Low-level requirements are accurate and consistent 4-4: Low-level requirements are verifiable. 4-7: Algorithms are accurate 4-8: Software architecture is compatible with high-level requirements 4-9: Software architecture is consistent 4-11: Software architecture is verifiable.	8.2.8
D2^C, D^M	**SwVP-DP-MB 9 –** Model coverage assessment	Simulation Result	Report	4-14: Simulation Cases are correct. 7-4: Test coverage of low-level requirements is achieved.*	8.2.9
* Under the assumption that the simulation procedures are equivalent to corresponding test procedures to a significant extent and discrepancies are separately reviewed.					

Table 10: Tasks for simulation testing

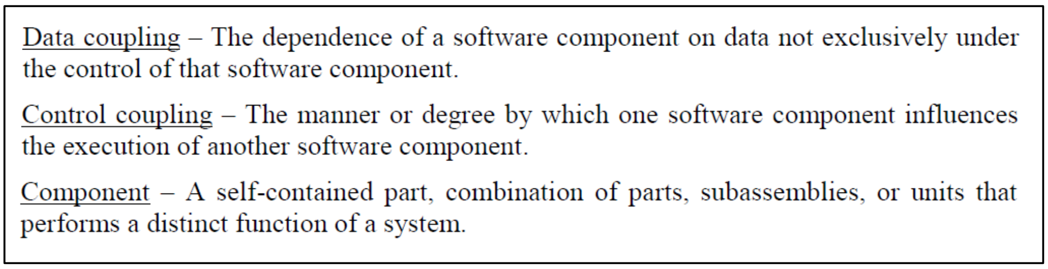
The simulation cases are executed as test cases in SIL. This is mainly a different invocation of the simulation execution. Main purpose is structural coverage assessment in SwVP-CP-MB 7. The outputs are reviewed in SwVP-CP-MB 6, but they are not considered as the final test results (therefore, PIL is used later on). To highlight this, objective 4-16, which addresses simulation results, has been listed instead of 7-2, which addresses “test results”. SIL testing contributes to show that the Source Code is testable, thus objective 5-3 is satisfied to some degree as well.

Proc	Task ID - Name	Inputs	Outputs (Software Verification Results)	Mapped DO-331 Objectives of Table MB.A	Section
C^M	**SwVP-CP-MB 6** – SIL testing & result review	Simulation Procedures and Cases	SIL Result Report	4-16: Simulation Results are correct and discrepancies are explained. 5-3: Source Code is verifiable	8.2.15
C^C, C^M	**SwVP-CP-MB 7** – SIL structural coverage assessment	Simulation Result	Report	7-6: Test coverage of software structure (decision coverage) is achieved. 7-7: Test coverage of software structure (statement coverage) is achieved.	8.2.16
Objective 7-5 describes modified condition/decision structural coverage and is omitted here, since it is only required for DAL A software, not DAL B (cf. assumption AS 8).					

Table 11: Tasks for SIL testing

4.6.4 Data coupling and control coupling analysis

In DO-178C, data coupling and control coupling (DC/CC) is defined as shown in Figure 27.


Data coupling – The dependence of a software component on data not exclusively under the control of that software component.

Control coupling – The manner or degree by which one software component influences the execution of another software component.

Component – A self-contained part, combination of parts, subassemblies, or units that performs a distinct function of a system.

Figure 27: DO-178C glossary definition of data and Control Coupling (DO-178C p.110f)

In other words, data coupling can be simply described “as the transaction of data values from a point in the source code where the value is set to another point where the value is referenced or used” and control coupling as “call sequence of the software”. [98, p. 237]

A technical report commissioned by the FAA outlines four types of coupling dependencies, which must be documented in requirements and be verified [99, p. 5]: Sequencing, Timing, Control dependencies (call/return of functions), and data dependencies (consistency and completeness of data passed, read/write order,…).

The idea of DO-178C is to evaluate the coupling between software components. DO-248C FAQ #67 states that DC/CC analysis is a combination of

- Review and analysis of Software Architecture
- Review and analysis of Source Code
- Requirements-based testing with structural coverage analysis (verification of the integration)

Only the third point is an explicit DO-178C objective. Test coverage of the software structure requires assessing data and control coupling according to DO-178C Table A-7: 8 and DO-178C 6.4.4.

Structural coverage of DC/CC

CAST-19 provides some clarifications about the method [100]. In general, the intention of structural coverage analysis is "to provide a measure of the completeness of requirement-based testing [100, p. 2]". According to CAST-19, measures like statement or decision coverage can be taken in isolation on module-level in order to reduce the complexity. "The intent of the structural coverage analyses of data coupling and control coupling is to provide a measurement and assurance of the correctness of these modules/components' interactions and dependencies. [100, p. 2]"

CAST-19 further states that this activity "should be conducted on R-BT [requirements-based testing] of the integrated components (that is, on the final software program build) [100, p. 2]". Hence, structural coverage of DC/CC is shown as part of the main DO-178C process and not by the SIL coverage collection with SL Coverage as previously described.

However, due to the strong modularization effort, it is reasonable to provide at least some measures to confirm that DC/CC is sufficiently tested during the MBSwD process, although the results are not considered for DO-178C Table A-8, since not measured on the final software.

For control coupling coverage, function and function call coverage provided by SL Coverage [101, pp. 4-5f] is assessed for all test cases executed from the component interface (cf. SwVP-CP-MB 7). This, for example, aligns with the "procedure coverage" proposed by LDRA [102]. In addition, an equivalent metric has been chosen for simulation testing (cf. SwVP-DP-MB 9).

Data coupling on code- and model-level can currently not be assessed with SL Coverage. A separate analysis tool is required. Data coupling coverage may log read and write of function arguments, local, and global variables (cf. "Dynamic Data Flow Coverage" or "Set-Use Coverage" in [102]).

Review and analysis of DC/CC

Review and analysis of DC/CC is not a specific task, but covered by the combination other tasks throughout the process. Such an approach aligns with [98] and [45, 223ff]. Appendix A lists DC/CC review and analysis contributions during the MBSwD process. The given mapping is specific for the process at hand and goes beyond the content of existing, published processes.

The application is, by assumption AS 8, a typical controller, with a single, completely executed step function as interface. All code runs at a single rate. To ensure that possible initialization functions are called first is part of the software application framework. In addition, further review and analysis may be necessary during integration (e.g., review or analysis of linker tables).

4.6.5 Complete testing approach

The testing strategy is a bottom-up approach from the lowest (unit) to the highest architectural entity (full software image) and summarized in Table 12.

Testing Phase	Process	Execution Target	Test Selection	Exit Criteria	Objective	Task
❶ **Module testing (requirements-based)**	D^M	Simulation on host computer	All tests developed in module	All tests passed. Requirement and model coverage goal achieved for each unit	Compliance of SW Design to HLRs/derived LLRs, requirements-based coverage for LLRs achieved.	SwVP-DP-MB 6 SwVP-DP-MB 7 SwVP-DP-MB 8 SwVP-DP-MB 9
	C^M	SIL on host computer		All tests passed. Code coverage goal achieved for each unit	Compliance of Source code to SW Design, shown, structural coverage	SwVP-CP-MB 6 SwVP-CP-MB 7
❷ **Equivalence testing (Simulation - SIL)**	After testing in C^M	Host computer	n/a	Recorded simulation and SIL module testing results are equivalent	Not used as certification evidence (prospective only).	SwVP-DP-MB 8
❸ **Component software/software integration testing**	D^C	Simulation on host computer	Tests testing the component interface (may overlap with to module testing)	Model coupling coverage goal achieved	Not used as certification evidence (prospective only).	SwVP-DP-MB 9
	C^C	SIL on host computer		Code coupling coverage goal achieved	Not used as certification evidence (prospective only).	SwVP-CP-MB 7
❹ **Component hardware/software integration testing**	I (not part of MBSwD)	PIL on target	All test cases	All tests passed	Compliance of Executable Object Code to HLRs/LLRs	Out of scope
	I (not part of MBSwD)	Hardware/software testing on full-image loaded in target	Full-image test cases	All tests passed and code coupling goal achieved.	Compliance of full-image Executable Object Code to HLRs/LLRs, DC/CC coupling.	Out of scope
❺ **Equivalence testing (SIL - PIL testing)**	After testing in I (not part of MBSwD)	Compares SIL and PIL results for equivalence	n/a	Recorded SIL and PIL module testing results are equivalent	SIL code coverage valid for Executable Object Code	Out of scope

Table 12: Bottom-up testing strategy

In phase D^M, simulation cases are developed from HLRs and derived LLRs. In principle, simulation cases are divided into two groups: Those testing a module or unit interface, and those testing the component interface. Latter are expected to support full-image testing in any case. Simulation cases based on module/unit interfaces may or may not support full-image testing (Figure 28).

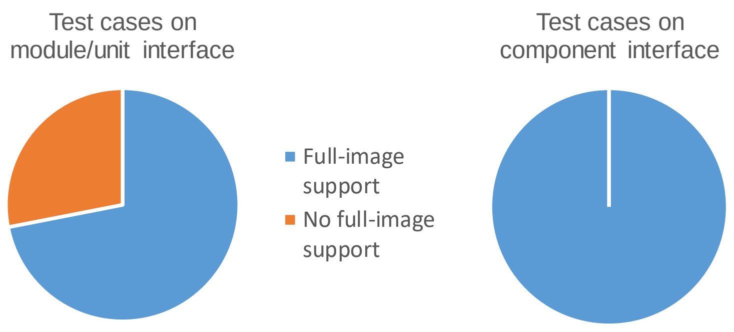


Figure 28: Types of test cases relevant for testing strategy (qualitative)

Module testing (1) is performed for every module. All test cases for a module are executed and the simulation output is recorded. By the end of module testing, all tests must have passed and model coverage must be complete for each unit.

After code generation (C^M), SIL testing (1) is executed with the same simulation cases and the output is recorded. At the end of SIL testing, all tests must have passed and code coverage must be resolved.

Equivalence testing between simulation and SIL (2) is optionally performed after that. It is does not contribute to certification evidence.

As part of the component module process, additional coverage criteria related to coupling are assessed both for the model and the code. The test cases have already been developed during module testing in the module process of the component module. They are just executed again with the new coverage criteria. At the end of D^C or C^C, respectively, coupling coverage must be satisfied. This phase is called *component software/software integration testing* (3). As mentioned earlier, this coupling coverage is not considered as part of certification evidence, since it is not collected in the target environment.

After that, the MBSwD workflow is left. The whole application code and all simulation cases are handed over to integration testing of the main DO-178C process. All simulation cases are executed as test cases on the Executable Object Code compiled for and loaded onto the target during PIL testing (5). For PIL testing mode, no full image of the software is used, which allows executing tests that do not support a full-image testing, too. PIL testing results are recorded and compared to SIL results in a subsequent step (6). This verifies that the SIL code coverage is valid for the Executable Object Code as well.

Those test cases supporting a full-image testing are additionally executed with the full image in the hardware/software integration testing stage afterwards.

For hardware/software integration testing, overlap with other processes may exist. For example, according to DO-254 6.2, “verification of hardware requirements during these processes [software/hardware integration and systems integration verification processes; note from the author] are a valid method of hardware verification”. Test cases basing on hardware and system requirements may be added.

5 Modeling framework for safety-critical MBSwD in SL

5.1 Objective

This section addresses the specified and implemented modelling framework for the modular process at hand. The modeling framework concretizes, how the developer shall implement the Design Model in the Design Process (cf. 4.5.2) by clear design, modeling, and coding rules. The rules are an essential part of the Software Design, Model, and Code Standard documents.

5.2 State-of-the-art

The implementation of models is not straightforward, since they are the crucial point of MBSwD. Models used for code generation (here Design Models) are the origin of various automatically generated artifacts. Their implementation has direct impact on simulation as well as compatibility with development, verification, and testing tools. The implementation of the Design Model also directly drives the quality of the generated C code. A violation of a coding rule can only be reasonably resolved by changing the Design Model.

DO-178C/DO-331 requires to describe the design and modeling tool/language with rules in a Software Design Standard (DO-178C 11.7) and a Software Model Standard (MB.11.23). The requirements for the generated code have to be documented in the Software Code Standard (DO-178C 11.8). All standards are intended to support developers in implementing Design Models or Source Code with a constant level of high quality. The expected content of the standards is roughly outlined by Table 13, Table 14, and Table 15 as described by DO-178C and DO-331.

Design Standard content according to DO-178C 11.7 (shortened)	
a	Design description method(s) to be used.
b	Naming Conventions to be used
c	Conditions imposed on permitted design methods
d	Constraints on the use of design tools
e	Constraints on design
f	Complexity restrictions

Table 13: Design Standard contents according to DO-178C 11.7

Model Standard Content according to DO-331 MB 11.23 (shortened)	
a	Method and tools used
b	Modeling languages used
c	Style guidelines and complexity restrictions
d	Constraints on the use of the modeling tools
e	Method to be used to identify and delimit requirements contained in model
f	Method to identify and delimit derived requirements and the method to provide derived requirements to the system processes
g	Identification of model elements that do not contribute to the representation of the software requirement or architecture
h	Rationale for the suitability of the technique for the type of information expressed by a Design Model

Table 14: Model Standard contents according to DO-331 MB.11.23

Code Standard content according to DO-178C 11.8 (shortened)	
a	Programming languages to be used and/or defined subset(s)
b	Source Code presentation standards
c	Naming conventions
d	Conditions and conventions imposed on permitted coding conventions
e	Constraints on the use of coding tools

Table 15: Code Standard contents according to DO-178C 11.8

Whereas the role of the standards is obvious in traditional software development, the separation of their content is blurred in MBSwD. For example, 11.7d requests constraints on design tools and MB.11.23d constraints on modeling tools, which are basically the same. Furthermore, a strong dependency between Model and Code Standard exists. The Code Standard imposes restrictions on the code implementation. In traditional DO-178C, Software Code Standards contain an extensive set of rules and are one of the most important documents for developers. In MBSwD with auto-generated code, the appearance of the generated code is fully controlled by the Design Model (and thus the Software Model Standard) and code generator settings.

Since SL/SF for MBSwD is broadly used, various companies have developed an own, rich set of modeling rules. For development of high-integrity software in connection with Embedded Coder, MathWorks provides so-called guidelines for High-Integrity System Modeling [103] and the guidelines for code generation [104]. Other code generators, like Target Link, also come along with separate sets of guidelines[15]. Also MISRA standardized modeling in the context of auto-code generation in general [105], and specifically for SL/SF [106, 107]. Different advisory boards from the automotive domain have been established world-wide over the last years and created guidelines (cf. MAAB [108] or JMAAB [109] basing on Orion GN&C MATLAB/Simulink Standards [110]) trying to satisfy simulation, code-generation, and verification requirements. Some companies also published their modeling guidelines, like the Ford Motor Company [111], or General Electric [112]. Other guidelines are a subset for a special purpose, e.g., to safeguard the translation process to SCADE [113, 114] or support the formalization of the SF language [115].

However, just selecting or combining sets is not the solution, since many sets

- base on the same origin, but have project or company specific extensions. They have contradictory or duplicate rules.
- focus on a specific use case. Some sets primarily cast an eye on simulation, others respect code generation with a specific code generator.
- are outdated and base on old SL/SF releases, except the guidelines shipped by MathWorks and JMAAB, which are under active development.

In addition, most of the sets are too general and specific at the same time. For example, the MathWorks and JMAAB guidelines address specific settings for safety, code generation, or MISRA compatibility reasons. However, they avoid imposing any restriction on whole features in order to preserve a broad applicability. The guidelines do not address limitations imposed by custom processes, like compatibility with defined development and verification tools or the target environment in the downstream workflow. Developers applying these guidelines get to a safe design, but often need a lot of rework until they have iterated to a compatible Design Model for the actual development process.

Furthermore, all known guideline sets just superficially touch topics like software design, conventions, or modularization. Even if different developers fully applied these sets, they would very likely implement totally different models, which would require significant rework during integration. Also proving the consistency and completeness of verification evidence would be difficult.

To sum up, independent, specific sets of modeling rules exist, but with significant gaps for a consistent design and coding concept embedded in an overall DO-178C process. In the work of this thesis, existing guidelines have been extended, and new guidelines have been added, restricting SL/SF modeling in symphony with the whole process.

[15] https://www.dspace.com/de/gmb/home/support/kb/supkbspecial/kbtl/tlmodguide/tlapp_modelguide.cfm [Accessed on: Jan. 6th 2020]

5.3 Structure

Ultimate goal of the whole standard documents and thus the modeling framework is to provide rules that help developers satisfying DO-178C/DO-331 objectives. These are

- objectives in Table MB.A-2 “Software Development Process” and referenced activities in MB.5.2.2
- objectives in Table MB.A-4 “Verification of Outputs of Software Design Process” and referenced review and analysis in MB.6.3.2 and MB.6.3.3.

The content is guided by the outline of the Software Design, the Code, and the Model Standard (cf. Table 13, Table 14, Table 15).

The modeling framework consists of the following components:

- Design rules
- Coding rules
- Module design rules
- Fundamental modeling rules
- Traceability rules

The rules are an essential part of the respective standard documents as depicted in Figure 29.

At first, design rules (DRs) are defined in section 5.4. They define concepts and requirements, which the software design has to fulfill from a functional and process point-of-view. These objectives are independent of the chosen modeling language, but rely on requirements of DO-178C/DO-331.

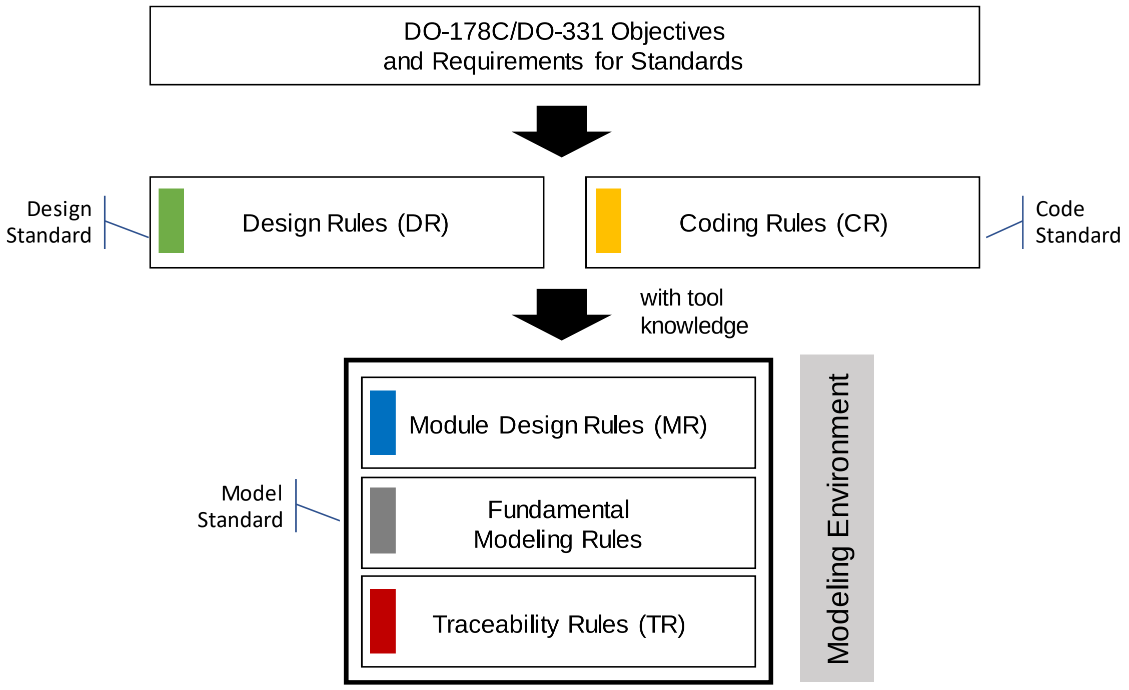


Figure 29: Modeling framework overview

Similarly, the requirements for the code are formulated as coding rules (CRs) in accordance with the design rules. They describe the language set and structure of the auto-generated code and introduce the impact of the target environment. They are discussed in section 5.5.

The defined DRs and CRs are the basis for three types of rules in the Software Model Standard:

- *Module design rules* (MRs) introduce basic constraints on SL/SF by limiting the feature set to a so-called *safe modeling subset*. Furthermore, they focus on transforming the design concepts in the DRs into SL/SF, like architectural design, design range contracting, encapsulation, deactivated, dead or unreachable functions.
- *Fundamental modeling rules* have been selected and slightly adapted from existing guidelines like the MathWorks high-integrity guidelines [103]. These rules are significantly more detailed than MRs and, in general, address specific model settings, but they do not handle all the facets of module design rules.
- *Traceability rules* (TRs) have been established to describe, which trace links have to be created and the applicable granularity as well as the handling of derived requirements in SL/SF.

A good example to illustrate the difference between module design rules and fundamental modeling is the use of signal ranges in SL models. Signal ranges in SL models define the expected or admissible value range of the signal. Specifying value ranges only makes sense if planned throughout the process and their meaning is clearly defined. Otherwise their use is extremely risky due to misinterpretation. The overall concept is a concerted interplay between run-time diagnostic settings, various verification methods on model and code level, and maintainability considerations. The respective high-integrity modeling guidelines (fundamental rules), however, just specify that value ranges shall be entered in the model at root-level and provide examples of tools, which can use them [103, pp. 2-34ff]. Any further overall concept is missing. The respective MRs are more detailed in this case by exactly specifying, for which architectural component and which model element values can be set in accordance with the given verification process.

The thesis presents the assembled module design rules in section 5.6. The fundamental modeling rules are just briefly discussed in section 5.7 for completeness. The whole topic of traceability is covered separately in section 6.

All rules are defined in gray boxes with a unique identifier, a short title and the actual content below. Different types of rules have a different color code, which aligns with the colors of Figure 29. A design rule (DR), for example, has the following format:

DR X – Rule Title

Rule Content

Each rule is followed by a detailed description and discussion afterwards substantiating the decisions of the author with arguments and references.

Each rule must be verified in at least one task of the process. DRs can be directly verified by a task, or implicitly via a MR, TR, or fundamental modeling rule. CRs must always be verified directly in a respective task. Verification cannot be argued via MR, since a non-qualifiable code generator is in-between model and code.

In section 5.8, the presentation of the modeling environment summarizes the artifacts required to realize an implementation complying to the rules.

5.4 Design rules

Contribution 5: A new set of design rules for modular development has been created. The rules differ from existing modeling rules by describing overall concepts of the software design in a development tool-independent manner, but with MBSwD in mind. The new design rules help developers to understand overall principles and provide reasoning for modeling rules. They cover topics, which are essential in modular workflows but fairly unaddressed in existing rule sets, i.e., they formulate requirements for high-level architectures, data encapsulation, and interface contracting and they specify the way to handle DO-178C Parameter Data Items or deactivated and noncovered design.

For the design rules, traditional software design concepts have been adopted for model-based design. The rules have been grouped according the objectives of DO-331 Table MB.A-4.

5.4.1 Summary of rules

5.4.2 Conformance

DR 1 - Conformance to Standards

A. The SW Design shall conform to the Software Design Standard. Deviations shall be handled according to the respective verification task.

B. The SW Design shall conform to the Software Model Standard for the selected modeling tool. Deviations shall be handled according to the respective verification task.

C. The SW Design shall support generation of Source Code, which conforms to the Software Code Standard.

Conformance to standards results from objectives DO-178C Table-4:5,12. (A) is the trivial requirement. The respective DO-331 objective requires (B). Due to automatic code generation, the Design Model highly influences the Source Code, which leads to (C).

DR 2 - Configuration management

The SW Design shall support the Configuration Management Process and considerations.

The SW Design artifacts must be stored and formatted in a way so that they can be handled according to the principles and processes of configuration management. This starts with unique names, checksums, goes over change diffing or merging, up to annotations in a review process.

5.4.3 Compliance

A core idea of DO-178C is that the SW Design bases on allocated requirements. This is referred to as “compliance” and further detailed in DO-178C Table A-5:1 and 8. The following rules have been derived from the given objectives.

DR 3 - Compliance with High-Level Requirements

The LLRs shall satisfy allocated higher-level requirements of the module and the System Architecture shall not conflict with allocated higher-level requirements.

The SW Design shall implement the behavior described by the HLRs. The type of higher-level requirements varies with the chosen workflow type (cf. section 4.5.1).

During development, new requirements may arise, the so-called “derived requirements” as specified by DO-178C according to Figure 30. To avoid subversion of DR 3, they must be handled with care.

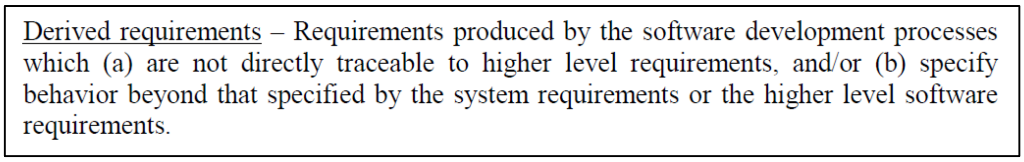

Derived requirements – Requirements produced by the software development processes which (a) are not directly traceable to higher level requirements, and/or (b) specify behavior beyond that specified by the system requirements or the higher level software requirements.

Figure 30: DO-178C Glossary - Derived requirements

The following DR addresses derived requirements in the SW Design, only. This means requirements, which appear during the Design process and do not trace up to higher-level requirements (cf. Rierson [45, p. 144]). In MB 5.0, DO-331 explicitly mentions that also “Design Models may contain derived requirements”.

DR 4 - Identification and documentation of derived Low-Level Requirements

A. To be considered as derived requirement, the following criteria shall be fulfilled:

- They shall not directly be traceable to higher-level requirements and/or specify behavior beyond that specified by higher-level requirements (DO-178C Glossary).
- They shall document a design decision, which is impacted by or has impact on externally visible behavior.

B. Derived requirements shall be identified by a unique ID.

C. Derived requirements shall clearly be visible as derived in Trace Data (DO-178C 5.5b).

D. Each derived requirement shall be documented and provide:

- a clear description
- a rationale for its necessity (where does this requirement originate from?)
- a rationale why it is not treated as HLR

E. Derived requirements shall be documented and the documentation shall fulfill the following:

- Derived requirements can be provided to system safety processes to assess any safety impact according to a method specified in the Software Design Standard and/or Software Development Plan (DO-248C FAQ #37).
- Derived requirements needed for hardware/software integration can be provided to the hardware life cycle process (DO-178C 2.2.3).

(A) provides some additional guidance on how to distinguish a derived from an ordinary requirement, since the definition in the DO-178C glossary entry (Figure 30) does not necessarily help identifying derived requirements. It is not always clear, whether a behavior goes beyond the system requirements, since the design is always a concretization of the requirements. Additional hints are provided in DO-248C. According to DO-248C FAQ #36, derived requirements may on the one hand be “design, performance, or architectural decisions”, which are typically not traceable to higher-level requirements. On the other hand, they may be design decisions, which introduce new additional behavior, which is not specified by higher-level requirements.

Also the idea of design decisions is difficult to apply, since finally any design of a requirement bases on design decisions. An additional aspect is given in the Requirements Engineering Management Handbook published by the FAA. The handbook describes derived requirements as means of flagging design decisions, which are "indicated by the fact that other design choices could be made that would affect the externally visible system behavior, while still meeting the system requirements". [116, p. 59] Hence, it has been decided in (A) that an impact on the "externally visible system behavior" must exist or the design decision must be impacted by any external behavior itself. Otherwise it is not a derived requirement. Table 16 discusses some cases.

Case	**Derived?**
Breakdown of a feature, specified by the high-level software requirements, into different functions to reduce complexity.	No – no impact on the externally visible behavior.
Algorithm (binary search) chosen to look up an element in a lookup table. The breakpoints and values are provided by Parameter Data Items.	Yes – the binary search imposes a requirement on the Parameter Data Item (the breakpoints must be ordered).
Algorithm (binary search) chosen to look up an element in a lookup table, which represents a hard-coded characteristic.	No – for the applicant of the function, the internal implementation is not relevant, as long as the output fulfills the requirements.
To achieve performance targets, a pre-sorting of (fix) values in an algorithm is performed the first time it is executed.	Yes – the first execution steps may be slower than all other execution steps. This is externally visible behavior.
A saturation is introduced in the design to avoid a division by zero. This changes the bandwidth of the output signal, but it still meets the higher-level requirements.	Yes – this has an impact on the externally visible system behavior. It is a design decision, when a division by zero is captured and what the reaction is. Typically, division by zero has a physical meaning.
Restrictions of the modeling language require a workaround.	No – as long as the workaround does neither affect performance nor any other externally visible system behavior.
A requirement states that a history of elements shall be stored in non-volatile memory. Storing values in a non-volatile memory requires additional functionality.	Yes – affected by hardware requirements and implementation or may affect the hardware requirements.
For testing purposes, the design of the algorithm is adapted. Additional modeling elements are needed to connect to testing tools. These model elements influence the generated code, but have no functional impact.	No – if it can safely be assumed that a functional impact does not exist. However, it also depends on how the absence of a function impact is shown.
Adding of scaling limits when using fixed point arithmetic (example from DO-248C FAQ #36).	Yes – has impact on externally visible system behavior and is additional functionality.
Interrupt handling is discovered to be needed during design (example from DO-248C FAQ #36).	Yes – affected by hardware requirements and implementation or may affect the hardware requirements.

Table 16: Examples of derived requirements

Derived requirements are discovered either during implementation or during traceability and requirement coverage review. Even though a design decision is not a derived requirement, its documentation may be worth it. Then, annotations in the SW Design should be used.

If a derived requirement is identified, (B) to (E) provide guidance on documentation. (B) is a critical requirement especially for model-based design. Scoping, uniquely naming, and documenting a requirement, which is, e.g., a part of a flow-diagram, may be challenging.

The rationales requested in (D) can be seen as countercheck. New derived requirements must be defined with care, since they may conceal, on the one hand, missing HLRs and, on the other hand, unintended functionality. Rierson [45, p. 144] recommends asking, why the derived requirement is necessary and why it is classified as derived and not covered by the HLRs? These rationales shall be documented, which also fulfills DO-178C 5.2.2b.

DR 5 - Unintended functionality

> There shall be no functionality, which is not specified by a (derived) higher-level requirement.

The requirement for the absence of unintended functionality has been derived from objective DO-331 Table MB.A-5:1. On the one hand, the SW Design shall not implement more functionality than actually specified. In addition, there shall be no dead design, i.e., functionality, which complies with HLRs, but is either not necessary to fulfill the requirements or is never transformed to Source Code. Dead design is explained in DR 30.

5.4.4 High-level architectural design

Designing software typically consists of two steps: The (high-level) architectural design and the detailed design. In DO-178C, this maps to the Software Architecture and LLR part of the design (cf. DO-248C FAQ #35). However, DO-178C does not emphasize a distinction as strong as in literature (cf. Saleh or Summerville [10, 82]) or other software standards like ISO 26262-6, which separates “Software Architectural Design” and “Software Unit Design” as process steps. DO-331 even allows combining Software Architecture and LLRs in a single artifact, the Design Model.

In this section, rules for the high-level architectural design have been established.

5.4.4.1 *Software modules*

In section 4, a new breakdown concept of the MBSwD into sub-processes for software modules has been introduced. This section provides the scope and definition of a *software module*.

Software modules hold software life cycle data from the Requirements to the Code process. They encapsulate cohesive functionality and have small interfaces, which eases concurrent development, maintenance, and lowers the integration effort later on.

DR 6 - Modules

A. The Software Architecture shall break down the whole software component into modules.

B. To any module, a set of higher-level requirements shall be allocated, which the module implements.

C. A module shall be handled as a separate Configuration Item according to DO-178C and shall be independently developed in the SW Requirements, Design, and Code Process.

D. A module shall define the properties of Table 17.

Module ID	Two letter identifier ([a-zA-Z][a-zA-Z0-9])
Module Name	Arbitrary length with characters [a-zA-Z0-9]
Component Module	Yes or No
Partially Usable Library	Yes or No

Table 17: Properties of modules

E. A module may depend on other modules. All modules must have been developed with the same software level under the MBSwD process, with the same modeling environment and same code generator settings.

As defined by (A), from the viewpoint of configuration management, a software module is defined as a separate DO-178C configuration item and thus, for example, subject to configuration identification, indexing, baselining, and change control (DO-178C 7.2). All artifacts inside a module go through the development process together until they are integrated.

Complexity of the process can be managed easier, if the properties listed in Table 17 are defined for each module. Analogously to the processes, there is one *component module* and several sub-level *modules*. A component module defines the component interface. Depending on the module type, a different process applies (cf. section 4.4.2).

In addition, modules can be tagged as *partially usable library* (cf. section 5.4.4.5).

5.4.4.2 *Software units and module data*

Below the module, the next architectural entity is the *unit* and *module data*. Units perform an operation with/on an input and provide an output. Module data are, for example, constants or type definitions. They can be used by units.

DR 7 - Units and module data

A. Modules shall consist of *units* and *module data*.

B. Units shall

- be independently verifiable (cf. DR 27).
- have defined interfaces.

C. Selected units and module data shall be accessible from other modules.

Figure 31 illustrates an example SW Design of a MBSwD component. It has a single component module, calling units of other modules. Some units are just called from inside its own module. There is a module at the bottom just holding module data, e.g., the interface type definitions. In this case, the partially used library module is not part of this component.

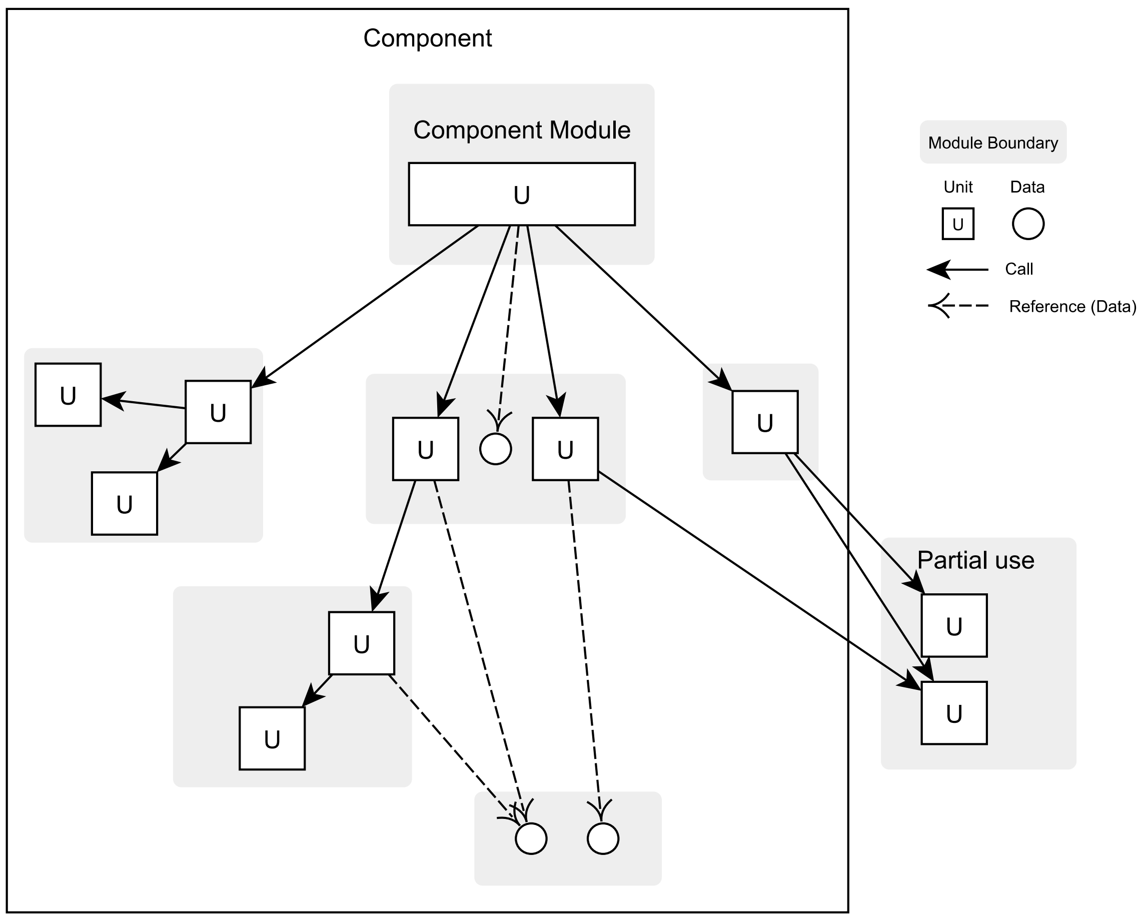


Figure 31: Module architecture example

5.4.4.3 *Module interfaces*

Interfaces are the key factor for a good design and have an important influence on verification interactivities. DO-178C 6.6.3b requires to ensure that “a correct relationship exists between the components of the software architecture”. The presented interface approach is an important foundation for this objective.

Modules have three types of interfaces of particular interest:

- *Component interfaces*
- *Inter-module interfaces*
- *Intra-module interfaces*

Component interfaces are those at the border of the whole software component, i.e., inputs, outputs, and Parameter Data Items. Inter-module interfaces connect units of different modules. Intra-module interfaces connect units of the same module.

The primary goal is to limit the quantity, size and scope of interfaces, since interface handling always bears additional risks. Three principles help to deal with interfaces:

- Encapsulation: Restricting access to internally used functions and data
- Coupling: Limitation of the number of interfaces
- Cohesion: Reduction of interfaces and preference of intra-module interfaces
- Contracting: Constraints on required interfaces

DR 8 – Encapsulation

> Access to functions or data over module boundaries shall be limited. Functions and data callable or usable from other software modules shall be explicitly identified.

Encapsulation is a technique to keep control of external interference and commonly used in traditional programming languages (like object-oriented languages, which allow the declaration of public or private variables). The design paradigm is also known as “information hiding” [82].

DR 9 – Coupling

> Coupling shall be measured and kept within thresholds.

Inter-module or external coupling describes the quantity of interfaces and dependencies between modules. Ultimate design goal is to keep these as small as possible.

Saleh [82] distinguishes the following four relevant types of coupling, from best to worst:

- Data-coupled: Data is passed between modules through inputs and outputs, which is necessary to execute the receiving module units
- Stamp-coupled: More data than required for execution is passed between modules.
- Control-coupled: Data, which changes the control flow, is passed to a module.

- Common-coupled: Modules are indirectly related by sharing of global data, structures, data types, or data files.
- Content-coupled: Statements that jump into another module and execute own functions in this context, or if one module can change the content of another module.

For all types of coupling, measures and thresholds are highly design tool and language specific.

DR 10 – Cohesion

Intra-module cohesion shall be measured and kept within thresholds.

Cohesion describes the reason, why the content of a unit, e.g., a module, belongs together. According to Saleh [82], different types of cohesion can be distinguished. Cohesion may be good or bad. Worst is coincidental cohesion. In this case, content is packed together without any reason. In the best scenario, functional cohesion exists, i.e., all content of a unit serves the same function ([82] Table 5.7 p.162).

DR 11 – Interface contracting

A. Data types and structure of data shall be fully specified for all module interfaces.

B. Interfaces shall be classified according to the contract types in Table 18.

	Component Input	Component Output	Inter-module Input	Inter-module Output	Inter-module Input (of Lib)	Inter-module Output (of Lib)
Unconstrained	x					
Wide contract					x	
Narrow contract		x	x	x		x

Table 18: Interface classification

C. Unconstrained and wide contract interfaces shall have a design range specified in HLRs.

D. Intra-module interfaces shall not have any type of contract.

Although coupling shall be minimized, interfaces are required, but wrongly understood interfaces can lead to completely invalid verification results. Constraining the module interface is called *contracting*.

All interfaces must specify the data type. This is a minimum requirement (A).

Interfaces have been categorized into three contract classes, depending on the preconditions, which are acceptable:

- *Unconstrained*: Input or output interfaces only have a data type specification. The value can be arbitrary (in the range of the specified data type). The software shall neither assume (for inputs) nor provide (for outputs) conditions on values.
- *Wide contract* (range unconstrained): The software can assume that the incoming values fulfill all software requirements (units, special quantities, …), but must assume an unlimited range (except the limitations provided by the data type). Vice versa, the software must make sure that outputs fulfill all software requirements, but must not limit them to a specific range.
- *Narrow contract* (range constrained): The software can assume that the incoming values fulfill all software requirements and are narrowed to the specified range. Vice versa, the software must make sure that outputs fulfill all software requirements and are limited to a specific range. The idea of narrow contracting is slicing the software into pieces.

The classification of (B) can be explained as follows. In terms of defensive programming, external incoming data shall not be trusted. These inputs can have any value, which is representable by the respective data type. This aligns with DO-248C FAQ #32, which proposes various defensive programming examples for “avoidance of input errors” and “avoidance of interface errors”.

Inter-module interfaces shall support a narrow contract (i.e., rely on and provide a range specification). Wide contracts always require a significant amount of robustness code, which decreases performance. Narrow contracts also ease the application of formal methods. However, due to their high impact on software safety, preconditions must be carefully and exhaustively verified. This is only possible, if all modules are developed under the same software level, otherwise it must “be confirmed that the higher software level component has appropriate protection mechanisms in place to protect itself from potential erroneous inputs from the lower software level component.” (DO-178C 6.3.3b).

Inter-module interfaces into library modules must have a wide contract for inputs, since library functions are used by different software developers for different purposes. The risk of violating a range constraint is thus much higher. In contrast, outputs shall provide a narrow contract. This interface can be controlled very well at a single place.

For (C), it is important to highlight the difference between the range specified in a contract and a *design range*. Design ranges are specified in HLRs or Interface Control Documents (ICDs) defining the normal operation range of the software (verified with normal range testing). However, the software must expect values outside this normal range, i.e., it must be robust, which is verified with robustness testing. A range specified by a contract is part of the SW Design and is not subject to robustness testing. Adherence of callers to contracted ranges must be formally provable, a violation is thus technically not possible and robustness concerning a violation must and cannot be shown. Any robustness code would be marked as unreachable in formal analysis.

Some examples for encapsulation and contract classification are given in Figure 32. Important to note is that, although the library module is not in the component, its interface is not considered as component interface, since the calling component ensures that no unconstrained data is exchanged.

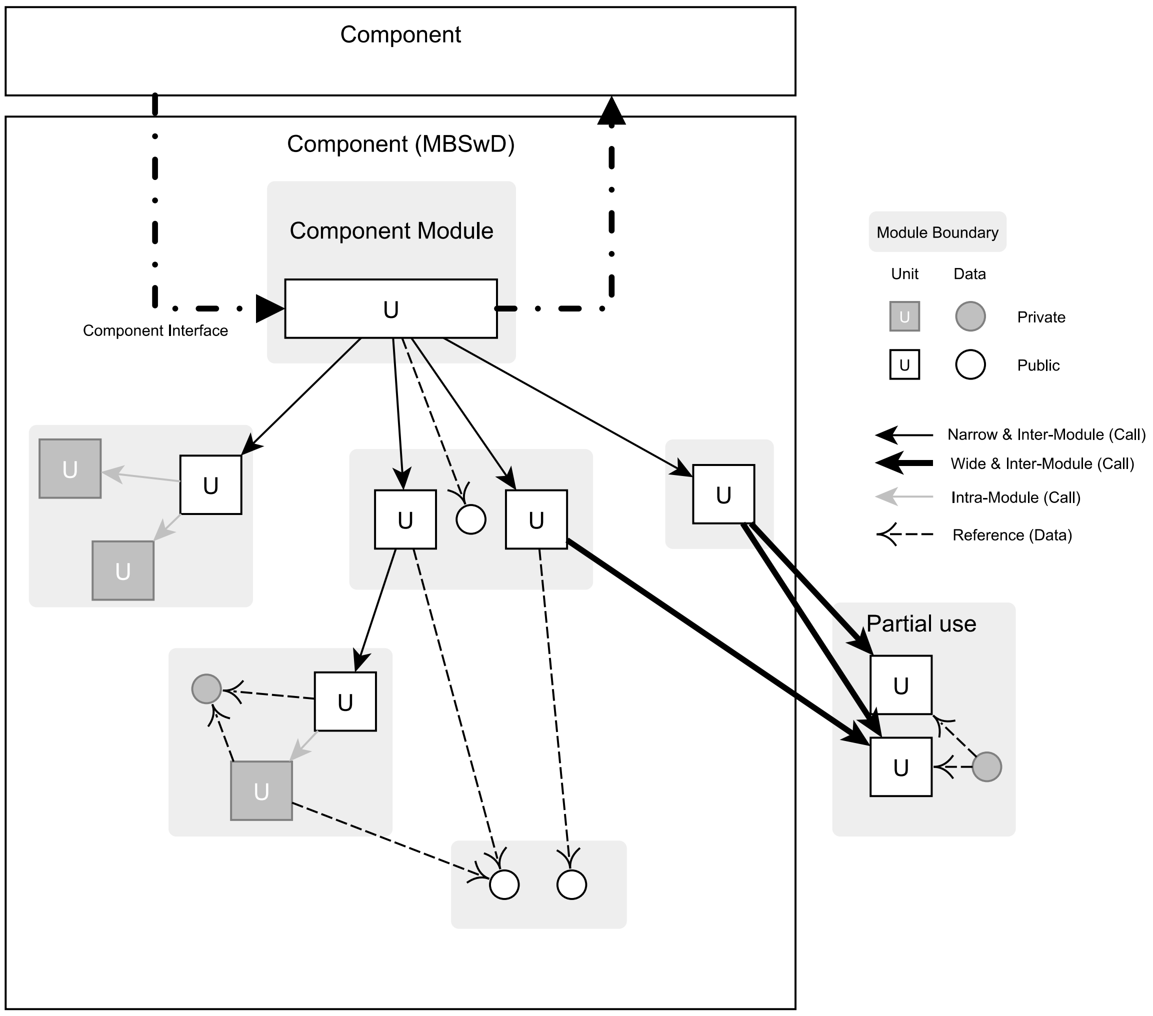


Figure 32: Module interfaces

5.4.4.4 *Component interfaces*

Component interfaces of the model-based software component only communicate with the software application framework (cf. section 3.1). An agreement must thus exist concerning the function call and data interface.

DR 12 – Component call interface

> The function call interface shall comply with the interface specified for the software component.

The function call interface depends on how the code shall be plugged into the surrounding framework. For example, whether tasks shall be triggered synchronously by a cyclic periodic interrupt timer. The rule above satisfies assumption AS 8.

DR 13 – Component data exchange interface

> The data interface shall comply with the data interface specified for the software component.

Selecting the right approach is highly project-specific and depends on the type of application and the capabilities of the software application framework

DR 14 - Unconstrained data handling strategy

A. Only the top-level model in the component module shall handle unconstrained data.

B. Cohesive and clearly identifiable data conversion and monitoring units shall be implemented in the top-level model for every incoming external data prior to any other operation and for outgoing external data after any operation.

C. External incoming data shall be monitored and converted prior to any use:

   - Special quantities shall be detected and removed.
   - Design ranges shall be checked and the values saturated or reset if necessary.
   - Data types shall be converted.
   - Scaling of integers shall be performed.
   - Units shall be converted.

D. External outgoing data shall be monitored and converted prior to sending:

   - Design ranges shall be checked and the values be saturated if necessary.
   - Data types shall be converted.
   - Scaling of integers shall be performed.
   - Units shall be converted.

According the definitions of the previous section, the format of external data is unconstrained, since it is often predefined by other components (e.g., commercial-of-the-shelf components) and cannot be influenced. Inside the software, however, limitations on units, special quantities (like Inf, NaN, …) exist. The target is thus to limit the access to unconstrained data and keep its “lifetime” as short as possible.

5.4.4.5 *Library modules*

The purpose of those modules is to share functions and data across different components. A module tagged as partially usable library can be used in other components as well. Such modules have special requirements concerning implementation and especially verification. For example, not all library functions have to be part of the component software, but are not dead design.

A typical example for library functions are repeatedly used filters or mathematical utilities.

DR 15 – Library modules

A. Units shall be interfaces for library modules, only (i.e., library functions). Their implementation shall follow the guidance for deactivated design in DR 31.

B. Requirements, from which the library module is developed, shall be independent.

C. The requirements shall state the conditions for usage.

D. Library functions shall be documented in addition to requirements.

E. The symbol of a library function shall be uniquely identifiable, non-misleading, and documented.

Units provide an independently testable interface by definition (A). Independent requirements simplify reusability across projects (B). (C) is a requirement from DO-331 MB.B.18.4. (D) is a requirement from DO-331 MB.B.18.9.

If library functions are used, the following rules apply:

DR 16 – Library module usage

If a library function is used, operational requirements for the library shall be documented.

This requirement arises from DO-331 MB.B.18.3.

5.4.4.6 *Parameter Data Items*

The purpose of Parameter Data Items (PDIs) in the context of the project has been discussed in assumption AS 9. For the described use case, no additional considerations (field-loadable, option-selectable, or user-modifiable software) apply.

The goal of PDIs is to decouple the verification of PDI Files from verification of the Executable Object Code. A PDI File is the exchangeable instantiation of a PDI, i.e., the values. This reduces verification effort, since values can easily be changed without touching the Executable Object Code. The conditions, which must hold, are provided in DO-178C 6.6:

- The structure of the PDI life cycle data allows separate management.
- Normal range testing on Executable Object Code for PDI values complying with the structure and attributes has been performed.
- Executable Object Code is robust with respect to PDI Files structure and attributes.
- All behavior of the Executable Object Code resulting from PDI values can be verified.

PDIs must have the same software level as the application they are used in (DO-178C 2.5.1).

Usage and implementation of PDIs must be planned. MathWork's reference workflow [32] does not address PDIs specifically, although they imply important design decisions. The following part of the section introduces the chosen way to handle PDIs from specification to implementation.

Figure 33 illustrates the intended processing of a PDI File in the software. Reading a PDI File and checking its values is delegated to the software application framework. The framework provides the PDI as *constant* (during the execution) to the software application. The software execution must be stopped to exchange the PDI file, i.e., it is loaded once at initialization of the software. The software application does not have to perform robustness checks on the PDI values.

In general, PDI File values must be considered unconstrained by the software, since DO-178C requires robustness testing for structure and attributes. For practical reasons, it has been decided that robustness code shall be part of the application framework code running prior to the software application. The reason is that PDI values are immutable and should be checked once during initialization and not every calculation step. SL/SF in the used release and in combination with the chosen verification tools has limited functionality to model initialize functions and efficiently check or update large arrays (e.g., lookup table values). In consequence, a *narrow contract* can and shall be assumed for PDI values (cf. section 5.4.4.3) in the model-based application.

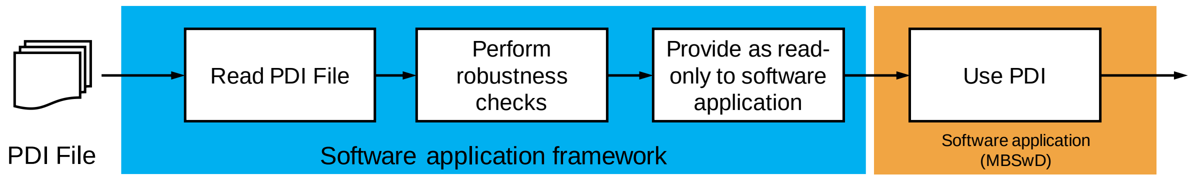

PDI File
Read PDI File
Perform robustness checks
Provide as read-only to software application
Software application framework
Use PDI
Software application (MBSwD)

Figure 33: PDI processing in software

Figure 34 shows the process responsibility for PDI development in the MBSwD process respecting DO-248C DP #20. Structure and attributes of the PDI are defined as HLRs in the main DO-178C process, since the software application framework has to handle reading and robustness checking. These *PDI Structure HLRs* may be derived or traced to system requirements. In the process at hand, they are documented in Polarion as separate work items incorporating structure and the following attributes (cf. DO-248C DP #20):

- Use case
- Data Type / structure
- Dimension
- Design range
- Unit

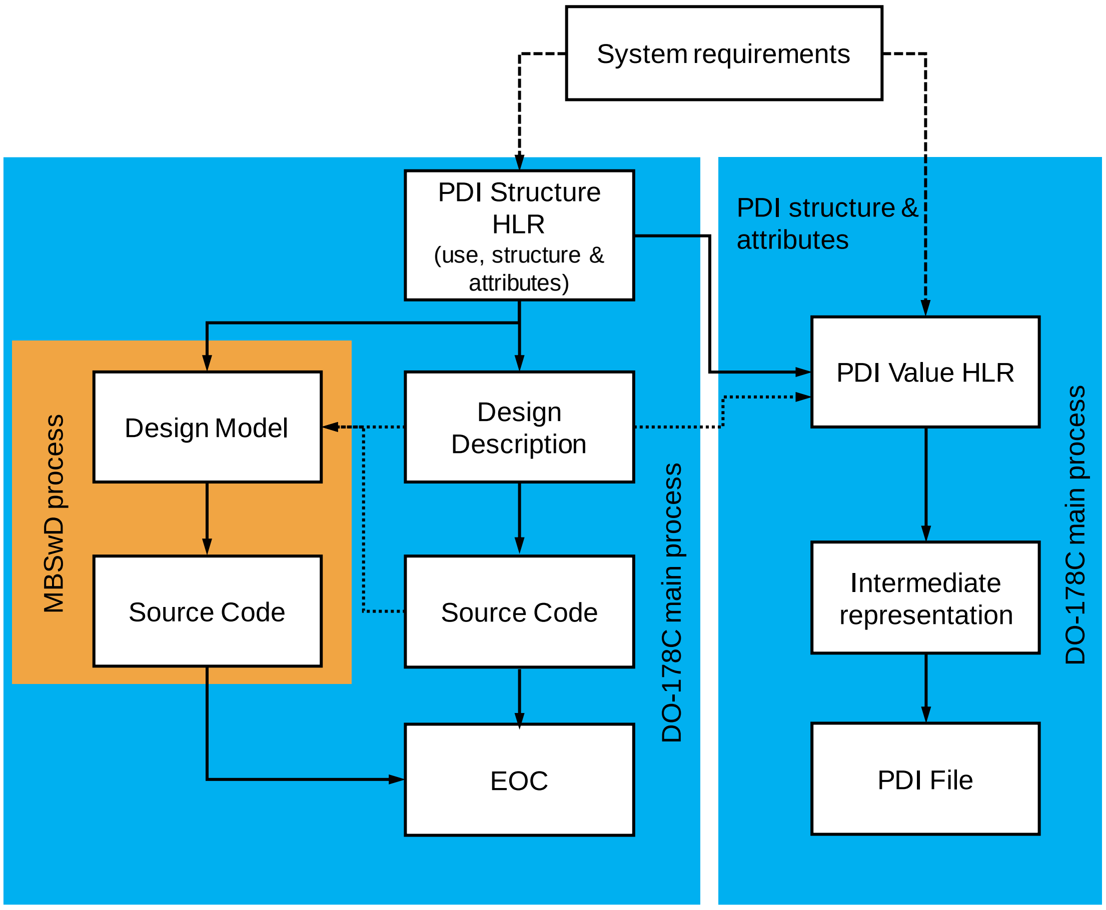


Figure 34: PDI view based on DO-248C Figure 4-4

Afterwards, the PDI Structure HLRs are relevant in three different workflows:

- Development of the software application framework (middle of Figure 34). It refines structure and attributes in the Design Description and implements the respective data structures as well as the reading functions in Source Code. The data structures must be accessible by the auto-generated code of the application later on, so it has been decided that the software application framework shall expose the values in global variables.
- Development of software application (left-hand side of Figure 34). The Design Model references the PDIs data structure and consumes the data. Therefore, both information from the Design Description and Source Code may be required. The reference in the Design Model provides traceability to the PDI Structure HLR.
- Development of the PDI File (right-hand side of Figure 34).The values are specified in separate *PDI Value HLRs.* PDI Structure HLRs drive the format of PDI values and the PDI File. The PDI Value HLRs are directly implemented into an intermediate representation (cf. DO-248C Figure 4-4), which is finally translated to a PDI File. All steps are part of the main DO-178C process.

DR 17 describes the relevant requirements from the perspective of the MBSwD process.

DR 17 - Parameter Data Items

A. Any reference to a PDI data structure of the framework code in the Design Model shall be traceable to the PDI Structure HLR.

B. To describe the MBSwD process, assumptions on the PDI implementation in the main DO-178C are necessary.

C. PDIs shall be referenced in the design and implemented in code in a way that they can be verified independently of the Executable Object Code.

D. PDIs shall not lead to deactivated design (cf. DR 31).

E. PDIs shall not lead to user-modifiable, option-selectable, field-loadable software, or Deactivated Code.

Reading of a PDI data structure in the Design Model underlies verification activities of the MBSwD. To verify the behavior of the Executable Object Code resulting from PDI values, normal range test cases shall be formulated as simulation cases in the MBSwD. Robustness test cases are formulated for the implementation in the application framework as part of the DO-178C process, only. PDIs shall not lead to Deactivated Code (cf. CR 15), the software must thus be tested thoroughly for all valid configurations.

5.4.4.7 *Error handling strategy*

DR 18 - Error handling strategy

A common error handling strategy shall be defined and evaluated, for example, if and how detected errors are reported to other functions and how they are logged.

If runtime-errors are caught in advance, a strategy is required to handle them. This may reach from reporting, over logging, to terminating the program. The error handling strategy is impacted by both the system design (e.g., whether a backup controller exists, which takes over in case of a runtime error) and software. Since there are many ways to implement it and since it has no further impact on the process itself, the error handling strategy is not detailed here.

5.4.5 Accuracy and consistency

Like for textual LLRs and software architecture, DO-331 also requires accuracy and consistency of the Design Model (as in DO-178C Table A-4:2,9). DO-178C 6.3.3b and DO-178C 6.3.3c provide some additional information on the objectives. DO-178C 6.3.3b specifically mentions data and control coupling considerations for software architecture. These have been covered with the high-level architectural design in previous sections. The following DRs focus on accuracy and consistency of the lower level detailed implementation.

DR 19 - Unambiguity, readability, and maintainability

A. The Design Model shall be non-ambiguous.

B. The Design Model shall be readable, understandable, and transparent.

C. The Design Model shall be maintainable.

D. The Design Model shall have low complexity.

DO-178C 6.3.2b explicitly mentions “non-ambiguity” and “accuracy”. “Non-ambiguity” is necessary to preserve the formal character of the model, hence ambiguous parts of the modeling language shall not be used (A). “Accuracy” is considered similarly to textual requirements. They are accurate, if they are short and precise. A Design Model is considered accurate, if it follows a readable, transparent, and maintainable modeling style (B). Readability means that structure and organization of the design support orientation of the developer and that the implemented functionality is clearly identifiable. This may be achieved by encapsulation of functionality, clear naming, readable charts, and so on. A maintainable SW Design can be easily changed and adapted. For example, if repeatedly used values are parameterized and the coupling with other functions is limited. Also unit and naming conventions contribute to an accurate design (cf. the following rules).

In addition, DO-178C 5.2.2 emphasizes, that the SW Design process “should avoid introducing complexity”. The complexity type of interest depends on the modeling language, e.g., a SL diagram requires different complexity considerations than a SF diagram. In consequence, complexity is detailed as part of the module design or fundamental modeling rules.

(B) to (D) are concretized with complexity measures in the derived modeling rules.

DR 20 - Technical Units

A. The used system of technical units shall be the International System of Units (SI) including the therein specified accepted non-SI units [117].

B. Angles shall always be given in radians.

C. Data at component interfaces may deviate from the unit specification.

A single unit system improves readability and interface compatibility and reduces the risk of failures. SI units have been a recommended system for years.

Most often, the format of external data cannot be influenced and it is thus unavoidable to have units deviating from the chosen system in the model. For example, English units like feet or knots are commonly used. However, those units shall only appear at component interfaces and be directly converted to SI units (cf. DR 14).

DR 21 - Design naming conventions

A. The provided naming conventions shall be applied throughout the SW Design.

B. Names shall be unambiguous, meaningful, and describe the labeled item properly.

C. Local language shall be English (US).

Design naming conventions are independent of the modeling tool. They describe the designation of mathematical expressions, units, or Greek variables. This set of guidelines has been developed by multiple employees of TUM-FSD. One example is provided in Table 19.

Vector	**Abbreviation**
Position of the point G relative to the center of the earth $\downarrow$ $(\vec{\mathbf{r}}^{G})_{B}$ $\uparrow$ Body-Fixed Frame is Notation Frame	`pos_G_B_m`

Table 19: Example design naming convention

DR 22 - Run-time errors

Algorithms shall be free of:

- arithmetic errors (division by zero, invalid use of mathematical functions outside their domain of definition).
- under- or overflow of integer and floating-point variables.
- index out-of-bounds access.

A general requirement is that the design shall avoid runtime-errors. Run-time errors do not necessarily lead to a program crash, but often result in unspecified or unexpected behavior. Consequences of run-time errors are strongly related to the programming language, the compiler, or floating-point implementation and may occur for both integers and floating-point numbers.

5.4.6 Algorithm aspects

Objective DO-178C Table A-4:7 is vague, it requires to “ensure accuracy and behavior of the proposed algorithm, especially in the areas of discontinuities” (DO-178C 6.3.2g).

DR 23 - Accuracy of mathematical algorithms

The accuracy of mathematical algorithms shall be confirmed.

Rierson proposes a review of mathematical algorithms with domain experts (cf. [45, 153f.]). In workflow 5, this DR may have already been considered on system level and documentation may be reused.

5.4.7 Traceability

Traceability is a pivotal concept of software development and mandatory in various accepted software standards, also in DO-178C as outlined in section 2.3. This section mainly focuses on the traceability between LLRs and HLRs as described in objective DO-178C Table A-4:6. This kind of traceability bases on semantical knowledge and must be manually created and maintained.

DR 24 - Traceability to higher-level requirements

A. Trace links shall be established from parts of the Design Model, which implement the feature, to HLRs.

B. Trace links shall be navigable in both directions.

C. Completeness of design shall be evaluable.

D. Trace links shall satisfy configuration management requirements, i.e., they shall be archived / under version control and restorable. Changes shall be traceable

D. Bi-directional impact analysis shall be leveraged between requirements and design.

Traceability means far more than just creating a link from one artifact to another:

- **Allocation:** Define, which requirements are implemented by which software component or module.
- **Realization and Storage:** Create a navigable trace between a model element and a requirement.
- **Restoration:** Store links in a way that they are restorable at any time and satisfy configuration management requirements.
- **Maintenance:** Keep the traces valid. Remove deprecated traces in case of removed requirements or model elements. Add new traces for new requirements or models.

- **Documentation:** Export trace data in a format, which allows archival and review (e.g., a traceability matrix).
- **Trace verification:** Check that all requirements trace to model elements and all model elements trace to requirements (requirement coverage).
- **Usage:** Check that all functionality has been implemented. In case of changes, find upstream or downstream impacted artifacts (impact analysis).

5.4.8 Target compatibility

Objective DO-178C Table A-4 shall ensure that no conflicts between the low-level requirements, software architecture, and the hardware/software features of the target computer exist (cf. DO-178C 6.3.2c, DO-178C 6.3.3c).

Since the model-based software only communicates with hardware/software features through the software framework application, the scope of this objective is limited.

DR 25 - Data type compatibility

> Only data types specified in the Software Code Standard shall be considered in the design.

Data types drive software low-level design or interface descriptions. They must comply with those supported by the programming language and the target environment. The data types are discussed in the Software Code Standard.

DR 26 - Floating-point arithmetic compatibility

> Floating-point numbers shall be preferred for mathematical algorithms and handled as defined in the Software Code Standard.

Calculations with fractional values are typically either carried out in floating-point or fixed-point arithmetic. C does not provide a built-in data type for fixed-point numbers, so the calculation in the code must be performed with integers. Floating-point calculations are typically performed by a separate floating-point unit (FPU) on the hardware. Build-in floating-point data types exist in C.

Thus, floating-point is easier to handle, but comes along with other problems. The semantics of floating-point operations are an interplay of hardware, compiler, and software libraries and the implementation of standards may slightly differ [118]. Another challenge is the precision and range. With a given scaling, the gap between adjacent fixed-point numbers is known and uniform. The precision of floating-point numbers is not uniformly distributed [119], floating-point numbers can represent a significantly higher dynamic range.

In this work, floating-point arithmetic has been chosen over fixed-point arithmetic due to various reasons. At the time project work has started, the support of fixed-point arithmetic in the SL/SF tool chain was very limited. Calculations had to be implemented by-hand in SL, which didn't seem to be practical. Nowadays, SL can abstract fixed-point arithmetic away from the user and Embedded Coder can auto-generate respective code. Another argument for floating-point arithmetic is that controllers are easier to implement with high dynamic range of values. And finally, the PPC has a highly performant double-precision FPU.

Floating-point arithmetic in the given avionics context has been investigated by Nürnberger [120].

5.4.9 Verifiability

DO-178C Table A-4:4 and 10 require that both the Software Architecture and LLRs are "verifiable". The meaning of this term is not further explained, except with the example that there shall be no unbounded recursive algorithms (DO-178C 6.3.3d).

Under this objective, Rierson mainly understands testability criteria (cf. [45, p. 153]). Since a Design Model has a higher degree of formality than textual LLRs, a couple of additional analysis techniques can directly be applied and extend the scope of "verifiability".

DR 27 - Units for model simulation and testing

A. Units must support the chosen testing strategy.

B. Inputs and required states of units shall be controllable in simulation and during Executable Object Code testing.

C. Outputs and required intermediate values of units shall be observable in simulation and during Executable Object Code testing.

A whole bunch of literature gives advices on testability criteria, but from the viewpoint of the SW designer and with the assumption of an independent process with black-box testing, an agreement on testability of units is the most important aspect.

The testing strategy has been introduced in section 4.6.3. Units are Design Model subsets, which provide observability and controllability interfaces both in model simulation and Executable Object Code testing. Although the term “unit” is used, simulation and testing is still requirements-based, and not white-box “unit testing”.

DR 28 - Testable design

A. The SW Design shall satisfy reachability criteria during simulation.

B. There shall be no recursive function execution.

C. There shall be no dynamic memory allocation.

D. There shall be no self-modifying code.

E. There shall be no user-modifiable, option-selectable or field loadable software.

The DR above relates to the system design assumptions presented in AS 8 Table 1, DO-178C 6.3.3d, and Rierson (cf. [45, p. 152]).

DR 29 - Analysis tool compatibility

The SW Design shall be compatible with all tools applied in the tasks for verification purposes.

The list of verification tools operating on the SW Design must be derived from the verification tasks and depends on the chosen modeling language/tool as in Table 20 for the process at hand..

Tool	Notes
SL Model Advisor as used in SwVP-DP-MB 1 and SwVP-DP-MB 2	
Simulink Code Inspector as used in SwVP-CP-MB 1 and SwDP-CP-MB 2	Amongst others, compatibility considerations under • [121, pp. 2–2ff.] • [121, pp. 3-2ff.]
Simulink Design Verifier as used in SwVP-DP-MB 5	Amongst others, compatibility considerations under • [122, pp. 3-2ff.] • [123, p. 6-29] In addition, the design must be compatible with the implemented preprocessing algorithms.
Simulation Case Management with Simulink Test as described in SwVP-DP-MB 6	
Model Coverage Evaluation (Simulink Coverage) as used in SwVP-DP-MB 9	Compatibility with the custom model coverage aggregation workflow must be given.

	Beyond that, *coverage preservation* shall be maximized. Coverage preservation is further explained in SwVP-DP-MB 9. The goal is that full model coverage leads to full or a very high degree of structural code coverage, when simulation cases are reused as test cases. The chosen model elements and settings may have a significant influence on preservation.
Model simulation as used in SwVP-DP-MB 8 including diagnostics	Amongst others, compatibility considerations under • [124, pp. 8-105ff.] Additional simulation requirements are: • Guarantee equality of model simulation and code execution results • Maximize simulation efficiency and robustness (simulation speed and avoidance of e.g., algebraic loops.) • Execution performance of compilation
SIL Simulation as used in SwVP-CP-MB 6	Amongst others, compatibility considerations under • [96, pp. 64-67ff.] • [96, pp. 66-35ff.]
Simulink Code Coverage as used in SwVP-CP-MB 7	Amongst others, compatibility with the custom model coverage aggregation workflow must be given.
Polyspace BugFinder as used in SwVP-CP-MB 2 and SwVP-CP-MB 3	
Polyspace Code Prover as used in SwVP-CP-MB 5	
Simulink Report Generator as used in SwDP-DP-MB 6	
SimPol and RMI as used in SwDP-DP-MB 2, SwDP-DP-MB 4, and SwDP-DP-MB 5	cf. section 6.4
PIL testing limitations	cf. section 4.6.3

Table 20: Analysis tool compatibility

DR 30 - Noncovered design

A. Noncovered design shall be justified.

B. There shall be no dead design in the Design Model.

Noncovered design is design, which is not reached by requirements-based model simulation and model coverage recording. Noncovered design should be resolved in the first place. If no resolution strategy is successful, it may be justifiable. If no proper justification applies, the noncovered design must be considered as dead design as illustrated in Figure 35. Dead design indicates unwanted functionality and requires rework of the Design Model.

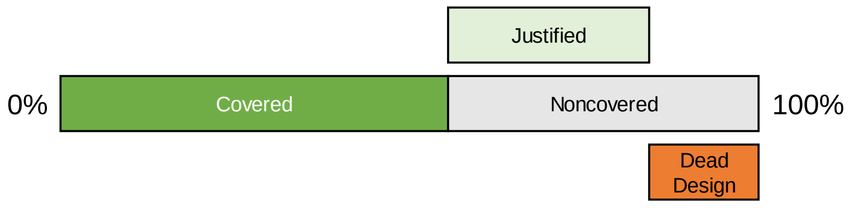


Figure 35: Covered and noncovered design (qualitative)

Resolution strategies for noncovered design are listed Table 21. They address the possible deficiencies given by DO-331 MB.6.7.2 and the specific resolution information for the MBSwD.

Cause (Numbering refers to DO-331 MB.6.7.2)	Resolution	Output
a. Shortcomings in simulation cases and procedures	Review and update of the test cases developed in SwVP-DP-MB 6 and SwVP-DP-MB 7	Covered design
b. Shortcomings in requirements from which the Design Model has been developed	Review and update of the higher-level requirements (cf. SwVP-DP-MB 9)	Covered design
c. Derived requirements	The noncovered part may be a derived requirement. Criteria for derived requirements and their documentation are as discussed in DR 4	Covered design
Shortcoming of analysis method	In some cases, a project- or company-wide deviation applies considering deficiencies of the analysis method. These deviations are safe to apply. Examples are given in 8.2.9	Justified noncovered design
d. Deactivated design	According to DR 31	Justified noncovered design
e. Dead design (unintended functionality)	If no other cause can be found, it is unintended functionality, which shall be removed	Dead design

Table 21: Model coverage resolution

The concept of deactivated design is similar to Deactivated Code of DO-178C (cf. CR 15), but instead of removing or disabling code in the compilation process, deactivated design is re-moved in the code generation process. Whereas Deactivated Code is separated in two cate-gories, i.e., code which is never executed, and code which is executed in some configurations, deactivated design is never executable, since it is never implemented in software. Thus, the main purpose for deactivated design are library functions.

Similar to Deactivated Code, the main difference between deactivated and dead design is that deactivated design is planned and traceable to requirements.

DR 31 - Deactivated design

A. All public units in modules declared as “partially usable”, and only those, can be deactivated design and shall underlie the following restrictions:

- They shall not call other units (in- or outside the module).
- The design shall be traceable to requirements and the partial use shall be specified in those requirements.
- The higher-level requirements of each of those units shall be independent of other units.
- Simulation cases of traced requirements from those units shall not contribute coverage to other units, i.e., they shall be independent.

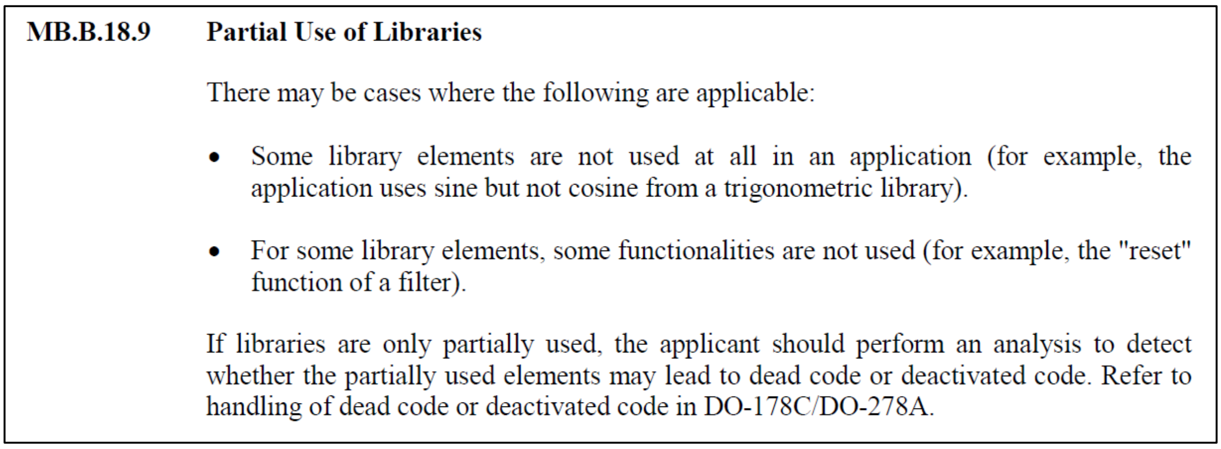
B. In the component process, deactivated design shall be identified, documented, and evidence shall be provided

- that deactivated design is not included in the generated code and cannot be inserted inadvertently.
- that test cases of traced requirements from deactivated design are not executed.

As a consequence, the safety challenge with deactivated design is not that it may lead to dead code, but that it may pretend that a specific functionality is implemented, although it is removed during code generation later on.

DO-331 addresses deactivated design as “deactivated functionality” in MB.6.7.2.d and requires to show prevention of realization, isolation, or elimination with “a combination of analysis, simulation and testing”.

Deactivated design can also be regarded as unused library functionality. The statement of DO-331 on unused and partially used model elements is given in MB.B.18.9 (Figure 36).



MB.B.18.9 Partial Use of Libraries

There may be cases where the following are applicable:

- Some library elements are not used at all in an application (for example, the application uses sine but not cosine from a trigonometric library).
- For some library elements, some functionalities are not used (for example, the "reset" function of a filter).

If libraries are only partially used, the applicant should perform an analysis to detect whether the partially used elements may lead to dead code or deactivated code. Refer to handling of dead code or deactivated code in DO-178C/DO-278A.

Figure 36: DO-331 MB.B.18.9 - Partial Use of Libraries

Both paragraphs MB.6.7.2.d and MB.B.18.9 require to make sure that deactivated design is not realized. In a modular approach, the decision, whether design is actually deactivated or not, can only be made on component-level, when all modules are integrated. On module-level, the design can only be marked as *potentially deactivated* or not. Assume a partially usable library module with various utility functions. A function is only deactivated design from a process perspective, if it is never used in the whole module hierarchy.

Thus, for public units of library modules, code is generated and they are subject to simulation, model coverage, and structural coverage on module-level. On module-level, no difference is made except that certain independence is explicitly required (A).

(B) describes the additional work performed in the component process, but also in subsequent testing processes. For example, the Source Code generated in the component process shall not contain the code of deactivated functions (i.e., functions never called in the application).

Deactivated design may be used for small library units (utility libraries). Deactivated design is not the technique to handle features and software variants. Therefore, whole software modules can be used.

Figure 37 extends the previously introduced example model architecture with deactivated and dead design. Units not called in the project and not part of a partially usable module are dead design. Any module data not referenced in a unit, is dead design. If the unit itself is dead design, the module data is dead design, too. Deactivated design can only occur in the partially usable library. If module data is linked to a deactivated unit, it can implicitly become dead design.

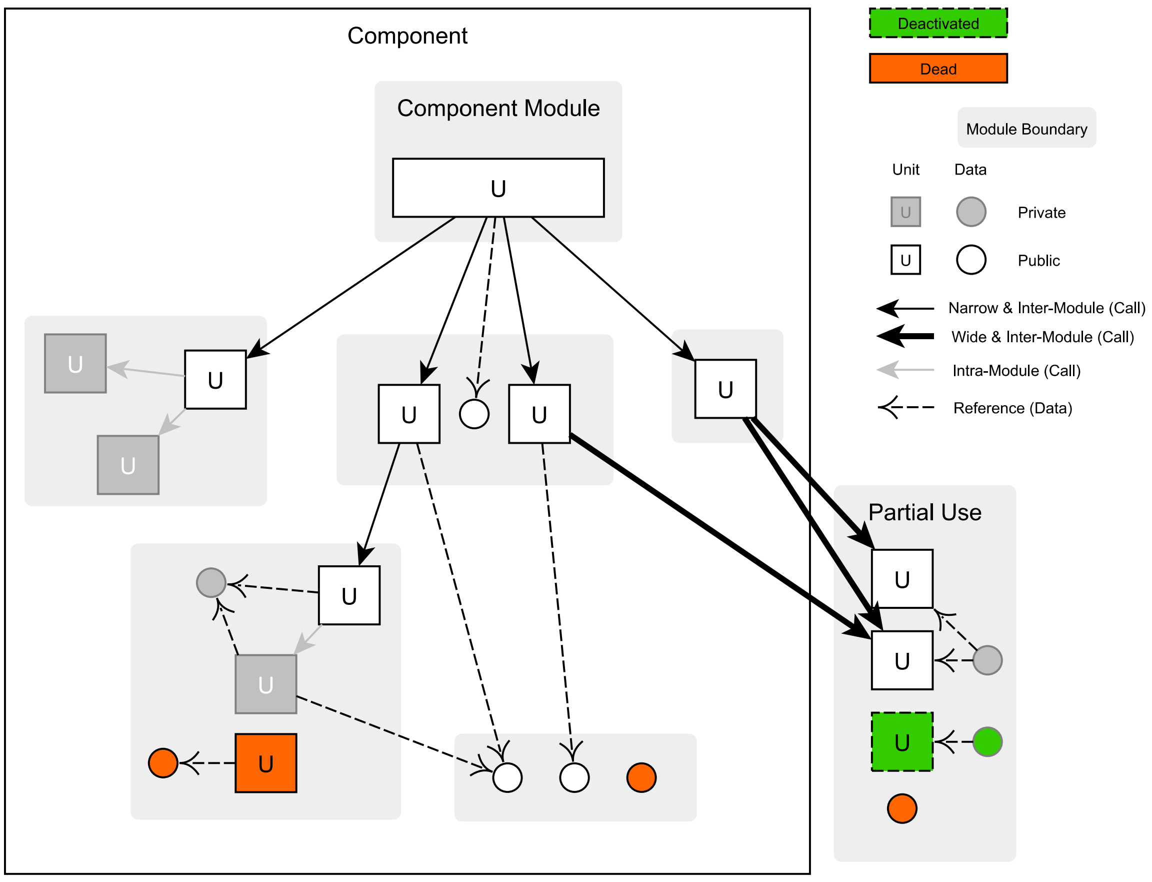


Figure 37: Deactivated and dead design

5.5 Coding rules for code generation

Defining the Software Code Standard in traditional DO-178C development is state-of-the-art. Not so common is its definition for MBSwD with automated code generation, since DO-331 does not provide an updated interpretation and definition of the Software Code Standard for model-based techniques.

Contribution 6: A set of coding rules specifically tailored to auto-generated code of Embedded Coder has been written. The coding rules have been specifically defined with respect to the used code generator, compiler, and hardware.

In the presented process, the generated Source Code is fully determined by the model and the code generator configuration, which themselves are specified in the Model Standard. The Code Standard is thus less a document for developers, but more for the process responsible to document the code requirements as well as to provide rationales for modeling rules and the basis for code verification activities. This is the reason, why the coding rules are discussed prior to the modeling rules and have a drastically reduced scope compared to traditional Code Standards.

In the work for the thesis, a consistent and complete set of code rules from this new point of view has been created. The rules are specifically tailored to the modular MBSwD, automated code generation, and the used compiler.

The target environment has been briefly introduced in sections 3.1 and 3.4, more detailed information provides the following material:

- PowerPC e300 target (PPC Manual [125])
- Compiler-support and implementation provided by the CompCert (cross-)compiler for PowerPC (CompCert Manual [57])
- GCC Preprocessor (GCC Preprocess Manual [56])
- GCC Linker (GCC Linker Manual [58])

The coding rules (CR) can be found in Appendix A. They are grouped according the objectives of DO-331 Table MB.A-5. A large part is compliance with the MISRA C:2012 standard (“Guidelines for the use of C language in critical systems”) [126], further on denoted as MISRA C.

5.6 Module design rules

This section presents a rule set, which goes beyond existing modeling guidelines. On the one hand, the new rules focus on transferring the architectural design and modularization concepts to SL/SF. On the other hand, they constrain SL/SF as modeling language (safe modeling subset). Objective is to provide a selection of features, which allows modeling that is compatible with the defined process tasks, as well as with respect to the previously defined design rules. It also leads to code generated with Embedded Coder that conforms to the coding rules.

The rules represent the best practice approaches gained in the various projects accompanied by the author. The rules, which are ready-to-use, save companies and engineers, which enter MBSwD with SL for the first time, from costly, iterative process adaptions to achieve Design Models that are compatible with a process.

In contrast to the design rules, the structuring of the module design rules in this thesis does not reflect the DO-objectives. They are roughly divided in overall concepts, the high-level as well as the detailed design and then describe the usable model elements.

5.6.1 Summary of rules

5.6.2 Naming convention

Contribution 7: A large set of naming conventions has been established ensuring consistent naming of model elements throughout the design. The rules are important to avoid identifier clashes or identification of responsibilities for model elements in a team-based development process.

In addition to the naming conventions in DR 21, further naming constraints are necessary for SL/SF and its model elements.

MR 1 - Naming conventions

The provided naming convention document shall be applied.

Publicly available naming conventions incorporated, as in the MAAB guidelines, only define absolute limitations, like usable characters and length of identifiers. However, the goal of naming conventions is far beyond that as discussed by Hochstrasser [36] in detail.

Therefore, a naming convention document has been assembled, which is separated into:

- design naming conventions, as specified in DR 21
- project-independent modeling naming conventions, standardizing the naming of model elements with respect to compliance with modeling and coding rules (e.g., MISRA C), conflict-free integration of software modules as well as readability of models and code
- project-dependent modeling naming conventions for specifics of the respective project

In the following, some rules just refer to the naming convention document, however a few rules directly provide naming conventions. In this case, they are just a repetition from the naming convention document for the sake of completeness.

5.6.3 High-level architectural design

Contribution 8: Rules for the high-level architectural design have been defined, which map the generic Design Rules to the SL/SF development environment. They dictate a consistent solution to specify interfaces of architectural entities, including rarely applied concepts of contracting and encapsulation. To the authors knowledge, the provided rule set is the only rule set addressing modular design / code generation from this perspective and thus significantly reduces the adoption effort for any reader.

5.6.3.1 *SW Design overview*

This section introduces a common terminology, discusses the repeatedly arising questions, what exactly the Design Model in SL/SF is, how formal the SL/SF "language" is, and how to handle a SL/SF model with respect to configuration management.

A SW Design in SL/SF composes of a few core components as shown in Figure 38. This view is not complete and just highlights the most important ones.

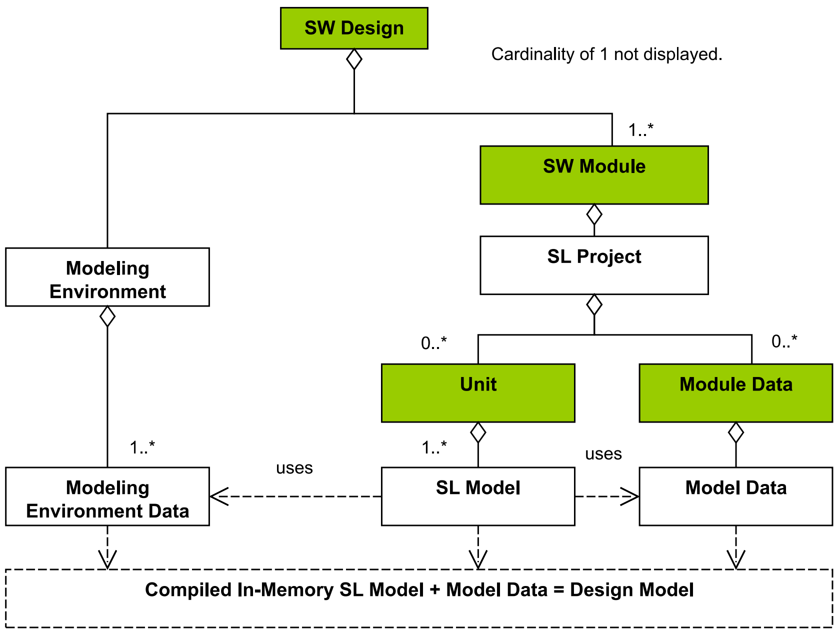

SW Design
Cardinality of 1 not displayed.
1..*
SW Module
SL Project
Modeling Environment
0..*
0..*
Unit
Module Data
1..*
1..*
uses
uses
Modeling Environment Data
SL Model
Model Data
Compiled In-Memory SL Model + Model Data = Design Model

Figure 38: Core components of SW Design in SL/SF

Multiple modules compose the SW Design. It has been decided that each module maps to exactly one SL project. A *SL project* is a feature provided by MathWorks serving as container for artifacts and leveraging additional functionality like dependency analysis, source control integration, and collaboration features for this scope [124, pp. 16-3ff.].

Functionality is implemented in *SL models* representing graphical data flow (SL) and state (SF) diagrams. Multiple SL models form a unit and multiple units form a module (cf. DR 7). Additionally, there is *model data,* which is equivalent to module data and has an independent life cycle of the SL model (cf. DR 7).

One SL model is one file in the file format `.slx`. SL models can call each other, this concept is called *model referencing* [124, pp. 8–2ff.]. SL models themselves can have multiple layers, which are called *subsystems* [124, pp. 3-10ff.].

The algorithms in SL models are implemented with *graphical model primitives* (e.g., blocks) and model data (e.g., data type definition or constants). Graphical model primitives form the SL model. Model data is associated with the SL model or graphical model primitives.

SL model and subcomponents reference and use *modeling environment data*, which is not part of a SL Project, but of the *modeling environment*. This contains for example settings controlling the behavior of the SL model, the *configuration settings*. The modeling environment can also contain additional graphical modeling primitives and model data. The data in the modeling environment is not adaptable by the developer as discussed in section 5.8.

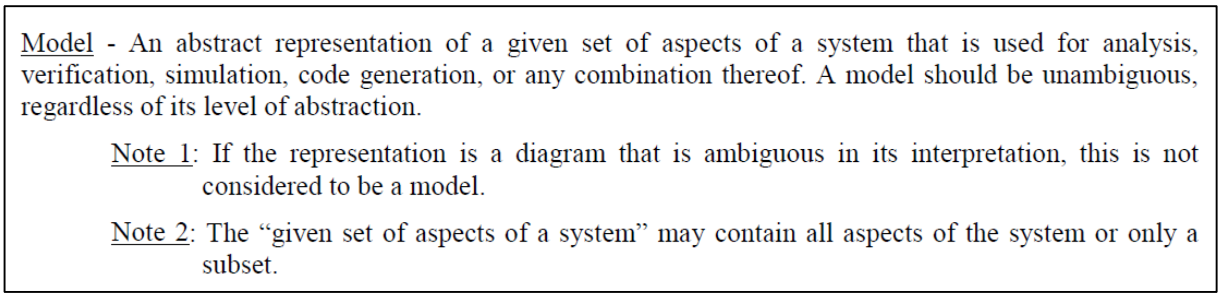
5.6.3.2 ***Design Model***

MR 2 - SL/SF as Design Model

A. The compiled in-memory SL model shall be considered as Design Model.

B. The compiled in-memory SL model shall be identified using the structural checksum, the file checksums, or both, whatever is appropriate for the given case.

The DO-331 Glossary describes a Model as in Figure 39.


Model - An abstract representation of a given set of aspects of a system that is used for analysis, verification, simulation, code generation, or any combination thereof. A model should be unambiguous, regardless of its level of abstraction.

Note 1: If the representation is a diagram that is ambiguous in its interpretation, this is not considered to be a model.

Note 2: The "given set of aspects of a system" may contain all aspects of the system or only a subset.

Figure 39: Definition of the term "model " according to DO-331 Glossary

Main requirement is its non-ambiguity in interpretation and its usability in analysis, verification, simulation, code generation, or any combination thereof. A "raw" SL model does not fulfill the "non-ambiguity" requirement, since it does, for example, not define an execution order of blocks or can have unspecified data types. However, SL models can be compiled, i.e., they can be converted into an executable form [124, pp. 3-17ff.]. This process adds the missing information, like the execution order, and performs semantic as well as syntactic checks. Uncertainties or errors are reported during the compilation process. The additional information of the compilation process can be displayed in the SL editor as well. The *compiled model in memory* is non-ambiguous. Thus, MathWorks defines the compiled, in-memory representation of the model as the actual LLRs [32, p.1-10].

Defining the Design Model as in-memory representation comes along with consequences:

- A possibility must be found to uniquely identify the compiled model (e.g., as reference for verification reports).
- There must be a possibility to generate a persistent export as part of the DO-178C Design Description (cf. DO-178C 11.10).
- Some verification activities must be performed on the compiled model.

Identifying the configuration of a compiled model is unfortunately not straight-forward. In general, three options exist:

1. *Structural checksum.* The behavior of a compiled model in memory is uniquely identifiable by its structural checksum [127, pp. 2-527ff.]. However, structural equivalence ignores fundamental non-functional parts of the model, e.g., the graphical layout, documentation, or the properties of some parameter constructs. Since the Design Model represents both architecture and LLRs, one of its primary goals is to explain, how the software works. Readability thus plays an important role and the structural checksum is not always sufficient.
2. *File checksum.* A file hash, as described in [127, pp. 2-636f.], is calculated over all files contributing to the SW Design. If all file dependencies are known and a combined file checksum is calculated, the configuration is, in theory, fully identified. Shortcoming is that the compilation process must behave equally.
3. *Model version.* The model version is an internal model property, which is incremented each time the model is saved [124, pp. 4-62ff.]. Similar to the file checksum, it only describes the Design Model holistically, if all dependencies are known. Main problem with the model version is that it can be easily tweaked.

In consequence, only a combination of structural checksum and file checksums is a valid approach for configuration identification of the SL model.

DO-178C requires a Design Description as output of the Design Process defining LLRs and software architecture (cf. DO-178C 11.10). A human-readable dump of the compiled model in memory must thus be possible. For the Design Description, MathWorks [32, p. 1-10] proposes to use the “System Design Description” report export (cf. section 8.1.1).

MR 3 - Model compilation

A. Any SL model shall compile without errors.
B. Warnings thrown during compilation shall be documented and justified.

With the definition of the Design Model, model compilation becomes a central feature, which should be fully understood. Model compilation can be invoked with the so-called `model` command [127, pp. 2-378ff.].

There are a couple of settings controlling the compilation process, from diagnostic to influential control and data flow settings. A few very important settings for the given safety-critical process have been listed in Table 22.

Parameter	Description	Proposed Value	Rationale
Block reduction (`BlockReduction`) [128, p. 2-99ff.]	Block reduction removes redundant data type conversion, dead design and fast-to-slow rate transition blocks from execution.	`Off`	All criteria for removal describe design flaws. If removed from the design, they will not be discovered in requirements-based simulation and testing. Reduced blocks may have traceability to requirements and imply, that a functionality has been implemented, although SL has removed it.
Conditional input branch execution (`ConditionallyExe-cuteInputs` [128, pp. 2-102ff.]	Data flow branches are only executed when needed, e.g., the inputs branches to a Switch block.	`On`	This setting totally changes the behavior of data flow, and may lead to severe safety issues and runtime-errors, if not consistently applied throughout all SL models, since developers may assume the wrong behavior. Example: If an index is greater than the maximum size of an array, the last element shall be selected, otherwise the element referred by the index. If the different decisions are implemented in SL in a conditional branches and conditional branch execution is off, this will lead to a runtime-error (both decisions are always executed, only the result selection is a true switch). It will work if the setting is on.
Solver type and solver (`Type/Solver`) [128, pp. 7-11ff.] [128, pp. 17–14ff.]	The solver type can be either fixed-step or variable-step, defining the step size. The solver is either discrete or continuous.	`Fixed-step` and `discrete`	Execution on the target happens with fixed-step size (the step() function is called periodically with a fixed time interval). Continuous states are not supported by a couple of verification tools (e.g., SLCI).

Table 22: Selected simulation configuration settings

5.6.3.3 *Modules*

A SW Design is defined as the aggregation of SW modules, which themselves consist of units and module data (cf. DR 6 and DR 7).

MR 4 - Module definition in SL/SF

A. Each module shall base on a single SL Project.

B. Each SL Project shall have a unique name with the following syntax: `<module-ID>_<module-name>`.

C. The properties of DR 6 shall be encoded in a project specification XML file located in the root-folder of the project.

A SL project defines the module scope, since it binds multiple file artifacts together. Listing 1 shows an example module description for (C).

```xml
<module xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="schemas/mrails-module-schema.xsd">
   <id>fc</id>
   <name>flightcontrol</name>
   <partial-use>0</partial-use>
   <component>1</component>
</module>
```

Listing 1: XML module description

From time to time, the name *mrails* is mentioned. *mrails* is the name of the process-oriented build tool introduced in section 7. Among others, it provides the infrastructure to create and manage the modules.

5.6.3.4 *Units*

In SL/SF, a unit must fulfill the requirements imposed by DR 7, which especially include compatibility with the testing strategy in section 4.6.3. In particular, this means independent simulation, SIL testing, and PIL testing of the unit interface.

MR 5 - Unit definition in SL/SF

Any SL model can be a unit. Subsystem blocks of SL models shall not be used as unit.

SL models have a defined interface and are easy to handle in configuration management, since they are a single file. Subsystem blocks (or just called subsystems) are not independently testable in SIL and PIL. Section 5.6.4.1 refers to further details on other options and their advantages or disadvantages.

MR 6 – PIL and full-image testable units

A. Inputs, outputs or intermediate values, which must be set or observed, shall be exported by a Simulink.Signal with a unique identifier as (SINGLETON SIGNAL as further explained in. MR 37).

B. Simulation cases with reusable models as tested interface shall never be declared as full-image tests.

C. Intermediate values (inputs, outputs, states,...) shall never be observed and controlled in simulation cases.

The main challenge is making input, outputs, states, and intermediate values accessible on the target. Inputs and outputs of the tested interface are always accessible, but if the SL model is tested in the context of another model (which is also the case in a full-image test), inputs and outputs become intermediate signals.

Intermediate signals and states in SL are only visible in C code, if they are translated to globally visible identifiers in C code. Embedded Coder can do this for every signal, but both PIL and the full-image testing framework need to know, how a signal in SL maps to its related variable in code. This traceability is the main challenge.

By default, signals are mostly translated to local variables and the naming is generated by Embedded Coder. The way of making signals global is attaching a so-called Simulink.Signal object to a signal line in SL. The Simulink.Signal object allows specification of further coder settings. Most important is the so-called *storage class*, which controls whether a signal is translated to a global variable and how it is named in code.

An `ExportedGlobal` storage class ensures that a globally accessible variable of the same name is generated in Source Code for the respective signal [96, pp. 19-160ff.]. This variable is written once during one execution step. In consequence, the storage class is only usable in SL models, which are executable once.

SL models, which have the ability to be executed multiple times (reusable models), cannot use this storage class. They have to use the `SimulinkGlobal` storage class [96, pp. 19-160ff.]. The storage class exports signals as global variables as well, but does not have the naming restriction. Embedded Coder adds a (random) postfix for every instance.

In the case of `ExportedGlobal` storage classes, the trace between signal name and identifier name in code can be estimated, since both have the same name.

For signals of type `SimulinkGlobal`, it is not straight-forward as shown in example 2 of Appendix C. At the time when this thesis was written, no direct way to obtain the mapping in R2017b was known to the author. Different workarounds to extract trace data from supported data exchange interfaces have been tried out, but came up with significant implementation effort (cf. C API [96, pp. 43-2ff.]) or were even not applicable (e.g., ASAP2 export does not support `SimulinkGlobal` storage class [96, p. 44-9]).

To sum up, in PIL, inputs of reusable models can be controlled and outputs be observed, if the reusable model is the model under test, and not a nested model. Intermediate signals of reusable models, i.e., also inputs and outputs of nested models, can never be written or observed. Since a reusable model is not the tested interface in a full-image test, no test case of reusable models can be executed on a full-image.

Models, which can only be executed once, support both, PIL and full-image tests.

A final word on states, like in the Unit Delay block [127, 1-1723ff.]. For all states, the same requirements as for intermediate signals apply. If they shall be controllable and observable, they must be exposed by defining the appropriate storage class and a name.

MR 7 – Unit encapsulation

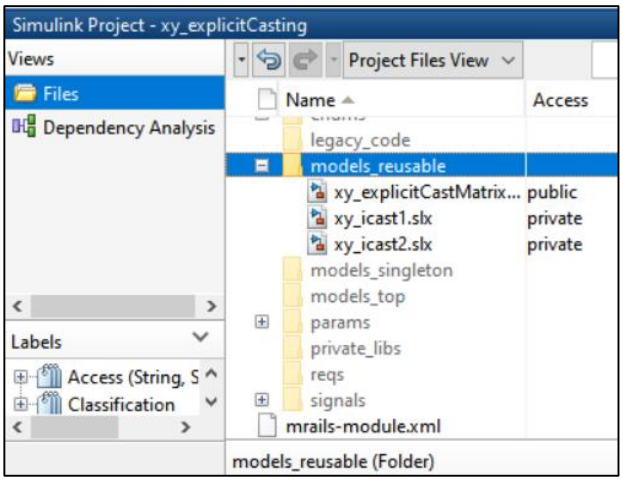
A. *Public* and *private* SL models shall be distinguished. Public models are usable in other modules, private models shall not be used in other modules.

B. SL models shall be labeled as *public* and *private* in SL Project in a category called *Access*. Unset labels shall be conservatively interpreted as private.

SL models are functions and shall be encapsulated on module-level according to DR 8. Distinguishing public and private accessibility is a known concept from object-oriented programming.

SL and SL Project have no built-in functionality to restrict the accessibility of models. Thus a workaround had to be found. SL Project allows placing meta data in form of labels on files, which are part of the project. Custom labels are allocated to custom categories. Figure 40 illustrates the usage of labels.

Labeling SL models does not preempt invalid referencing. However, it can be verified with a check afterwards.



Name	Access
legacy_code	
models_reusable	
xy_explicitCastMatrix...	public
xy_icast1.slx	private
xy_icast2.slx	private
models_singleton	
models_top	
params	
private_libs	
reqs	
signals	
mrails-module.xml	

Figure 40: Custom encapsulation (access) labels in SL Project

5.6.3.5 *Module architecture*

Modular development always reveals a dilemma of affiliation. In Figure 41, the component module references (integrates) two other modules, the auto flight and the system automation module. `fc_main()` calls `af_core()` and `sa_moduleSwitch()` with passing the sensor data. The data type of the sensor data is specified by a Simulink.Bus. Simulink.Buses are model data, which allow the specification of a structured data type in SL (cf. MR 30). The question is now, in which module the Simulink.Bus object is stored. The answer is that it is not possible in any of the three modules. If it was stored in the component module, both the auto flight and the system automation module would not be able to access it. If it was stored in the auto flight module, the system automation module would not be able to access it.

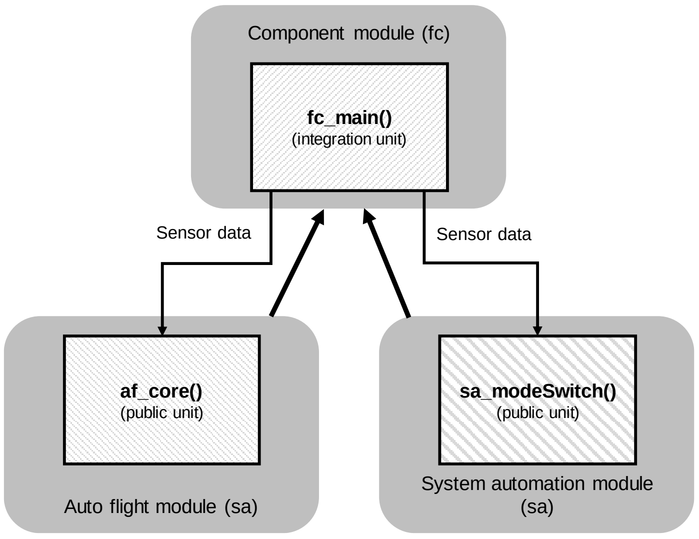


Figure 41: Interface affiliation dilemma

MR 8 – Shared module data

Save shared data in defined, globally managed modules.

In software design, it is good practice to keep shared data between modules in a separate module [129, p. 15]. Suggested is a module structure according to Figure 42 managing shared data globally. The figure shows example module dependencies for a flight controller. The “Auto Flight Control” embeds the “Inner Loop Control”. The “Communication Protocol” and “Monitoring” modules are independent of the “Inner Loop”. Functions and data shared between projects and modules is sorted out into a *project-specific global module*. All modules integrate this shared module, either directly or through transitive dependencies (e.g., in Figure 42, module 2 includes the global module through module 3). This eases configuration management dramatically. All modules are finally integrated into a single component module. All modules are subject to the module process and the component module is additionally subject to the component process.

In the previous example, the project-specific global module would hold the Simulink.Bus data type specifications for inter-module interfaces.

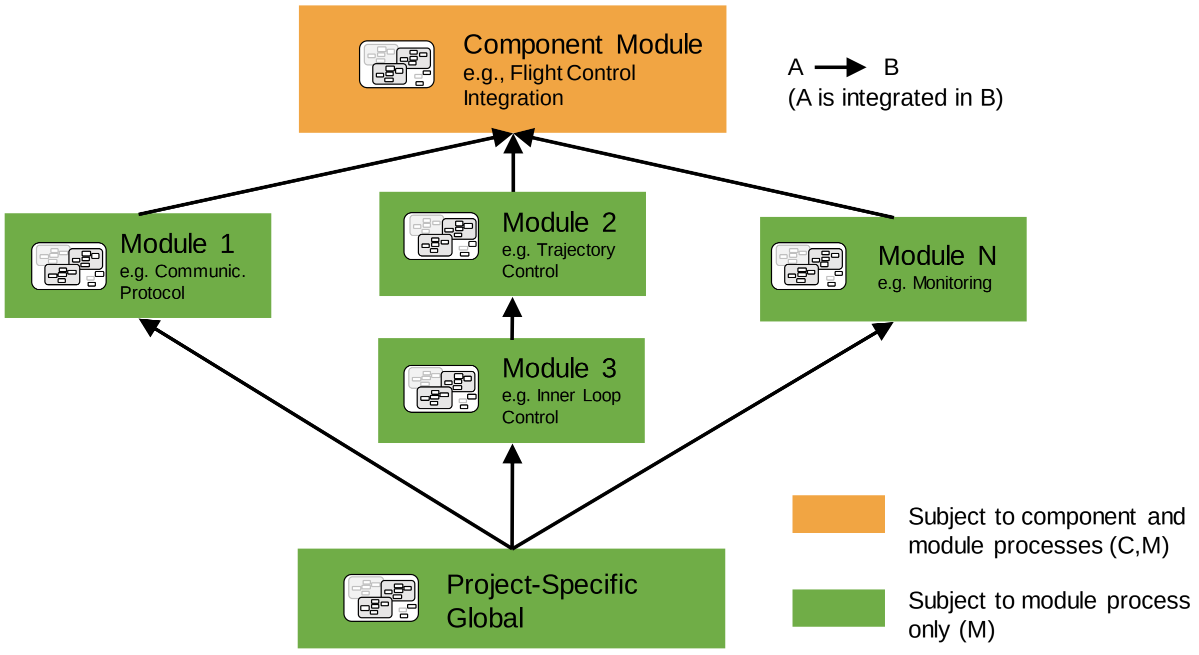


Figure 42: Example module dependencies/architecture

In the best case, a module architecture serves multiple purposes:

- It defines the required modules, their naming, the dependencies and interfaces between the modules a priori to implementation.
- It helps to allocate requirements to modules.
- It can be reused as integration model that allows both simulation as well as code generation of interfaces in the end.

The following outlines a simple approach for such a “multirole” module architecture.

MR 9 – Multirole module architecture

A. A module architecture shall be defined during D1^C using SL by

- creating empty harnesses for units with Subsystem blocks in the component module
- defining interfaces between units in different modules, preferably with Simulink.Bus objects in a project-specific global module.

B. Implemented units shall be integrated into the component module by replacing the harnesses during D^M of the component process.

The idea of the architecture process shall explained using the example architecture in Figure 43. The boxes represent units. The two-letter notation indicates the module they belong to. Note that a single module can contain multiple relevant units. All external inputs are routed through “Input conversion, monitoring, and voting”, and all outputs through “Output conversion, monitoring, and voting”, which are implemented in the top-level model.

Such a module architecture is defined in D1^C. Therefore, the modeling tool (SL/SF) is already used, since the module architecture shall transition into the future top-level model later on. Each public unit is replaced by a *unit harness*, since the actual implementation does not exist yet. As unit harness, a Subsystem block is used, which is exchanged by the actual SL model later on. Each unit harness is assigned to a module and to each module, requirements are allocated. Since units belong to modules, they implicitly define *module dependencies*.

The interfaces of each unit harness are modeled in SL/SF and should use Simulink.Bus objects. The interface definitions (Simulink.Bus objects) are stored in the project-specific global module. The module architecture is stored in the component module, since it becomes the top-level model.

In D^M of the component module, implemented units are integrated into the module hierarchy by replacing the harnesses with the actual SL models. The module architecture is continuously transformed into the final, executable top-level model.

A nice side effect is that architecture and interfaces are available prior to the implementation and test cases can be developed in parallel and independently from the actual implementation.

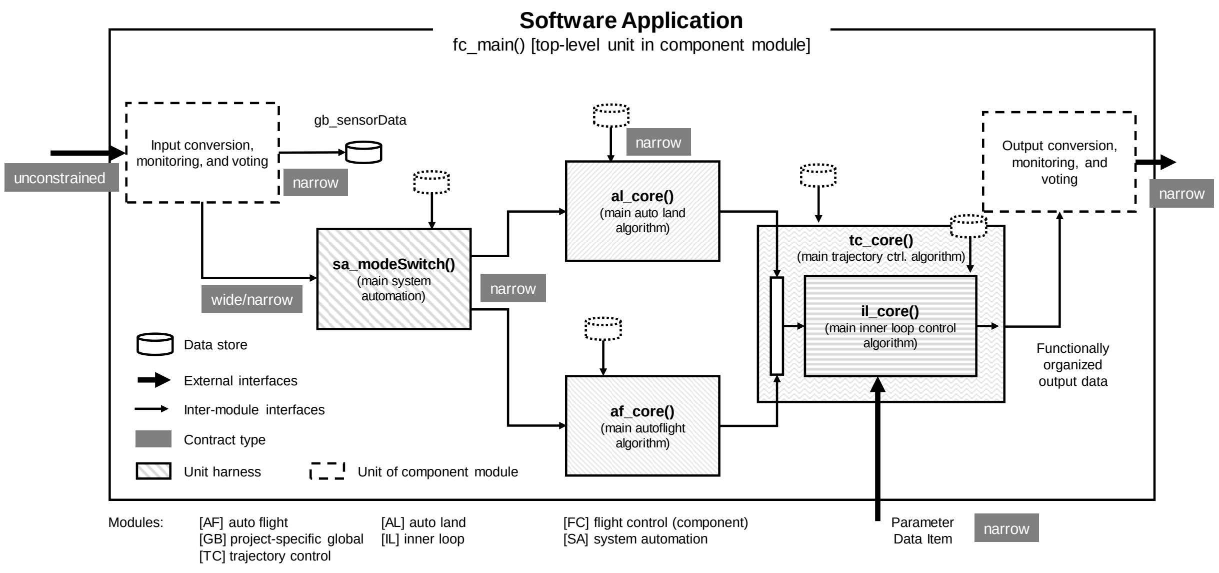

Software Application
fc_main() [top-level unit in component module]
unconstrained
Input conversion, monitoring, and voting
gb_sensorData
narrow
wide/narrow
sa_modeSwitch() (main system automation)
narrow
al_core() (main auto land algorithm)
narrow
af_core() (main autoflight algorithm)
tc_core() (main trajectory ctrl. algorithm)
il_core() (main inner loop control algorithm)
Output conversion, monitoring, and voting
narrow
Functionally organized output data
Parameter Data Item
narrow
Data store
External interfaces
Inter-module interfaces
Contract type
Unit harness
Unit of component module
Modules: [AF] auto flight [GB] project-specific global [TC] trajectory control [AL] auto land [IL] inner loop [FC] flight control (component) [SA] system automation

Figure 43: Example module architecture

The presented approach is a workaround, which tries to cope with the missing architecture design features of SL/SF in the used release. It works best for flat hierarchies, if units are not nested and requires that a limited number of public units exists in each module.

5.6.3.6 *Model data*

Model data in Simulink can be broken down into two main classes:

- Simple *MATLAB variables*
- Complex *SL Data Objects*, i.e., Simulink.Parameter, Simulink.Bus, Simulink.Signal, … [124, pp. 59-53ff.]

MATLAB variables are of basic MATLAB data types like double, uint8, or structure and can be scalar, arrays, or matrices. They are independent of SL, i.e., they can be referenced in SL, but cannot deeply interact with SL. In contrary, *SL data objects* are variables, which can (re-)define data types, aliases, or parameters. They have pervasive impact on simulation or code generation behavior of SL and Embedded Coder.

MR 10 - Model data

A. All data, which is stored outside SL models (in so-called *workspaces*), shall be considered as model data.

B. Model data is equal to module data (DR 7).

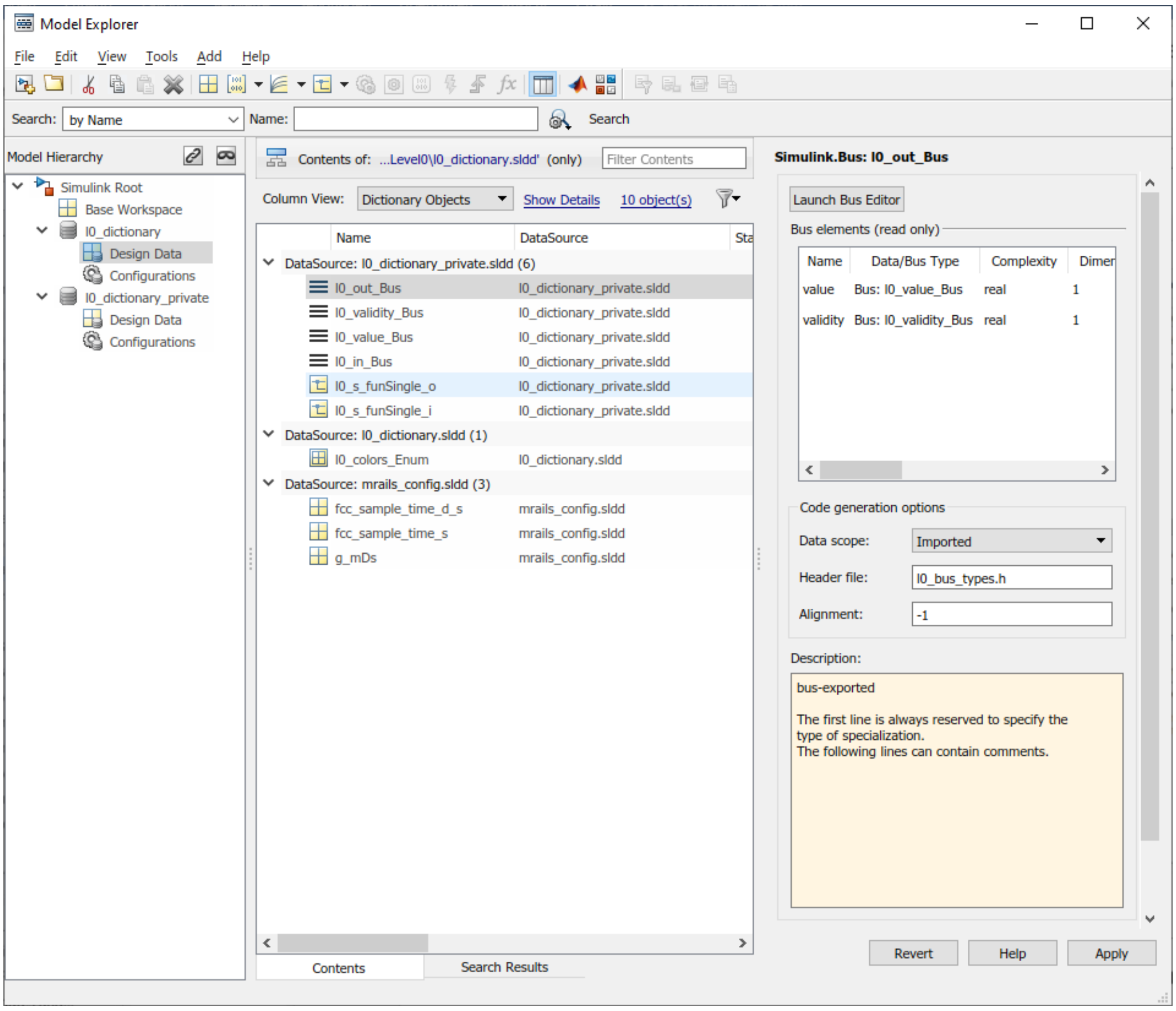
All model data referenced in SL/SF diagrams must be stored apart from the model in a *workspace*, otherwise we do not consider it as model data. Scalar values explicitly inserted in a block dialog are not considered as model data.

To store model data, SL provides three workspaces: the *base workspace*, the *model workspace*, and *SL data dictionaries* [124, pp. 63.2ff.].

The *base workspace* is volatile. Data must be written into the workspace at project startup. It is a global storage, which can be accessed, overwritten, and cleared from everywhere within MATLAB. The base workspace has the weakest usability features (e.g., sorting, filtering, searching,…).

The *model workspace* is persistent and either stores data in a separate file or directly in the model. The scope of the data is limited to the model. To avoid identifier clashes during code generation, the model workspace does not support data with global storage class including Simulink.Bus objects in R2017b.

Data dictionaries store data persistently in a binary file. Data dictionaries can reference each other. In R2017b, duplicate entries (same name) caused by referenced data dictionaries throw an error. In contrast to all other workspaces, data dictionaries support the insertion of traceability links to requirements for the model data. Coming along with various features to work with the data content, data dictionaries provide the highest usability. Usage of base workspace and data dictionary is exclusive in R2017b, a model can either link to the shared base workspace or to a data dictionary. Figure 44 shows the screenshot of a SL data dictionary.



Model Hierarchy:
- Simulink Root
  - Base Workspace
  - I0_dictionary
    - Design Data
    - Configurations
  - I0_dictionary_private
    - Design Data
    - Configurations

Contents of: ...Level0\I0_dictionary.sldd' (only)

Column View: Dictionary Objects — Show Details — 10 object(s)

Name	DataSource
DataSource: I0_dictionary_private.sldd (6)	
I0_out_Bus	I0_dictionary_private.sldd
I0_validity_Bus	I0_dictionary_private.sldd
I0_value_Bus	I0_dictionary_private.sldd
I0_in_Bus	I0_dictionary_private.sldd
I0_s_funSingle_o	I0_dictionary_private.sldd
I0_s_funSingle_i	I0_dictionary_private.sldd
DataSource: I0_dictionary.sldd (1)	
I0_colors_Enum	I0_dictionary.sldd
DataSource: mrails_config.sldd (3)	
fcc_sample_time_d_s	mrails_config.sldd
fcc_sample_time_s	mrails_config.sldd
g_mDs	mrails_config.sldd

Simulink.Bus: I0_out_Bus

Bus elements (read only)

Name	Data/Bus Type	Complexity	Dimen
value	Bus: I0_value_Bus	real	1
validity	Bus: I0_validity_Bus	real	1

Code generation options

Data scope: Imported

Header file: I0_bus_types.h

Alignment: -1

Description:

bus-exported

The first line is always reserved to specify the type of specialization.
The following lines can contain comments.

Figure 44: SL data dictionary with model data

MR 11 - Workspace usage

A. The concept of Figure 45 shall be followed to decide on the used work-space.

(1) All data, which is not used by the Design Model, shall be exclusively initialized into the base workspace (e.g., signal logging, test input and output data). This data is not considered as model data.

(2) Data, which is referenced by different models or across modules shall be stored in the data dictionary.

(3) If the data is only used in a single model, and if it is scalar, numeric and no tunability is required, it can be stored in the model workspace.

B. Names of model data in the model workspace shall be prefixed with `m_`. Data in the data dictionary shall be prefixed with the module ID.

Figure 45 outlines the proposed strategy for workspace usage in a flow diagram.

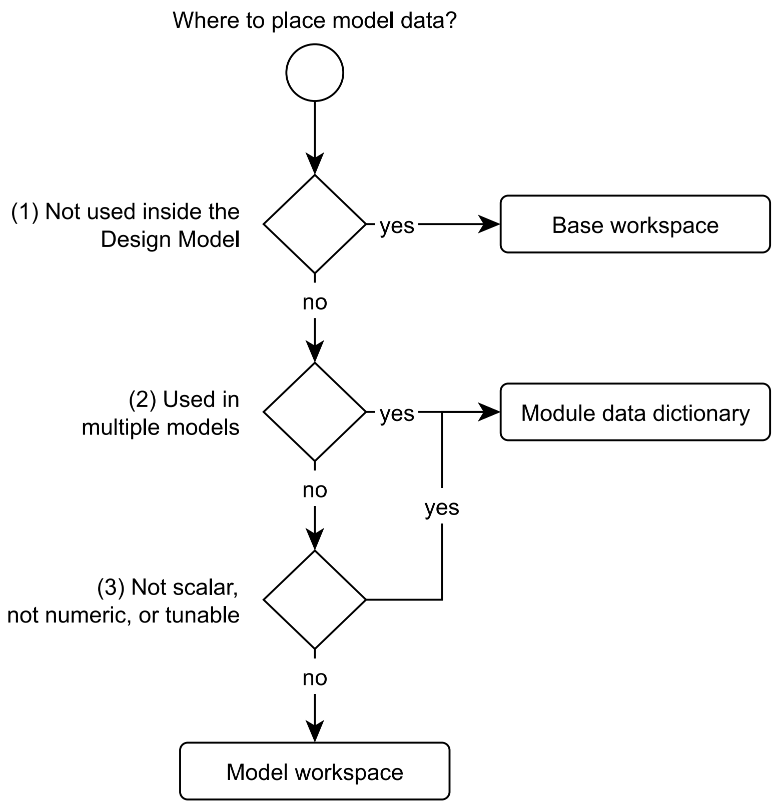

Where to place model data?
(1) Not used inside the Design Model
yes
Base workspace
no
(2) Used in multiple models
yes
Module data dictionary
no
yes
(3) Not scalar, not numeric, or tunable
no
Model workspace

Figure 45: Workspace usage

The concept extends the suggestions of MathWorks [124, pp. 59–96ff.]. Data referenced by a Design Model shall only be placed in data dictionaries and model workspaces (1). Especially in case of modularization, scoping and overwrite protection of variables is of importance, which is not supported by the base workspace. From experience, errors can occur, which are difficult to detect, if independently developed modules initialize the variables and overwrite each other in the base workspace.

If data is used in different models or across modules, it must be placed in a data dictionary (2). If it is only locally used, the developer can place it in the model workspace, presumed it fulfills the requirements. In order to control parameter and signal coding properly, as discussed in section 5.6.4.7, and since the model workspace does not support global data, it was a design decision to restrict the variables to numeric, scalar values.

To distinguish local model workspace from global data in the data dictionary easily, names of model workspace data shall be prefixed by `m_` (B).

Finally, it shall be mentioned that data dictionaries have two major shortcomings. At first, they store persistent data in a binary file. From the viewpoint of a version control system, these files are black boxes. The tools cannot display the actual changes, nor can the user identify or merge changes retrospectively in the source control tool without MATLAB. No plugin for source control systems easing this pain is known to the author.

Second, data is sometimes more than a scalar value. For example, large matrices are hard to enter in the data dictionary and calculations during initialization are not possible. For parameters, it makes sense to have scripts for initialization, which update the respective data dictionary entries at initialization. However, this changes a design artifact during each initialization, which is a highly undesired behavior. A good solution could not be found yet.

Figure 45 anticipated the term *module data dictionary*, which is further elaborated in the following. In general, each SL model can link to a single SL data dictionary. Data dictionaries itself can reference an unlimited number of other data dictionaries.

MR 12 - Module data dictionary and encapsulation

A. Each module shall contain two SL data dictionaries, the *private* and *public module data dictionary.* Model data placed in the private dictionary shall be usable in the module of the data dictionary, only. Model data in the public dictionary shall be accessible/usable in other modules, too.

B. All SL models of the module have to link to the public dictionary, which references the private dictionary.

C. The private dictionary shall reference a global *configuration data dictionary*, which is provided with the modeling environment and contains, for example, configuration settings.

D. The public dictionary shall reference the private dictionary.

E. The public dictionary may reference public dictionaries of other modules, on which it depends.

The implementation in R2017b as described is shown in Figure 46. This is not the desired referencing, since module data cannot be truly hidden this way and an analysis is necessary to identify inappropriate access. The public module dictionary sees all content of the dependent public module dictionary, which itself sees all content of the dependent private module dictionary. So also the module dictionary sees private module data of the dependent module. Unfortunately, this is the only way to implement it in R2017b.

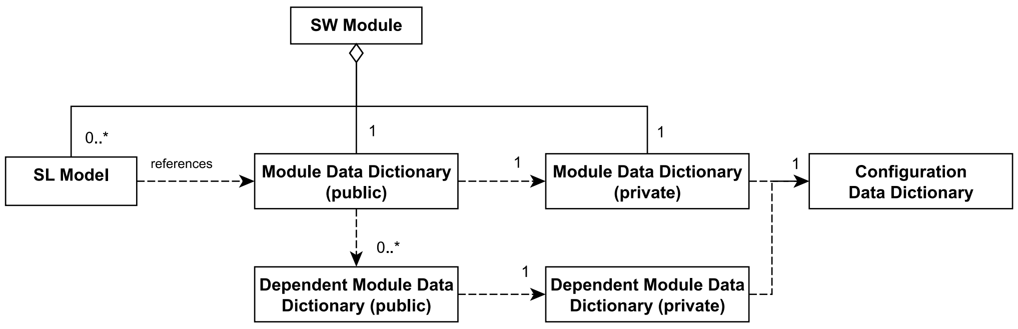

SW Module
0..*
1
1
SL Model
references
Module Data Dictionary (public)
1
Module Data Dictionary (private)
1
Configuration Data Dictionary
0..*
1
Dependent Module Data Dictionary (public)
Dependent Module Data Dictionary (private)

Figure 46: Implemented SL data dictionary dependencies

The desired referencing of data dictionaries is illustrated in Figure 47. In this case, the SL model would only have access to the private module data dictionary of its own module and to all public data dictionaries. However, this was not implementable in R2017b, since model data, which is not visible for the user, is also not accessible for SL model simulation. So the simulation will not run, if not all data is visible to all models.

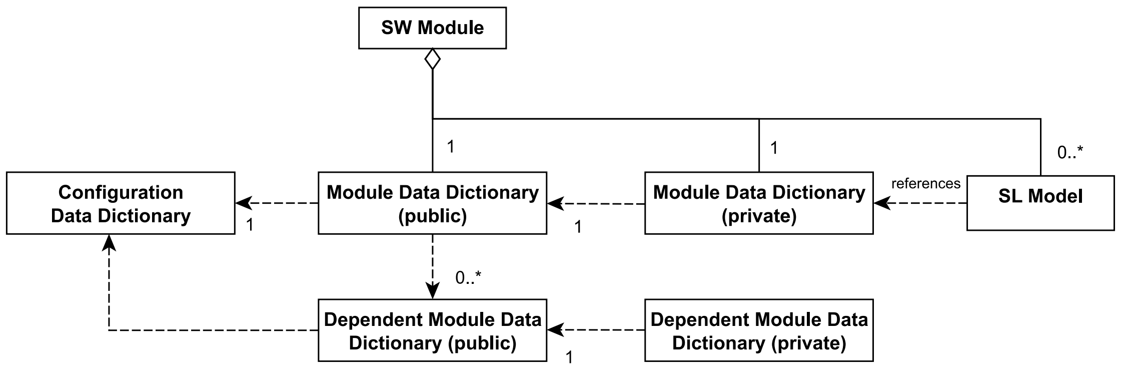

SW Module
1
1
0..*
Configuration Data Dictionary
1
Module Data Dictionary (public)
1
Module Data Dictionary (private)
references
SL Model
0..*
Dependent Module Data Dictionary (public)
1
Dependent Module Data Dictionary (private)

Figure 47: Desired SL data dictionary dependencies

Every SL model requires an active configuration set, which defines simulation and code generation behavior. Technically, configuration settings can either be stored in the model (default), or externally.

MR 13 - Configuration settings and configuration data dictionary

A. Configuration settings shall be stored externally to SL models in the *configuration data dictionary*.

B. Each SL model shall only contain a single (active) configuration reference to the correct configuration setting.

If configuration settings were stored in the model, every developer would be able to locally change the settings for each model independently. This might lead to incompatibilities and safety issues. The recommend approach is thus to use *configuration references*, which, literally spoken, place a link to an externally stored, centrally managed configuration set [124, pp. 13-29ff.]. The best solution is to store these configuration settings in a separate configuration data dictionary and provide it as read-only part of the modeling environment.

5.6.4 Detailed design

Contribution 9: A safe modeling subset for SL/SF has been assembled. It is a set of well-defined and justified rules limiting SL/SF features and adding conventions. They cumulate best practices collected by the author in the various accompanied projects. In contrast to many other existing guideline sets, which blacklist prohibited features, a whitelisting approach is followed. Only features from a permitted subset shall be used. This safe modeling subset targets compatibility of the process, tools, and tasks. It is usable out-of-the-box and significantly lowers the adoption risk, with which large and small companies struggle. Incremental try-and-error to reach a compliant guideline is significantly reduced. The rules are also an important pillar for generation of modular code.

The previous sections demonstrated the general organization of the Design Model in SL/SF and discussed high-level architectural guidelines. The content of SL models or model data has not been introduced.

For the detailed design, a safe subset of SL/SF features, which define all *model elements* "from which the model is constructed" (DO-331 p. 82), has been defined. Figure 48 illustrates a structural breakdown of the complete safe subset of model elements chosen for SL/SF. Only such a drastically constrained subset supports robustness of the process and compliance with all tasks to be done for development and verification.

Most general groups are the diagrams itself, the graphical primitives, from which they are composed of, as well as the model data. These three groups are further broken down into selected SL and SF elements or features. From all of the features in SL/SF, only a few have been selected as shown in Figure 48. For the third level, the selected features are further constrained concerning their usage, settings, or parameterization. This finally leads to so-called safe specializations. They are highlighted in the following with DIFFERENT FORMATTING.

Graphical model primitives require some additional explanation. They have been divided into *atomic primitives* and *container primitives*. In contrast to container primitives, atomic primitives cannot be further decomposed into further atomic or container primitives. Safe Specializations for all graphical model primitives are given in the *DO-331 Foundation Library*, which is a collection of atomic blocks.

In the following, the safe specializations will be presented.

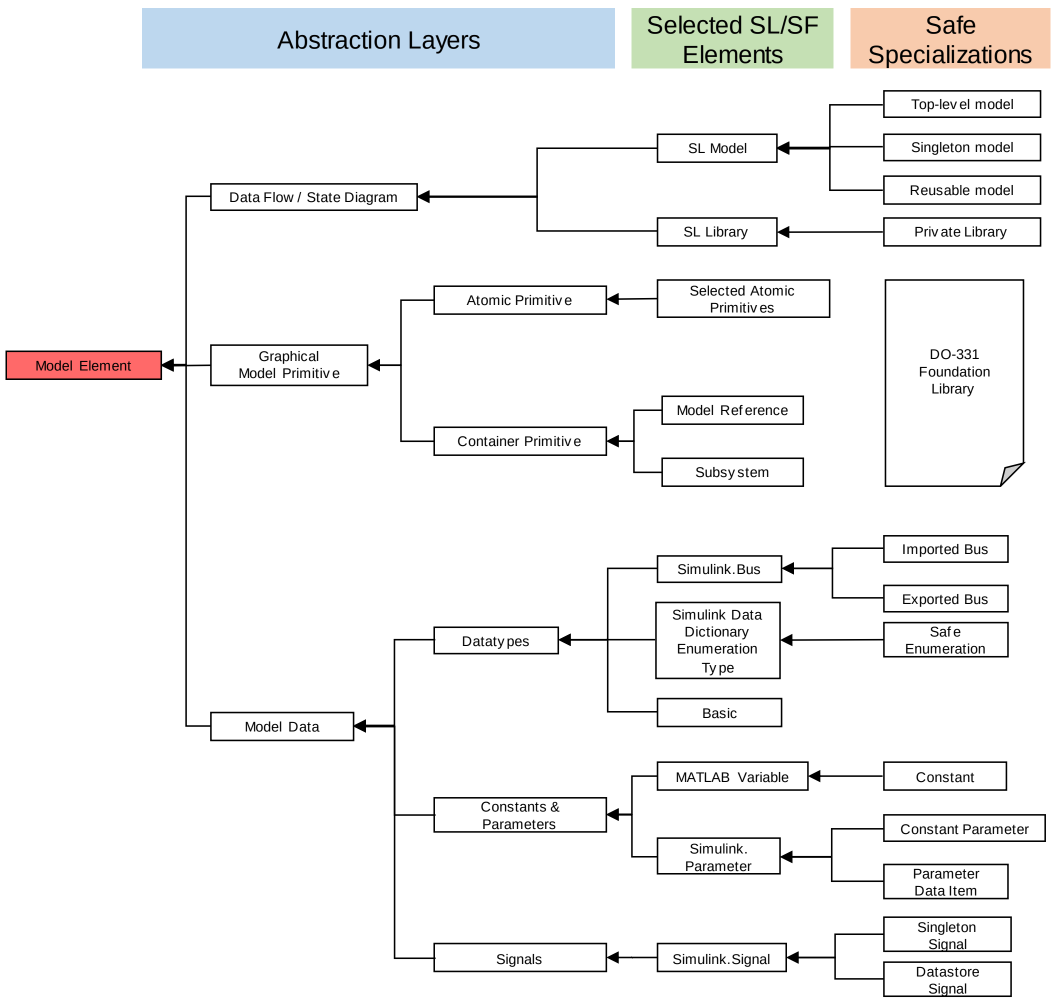


Figure 48: Model element hierarchy overview

5.6.4.1 *SL models*

Some deeper considerations for SL models are necessary. Theoretically, models within the same call hierarchy may also be executed with different sample times. However, the following restrictions apply.

MR 14 - Software application model hierarchy and execution rate

A. In the whole software application, there shall be a single model (call) hierarchy with a single root model.

B. All models within the hierarchy shall be executed with a single execution rate and thus be simulated with the same *application sample time*.

Multi-rate execution immediately raises verification questions and tool compatibility issues (e.g., with SLCI). It also complicates worst-case-execution-time (WCET) analysis. Thus, this feature has been excluded by (B).

Each SL model must be linked to exactly one configuration set, which defines the simulation and coding behavior of the model. For units, different SL model types are distinguished by different configuration settings.

MR 15 - SL model types

A. Separate configurations settings for three different model types shall be used.

- TOP-LEVEL MODEL
- SINGLETON MODEL
- REUSABLE MODEL

B. A TOP-LEVEL MODEL always represents the root model of the software application model hierarchy. It shall not be included somewhere else in the hierarchy or in other hierarchies. Only one top-level model can exist in the component module.

C. A SINGLETON MODEL shall be referenced once in a model hierarchy. Any duplicate use shall cause a simulation and coding error.

D. A REUSABLE MODEL can be referenced multiple times in different contexts.

Having three SL model types has been proved as best practice. They differ in the number of allowed instances, intended use and code interfaces as listed in Table 23. The referenced parameters are documented in the Simulink and Embedded Coder Reference [130, 131].

	Top-level model	Singleton model	Reusable model
Indented use	Root-level model	Uniquely used functionality	Library / utility functions
Number of instances in model hierarchy (`ModelReference NumInstancesAllowed`)	`One`[16]	One	Multiple
Nested model types	Singleton / reusable	Singleton / reusable	Reusable
Periodic sample time constraint (`SampleTimeConstraint`)	Unconstrained	Unconstrained	Ensure sample time independent
Fundamental sample time (`FixedStep`)	Application sample time	Application sample time	n/a
Code function interface	initialize step	<model-name>_initialize <model-name>_Init <model-name>_Reset <model-name>_Disable	<model-name>_initialize <model-name>_Init <model-name>_Reset <model-name>_Disable
Code data interface (`CodeInterfacePackaging`)	Nonreusable function	Nonreusable function	Reusable function
Pass root-level I/O as (`RootIOFormat`)	Structure reference	Individual arguments	Individual arguments
Remove Reset Function (`RemoveResetFunc`)	Yes	No	No
Remove Disable Function (`RemoveDisableFunc`)	Yes	No	No
Terminate function required (`IncludeMdlTerminateFcn`)	No	No	No
Support of Inf and NaN in code generation (`SupportNonFinite`)	Yes	No	No
Inf / NaN Diagnostics (`SignalInfNanChecking`)	None	Error	Error
Data store – Detect read before write (`ReadBeforeWriteMsg`)	Ensure all error	Disable all	Disable all

Table 23: Model type properties

[16] The setting also allows `Zero` instances, which seems reasonable for the root model at first glance, but also prohibits embedding the model into test harnesses or other models for simulation testing purposes.

REUSABLE MODELS have various limitations concerning simulation [124, pp. 8–105ff.] and coding [96, p. 4-28]. The following list summarizes the most important ones:

- Hierarchical restriction: SINGLETON MODELS cannot be nested in REUSABLE MODELS, since it would on the one hand undermine their uniqueness, and on the other hand not be compatible with exchangeable workspace structures on code level. In other words, REUSABLE MODELS can only reference other REUSABLE MODELS. For example, “Reusable Model D” cannot contain “Singleton Model E” in Figure 49.

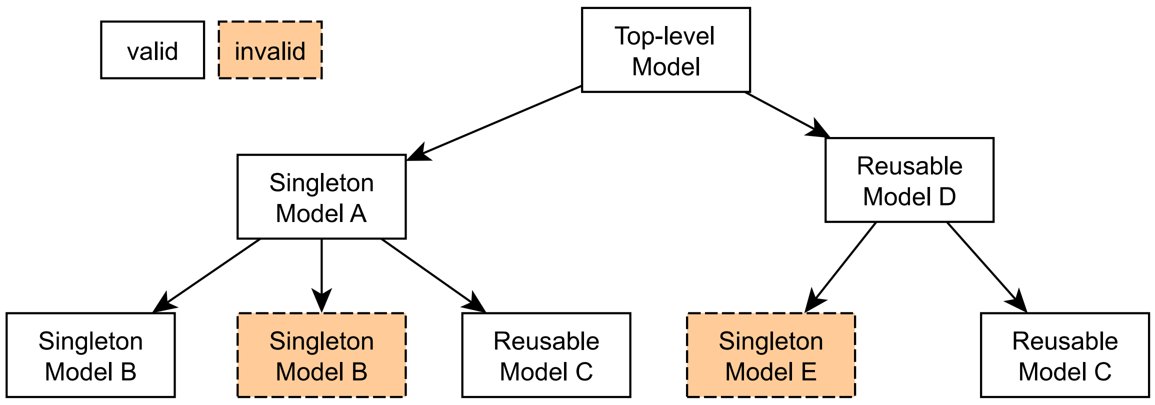


Figure 49: Model Hierarchy

- Iterative Subsystems blocks: Only REUSABLE MODELS can be nested in iterative subsystems.
- Signals usage: TOP-LEVEL MODELS and SINGLETON MODELS have different requirements concerning the storage class of signals compared to REUSABLE MODELS (cf. MR 37).
- In SF, reusable models impose restrictions, especially for graphical functions.

In order to embed REUSABLE MODELS into iterative subsystems (for-iterator Subsystem block), they are not allowed to contain sample times other than Inf or -1. Since SL models with unconstrained sample time value `Auto` cannot be coded without disabling important diagnostics, the sample time constraint has been set to `Ensure sample time independent`, which is on the other side a non-fatal code inspector compatibility (cf. handling of SLCI incompatibilities in section 8.2.10).

The function and data code interfaces for the TOP-LEVEL MODEL are mainly driven by the interface requirements (cf. DR 12 and DR 13). By setting the interface mode to “Structure Reference”, Embedded Coder generates a `_U` structure containing fields for all Inport blocks and a _Y structure containing fields for all Outport blocks. Furthermore, a `void-void` call interface is generated for the TOP-LEVEL MODEL. For nested models, passing variables as individual arguments improves performance, since the values do not have to be copied into a single structure before being passed.

Conditionally executed models

A special topic are models in conditionally executed contexts, this means for example in atomic subsystems, which are only executed, if certain conditions hold [124, pp. 10-3ff.]. If models with states are placed in a conditionally executed context, they require a reset and disable function in C code. For example, they may be configured to reset all states when activated. To generate a reset function, the general model configuration setting `RemoveResetFunc` must be disabled. The reset function is then always generated, if the model contains states, independently of whether it is called or not.

If a SL model is never used in a conditionally executed context, the reset function is Dead Code. All uncalled (and finally noncovered) functions in the Executable Object Code are Dead Code according to DO-178C, if well-defined exceptions do not apply (cf. section CR 14).

One way to resolve this issue is the introduction of additional configuration sets, e.g., a conditional singleton and reusable model. This increases the number of model types to five. However, the problem are models, which are in partially usable library modules. They are independently developed, so it is not possible to predict, whether they are used in a conditional context or not later on. Consequently, different variants would be needed, leading to significant overhead.

An alternative way is to ensure that uncalled functions are safely removed by the compiler, so that the exception for safely removed code applies (cf. section CR 16). This approach has been chosen here. Ensuring safe removal is part of the Integration Process not covered by this thesis.

Support of special quantities

Most modeling guidelines prohibit support of e.g., Inf and NaN floating-point numbers throughout all models. The support can be globally disabled in the configuration and SL throws an error as soon as those quantities occur during simulation. However, the top-level model has to cope with unconstrained inputs (cf. DR 11) and must allow robustness simulation testing and generating code, which checks for special quantities. Thus the settings of the top-level model have been chosen to support special quantities. SLCI flags these settings as non-fatal incompatibilities, but as long as no specific blocks for special quantities are used, no verification problems could be observed.

MR 16 - Model simulation mode

Models and model references shall be saved with the Normal mode selected.

SL supports different simulation modes to execute the model. The respective compilation is subsequent to the model compilation in MR 3.

The different simulation modes use different methods to build an executable model. *Normal*, *Accelerator*, and *Rapid* Accelerator modes use in-memory representations for simulation or generate optimized code and compile it. *SIL* (“software-in-the-loop”) and *PIL* (“processor-in-the-loop”) modes actually generate the final Source Code and compile this code for simulation on either the host or the target computer. [124, pp. 8-37f.]

Only Normal mode is considered as simulation of the Design Model. In all other modes, the compiled model passes a code generation process afterwards (also in Accelerator mode, separate executable applications are generated [124, pp. 8-54ff.]). Furthermore, only the Normal mode supports all runtime diagnostics [124, p. 8-39] and the Accelerator mode cannot be used for model coverage recording [101, p. 1-2].

Setting the Normal mode as default also has a practical reason. Accelerated execution can be forced for a model hierarchy with models in Normal mode by setting the simulation mode of the root model to Accelerator mode, but nested models that are in Accelerator mode can never be forced to run in Normal mode from the root model [96, pp. 64-37f.].

MR 17 - SL data types

A. Only the basic SL/SF data types and their mapping to the data types supported by the compiler and target hardware according to Table 24 shall be used.

Simulink Data Type	Typedef in Source Code	Mapping to CompCert Compiler Data Types as defined Section 6.5.4
Boolean	boolean_T	unsigned char
uint8	uint8_T	unsigned char
uint16	uint16_T	unsigned short
uint32	uint32_T	unsigned int
int8	int8_T	signed char
int16	int16_T	short
int32	int32_T	int
single (IEEE 754 single (32 total bits, 8 exponent bits))	real32_T	float
Double (IEEE 754 double (64 total bits, 11 exponent bits))	real64_T	double

Table 24: Data type mapping between Simulink and CompCert language set

B. Simulink.Bus data types shall be supported.

The data types supported by the compiler and target hardware are mapped to built-in SL data types with identical bit and byte representation as shown in Table 24. In terms of tool compatibility, only the built-in SL data types for integers and floating-point arithmetic are used and custom data and alias types [127, pp. 5-157ff.] are avoided. Simulink floating-point data types are used as defined in [127, p. 2-283]. Built-in fixed-point data types and complex numbers are convenient but not supported by SLCI in R2017b [121, p. 3-6].

Simulink.Bus data types [127, pp. 5-207ff.] combine multiple basic data types into a single, hierarchical structure. Section 5.6.4.7 provides a detailed discussion.

MR 18 - Special floating-point quantities

> Introduction of special floating-point quantities aside from unconstrained interfaces shall be avoided.

Special floating-point quantities, like NaN or Inf, are part of the floating-point specification, but require additional attention. The behavior in C is often unspecified, undefined, or implementation-defined. The values most often indicate a calculation error and behave differently than ordinary values. They do not represent mathematically treatable values anymore and in many cases lead to unexpected behavior or malfunction of the application software.

The first source of special floating-point quantities to be considered are constants used in SL/SF models. For example, automatically calculated, multi-dimensional gain tables used in lookup table functions. This case is covered in simulation and code generation. During Normal mode simulation, the diagnostic `SignalInfNanChecking` throws an error, if the lookup table block (or any other block) outputs a NaN. However, if the related breakpoints never hit the field with the special quantity, it remains undiscovered. During code generation, Embedded Coder throws an error if the configuration setting `SupportNonFinite` is disabled. In the special case with the lookup table blocks, a non-finite parameter is also a SLCI incompatibility flagged by respective checks.

The second source for special floating-point quantities are component interfaces. Component interface handling is discussed in DR 14 and MR 17. The third source are arithmetic run-time errors (cf. MR 42).

5.6.4.2 *SL container primitive usage*

Container primitives are core components of the SL/SF modeling language with various configuration settings. They are used to structure models into layers and influence simulation behavior or generated code. Choosing the wrong types of container primitives can cause significant difficulties for configuration management, verifiability, compatibility with verification tools, and maintainability of the Design Model. In the SL documentation [124, pp. 15-31ff.], some aspects of the different modeling approaches are discussed, but not from process-related viewpoints.

Therefore, a detailed comparison between container primitives in the scope of MBSwD has been made. It highlights the pros and cons of each approach and gives a final recommendation. The criteria are a valuable reference for process planners.

An overview of all container types is depicted in Figure 50. Main group of SL container primitives are model references and subsystems. *Model references* are graphical model elements, which can embed other SL models. Counterpart of model references are *subsystems*. They allow layering within a model.

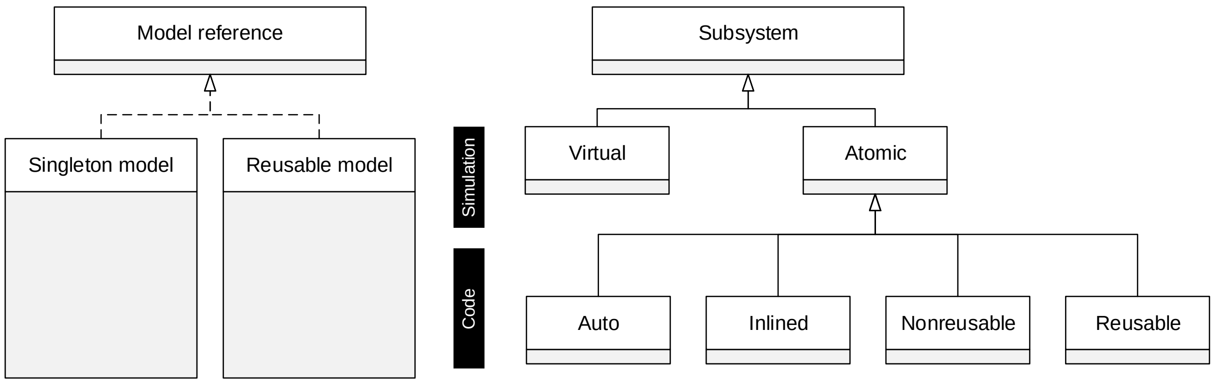


Figure 50: Types of container primitives

Settings of container primitives determine the behavior during simulation and code generation. Model reference behavior is defined by the type of the linked SL model, whereas subsystem behavior can be controlled in the properties of each Subsystem block or is inherited from the calling context.

Virtual subsystems are for visual support only. Execution order optimization of blocks and various other simulation optimizations are performed over subsystem boundaries. In contrast, blocks inside an *atomic* subsystem are treated as a unit, they are executed together and the user has the possibility to influence, when and if they are executed.

One goal of container primitives is encapsulating content and reuse it at different places (reuse for modeling purpose). Model reference blocks call other models. They can be called multiple times, if they have a configuration, which allows this. The behavior and execution of the linked model is independent of the calling context. Models have a defined interface specification (some values can be propagated, but must always depend on the content of the model itself). Every model links to a fix configuration set (cf. MR 13). Configuration parameters are not inherited from calling contexts.

The mechanism of a subsystems is different. Subsystems can be reused only, if they are placed in a SL library (cf. Figure 52). SL libraries are SL models, which are not executable and do not have an attached configuration set. In the calling SL models, a *subsystem library link* can be added wherever needed. In a non-compiled model, a linked subsystem just synchronizes with the subsystem in the SL library. A change in the SL library propagates to all linked subsystems, i.e., they can be centrally changed. But when compiling the model, a new, independent instance is created for each linked subsystem. The instances of linked subsystems inherit the configuration from the model they are embedded in. If not fully defined, also the interface may be inherited from the context. The behavior of the subsystem thus always depends on the context and can never be verified sufficiently apart from it. In other words, during modeling, linked subsystems have a common source. During compilation, they become separate instances. Subsystems in SL libraries should be considered as underspecified design patterns, like templates in a C++ programming language.

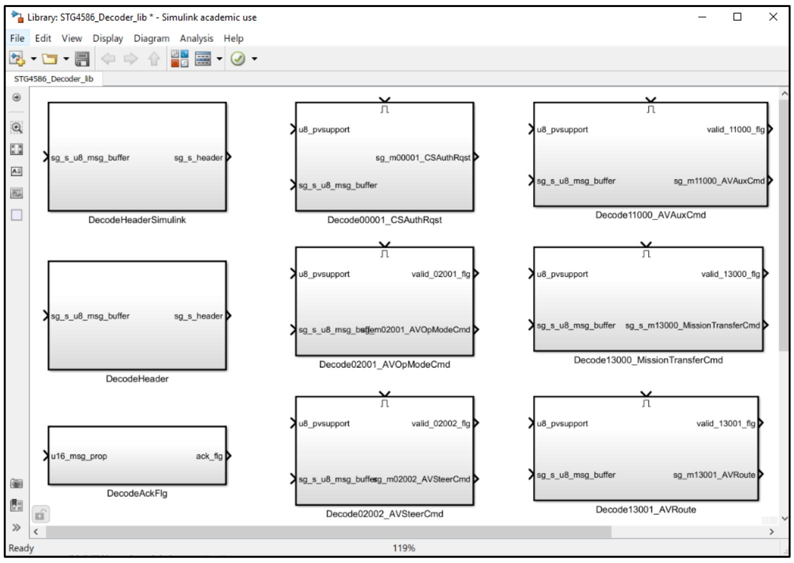


Figure 51: Example SL library

The container primitives also have influence on the reusability on code level, i.e., that C functions are generated, which can be called multiple times and from multiple contexts. Code of model references always contains independent functions. Whether they are callable multiple times is equal to the behavior on model-level and depends on the SL model type.

For subsystems, there is no correlation between reusability in the model and reusability of the code. A subsystem can be reused through a SL library on model-level, but may generate separate code each time. At the same time, depending on the setting, Embedded Coder may generate a common C function for subsystems, which are not linked libraries. Since this can only happen, if Embedded Coder considers two subsystems as equivalent, it is good practice to keep those subsystems in SL libraries as well.

Virtual subsystem code can never be encapsulated in functions on code level (due to the blurry boundary and optimizations). Atomic subsystems are always kept together in code, either as cohesive, inlined code section, or as separate function. The function packaging setting controls code placement and is independent of whether the subsystem is in a SL library (and thus reusable on model-level) or not. The following options are available as explained in [127, pp. 1-1538ff.]:

- Atomic subsystem with function packaging setting `Inline`. Embedded Coder integrates all code into the code of the higher context, but keeps it in cohesive code lines.
- Atomic subsystem with function packaging setting `Nonreusable Function`. Embedded Coder generates a separate function for each subsystem.
- Atomic subsystem with function packaging setting `Reusable Function`. Embedded Coder uses the structural checksum to identify subsystems of the same functionality and unites them to a single function in C.

In principle, there are two extreme cases for container usage. The whole hierarchy is structured with models only and subsystems are not used, or a single SL model exists that contains a single large diagram structured with subsystems only.

Both subsystems and model references have their advantages and disadvantages. A qualitative evaluation of various criteria has been performed for each container primitive. The results are displayed in Table 25. (●) indicates that the aspect is well supported, (○) that it can be partially fulfilled, and ( ) means that the requirement cannot be fulfilled at all. The rating is discussed in detail below.

Requirements	**Model Reference**		**Subsystem**			
	Singleton	**Reusable**	**Virtual**	**Inline**	**Nreusable**	**Reusable**
SLCI support	●	●	●	●	○	
Hiding of details of the design and introduce layers into the model	●	●	●	●	●	●
Usability (less effort to create new containers and manage them)	○	○	●	●	●	●
Simulatability (unconstrained simulation)	○	○	●	○	○	○
Simulation efficiency	○	○	●	●	●	●
Debugging capabilities	●	○	●	●	●	●
Separate review and analysis of design	●	●		○	○	○
Reusability/re-entrance on model-level		●	●	●	●	●
Independent simulation testing	●	●	○	○	○	○
Model coverage accumulation of different instances	n/a	●				
Code componentization	●	●			●	●
Independent code generation	●	●				
Code efficiency	○	○	●	○	○	○
Reusability/re-entrance on code level		●				○
Separate review and analysis of code	●	●				
Independent SIL testing	●	●				○
Independent PIL testing (with custom PIL tool)	●	●				

Table 25: Qualitative usability analysis of container primitives in R2017b

SLCI Support

SLCI supports all types of models. For atomic subsystems, only the `Inline` and `Non-Reusable Function` settings can be verified. Latter is restricted to a narrow configuration requiring a manually named function with a `void_void` interface. The function is always coded into main model source file (file name must be `Auto`). [121, p. 3-55]

Although `Reusable Function` code packaging of subsystems is not supported by SLCI in R2017b, it has been added to the comparison, since support has been added in R2018a[17].

Usability

The use of SL models is less convenient. For each SL model, a separate file has to be created, a configuration set as well as a data dictionary must be linked. Subsystems are created by dragging them from a library into the model. In R2017b, SL models are always separate windows, which may become confusing.

Simulatability (unconstrained simulation)

All containers except virtual subsystems impose requirements on the design.

If an optimization over the subsystem boundaries is beneficial, virtual subsystems are preferable. This is the case, if algebraic loops [124, pp. 3-37ff.] shall be resolved automatically or the purpose of the system is mainly signal routing.

Atomic subsystems are required for conditional and iterative execution (e.g., enabled subsystems).

Models have stronger constraints on interfaces than subsystems, e.g., subsystems can inherit data types from the higher context, models must explicitly specify or derive them from their own content. Model references, especially REUSABLE MODELS, impose further restrictions as discussed in MR 17.

Simulation efficiency

From experience, subsystem-based approaches are faster than separate models concerning loading and initialization in normal simulation mode, since each model causes some overhead. However, in Accelerator mode, models may leverage incremental build capabilities (dependent on the settings).

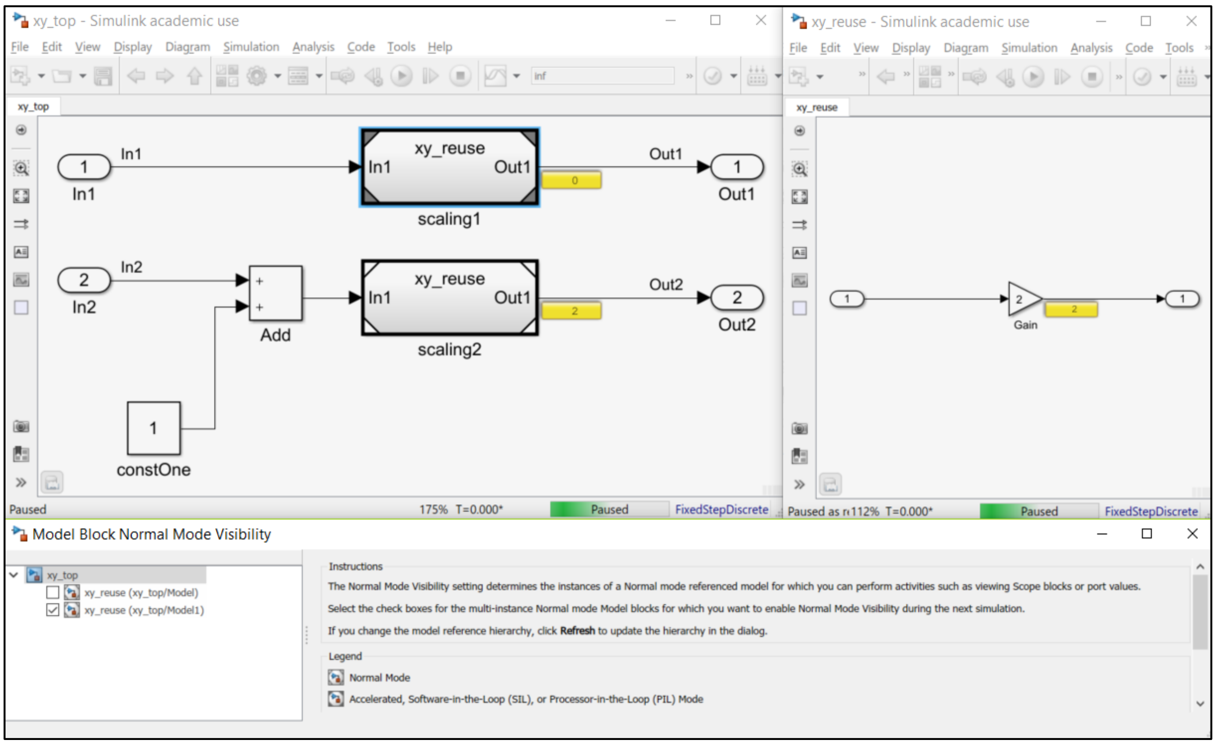
Debugging capabilities

The debugging capabilities of model references are more limited than those of subsystems. In contrast to subsystems, debugging in nested models is only possible in normal execution mode, not in Accelerator mode (R2017b).

Additionally, in R2017b, only one instance can be opened in a separate window at the same time. The visible instance must be separately and explicitly selected in the “Model Block Normal Mode Visibility” dialog. Simply opening the model does not change the visible instance and it can also not be changed during execution. If the visible instance is not correct, wrong data is displayed. In Figure 52, the second instance of model reference `xy_reuse` is selected in the dialog at the bottom and thus displayed on the right (which is not obvious from the window itself). Debugging both instances in the same run is not possible.

[17] https://de.mathworks.com/help/releases/R2018a/slci/ref/block-compatibility-alphabetical-list.html#bs6de_k-1 [Accessed on: Nov. 29 2019]

However, it should also be noted that other ways of debugging, like signal logging, streaming and plotting are typically more relevant for controller development and not limited to this degree.


xy_top - Simulink academic use
File Edit View Display Diagram Simulation Analysis Code Tools Help
xy_top
In1
In2
xy_reuse
scaling1
scaling2
Out1
Out2
Add
constOne
Paused
175% T=0.000*
FixedStepDiscrete
xy_reuse - Simulink academic use
Gain
Paused as r 112% T=0.000*

Model Block Normal Mode Visibility

- xy_top
  - ☐ xy_reuse (xy_top/Model)
  - ☑ xy_reuse (xy_top/Model1)

Instructions

The Normal Mode Visibility setting determines the instances of a Normal mode referenced model for which you can perform activities such as viewing Scope blocks or port values.

Select the check boxes for the multi-instance Normal mode Model blocks for which you want to enable Normal Mode Visibility during the next simulation.

If you change the model reference hierarchy, click **Refresh** to update the hierarchy in the dialog.

Legend

Normal Mode

Accelerated, Software-in-the-Loop (SIL), or Processor-in-the-Loop (PIL) Mode

Figure 52: Reusable model debugging

Separate review and analysis of design

Manual reviews can be independently performed for each container. All of the used tools support an analysis per model, and some also support the analysis of subsystems. For example, SL Model Advisor supports analysis of all types of subsystems, and SL Design Verifier supports atomic subsystems.

Anyway, a subsystem-based workflow is questionable, since a subsystem is never fully independent from the model, into which it is embedded. It for example inherits interfaces or configuration settings. In consequence, virtual subsystems do not and atomic subsystems do just partially allow calculation of checksums to identify outdated parts. The best checksums are supported by models (cf. section MR 2).

Reusability/re-entrance on model-level

Reusing parts of the model can be achieved with model references as well as with subsystems in SL libraries.

Independent simulation testing

Separate simulation testing requires (amongst others) that the tested entity has a fix interface and behavior independent of the embedded context and that the selected testing tool, here Simulink Test (cf. section 4.6.3), can access the interface.

Separate simulation testing is always possible for models, since they can be simulated independently.

All types of subsystems are embeddable in a SL test harness and can thus be tested independently in simulation, too. However, subsystems do not fulfill the requirements of *independent* simulation testing. Their interface and behavior depends on the context.

Also JMAAB advises to carefully investigate introduction of atomic subsystems merely for testability reasons ([109] p. 228).

Model coverage accumulation of different instances

If containers are reused, model coverage shall typically be accumulated over all instances, as it is the typical approach for functions when assessing structural code coverage. Depending on the calling context, different decisions may be triggered.

SL Coverage supports automatic accumulation of coverage for instances of REUSABLE MODELS, but not for linked subsystems (independent of the code function packaging). Even if a rudimentary manual workaround exists [101, pp. 8-8ff.], accumulating coverage of linked subsystems is not appreciated, since they are underspecified and depend on the context. For example, a subsystem in a model with conditional branch execution returns different coverage results as the same subsystem in a model without conditional branch execution (cf. MR 3).

In addition, if the subsystem code function packaging is not reusable, model and code coverage do not map.

Code componentization

Code componentization means the ability to split code in separate functions. Both model references and subsystems have features to achieve code componentization.

Independent code generation

Embedded Coder supports code generation from both models and subsystems. However, in R2017b, code generation from a subsystem satisfies another purpose and workflow [96, pp. 3-2ff.]. It is incompatible to code generated from the surrounding model and the code function packaging options (subsystem is considered as model) and targets a standalone executable.

Code efficiency

Most efficient code can be generated from virtual subsystems, since SL and Embedded Coder can perform optimizations across boundaries. For all other implementations, Embedded Coder must respect boundaries. Code of inlined subsystems may provide optimization potential for the compiler later on.

Reusability/re-entrance on code level

If reusability/re-entrance is given on model-level, it is favorable to structure the code similarly in order to reduce the lines of code, ease verification and coverage collection. Reusable code is generated by REUSABLE MODELS and subsystems in SL libraries with the function packaging setting `Reusable Function`.

Beside SLCI incompatibility, the latter approach has some significant shortcomings in a process context:

- Embedded Coder uses the structural checksum to identify subsystems of the same functionality and unites them to a single function in C. A linked subsystem in a SL library may have different structural checksums dependent on the caller context – no matter whether it is fully specified or underspecified (see example 1 in Appendix C). Therefore, it is almost impossible to control, which subsystems really map to a C function and how functions or files are named. To create C functions in a controlled and configuration management-friendly manner, models are the only choice.

- According to the documentation, Embedded Coder can detect identical reusable subsystems across models and generate a single function, but the naming depends on the structural checksum [96, pp. 6-54ff.]. This is difficult for configuration control and testing.

Separate review and analysis of code

Manual reviews can be independently performed for each model. The chosen analysis tools, like SLCI or Polyspace work model-centric. The subsystem verification capabilities of Polyspace require a standalone-build of the subsystem as described beforehand and are thus not relevant.

Independent testing in SIL

SL models can be independently simulated in SIL mode.

Although a workflow for subsystem SIL testing exists as well [132, pp. 4-2ff], this approach raises consistency questions in the process at hand, since code cannot be generated for subsystems independently.

Independent testing in PIL

With the custom PIL tool outlined in section 4.6.3.5, only SL model interfaces are supported for testing. At the time this thesis has been written, standalone testing support for REUSABLE MODELS was ongoing research work.

Notes on features after R2017b

After R2017b, MathWorks has improved the subsystem support. In R2019a, the so-called *library-based code generation for reusable library subsystems* has been introduced[18]. The user can attach a code configuration set to a reusable system in a library and make it independent from the context, from which it will be called. Code functions can be separately generated and will be called from the context. Still, there is no guarantee, that a code function can be reused in all contexts, but means are provided to throw an error during code generation of the context model the code cannot be reused. The feature also supports iterative code generation for different input interfaces. How compatible this workflow is, especially with verification tools, must be evaluated (e.g., model coverage accumulation, valid interface for SIL or PIL support, …). Compatibility will probably incubate in future releases, so this workflow may be a welcome addition.

In R2019b, so-called *subsystem references* have been added[19]. They allow storing the content of a subsystem in a separate file (not a library). The behavior remains as known for subsystems, this means that almost all considerations from above still apply, except that separate files ease configuration management.

MR 19 - Container primitive selection

A. Only the containers illustrated in Figure 53 shall be used.

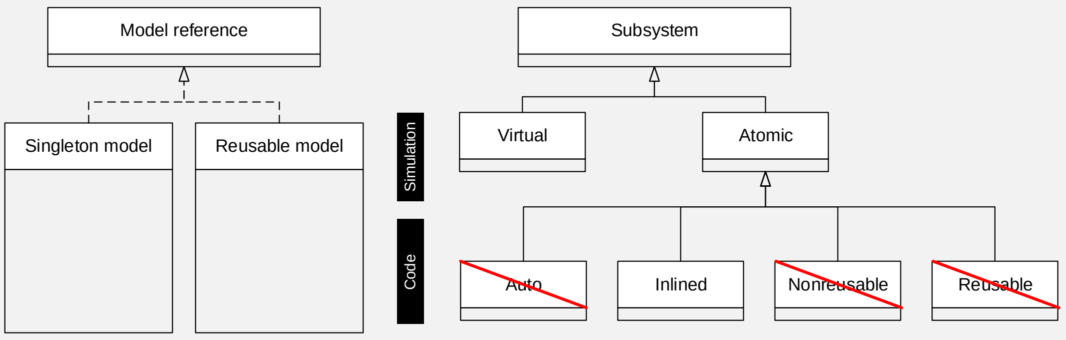


Figure 53: Usable container primitives

B. The selection of the respective container primitive should follow Figure 54.

[18] https://de.mathworks.com/help/releases/R2019a/ecoder/ug/library-based-code-generation-for-subsystems-shared-across-models.html [Accessed on: Nov. 29 2019]

[19] https://de.mathworks.com/help/releases/R2019b/simulink/ug/referenced-subsystem-1.html [Accessed on: Nov. 29 2019]

Table 25 shows clearly that none of the edge cases is a good choice. Subsystems have their strength for the development of the design, because they are easier to handle and have less restrictions. However, they have strong limitations in R2017b, when it comes to verification, code generation, and configuration management. Here, model references are the better choice. This aligns with the recommendations in [133].

In summary, the recommendation is to use a mix of model references and subsystems for componentization. Although supported by SLCI, the benefit of subsystems with a nonreusable code function interface is limited and shall, in order to reduce complexity, not be used. With the selected container set, the hard selection constraints can be boiled down to those in Figure 54.

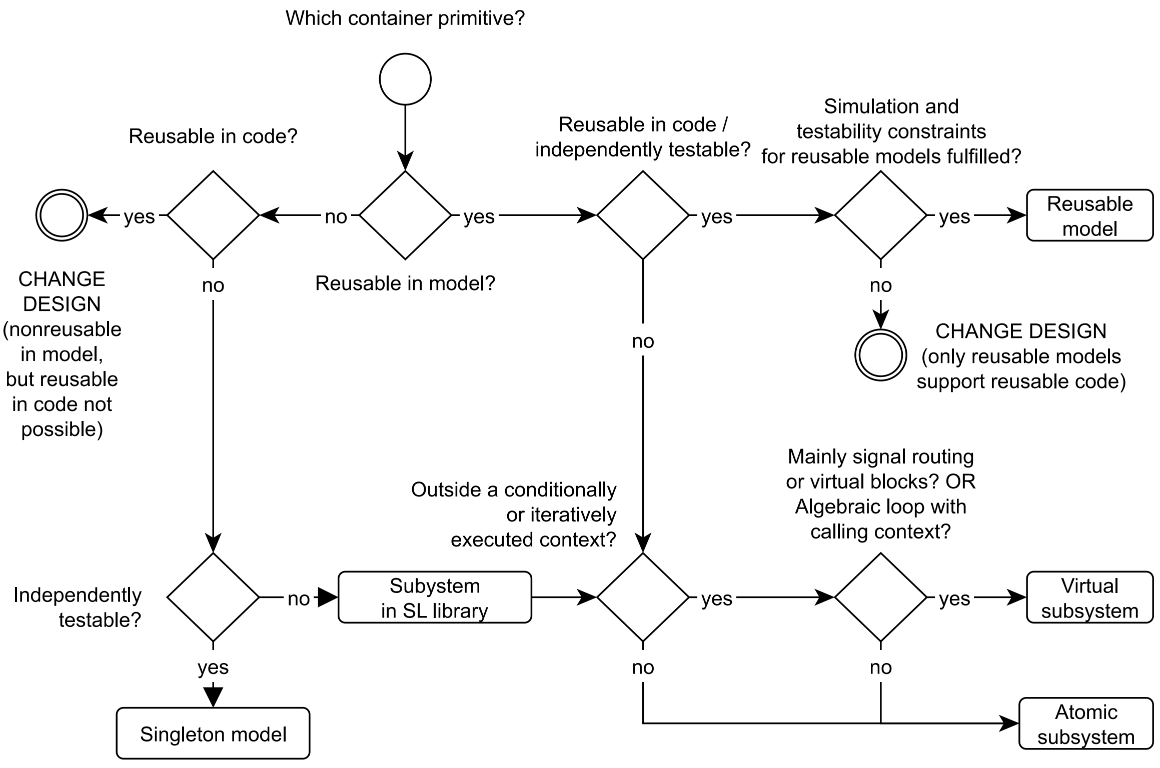

Which container primitive?
Reusable in code?
yes
no
yes
CHANGE DESIGN (nonreusable in model, but reusable in code not possible)
no
Reusable in model?
Reusable in code / independently testable?
yes
no
Simulation and testability constraints for reusable models fulfilled?
yes
Reusable model
no
CHANGE DESIGN (only reusable models support reusable code)
Independently testable?
no
Subystem in SL library
yes
Singleton model
Outside a conditionally or iteratively executed context?
yes
no
Mainly signal routing or virtual blocks? OR Algebraic loop with calling context?
yes
Virtual subsystem
no
Atomic subsystem

Figure 54: Container primitive selection

MR 20 - Model layers

A. Two model layers should be distinguished:

- Routing layer
- Functional layer

B. The routing layer should only contain subsystems and blocks that can be placed on any model level as defined in MAAB guideline db_0143.

C. The functional layer can contain any type of block.

Each container primitive forms a layer. To avoid arbitrary layering, some guideline sets recommend a distinction according to their function. The concept of layers has been introduced by JMAAB 10.2 [109] and has been adopted by the MAAB guidelines [108]. Although the JMAAB architecture defines more layers and a strict hierarchical breakdown, experience showed that separating between routing and functional layers is sufficient and does not restrict developers too much. Both layers are also checkable, since the usable blocks are defined.

5.6.4.3 ***Private library***

SL libraries and the behaviors have been introduced in section 5.6.4.1. To repeat, SL libraries in R2017b have no own simulation or code generation configuration set, they can neither be compiled nor executed, and no code can be generated. In consequence, many verification techniques are not directly applicable.

SL libraries are easy to use, but can significantly increase the complexity of modeling, verification, and traceability. Thus, the specialization of PRIVATE LIBRARIES is introduced to leverage the power of SL libraries in a controlled way.

MR 21 - Private libraries

SL libraries shall only be used as PRIVATE LIBRARIES under the following restrictions (exception is the *DO-331 Foundation Library*):

A. SL library subsystems shall only be used in the module they are defined.

B. SL libraries shall not contain further library links (except to the DO-331 Foundation Library).

C. Only subsystems shall be on the root SL model layer.

D. Links shall only be allowed to the top-level subsystems of a SL library.

E. Model references shall not be placed in SL libraries.

Theoretically, a SL library can contain multiple levels of subsystems. Each subsystem can link to another library, which can link another library, which can link another library, and so on. Or it can embed model references. Furthermore, theoretically any subsystem layer of a SL library can be referenced. So parts of the SL library function could be picked.

In sum, the danger of generating complex “spaghetti” models is very high. (B-E) limit the application of SL libraries. In addition, if SL libraries are used across modules, the ownership is difficult. A SL library in another module can change a model and generated code significantly without changing any file of the module. Thus their scope shall be limited to a single module (A).

In addition, SL libraries raise problems concerning traceability and verification. If they have linked requirements, the links would be copied to every instance (since references are just “duplicated” block patterns). One requirement would have multiple (different) implementations (in the worst case across modules) and testing in the different contexts would be required. Furthermore, it is hard to detect unused library functionality, since model coverage is only collected for instantiations and never for a SL library. With sole coverage assessment, it is not possible to find unused or uncalled library functionality. And many other verification tools do also not support libraries, since they require fully specified models, which can be compiled. Only a subset of modeling rule checks can be executed on libraries. Traceability rules are discussed in section 6.5.4.

5.6.4.4 ***Module interfaces***

This section explains, how the different kinds of module interfaces introduced in section 5.4.4.3 are implemented. Data is mainly exchanged through interfaces of SL models, when a model is called. PDIs are a separate interface not discussed in this section.

MR 22 – Interface constructs

Only the following constructs are allowed to exchange data with a nested SL model:

- *Port interfaces* modeled with Inport and Outport blocks [127, pp. 1-765ff.]
- *Data Store interfaces* modeled with Data Store Memory, Data Store Read, and Data Store Write blocks [124, pp. 62-2ff.]

The supported mechanisms to exchange data with a called SL model are listed in MR 22. Other mechanism, like *GoTo interfaces* modeled with GoTo and From blocks [127, pp. 1-713ff.]), exist, but shall not be used. This subset limits the complexity of verification, suffices in most cases, and has high compatibility with the used verification tools.

For narrow interfaces, value range contracts have to be defined. Therefore, SL offers *signal ranges* [124, pp. 64-54ff.]. Both Port and Data Store interfaces support signal ranges. Sometimes signal ranges are also called design ranges in SL. However, here the term “design range” is only used for ranges specified in HLRs. They are not part of the SW Design and thus shall not be inserted in SL. All ranges inserted in SL are signal ranges.

MR 23 – Signal ranges

A. Design ranges specified in HLRs shall not be inserted into SL models as signal ranges.

B. Signal ranges shall only be set for interfaces with narrow contracts.

C. Signal ranges shall only be defined in Simulink.Bus (or more specific the nested Simulink.BusElement) objects.

D. Signal range minimum and maximum values shall be scalar and not NaN, within the range of the basic data type, and exactly representable in the basic data type

E. Signal ranges shall not be specified for signals of type boolean and enumerations.

F. Signals passed to models and calculated in models shall always be within the specified signal range. This shall be proven by formal analysis.

Managing signal ranges is difficult in SL/SF. Signal ranges can be specified redundantly and with differing values for the same model element. For example, an Inport block referring to a Simulink.Bus object may have minimum and maximum specifications in the port block mask or in the Simulink.Bus object. Although there is a defined priority, multiple declarations are confusing and they are not checked for consistency. In addition, signal ranges can be redundant and incompatible between model elements. For example, an Output block can specify a range, and a directly connected Inport block can specify a separate range, too. Again, no consistency check exists.

Application must thus be constrained. Signal ranges shall just be used for narrow contract interfaces as listed in (B). These are inter-module interfaces (cf. DR 11). Only attaching signal ranges to Simulink.Bus data types ensures consistency (C). It avoids the above listed challenges, since Simulink.Bus objects are centrally stored and apply for signals, not for either an in- or outport block.

Another option, which has not been further considered here, is using Simulink.Signal objects for single value signals (cf. MR 37). They are also centrally stored and can save signal ranges as well. However, this introduces additional design options, which the verification steps have to support.

Signal ranges not exactly representable in the basic data type are handled differently depending on the tool. For example, the specified range of $[-3.14, 3.14]$ for a inport of data type int32 is extended by SL Design Verifier to $[-4, 4]$. However, the runtime diagnostics use the original range for range checking. It is thus mandatory to ensure that the entered signal ranges are representable in the basic data type to avoid confusion (D).

Signal ranges are verified in the following tasks (cf. data coupling and control coupling in section 4.6.4):

- SwVP-DP-MB 5 – **Design error detection**
- SwVP-DP-MB 8 – **Simulation testing & result review** (runtime-error diagnostics)
- SwVP-CP-MB 5 – **Code proving**

MR 24 – Component interfaces

A. Component interfaces shall be modeled as Port interfaces in the top-level model.

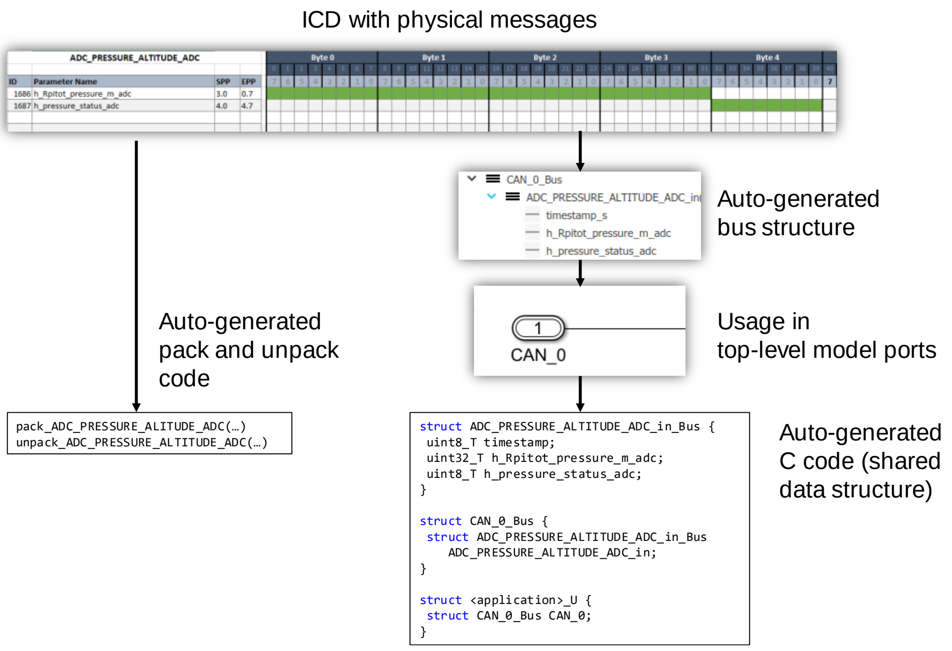
B. The Port interfaces shall have a data type of class EXPORTED BUS (cf. MR 30).

C. The Simulink.Bus objects shall be auto-generated from ICDs and stored in the component module.

D. The Simulink.Bus objects shall not specify signal ranges, since component interfaces are unconstrained (cf. DR 11).

In section 3.3, the interaction of the software application with the framework has been outlined. Before each execution step, the application framework unpacks new received incoming data (of physical interfaces) and copies it into a shared data structure in memory. The application reads from this structure during execution. Outgoing data is written into another data structure. After the execution of the application step, the framework searches the data structure with outgoing data for updated fields, packs messages and forwards them.

The shared data structures are auto-generated with the application Design Model. Therefore, Simulink.Bus objects for Inport and Outport blocks are automatically derived from the ICDs and referenced in the top-level model (cf. Figure 55). The top-level model has, according to MR 15, a `void-void` interface, and thus forces Embedded Coder to generate a separate input and output structure. Also the code for unpacking and packing physical messages is auto-generated.


ICD with physical messages
ADC_PRESSURE_ALTITUDE_ADC
Byte 0
Byte 1
Byte 2
Byte 3
Byte 4
Auto-generated bus structure
CAN_0_Bus
ADC_PRESSURE_ALTITUDE_ADC_in
timestamp_s
h_Rpitot_pressure_m_adc
h_pressure_status_adc
Auto-generated pack and unpack code
Usage in top-level model ports
1
CAN_0
Auto-generated C code (shared data structure)

ID	Parameter Name	SPP	EPP
1686	h_Rpitot_pressure_m_adc	3.0	0.7
1687	h_pressure_status_adc	4.0	4.7

```
pack_ADC_PRESSURE_ALTITUDE_ADC(…)
unpack_ADC_PRESSURE_ALTITUDE_ADC(…)
```

```
struct ADC_PRESSURE_ALTITUDE_ADC_in_Bus {
 uint8_T timestamp;
 uint32_T h_Rpitot_pressure_m_adc;
 uint8_T h_pressure_status_adc;
}

struct CAN_0_Bus {
 struct ADC_PRESSURE_ALTITUDE_ADC_in_Bus
    ADC_PRESSURE_ALTITUDE_ADC_in;
}

struct <application>_U {
 struct CAN_0_Bus CAN_0;
}
```

Figure 55: Auto-generation of component interface code

The general structure of the auto-generated buses is given in Listing 2.

```
<application-name>_U [_Y]
 .PHYSICAL_INTERFACE
   .MESSAGE_NAME
     .FIELD-1
     ...
     .FIELD-N
     .timestamp_s (uint8) [update_flg (boolean)]
```

Listing 2: Pseudo-structure for external data exchange

`<application-name>_U` is the input structure, `<application-name>_Y` the output structure. The application framework writes into the input structure and reads from the output structure. A substructure on the first level is generated for each physical hardware interface (each port in the SL model). Below, a substructure for each message is added, containing fields for each parameter. The `timestamp_s` field is a running number, which is increased by the framework software every time a new message has been received. This provides a certain amount of information about the up-to-dateness of the data.

The data exchange structure for outputs is similar. Instead of the timestamp, a boolean called `updated_flg` is provided. If it is set, the corresponding message data is forwarded by the software application framework to the physical I/O interface after the execution step of the application.

MR 25 – Inter- and intra-module interfaces

A. Inter-module interfaces of units shall be specified with an EXPORTED BUS data type (cf. MR 30), if

- complex data is exchanged.
- a narrow contract shall be formulated (cf. MR 23).

B. All Simulink.Bus objects specifying inter-module interfaces shall be stored in the project-specific global module (cf. Figure 42).

C. Inter-module interfaces with a narrow contract shall have a SINGLETON SIGNAL assigned (cf. MR 37). In consequence, only SINGLETON MODELS can have a narrow contract interface (cf. MR 6).

D. For intra-module interfaces, no special limitations apply.

(A) and (B) come along with practical aspects. In contrast to basic data types, Simulink.Bus data type specifications can be stored externally to the SL model including signal ranges. So they are not managed by one or another module, but globally. This significantly improves interface control.

(C) significantly simplifies assertion of narrow interface on code level (cf. section 8.2.14). Polyspace Code Prover can only assert value ranges for global variables, which are only easily identifiable in auto-generated code, if SINGLETON SIGNALS are used. A SINGLETON SIGNAL is a specialization of a Simulink.Signal [127, pp. 5-490ff.], which controls the code generation of a specific signal. It creates a uniquely named exported global variable in code for each signal. The concept of SINGLETON SIGNAL is explained in MR 37. An additional side effect is that SINGLETON SIGNALS simplify testing due to higher accessibility of variables. On the flipside, inputs normally passed as arguments are now exposed as global variables, which makes data coupling analysis more challenging. However, Embedded Coder ensures that these signals are just written once per step.

MR 26 – Data store interfaces

A. The use of data store constructs should be avoided.

B. If unavoidable, data stores shall be global and written once in the top-level model before the execution of any nested model. They shall only be read in sub-modules.

C. Each global data stores shall be based on the DATA STORE SIGNAL specialization (cf. MR 37).

D. Global data stores shall only be read or written with the Data Store Read and Write blocks in SL. They shall not be directly read or written in SF.

E. Data stores shall be written completely and once only.

F. If a narrow interface is specified, the underlying Simulink.Signal object must be of Simulink.Bus data type.

Data store constructs create an invisible data flow, make testing as well as coupling analysis difficult, and should thus be avoided (A). Anyway, best practice showed, that in some cases, data stores are reasonable. For example, to form a globally accessible data pool of sensor signals and avoid inefficient Simulink.Bus copies in the code. MR 26 is defined for these cases only.

Data stores consist of a Data Store Read, a Data Store Write and, optionally, of a Data Store Memory block. The code generation of data stores can be specified with Simulink.Signal objects. A global data store is a data store available across SL model boundaries. It does not have a Data Store Memory block, but needs a specifically configured Simulink.Signal.

(B)-(E) limit the usage of data stores. Global data stores are a deviation from high-integrity guideline “hisl_0013” [103, p. 2-38], which recommends to “avoid data store reads and writes that occur across model and atomic subsystem boundaries”, since the SL sorting algorithm does not take into account this type of data coupling. However, with (B), an explicit execution order is ensured. (B) may require changing the execution priority of the Data Store Write block, so that it is executed at first. This is the only situation, in which changing the execution priority is allowed. (D) has been introduced to simplify the cases to be considered for formal value range verification with SL Design Verifier (cf. section 8.2.5).

Only elements in Simulink.Bus objects shall specify signal ranges to guarantee the same signal ranges everywhere, where the Data Stores are read. Whenever a narrow contract shall be defined, the DATA STORE SIGNAL must be of type Simulink.Bus (F).

In order to use data stores the recommend way, also the diagnostic settings proposed in “hisl_0013” had to be adapted. If `ReadBeforeWriteMsg` always throws an error, no reading SL models can be independently simulated, because it throws an error. The configuration settings of sub-level models thus disable the `ReadBeforeWriteMsg` diagnostics, but not the top-level model (cf. Table 23). This sounds unsafe for nested models, but actually is not a different behavior than the simulation of non-instrumented Inport blocks (default initialization with zero).

Figure 56 shows a global data store architecture. Model C is the model referenced in models A and B as `fc_test`. The referenced model C can be independently simulated with the diagnostic settings described above. Model A cannot be simulated, since it has a read before a write. In model B, the execution order has been changed and it can be simulated without error.

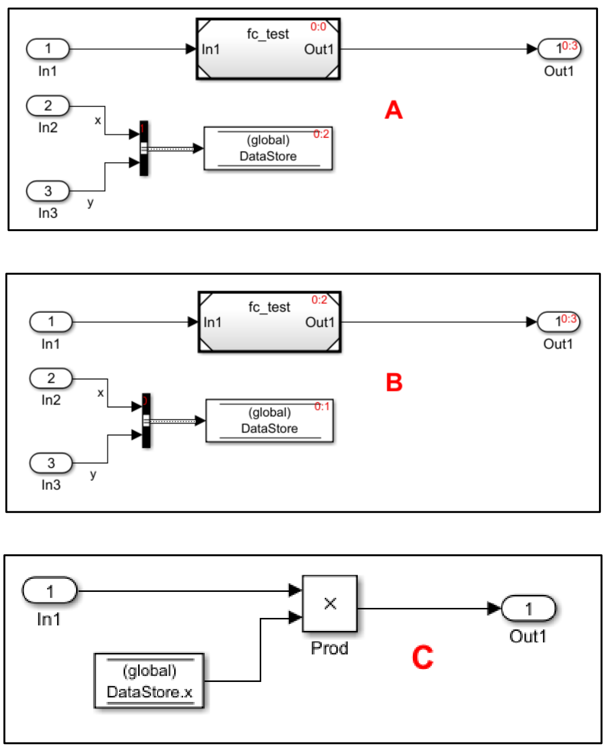


Figure 56: Data store interface usage examples

5.6.4.5 ***SL atomic primitives***

Figure 48 distinguishes graphical model primitives into atomic and container primitives. Atomic primitives can graphically not be broken down any further, in contrast to container primitives.

MR 27 - Supported atomic model primitives

Only the atomic primitives of the *DO-331 Foundation Library* shall be used.

A block library for the DO workflow does not exist in the shipped version of Simulink (R2017b) and is not publicly available. A few partially compliant libraries, like the supported block library of SLCI (`slcilib`)[20], exist, but they do neither fulfill modeling rules of MAAB, JMAAB, or Math-Works High-Integrity Guidelines, nor do they comply with project specific requirements. The *DO-331 Foundation Library* is a specific SL library containing all preconfigured atomic primitives, which can be used.

Contribution 10: A novel foundation block library limiting the usable model elements has been created. This library deviates from block libraries shipped by MathWorks in R2017b, since it is compliant with the rules at hand (e.g., naming conventions) and contains new blocks.

The library has been derived from the `slcilib`, since SLCI imposes the most significant constraints on usable block sets. Blocks in the DO-331 Foundation Library are considered as atomic on model-level from the viewpoint of the process (e.g., static model analysis does not traverse them). Most of the blocks are actually atomic (*simples primitives and complex primitives*), others, however, are subsystems with nested blocks, further on called *pseudo primitives* (cf. the discussion about the Model Element Library in section 5.6.5.5).

The following refactoring was necessary to make the existing `slcilib` compliant to the process:

- Adaption of blocks labels to naming rules
- Adaption of default block settings compliant to the modeling guidelines and most frequently used options
- Removal of library link breaking
- Removal of unsupported blocks
- Inclusion of custom real and pseudo primitives

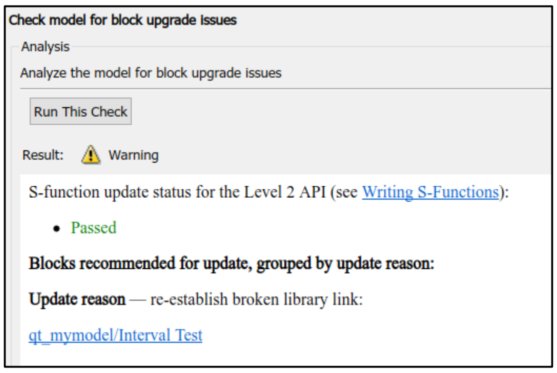
[20] The `slcilib` is a block library shipped with SLCI. It can be opened by typing `slcilib` in the MATLAB command line window.

Removal of library link breaking

If simple or complex primitives are added from a SL library to a SL model, a reference to the globally available low-level implementation is inserted. The model behavior may change if the low-level implementation changes, e.g., due to a MATLAB version upgrade.

For pseudo primitives, the `slcilib` compels a different behavior. Adding the primitive to the SL model creates an unlinked duplicate of the nested block pattern. This is a non-default behavior for libraries, which typically keep a link. The behavior is realized with hidden callbacks. In consequence, an update of the library is not propagated to already added block patterns.

Practice showed that this mitigates on the one hand the risks connected to changes of the DO-331 Foundation Library, since they do not influence existing models. On the other hand, important fixes cannot easily be deployed. In addition, the Simulink Upgrade Advisor [124, pp. 6-2ff], a tool which supports the migration from one to another MATLAB release, detected many pseudo primitives as taken from a built-in library (even if they were modified copies) and proposed to automatically reestablish the links to the original version (cf. Figure 57[21]). It occurred several times that developers unsuspectingly executed the tool and observed unexpected behavior due to wrong reestablished links. Thus, the respective callbacks of the `slcilib` have been removed.


Check model for block upgrade issues
Analysis
Analyze the model for block upgrade issues
Run This Check

Result: ⚠ Warning

S-function update status for the Level 2 API (see Writing S-Functions):

- Passed

Blocks recommended for update, grouped by update reason:

Update reason — re-establish broken library link:

qt_mymodel/Interval Test

Figure 57: `slcilib` upgrade issue

[21] Run upgrade advisor by selecting Analysis > Model Advisor > Upgrade Advisor from the model editor.

Removal of unsupported blocks

The following blocks have been removed from the `slcilib`:

- Discrete time integrator (replaced by new custom integrator blocks)
- Difference (not supported by [108] guideline "jm_0001")
- Doc block (replaced by new Doc block)
- Variant subsystems, model variants (unsupported according to MR 28)
- Unit conversion (unsupported by process at hand according to MR 28)
- Probe (not supported by [108] guideline "jm_0001")
- MATLAB Function (unsupported by process at hand according to MR 28)

New custom pseudo primitives

In general, the number of pseudo primitives is kept as small as possible. The preferred method of implementing reused functionality is the REUSABLE MODEL specialization. However, in some cases, convincing arguments exist to add a new primitive to the DO-331 Foundation Library.

The design pattern *must be sufficiently universal and simple* and one of the following criteria must be met, in order to include a new primitive into the DO-331 Foundation Library:

A. A mask significantly reduces the number of variants or simplifies the interface. Primitives are allowed to have masks, developers, however, shall not implement masks in the SL model by their own (cf. MR 28).

B. The primitive avoids artificial algebraic loops. SL can resolve algebraic loops across virtual subsystem borders, but not across model reference or atomic subsystem borders.

C. The functionality is not or just inconveniently realizable with existing blocks.

The following groups of blocks have been added after intensive testing:

- Doc Block (real primitive, C)
- Integrators (pseudo primitive, B)
- Trigonometric flip functions (pseudo primitive, C)
- 2D-Selector[22] (pseudo primitive, A+C, natively not supported by SLCI in R2017b)
- NaN-Checking (real primitive, C, SL built-in NaN checking blocks are not supported by SLCI)

[22] Prior to R2017b, SLCI did not support multi-dimensional indexing in the Selector block.

5.6.4.6 *SL atomic primitive restrictions*

The settings and use of atomic primitives underlies certain constraints. Only a few important concepts are discussed in the following, since most of the restrictions are part of the fundamental modeling rules.

MR 28 - Excluded SL functionality

The following functionality shall not be used in any atomic primitive:

- Prioritization of execution order
- Absolute time
- Units
- Variable size signals
- MATLAB code
- Custom block masks
- Variants

There are a couple of SL features, which have their impact at various places in the model. They are excluded by this rule. For some of the features listed above, there is no rationale, they just have not been investigated and conservatively been excluded.

Every block provides the possibility to increase or decrease prioritization in the execution order. This priority is respected during model compilation as far as possible. Changing the priority has significant impact on the outcome of the model and may lead to counter-intuitive outputs. It is thus highly recommended to not use this feature. Single exception are Data Store Write blocks (cf. MR 26).

The exclusion of MATLAB code results from two considerations. At first, it was not well supported by SLCI in earlier releases. Second, it would need the definition of MATLAB code guidelines. Many tasks can be performed more efficiently in MATLAB code and its support should be regarded for the future. All controllers of the projects accompanied by the author could be implemented without MATLAB code.

Developers shall not be allowed to define own block masks. Block masks define behavior, which is not documented in the Design Description (cf. section 8.1.1), and may totally redefine block behavior (e.g., self-modifying masks [124, pp.38–56ff.]). Practice showed that, if masks are allowed, they quickly lead to complex constructs and hidden logic.

Variants allow conditionally exchanging parts of SL models or whole SL models based on global parameters. This “dynamic” significantly increases verification effort and the identification of dead design. Variants were not used in the Design Model to keep the process simple.

5.6.4.7 ***Model data***

SL features open various ways to define and use model data. If all features were allowed, it would be hard to predict the impact on the generated code and whether verification tasks are compatible and do the right thing. The novel, composed minimum viable set in this section covers many use cases and forms a manageable feature set for verification.

Almost all model data are objects in the SL data dictionary. Each entry in the SL data dictionary is required to be mapped to a safe specialization.

MR 29 – Mapping of model data to safe specializations

> Every model data in the SL data dictionary shall be mapped to safe specializations by a respective reference in the *first line of description field* of each entry.

Figure 44 shows a screenshot from a SL data dictionary. In the bottom right, the description field shows, how the type of the specialization is defined, in this case as `bus-exported`.

5.6.4.7.1 Buses

SL provides the possibility to define *bus data types* in order to structure data signals. Buses are globally defined as Simulink.Bus objects in the SL data dictionary [127, pp. 5-207ff]. They combine different signals with different data types. Buses can contain other buses and thus form arbitrary hierarchies.

Two specializations are used:

- EXPORTED BUSES
- IMPORTED BUSES

EXPORTED BUSES are those, which become part of the generated code. IMPORTED BUSES consider the data types as existing in code. Code generation does not translate, but just reference them. In most cases, EXPORTED BUSES are the right choice, since the type definition shall be part of the generated C code. IMPORTED BUSES are used, if C type definitions are externally programmed or generated (component interfaces, or, for example, the communication protocol implementation by the author in [31]).

MR 30 – Exported and imported buses

A. Buses shall be created in the SL data dictionary with the properties of Table 26.

Property	EXPORTED BUS	IMPORTED BUS
Data Scope	`Imported` (`Exported` during shared code generation)	`Imported`
Header File	`<module-ID>_bus_types.h`	Any header file
Specialization name in description	`bus-exported`	`bus-imported`

Table 26: Bus properties

B. Naming conventions shall be applied (cf. MR 1).

The data scope determines, whether the bus declaration is generated in C code (`Exported`) or are considered as existing (`Imported`). For example, the `typedef struct` in Listing 3 is generated from an exported Simulink.Bus object.

```c
typedef struct {
    real32_T Phi_rad;
    real32_T Theta_rad;
    real32_T Psi_rad;
    boolean_T Phi_rad_valid_flg;
    boolean_T Theta_rad_valid_flg;
    boolean_T Psi_rad_valid_flg;
} xy_rawAHRSData_Bus;
```

Listing 3: Bus structure translated to C code

Nevertheless, independent of whether an imported or exported bus is specified, the data scope shall be set to `Imported` (A). In consequence, type definitions are never generated in C code. During the *shared code generation process*, which will be detailed in section 8.1.2, the data scope is changed to `Exported`.

All exported data types declarations in the code shall be collected in one header file per module for modular code and readability reasons. Thus the header file is explicitly set (A).

Since Simulink.Bus objects are globally visible in all modules, naming conventions are important to avoid identifier clashes and identification of module ownership (B). The chosen syntax is `<module-ID>_<busName>_Bus`. Suffixing the name of the type with `_Bus` clearly distinguishes data types from other model data and especially from the field names in nested bus structures.

MR 31 - Exported/imported bus usage

A. Virtual buses shall not be used for in- or outputs of SL models.

B. Array of buses shall not be used.

C. Subsequent bus assignments on the same bus should be avoided.

In terms of usage, it is distinguished between virtual and non-virtual buses. Virtual buses, like virtual subsystems, are only graphical elements. They are not visible as structures in the generated code, whereas data of non-virtual buses is stored in contiguous memory sections [124, pp. 65-4f.]. Whether a bus is virtual or non-virtual is decided by the usage in the model, not by a Simulink.Bus object. Simulink.Bus objects are optional for virtual buses [124, pp.64-64ff.].

By construction, virtual buses promise higher performance of code, since the code generator has more freedom for optimization. Virtual buses, however, must be used with thought, since experience showed that

- migration to new MATLAB releases caused problems, if virtual busses crossed model references and were mixed with non-virtual buses.
- SLCI may require to make buses non-virtual in order to analyze traceability.

For testability reasons, buses at in- or outputs of SL models shall directly represent the data type in the code (A). Buses that never cross the border of a model reference may be virtual.

Array of buses are not supported by SLCI [121, p. 3-6] (B).

In the projects, to which the author contributed, significant effort was spent to improve execution performance. One identified performance lack was wrongly used buses. In R2017b, buses tended to introduce data copies in the generated Source Code. Especially in case of large buses, this can have big impact on execution performance of the code. A change from non-virtual to virtual buses did not bring the expected performance increase, however optimized block patterns as in Figure 58 drastically reduced the amount of copies.

In Figure 58, Model A has two Assignment blocks. Each creates a copy of the whole data structure in R2017b. By combining both Assignment blocks in model A*, one data copy can be avoided.

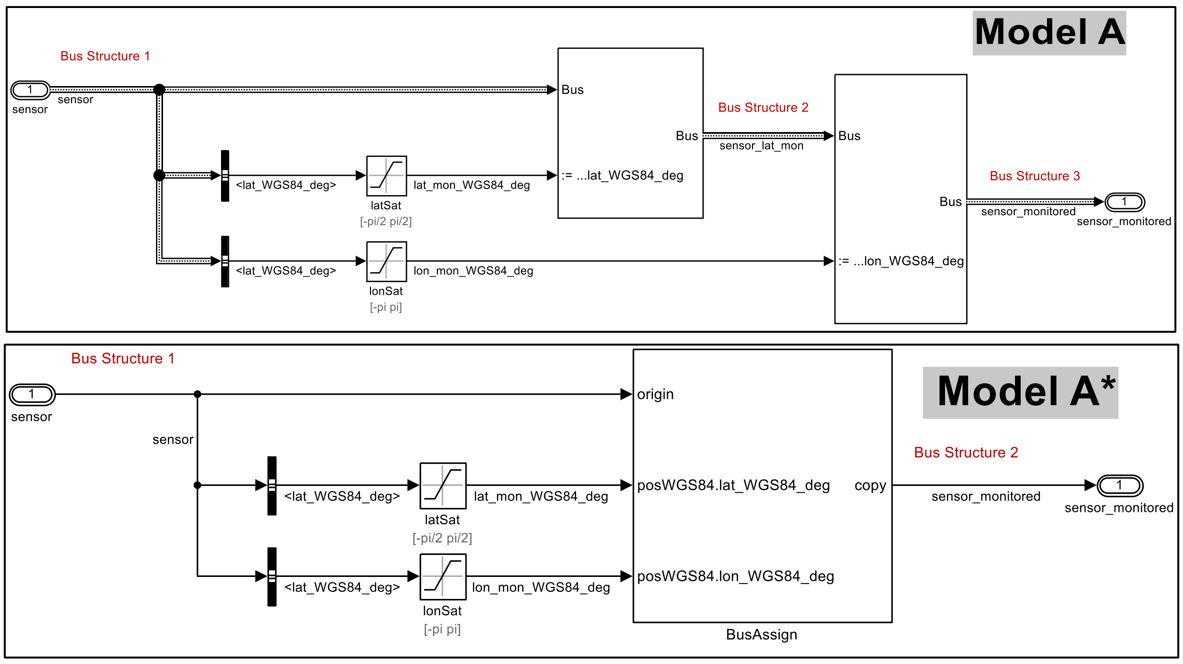


Figure 58: Simulink.Bus performance best practices with optimized model (*)

MR 32 - Data alignment in buses

> If the data type length of all elements in a bus is less than 4 bytes (not considering nested buses), the alignment shall be enforced with compiler statements. Therefore, the alignment property of the respective Simulink.Bus object shall be set to 4.

From a modeling point of view, elements can be arbitrarily placed in buses. However, the translated structures of non-virtual buses in C immediately rise the question of alignment requirements imposed by the processor. If misaligned words are accessed, an alignment fault may be thrown or additional instructions and calculations are required to access the data. Many processors just support so-called natural alignment, where the address of a word is the multiple of its size [129, pp. 245ff].

Misalignment is normally no problem, since either the processor itself handles the misalignment or the compiler or linker introduces padding bytes. However, in two areas further considerations were necessary:

Firstly, padding bytes inserted at external input interfaces may cause problems, especially memory areas are completely copied into a structure. The application framework presented in section 3.3 copies field per field into the input structure, so padding bytes have no impact.

Secondly, compatibility to verification tools should be considered. For the tool chain presented in this work, especially the WCET analysis with aiT WCET Analyzer caused problems with misaligned data access on the used PowerPC. Time measurements, which form the basis for the evaluation, only exist for aligned accesses. A misaligned access forces the compiler to introduce multiple instructions, which are not covered by timing measurements [37].

The brute way to avoid misaligned data access is enforcing alignment by the compiler. CompCert therefore knows the language extension `__attribute__((aligned(%n)))`, which forces the compiler to insert padding bytes [57]. By registering a target info in combination with a code replacement library in Embedded Coder, an alignment specification and rules for application can be defined for structs [96, pp. 51-135ff.]. In combination with the alignment field in the Simulink.Bus element, prepending of the extension can be triggered.

Another possibility is to modify the bus. Either padding bytes are manually added or so-called “natural alignment” [129] is applied. A structure is naturally aligned, if the starting address of every field (also of nested structs) is a multiple of the field word size. It is implicitly fulfilled, if fields are arranged by increasing word size as in Listing 3.

Here, bus reordering is not considered as feasible approach due to complex and deep bus structures. It has been decided to force alignment with compiler statements, which proved to be compatible with the WCET analyzer [37] as well. In the special case with CompCert, forcing alignment was only necessary, if no element in the bus had a byte length of four or more. Otherwise the compiler automatically used a 4-byte alignment. Developers are supported with a check identifying potential alignment problems in the bus structure.

5.6.4.7.2 Safe enumerations

Enumerations (enums) are a finite set of values with a unique name (literal). Value and literal are called *enumeration constants*. The literals have a meaning in the context they are applied, and often increase the readability of SL models and SF charts. If enumerations are used as data types, they are called *enumeration data types*.

The following paragraphs specify definition and usage of enumeration data types and constants.

MR 33 - Definition of a safe enums

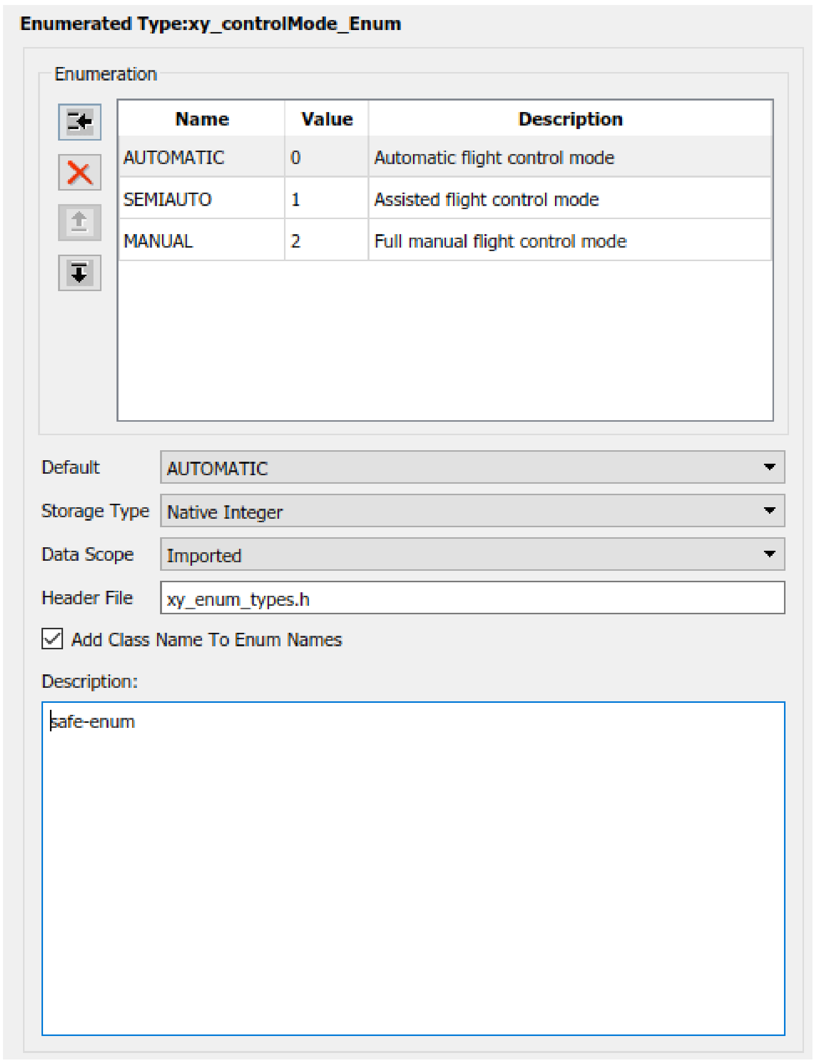
A. Enumerations shall be created in the SL data dictionary with the properties of Table 27. These are called SAFE ENUMS.

Property	**Value**
Literals/Values	Value range `[0,2147483647]` Literal for 0 value is required. No further constrains, but shall be defined in monotonically increasing order.
Default	Literal of 0 value
Storage Type	Native Integer
Description	Any
Data Scope	`Imported` (`Exported` during shared code generation)
Header File	`<module-ID>_enum_types.h`
Add Class Name To Enum Names	Enabled

Table 27: Safe enum properties

B. Naming conventions shall be applied (cf. MR 1).

In SL data dictionaries, SL Enumerated Type objects are globally visible entries, which can be created, modified and removed just the same as Simulink.Bus data types. They are instances of the `Simulink.data.dictionary.EnumTypeDefinition` class [127, pp. 5-743ff.]. Since SL data dictionaries are used, this is the only way to create and use enumerations in SL models. Figure 59 shows a SAFE ENUM in a SL data dictionary with the *enumeration constants* AUTOMATIC, SEMIAUTO and MANUAL.



Enumerated Type:xy_controlMode_Enum

Enumeration

Name	Value	Description
AUTOMATIC	0	Automatic flight control mode
SEMIAUTO	1	Assisted flight control mode
MANUAL	2	Full manual flight control mode

Default	AUTOMATIC
Storage Type	Native Integer
Data Scope	Imported
Header File	xy_enum_types.h

[✓] Add Class Name To Enum Names

Description:

safe-enum

Figure 59: Safe enumeration screenshot

The rationales for (A) are influenced by verification aspects and code generation considerations.

Default value

The constrains on the default value are given by SLCI [121, p. 3-6]. The value is used as fallback for safe casting.

Storage type

SL Enumerated Type objects allow the specification of the underlying storage type, which is used in SL and for C code generation. It must be distinguished between *basic integer storage type* and a *native integer storage type*. If basic integer types are used, like `uint8` or `int16`, the bit length and value range is fix. In C code, Embedded Coder fakes enumeration constants with preprocessor directives and the enumeration data type with a redefinition of the basic integer type (Listing 4). “Real” C99 enumerations are not used.

```c
/* This defines an enumerated type for control modes */
typedef uint8_T xy_u8controlMode_Enum;

#define xy_u8controlMode_Enum_AUTOMATIC ((xy_u8controlMode_Enum)0) /* Default value */
#define xy_u8controlMode_Enum_SEMIAUTO  ((xy_u8controlMode_Enum)2)
#define xy_u8controlMode_Enum_MANUAL    ((xy_u8controlMode_Enum)3)
```

Listing 4: Enumerations realized with macros

If the native integer storage type is used, the underlying integer data type in SL is chosen based on the range of the enumerated constants by the simulation engine. In C code, “real” enum types are used (Listing 5). The compiler choses the data type in the Executable Object Code during the compilation process. It is thus implementation-defined (C99 6.7.2.2 §4). An optimizing compiler will realize that the integer range in Listing 5 does not exceed 1 and would choose an unsigned 8-bit integer.

```c
/* This defines an enumerated type for control modes */
typedef enum {
       xy_controlMode_Enum_AUTOMATIC = 0,      /* Default value */
       xy_controlMode_Enum_SEMIAUTO,
       xy_controlMode_Enum_MANUAL
} xy_controlMode_Enum;
```

Listing 5: Enumerated type

From a functional point of view, implementations with `typedef` and `#define` are similar in these cases ([129, p. 83]), but SLCI only supports native integer storage types [121, p. 3-6].

Enumeration constants

Enumeration constants are the values and literals in an enumeration. C99 knows explicitly and implicitly specified enumeration constants (C99 6.7.2.2 §3). Latter just have a literal and no value assignment. Their value is an increment by one of the previous constant in the list. For implicit definitions, MISRA C Rule 8.12. requires uniqueness of derived constant values.

The SL data dictionary only allows the specification of enumeration lists with explicitly defined enumeration constants. Whether they are turned into implicit definitions in code is decided by the code generator, and thus also the final MISRA C compliance (cf. Listing 5 with the explicit enumeration constant AUTOMATIC, and the implicit enumeration constant MANUAL).

The value range has been limited to $[0,2147483647]$. As explained before, since native integers have to be used, the data type size is compiler-dependent. According to [57] §6.7.2.2, CompCert uses a 4-byte integer for all enumeration values. The definition reflects the positive value range.

SL Design Verifier supports enumeration data types and narrows the range to the maximum value. Figure 60 shows verification results of a model using enumeration data types in the range $[0,2]$. Green blocks have valid objectives; red blocks have falsified objectives.

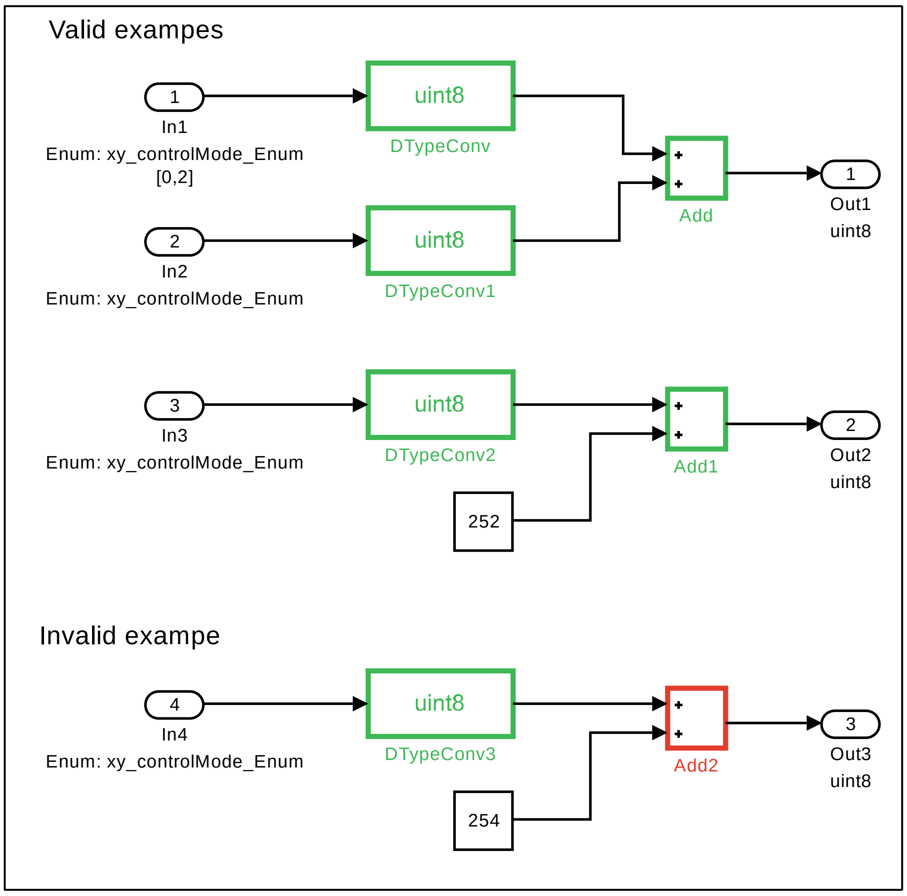


Figure 60: Example of enumerations in SL Design Verifier

Add class name to enum names

In SL, the literals of enumeration constants are always scoped for the enumeration, i.e., two enumerations can have the same literals. Enum constants are always addressed by a combination of enum class name and literal (e.g., `xy_controlMode_Enum.MANUAL`).

In C, the literals are global (e.g., `MANUAL` can directly be used anywhere in the code). In consequence, to avoid identifier clashes during code generation, the enum class name is prepended during code generation by enabling the option “Add Class Name To Enum Names”.

Data scope

The specification of an explicit header file name and `Imported` data scope is necessary to modularize the code. Imported data scope avoids building the code in situ with SL models. In the shared code generation process (cf. section 8.1.2.2), the data scope is changed. If the explicit header file were not set, Embedded Coder would generate guarded definitions into the code of each model, which uses the enumeration (cf. [35] and Listing 6).

```c
# ifndef DEFINED_TYPEDEF_FOR_ENUM_xy_controlMode_Enum_
# define DEFINED_TYPEDEF_FOR_ENUM_ xy_controlMode_Enum_

typedef enum {
       xy_controlMode_Enum_AUTOMATIC = 0,       /* Default value */
       xy_controlMode_Enum_SEMIAUTO,
       xy_controlMode_Enum_MANUAL
} xy_controlMode_Enum;

# endif
```

Listing 6: Guarded enumeration definitions

MR 34 - Usage of a safe enums

SAFE ENUMS shall only be casted from integer types. A safe cast, which falls back to the default value, should be used.

SL already restricts usage of enumerations drastically, especially as data types. For example, arithmetic operations are not possible, as well as casting from a floating-point data type to an enum.

SL and Embedded Coder can automatically insert a so-called safe cast. The safe cast falls back to the default value, if the integer value is not supported by the enumeration. This also works for non-contiguous enumeration values. Safe casting is applied, if the option “Saturation on overflow” is activated in the Data Type Conversion block [96, pp. 19-74f]. Listing 7 shows the code generated for safe casting. However, this feature deteriorates model to code coverage preservation.

```c
static int32_T safe_cast_to_xy_controlMode_Enu(int32_T input)
{
  int32_T output;

  /* Initialize output value to default value for xy_controlMode_Enum (AUTOMATIC) */
  output = 0;
  if ((input >= 0) && (input <= 2)) {
    /* Set output value to input value if it is a member of xy_controlMode_Enum */
    output = input;
  }

  return output;
}
```

Listing 7: Safe casting of enumerations

5.6.4.7.3 Constants and Parameter Data Items

The terms *constant* and *parameter* are widely used and have different meanings across disciplines. Here, the following process-driven definition shall be used: Both constants and parameters are constructs that allow the variation of a value from outside a function between executions without modifying the function. They hold values constant during the execution. A parameter is a specified feature (e.g., expressed by requirements), whereas a constant is a simple design pattern to increase maintainability, readability, reusability, or tunability. Constants are resolved to a specified constant value at the end of the development phase, in which they are introduced. All subsequent verification activities have to assume a constant value.

One example for constants are controller gains allowing the application of optimization algorithms in the closed loop simulation. The gain is tunable in simulation, but this tunability is no feature of the final software. A new optimized parameter value must be updated in the requirements afterwards and is then set to a specific constant value in the model. Subsequent steps consider it as constant.

Parameters are specified in requirements with type, range, or a set of allowed values. The value ranges must be considered in verification. In this case, the only true parameters are PDIs (cf. DR 17).

Technically, SL accepts three ways of defining constants and parameters in a model:

1. Block mask parameters,
2. MATLAB variables,
3. Simulink.Parameter objects.

Variables are either directly written into block dialogs (block mask parameters), referenced from MATLAB variables in a connected workspace, or they are wrapped in a Simulink.Parameter object [127, pp. 5-361ff]. Block mask parameters are not further discussed in this context, since they are not model elements stored in a workspace.

The implementation and use of MATLAB variables and Simulink.Parameter objects has significant impact on code generation and verification. Thus, it proved reasonable to limit the variety of implementations and support developers with guidance.

MR 35 - Constant and parameter specializations (general)

A. The specializations `CONSTANT`, `PARAMETER CONSTANT`, and `PARAMETER DATA ITEM` shall be used as defined in Table 28.

Criteria	**Constants**		**Parameters**
Specialization	`CONSTANT`	`PARAMETER CONSTANT`	`PARAMETER DATA ITEM`
Implementation	Basic MATLAB variable	Simulink. Parameter	Simulink.Parameter
Storage location	SL data dictionary, model workspace	SL data dictionary	SL data dictionary
Prerequisites			Specification as PDI in HLRs
Reusable	X	X	X
Accessibility	Model only, Private, Public	Private, Public	Private, Public
Tunable in Normal mode simulation	X	X	X
Tunable in SIL simulation			X
Allowed data types	Any basic + enumeration	Any, but not struct	Any, but not struct
Allowed dimensions	Any	Any, but not auto	Any, but not auto
Unit	n/a	Any	Any
Complexity	n/a	Real	Real
Allowed values	Any except special quantities like Inf, -Inf, NaN, []	Any except special quantities like Inf, -Inf, NaN, []	Any except special quantities like Inf, -Inf, NaN, []
Naming (cf. MR 1)	`<module-ID>_c_<name>`	`<module-ID>_p_<name>`	
Code placement	Inline or pool per model	Shared for module	Globally shared
Value influences structural checksum	Yes	No	No

Table 28: Usage of constant and parameter specializations

B. None of those specializations shall be used in in constant expressions.

Simulink.Parameter objects have the benefit that they support strong data typing. The developer can explicitly set a data type, whereas MATLAB variables derive a data type from the value. For example, in Simulink.Parameter objects, a Simulink.Bus data type can explicitly be selected. Then, the structured values must satisfy the constraints of the bus specification. In addition, Simulink.Parameter objects allow granular code generation settings as well as the specification of signal ranges.

However, Simulink.Parameter objects are conceptually made for tunable parameters. They are independent specifications. Neither do all of their properties (signal range, storage class, value, …) influence the structural checksum of the model, nor does their value appear in the generated code by default. Why Simulink.Parameter objects have been chosen for constants anyway is explained in the following paragraphs.

(B) originates from a SLCI limitation with tunable parameter expressions [134, p. 1-4]. The specializations of (A) are explained in the following.

Simulation tunability

At first, constants and parameters play an important role during model simulation. All specializations allow accessing and modifying values during execution of SL/SF in Normal, Accelerator and Rapid Accelerator mode (aside from a few exceptions [124, pp. 36–42f.]), even though different APIs must be used. These modes generate executables, which do not base on the Source Code, and include methods providing the required tunability.

In simulation modes depending on Source Code, like PIL or SIL, and in Executable Object Code, tunability depends on how Embedded Coder transforms the parameters into C [96, pp. 64-67ff.].

General code generation behavior

In the following, the tunability in the Source Code is further addressed. Code generation of MATLAB variables basically depends on the configuration setting "Default parameter behavior" (`DefaultParameterBehavior`) [135, p. 15-92], which predefines the behavior, if no further specification is given. If it is set to `tunable`, Embedded Coder tries to globally expose a maximum number of parameters in C. If set to `inlined`, no parameter is tunable in the C code unless tunability is individually enforced.

Since the majority of parameters are constants and resolved after the design phase, the default behavior is set to `inlined`. This leads to more efficient code.

If parameters cannot be inlined during code generation, Embedded Coder adds them to a *data pool*. The pool is by default generated in the source file `<modelname>_data.c`, or in a globally shared file, if the respective Embedded Coder parameter (`GenerateSharedConstants`) is set [96, p. 6-84]. In principle, this is a beneficial behavior, since it can drastically reduce the amount of memory and code, if large parameters are used.

Anyway, parameter pooling has some restrictions:

- Readability is decreased – the numbered identifiers are meaningless at the place where they are used (cf. example 3 in Appendix C) or even cryptic in shared pools.
- The large parameter pool struct is not readable, since the field names do not occur at the place where the values are defined, but only in the type definition (Example 3).
- In the given case, a registered Code Replacement Library for data alignment (cf. section 8.1.2.5) prevents the generation of a shared pool. Scope of the data pool is always the model.

The only possibility to share constants across models, avoid parameter pooling and control coding behavior more granularly, is wrapping the constant with a Simulink.Parameter object and specifying coder options. Specifications within the Model Parameter Configuration dialog shall be avoided, since these are ignored, if the model contains any further model references [128, pp. 5-26ff.].

MR 36 – Parameter object specializations

A. The coder settings for PARAMETER CONSTANT and PARAMETER DATA ITEM specializations shall be configured as defined in Table 29.

	PARAMETER CONSTANT	PARAMETER DATA ITEM
Signal range	Unset (interpreted as constant)	Set to a finite value (narrow contract)
Storage Class	`ImportedFromFile` (`ExportedToFile` during shared code generation)	`ImportedExtern`
HeaderFile	`<module-ID>_param_constant.h`	n/a
Owner	Set to model name of helper model used for shared data code generation (cf. section 8.1.2.2)	n/a
SourceFile	`<module-ID>_param_constant.c` during shared code generation	n/a

Table 29: Simulink.Parameter specializations

B. PARAMETER DATA ITEMS shall only be referenced in Constant blocks. They shall not be directly referenced in SF.

The two specializations differ in their storage classes and other properties to be set as listed in Table 29. The PARAMETER DATA ITEM specialization fulfills the requirements specified in DR 17. PARAMETER CONSTANTS shall be resolved during code generation and do not have to be accessed on code level. The compiler can assume them constant and perform optimizations.

(B) has been introduced to ease handling in formal methods (cf. section 8.2.5).

Signal Range

PARAMETER DATA ITEM specializations shall provide a signal range due to their narrow contract (cf. argumentation and implementation strategy in DR 17).

PARAMETER CONSTANTS shall not be tunable from a process point of view and, in consequence, the signal range shall not be set to avoid wrong results in verification.

Storage Class

As shown in Listing 8, storage class `ImportedExtern` of the PARAMETER DATA ITEM only generates an `extern` (forward) declaration in the model code [131, p. 1-111]. The statement tells the compiler that the variable is defined elsewhere [129, p. 222]. A header file does not need to be known.

```
/* In <modelname>_private.h> (code of consumer model)
   Imported (extern) block parameters */
extern real32_T xy_p_pdi;  /* Variable: xy_p_pdi
                             * Referenced by: '<Root>/Constant'
                             * param-data-item */
```

Listing 8 : Code snippets for PARAMETER DATA ITEM

For PARAMETER CONSTANTS, the coder settings are changed during shared code generation. An `ImportedFromFile` data scope avoids that declarations and definitions are generated with SL models [96, p. 23–7]. This shall be done during the shared code generation once. `ImportedFromFile` only places an `#include` statement into model code (cf. Listing 9).

In the shared code generation process (cf. section 8.1.2.2), the data scope is changed to `ExportToFile` [96, p. 23–7]. Embedded Coder then creates both a declaration and a definition into the shared code (cf. Listing 9).

Since `PARAMETER CONSTANT` specifies a header and source file name with module prefix during shared code generation, module parameters are kept separate from parameters of other modules. The explicit header naming in combination with correct model configuration settings also forces Embedded Coder to place the declarations in the shared code directory and make them accessible for other models.

```
/* In <module-ID>_param_constant.h (shared scope)
   Declaration for custom storage class: Const */
extern real32_T fc_p_max_ego;

/* In <module-ID>_param_constant.c (shared scope)
   Declaration for custom storage class: Const */
real32_T fc_p_max_ego = 400.0F;

/* In <modelname>_private.h> (code of consumer model)
   Includes for objects with custom storage classes. */
#include "xy_param_constant.h"
```

Listing 9 : Code snippets for PARAMETER CONSTANT (summary from different files)

5.6.4.7.4 Signals

The term *signal* in the context of SL/SF is not clearly defined. Generally spoken, signals define the data flow and data properties in a model. In a programming language, signals would map to intermediate variables. Signals may be a graphical representation or not. A line connecting two blocks in SL is always a signal, but a state of a Unit Delay block or a Data Store may be a signal, too.

Each signal can have meta data, like a name, attached test points, but also a storage class. In this section, the interest focuses particularly on the storage class. The storage class determines, how Embedded Coder transforms the signal into code.

The challenge is to manage the various ways to specify signals and to avoid ambiguities. Signal properties can be specified in separate *Simulink.Signal objects*, which are attached to signals, or directly defined in the model. Simulink.Signal objects are intended to store additional signal data and reside in the data dictionary [127, pp. 5-490ff.].

MR 37 – Signals

A. The SINGLETON SIGNAL and DATA STORE SIGNAL shall be the only used Simulink.Signal objects, specialized with the properties of Table 30.

	SINGLETON SIGNAL	DATA STORE SIGNAL
Data type	Auto	Any but not auto
Dimensions	-1	Any but not -1
Min/Max	Unset	Unset
Dimensions mode	Auto	Auto
Complexity	Auto	Real
Unit	Unset	Unset
Sample time	-1	-1
Storage Class	`ExportedGlobal`	`ExportedGlobal`
Alias	Unset	Unset
Alignment	-1	-1
Initial Value	Unset	Unset

Table 30: Signal specializations

B. Signal properties, except names, shall only be specified with Simulink.Signal objects. If attached to a signal line or port in SL, the signal shall be explicitly resolved (indicated by ⊐).

C. Naming conventions shall be applied (cf. MR 1).

DATA STORE SIGNAL objects are used for global data stores as described in MR 26. They require a Simulink.Signal with `ExportedGlobal` storage class, but also a specified data type, complexity and dimensions.

The second use case of (A), to which SINGLETON SIGNALS map, is to export narrow inter-module interfaces and form testable units. Only the `ExportedGlobal` storage class fulfills this purpose (cf. MR 6). In this case, the SL model specifies many properties of the signal already. To keep a single source of truth, a Simulink.Signal object shall have a storage class, but inherit all other properties.

(B) restricts, how Simulink.Signal objects are attached in SL models. Only a few variations are shown Figure 61.

1) On line, internally resolved to SINGLETON SIGNAL
2) On line, storage class specified in line properties
3) On line, without specified storage class
4) On state, internally resolved to SINGLETON SIGNAL
5) On Outport block, internally resolved to SINGLETON SIGNAL

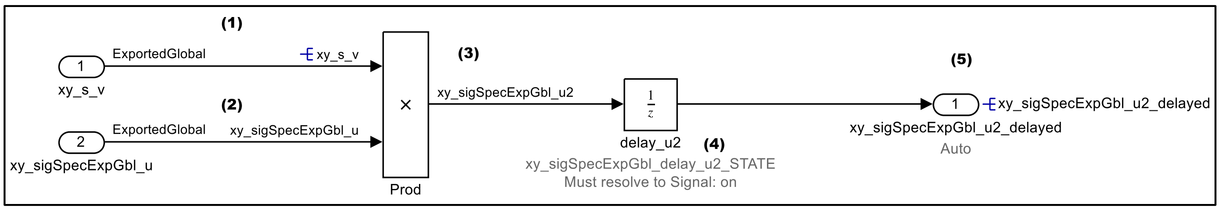


Figure 61: Simulink.Signal specifications

Here the approach is followed, that a signal name is only resolved to a Simulink.Signal object, if explicitly activated. This behavior is enforced in the configuration settings of the model [128, pp. 9-7f.]. This avoids unintended mapping. Signals 1, 4, and 5 are valid, since they resolve to a SINGLETON SIGNAL. Signal 2 is exported global, but does not resolve to a Simulink.Signal. Signal 3 does not even have a storage class.

5.6.5 Implementation of DO-178C concepts

Contribution 11: DO-178C/DO-331 concepts relevant for model-based design have been interpreted for SL/SF and constituted in rules. Consistent concepts for the definition of model elements contributing to the design, noncovered and deactivated design, algorithm correctness and the usage of model element libraries have been prepared. These concepts significantly determine, how SL/SF is used and are not covered by documentation offered by MathWorks. They lower the adoption risk of model-based design and ease discussions with authorities in a certification project.

5.6.5.1 *Quality restrictions*

MR 38 – Quality restrictions

Complexity limitations of the project shall be followed.

Quality restrictions, like complexity, coupling or cohesion measures, are subject to many other research projects and not further considered here. Olszewska [136] and Dajsuren [137] summarize industrially used complexity metrics for SL models and compare them. The most popular complexity measure is the cyclomatic or McCabe complexity. In [138], Stürmer highlights the weaknesses of McCabe complexity and adopts the Halstead Metrics of traditional software development for MBSwD. Other approaches base on structural and data complexity, like the weighted block count per layer or dependency measures like instability, abstractness, and distance [139, 140]. Mäurer [141] adopts object-oriented metrics for MBSwD additionally focusing on modularization and encapsulation.

Cohesion and coupling as defined in DR 9 and DR 10 is part of these quality metrics.

5.6.5.2 *DO-331 Model Elements not contributing to LLRs and Software Architecture*

Objectives MB 9 and 10 of DO-331 Table MB.A-2 require the identification of *Model Elements,* which do not contribute to implementation or realization of any software architecture or low-level requirement.

MR 39 – DO331 Model Elements not contributing to Low-Level Requirements and Software Architecture

A. All model primitives allowed in the safe subset contribute to LLRs or Software Architecture except
   - DocBlocks
   - Annotations
   - Virtual subsystems

B. Model elements of modules not referenced anywhere in the full model hierarchy do not contribute (cf. MR 40 and MR 41).

DO-331 does not provide further details except mentioning the example of "comment blocks". The MathWorks DO Workflow touches on a difference between virtual and non-virtual blocks, but remains vague concerning a detailed list of modeling primitives [32, p. 2-17]. The documentation defines nonvirtual blocks as blocks, which "play an active role in the simulation of a system [and] if you add or remove a nonvirtual block, you change the model's behavior" [124, pp.35-2f.]. In contrast, virtual blocks play no active role in simulation. Whether a block is virtual or not depends on both of its type and settings.

In consequence, a common assumption is thus that virtual blocks in Design Models do not contribute information to the implementation or realization of LLRs or Software Architecture. In fact, examples can be found where virtual locks provide visual help only, but many of them also have an impact on code generation. For example, a virtual Selector block, which selects a specific value from a root input for a subsequent operation, contains important information, which is afterwards found in Source Code. An Inport or Outport block of a subsystem may define the data type of a signal, although it is virtual.

Since the separation is difficult and cannot be formulated absolutely, it is stated that all modeling primitives contribute to low-level requirements and software architecture, except a few clearly arguable exceptions.

In addition, model elements of modules not referenced anywhere in the full model hierarchy do not contribute to the implementation or realization and are considered as unused model elements (cf. MR 40 and MR 41).

5.6.5.3 *Noncovered design*

DR 30 outlines the handling of noncovered design. Noncovered design is typically detected by model coverage analysis recorded during requirements-based simulation testing.

Unused design has been identified as additional contribution to noncovered design. Unused design may be unveiled in coverage analysis. For example, an unused subsystem in a `PRIVATE LIBRARY` is not detected in model or code coverage analysis. In contrast, uncalled SL models are detected by execution coverage or function call coverage.

MR 40 - Unused model elements

Unused model elements shall be identified on module-level as

A. Private SL models and model data not referenced in any other SL model of the same module

B. Subsystems in a `PRIVATE LIBRARY`, which are not used in an SL model of the same module

and on component-level as

C. Public SL models, which are not deactivatable, and never called

D. Public model data not referenced in any SL model in the integrated Design Model.

The requirements for deactivatable design from DR 31 have been further detailed for SL/SF.

MR 41 - Deactivatable model elements

A. Only public SL models as whole shall be deactivatable.

B. Deactivatable SL models shall

- reside in a module declared as “partially usable”.
- be accessible from other modules.
- be specified in and trace to higher-level requirements.
- have higher-level requirements, which are independent from project requirements (e.g., derived).
- not reference other SL models (in- or outside the module).

C. Deactivatable SL models shall not reference model data that leads to shared code, i.e., specifications of type `EXPORTED BUS`, `SAFE ENUMERATION`, or `PARAMETERS CONSTANT`.

(C) is the strongest restriction. It originates from the fact, how code is generated for these specializations. In order to share declarations and definitions across model code, they specify common header and source files, which are generated *once per module* (cf. section 8.1.2). At the time of code generation, it is not known, whether the SL models referencing these values will be deactivated design or not, so all parameters have to be transformed to Source Code. If a SL model finally becomes deactivated design, unused definitions and declarations exist in the code. This is a restriction solely introduced by generation of modular code. In contrast, if all code would be generated from the top-level model, the code generator could selectively generate or exclude declarations and definitions and no unused code would exist.

The idea of partially usable library modules is to provide utility functions. Thus, (C) is not considered as a blocking issue, since those functions typically do not rely on the mentioned specializations.

5.6.5.4 *Algorithm correctness*

MR 42 - Run-time error handling

Run-time errors shall be avoided.

Run-time errors on model-level are not critical itself. However, they always bear the risk of also appearing in the code. There, they often lead to undefined or implementation-defined behavior, which doesn't have to comply with the behavior of SL/SF.

The approach is to look at the run-time errors, which can appear in code, and find out, how they can be avoided on model-level. MISRA C:2012 Dir 4.1 distinguishes several groups of run-time errors, which are evaluated with respect to model-based design in SL/SF in the following.

Arithmetic errors – Under- and overflow

Unsigned integers perform a silent wrap and thus can never overflow, whereas the behavior of signed integer overflows is basically implementation-defined according to ANSI C99 6.3.1.3. However, implementations defining "signed integer types as also being modulo need not to detect integer overflow" (ANSI C99 H.2.2 §1). For the CompCert compiler, this is the case as defined in [57] §6.3.

This complies with the behavior of SL/SF. Although this argumentation releases integer overflows from being a run-time error in the given case, they are typically unwanted in SL/SF applications and not a part of the algorithms (for example, opposite to cryptography applications). Since they most often indicate a design flaw, they shall be avoided on both model and code level.

Floating-point overflows are run-time exceptions according to MISRA Dir 4. From a functional point of view, floating-point overflows like integer overflows indicate design flaws. Thus, they shall be avoided on both model and code level. Underflows indicate values, which are not distinguishable from zero anymore and are thus converted to zero. Underflows are uncritical from a functional point of view.

Both integer and floating-point overflows are avoided by explicit dynamic range checking on model-level or limitation of admissible signal ranges (cf. section MR 23). Fundamental modeling rules also limit block functionality to avoid overflows (e.g., extrapolation method in lookup table blocks).

SL diagnostics have been chosen to throw an error upon any simulated overflow (`IntegerOverflowMsg`, `SignalInfNanChecking`)or detected parameter under- overflow (`ParameterOverflowMsg`, `ParameterUnderflowMsg`). This reveals overflows during model simulation, however diagnostics only react, if the test cases actually trigger an overflow. Further formal analysis methods are applied to prove the absence of overflows (cf. section 8.2.5).

Arithmetic errors – Division by zero

The result of a division by zero is undefined in mathematics and according to ANSI C99 6.5.5 §5 and shall thus be avoided.

Division by zeros are avoided by explicit dynamic range checking on model-level or limitation of admissible signal ranges. If a division by zero occurs, SL diagnostics always terminates the simulation. This reveals division by zero during model simulation, however diagnostics only react, if the test cases actually trigger a division by zero. Further analysis methods are applied to prove the absence of such an event (cf. section 8.2.5).

Arithmetic errors – Array out-of-bounds

Since C internally uses pointer arithmetic to address array fields, index values beyond the bounds of the array lead to memory access violation or unexpected results.

Array out-of-bounds accesses are avoided by explicit dynamic range checking on model-level or limitation of admissible signal ranges.

If an array out-of-bounds access occurs, SL diagnostics always terminates the simulation. This reveals invalid array access during model simulation, however diagnostics only react, if the tests actually trigger such a case. Further analysis methods are applied to prove the absence of array out-of-bounds accesses (cf. section 8.2.5).

Function parameters

MISRA defines this run-time error as any call to a library function, which is outside the admissible range. For example, the behavior of the `fmod` function for the second argument equals to zero is implementation-defined (ANSI C99 7.12.10.1 §3).

All callable C library functions are robust for any input (cf. section 8.1.2.5)

Pointer arithmetic and pointer dereferencing

As example, MISRA C claims that dynamically calculated pointers must point to “somewhere meaningful” and NULL pointers are identified before dereferenced. Pointer handling errors can only be introduced by Embedded Coder, since the user has no direct control of pointers. SL does not allow explicit pointer access and modification on model-level. SF pointer operations are forbidden by MAAB jm_0001 [108, pp. 7-55ff.].

Dynamic memory

Dynamic memory allocation shall not be used in C code (cf. DR 28). Dynamic memory allocation is disabled in SL in configuration settings by not using variable size signals `SupportVariableSizeSignals` and a disabled dynamic memory allocation at initialization (`GenerateAllocFcn`).

MISRA C Dir 4.12 forbids dynamic memory allocation with the C Standard library or any third-party implementation.

Model checks detect any deviation from the chosen configuration settings. Calls to the dynamic memory allocation functions are found, when checking code compliance to MISRA C.

5.6.5.5 *DO-331 Model Element Libraries*

A known development challenge along a software process are libraries. Main intention of library functionality is reuse and distributed, independent development. In consequence, finally ensuring consistent traceability, completeness of software life cycle data, as well as consistent configuration management requires special attention. Objective of this section is to identify DO-331 *Model Element Libraries* in the MBSwD process at hand. Both DO-178C and DO-331 impose additional requirements on libraries.

MR 43 – DO-331 Model Element Libraries

The following design constructs in SL/SF shall be considered as Model Element Libraries and underlie the respective requirements of the standards:

- Partially usable module library
- Pseudo primitives in the DO-331 Foundation Library

DO-178C defines *Software Library* as a "controlled repository containing a collection of software and related data and documents designed to aid in software development, use, or modification. (DO-178C p. 116)". The term is mentioned a few times, but further guidance is not given.

DO-331 introduces the definition of a *Model Element Library* as "a collection of elements used as a baseline to construct a model. A model may or may not be developed using Model Element Libraries (DO-331 p. 82)". Discussion Paper DP#2 (MB.B.18) "Information on the usage of libraries in a Model-Based Development and Verification process" of DO-331 addresses the topic more intensively and formulates various requirements to be fulfilled.

Anyway, it is still controversial, what exactly a Model Element Library in MBSwD is. For example, is the set of atomic primitives considered as Model Element Library or does it belong to the modeling language? If it is a library, how can a model not use Model Element Libraries?

In both definitions given by DO178C and DO-331, configuration management plays an important role by mentioning a "controlled repository" and a "baseline". Furthermore, DP#2 distinguishes between library developers and users and the fact that libraries "are frequently developed by third parties, such as tool vendors, departments other than the one using the library". DO-331 MB.B.18.2 also states that "some libraries are directly developed in a programming language and others are developed in a modeling language."

In consequence, it is defined that if the following two conditions are met, software must be handled as library:

1. The software must have been developed in an independent life-cycle.
2. The software is partially usable. It is not required to achieve full coverage of all library functions in the product software.

For the presented modular approach, three types of library candidates can be identified and play a role in the process (cf. Table 31). They are treated as described in Table 31 and are discussed in the following.

	Independent Development Cycle	**Independent Verification Cycle**	**Partially Usable**	**Library Type (DO-178C/DO-331)**	**Reference**
Partially Usable Module Libraries	X (if in a separate module)	X (partially if in a separate module)	X	Model Element Library and Software Library	MR 9
DO-331 Foundation Library	X	X	X	Model Element Library (partially)	MR 27
Private Libraries				n/a	MR 21

Table 31: Model Element Library assessment in the MBSwD process

Partially usable modules libraries

Partially usable module libraries have been introduced in DR 6. These are modules developed in another process/project and are partially usable. They contain publically accessible SL models (Model Element Library), but also generated and (partially) verified Source Code (Software Library). Both requirements are met on model and code level, so they are considered as Software and Model Element Libraries.

Private libraries

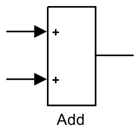
PRIVATE LIBRARIES have been introduced in MR 21. The specialization restricts library use to a single module, it is developed and used in the same controlled environment and by the same person. Verification is done where the patterns are instantiated and fully integrated into model verification. They are not partially usable (cf. MR 40). They are neither Software nor Model Element Libraries.

DO-331 Foundation Library

The DO-331 Foundation Library, which has been further explained in MR 27, is a single SL library included in the modeling environment providing all usable atomic primitives. It follows the Software Environment Life Cycle. The same library is available and must be identical in all modules, which are integrated into each other.

It is quite controversial, whether it is a Model Element Library or belongs to the modeling language. To make a more nuanced point, simple, complex, and pseudo-primitives should be distinguished.

Simple primitives are atomic blocks directly connected to a low-level implementation in SL. During code generation, they expand to a limited block of C code, which is inlined in the code of the higher context (Figure 62). Most of the primitives are unchangeable atomic elements of the current SL release and undoubtedly part of the modeling language.

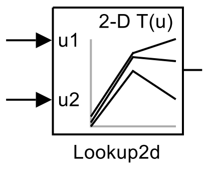


```
/* Sum: '<Root>/Add' incorporates:
 *  Inport: '<Root>/In1'
 */
rtb_Add = xy_model_U.In1 + xy_model_U.In1;
```

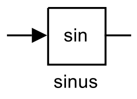
Figure 62: Simple primitive

Complex primitives are atomic blocks connected to a low-level implementation and expand to complex shared C functions during code generation or call non-generated C functions (e.g., from the Standard C Library or a Code Replacement Library). Typical examples are the `LookupTable` and `TrigonometricFunction` blocks (Figure 63). The first generates a shared utility function `look2binlca`, the second uses the `cert_sin` function of the Cert Standard C Library. Also blocks created from legacy C code (cf. section 8.1.2.5) extend the set of built-in primitives.

The blocks are unchangeable, atomic elements of the modeling language, but the called C functions are exchangeable and must developed under DO-178C as Software Library.



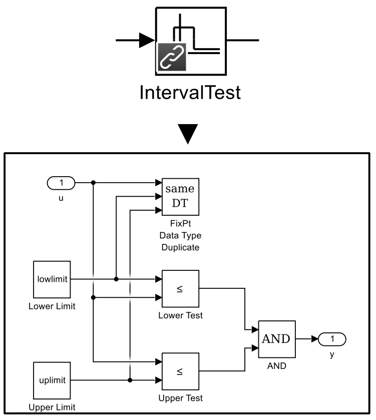
```
b_Lookup2d = look2_binlca(xy_model_U.In1,
  xy_model_U.In2,
  xy_model_ConstP.pooled1,
  xy_model_ConstP.pooled1,
  xy_model_ConstP.Lookup2d_tableData,
  xy_model_ConstP.Lookup2d_maxIndex,
  3U);
```



```
/* Trigonometry: '<Root>/sinus' incorporates:
 *  Inport: '<Root>/In1'
 */
b_sinus = cert_sin(xy_model_U.In1);
```

Figure 63: Complex primitives

Pseudo-primitives are blocks, which appear atomic, but actually are subsystems with nested blocks, e.g., the Interval Test "block" (Figure 64). Various pseudo primitives are provided by MathWorks ("built-in"), but it was also necessary to add own in some situations.



```
/* RelationalOperator: '<S1>/Lower Test' incorporates:
 *  Constant: '<S1>/Lower Limit'
 *  Inport: '<Root>/In1'
 */
rtb_LowerTest = ((-0.5) <= xy_model_U.In1);

/* RelationalOperator: '<S1>/Upper Test' incorporates:
 *  Constant: '<S1>/Upper Limit'
 *  Inport: '<Root>/In1'
 */
rtb_UpperTest = (xy_model_U.In1 <= 0.5);

/* Logic: '<S1>/AND' */
rtb_AND = (b_LowerTest && b_UpperTest);
```

Figure 64: Pseudo-primitives

Simple and complex primitives are considered as part of the modeling language, but pseudo primitives are not considered as part of the modeling language in the process at hand, although partially shipped by MathWorks and verified in the context. At least for these blocks, guidelines for a Model Element Library have to be followed. Rational is that pseudo primitives are independently developed and often contain complex functionality, which should base on requirements and thoroughly be verified on model-level before distributed. Any change to a pseudo primitive has significant influence on the behavior of models.

Anyway, this conservative decision is controversial and should be clarified with the authority. One could argue, that pseudo primitives are also pure design patterns fully verified in the context they are instantiated. Size and complexity of the nested block diagram in pseudo primitives and the capability of verification tools to traverse into linked subsystems and include their content into verification should be considered as criteria for the decision. Also the action performed when copying a pseudo primitive from the DO-331 Foundation Library may influence the viewpoint. For example, the SL library (`slcilib`) shipped with SLCI removes the link between the subsystem and the library as soon as placed in a model. Then, the subsystem is an unlinked, pure copy, which does not update with library changes anymore.

The additional objectives to be considered for a Model Element Library are listed in DP #2. The MBSwD process at hand covers the objectives for partially usable module libraries, since any module has similarities with libraries. Due to their more independent character and usage across projects, additional effort is necessary to document operational requirements. The symbols of library elements are predefined by the symbols of model reference blocks. Partial use is covered by the MBSwD (for example, DR 6, DR 31, or MR 41).

The process for the DO-331 Foundation Library needs further considerations, since SL libraries need extensions and deviations to the modeling rules, have to be separately verified, need separate requirements and have additional considerations concerning process coverage and reuse. This process is not further regarded in this work.

5.7 Fundamental modeling rules

Contribution 12: A consistent set of modeling rules as been condensed from existing guidelines sets. The rules address detailed settings of model elements. Various own, custom rules have additionally been added. These rules target standardized use of settings beyond the pure safety aspects.

The fundamental rules consist of a subset of the high-integrity guidelines [103], the guidelines for code generation [104], and the MAAB guidelines [108] as well as custom guidelines.

In fundamental rules, low-level block and configuration settings are restricted. For example, “hisl_0001 – Usage of abs block” [103, p. 2-2] says:

a. “Avoid Boolean and unsigned integer data types as inputs to the Abs block.”
b. “In the Abs block parameter dialog box, select ‘Saturate on integer overflow’”.

Since the sets have been independently developed, significant overlap and also contradictions exist. Furthermore, many MAAB guidelines are outdated and almost not applicable. Significant effort was necessary to condense them.

The new custom rules standardize settings throughout the models. For example, the integer rounding mode or the use of automatic overflow saturation, which almost every block provides. Another large part predefines configuration settings. Only a few important settings are settled by the existing guideline sets. For all others, a project-specific value had to be chosen and rationales had to be documented.

5.8 Modeling environment

Contribution 13: A consistent modeling environment has been assembled. The modeling environment comprises all resources to setup SL/SF in the light of the given design, modeling, traceability, and coding rules. As distributable package, they allow any developer to quickly turn SL/SF in the controlled environment necessary to implement Design Models.

The S*oftware Life Cycle Environment* "defines the tools, methods, procedures, programming languages, and hardware that will be used to develop, verify, control, and produce the software life cycle data (DO-178C 4.4)" and is included in the Software Development Plan in the Planning Process (objective DO-178C Table A-1:3). It describes the following:

- Software Development Environment (DO-178C 11.2c, 4.4.1)
- Language and Compiler Considerations (DO-178C 4.4.2)
- Test Environment (DO-178C 4.4.3)
- Simulation Environment (DO-331 MB.4.4.4)

Tools specified as part of the Software Life Cycle Environment are subject to *Software Life Cycle Environment Control*, which is a configuration management action (DO-178C Table A-8:6 and DO-178C 7.5). The connected software configuration management data is the *Software Life Cycle Environment Configuration Index* (DO-178C 11.15). According to DO-178C 7.5a, "configuration identification" is required as minimum.

The modeling environment defines the Software Development Environment and Simulation Environment for the Design and Coding Process to a high degree. The *modeling environment* is "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" [36, p. 70]. It must underlie a strict configuration management process [142]. Changes must be replicable at any time.

The content of the modeling environment has been presented by Hochstrasser [36], but evolved over time and had to be adapted for the modular process. Figure 65 provides an updated overview. Essential part are the modeling rules, naming conventions, and configuration settings constraining the modeling language as presented in the previous sections.

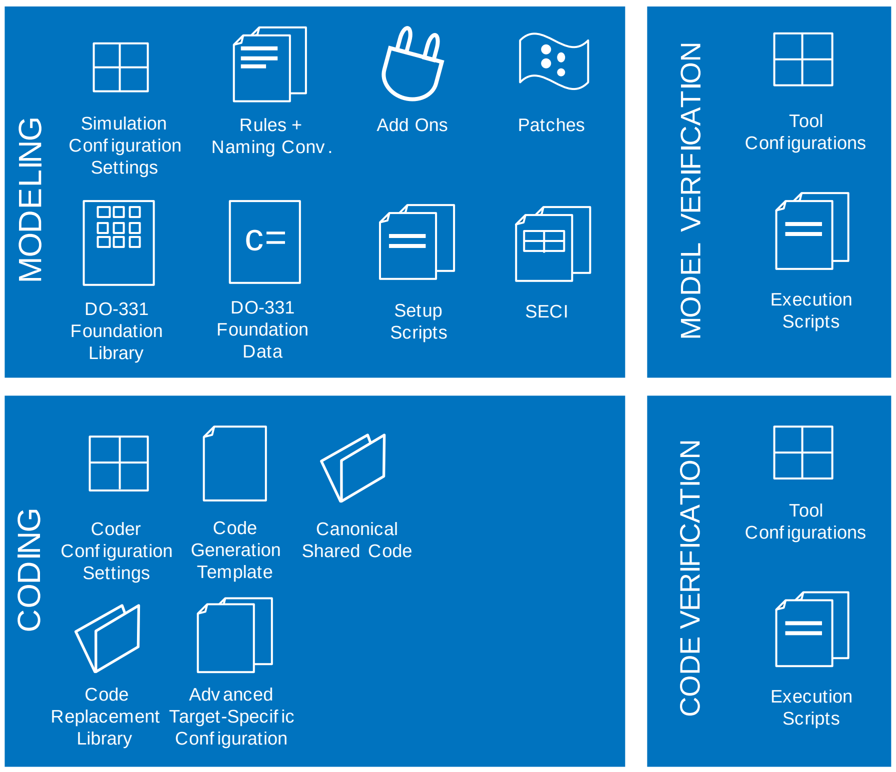


Figure 65: Content of modeling environment based on [36]

The simulation and configuration settings are the three sets introduced in MR 15. They are part of a general configuration SL data dictionary.

AddOns are little programs, which extend the functionality of SL/SF. On the one hand, this is the traceability tool *SimPol* presented in section 6.4, on the other hand the MinGW compiler used by SL/SF to compile simulation targets. The compiler has been chosen, since it is, especially for SIL simulations (cf. section 8.2.15), closer to the used cross-compiler CompCert for Executable Object Code.

Beside the DO-331 Foundation Library, which has been introduced in MR 27, commonly used constants (e.g., physical constants) have been added to the modeling environment, in Figure 65 entitled *DO-331 Foundation Data*. This is the only model data, which can, but does not have to be used in a project, since model data shall not be deactivated design according to section MR 41. The constants are designed to be inlined in code.

The Software Life Cycle Environment Configuration Index is basically a table with all tool versions. It is checked when initializing the environment.

The coding artifacts as well as the scripts and tool configurations for verification are discussed with the respective tasks in section 8.1.2.

5.9 Summary and outlook

With the modeling framework, a systematic approach to derive design, modeling, and code rules has been introduced in section 5.3 and applied in subsequent sections.

In a first step, a complete set of tool-independent design rules bridging the gap between abstract requirements of the process standard and detailed settings of modeling rules has been created. Amongst others, the design rules introduced a concept of modularization in section 5.4.4 by defining modules, units, or library modules. Special attention has been paid on interface specifications in sections 5.4.4.3 and 5.4.4.4, which are the basis for formal and classical modular verification. Also the development of Parameter Data Items has been planned in detail in section 5.4.4.6.

Additionally, coding rules for auto-generated code have been written for the chosen code generator, the selected compiler tool chain around CompCert, and the state-of-the-art coding standard MISRA C. The rule set is significantly smaller than coding guidelines applicable for hand-written code, since the requirements for code generators are less strict, e.g., it is assumed that they handle global variables much better and in a more organized way than a human programmer would, but also since no guidance for programmers is necessary.

Both design and coding rules served as basis for module design and fundamental modeling rules specifically applicable for SL and SF in section 5.6. The module design rules extended traditional modeling rule sets with architectural considerations, like the definition of modules and units, or the exact specification of component and module interfaces in section 5.6.4.4. Additionally, rules for the application of container primitives like SL models or subsystems (section 5.6.4.2) have been assessed in the light of the given process context and specified. To constrain the detailed design, a new approach has been followed. Instead of “blacklisting” features that are not compliant or safe in SL/SF, permitted features have been “whitelisted”, e.g., the safe specializations for model data in section 5.6.4.7 or the DO-331 Foundation Library for atomic blocks in section 5.6.4.5. Finally, rules have been created that address topics imposed by the standards, like noncovered design or additional considerations for so-called Model Element Libraries in section 5.6.5.

A considerable amount of complexity in the rules is subject to modularization. Scalability is only provided and agile work in different teams only possible, if the interfaces are clear and robust. Converting the higher-level design rules into the SL/SF environment revealed several gaps in the tool chain with respect to these topics. Best example is encapsulation. SL does not know a concept for encapsulation for SL models or subsystems. Also encapsulation of model data is cumbersome.

In addition, the rules show that the intersection of features drastically diminishes, if compatibility throughout the whole tool chain is required and the workflow deviates from integral code generation. Carving out this subset is connected to time-consuming trial-and-error. On the opposite, the fact that for almost every situation a workaround could be found, demonstrates the flexibility of the tools.

The compatibility requirements with the tool chain resulted in a very “model-centric” workflow. The concept of model references is preferred over library linking. As pointed out in the discussion of section 5.6.4.2, several disadvantages exist in this case, too, like performance or tool limitations. This drawback has been realized by MathWorks as well. New concepts for library subsystems in later releases promise alternative workflows and should be investigated to combine the best of both SL models and SL libraries. All guidelines base on SL release R2017b. In subsequent work, concepts and rules should be assessed in the light of enhanced features of new releases. Certainly some limitations are not necessary anymore.

In summary, a holistic, adaptable modeling framework has been created with rule sets covering many aspects of safety-critical model-based software development and guaranteeing consistency as well as a high level of compatibility with development and verification tasks. Although the rules constrain the features significantly, they either offer a valuable “out-of-the-box” solution or profound foundation for many controller development applications.

6 Traceability tooling and rules

6.1 Objectives

Gotel et al. describe traceability as “the potential to relate data that is stored within artifacts of some kind, along with the ability to examine this relationship. […] The value of traceability lies in the many software and systems engineering activities and tasks that the information provided through such interrelations can enable, such as change impact analysis, coverage analysis, dependency analysis etc.” [143, p. 3]. DO-178C requires extensive traceability and it is a core evidence for showing process compliance to authorities. However, establishing and managing traceability is a significant effort without direct benefit. In consequence, trace data does not get the required attention and is often incomplete or does not provide the necessary granularity.

This section introduces, how traceability is established in the process at hand and presents formulated rules. Since requirements are managed in Polarion, traceability must be established between Polarion work items and model elements in the SL/SF environment. Bridging this gap is thus an important objective.

6.2 State-of-the-art

Traceability has been a research topic for many years. However, the industry still suffers from traceability gaps. Mäder et al. point out the importance of traceability in safety critical projects, but also list adoption problems reaching from invisibility of the benefit for developers to granularity issues [144]. Rempel [145] assesses the need for an explicitly defined traceability strategy. In the empirical study, several projects have been analyzed and show traceability gaps. In their recent study, Salome et al. [146] provide a summary of the traceability challenges in the automotive domain. In the areas of tool support, human factors, organization and process, as well as exchange of traceability, a variety of gaps and inefficiencies could be identified. Overall, the excessive manual work to create and maintain traceability throughout the development life cycle is time-consuming and does not appear beneficial to developers. Also, the lack of flexible tools is mentioned as challenge, which is rarely addressed in literature.

Especially, if the linked artifacts are of a different type, managing traceability is far more challenging. Even traceability between textual requirements and Source Code, which is applied for decades, is still unsatisfactorily solved and topic of current research [147]. According to Paz [69], “traceability appears as fairly unattended” in many MBSwD approaches, especially concerning coverage analysis.

Different strategies are under research depending on the type of the tracing activity. Gotel et al. [143] distinguish *manual tracing activities*, where a human tracer establishes the trace links, as well as *automatic and semi-automatic tracing activities*. Automatic means that the trace link is automatically generated, whereas semi-automatic implies the involvement of a human in some way. In MBSwD, both manual tracing and automatic tracing is necessary. In automatic code generation with Embedded Coder, traceability from code to design is automatically created, whereas the traceability from design to textual requirements must be manually established.

In particular, manual tracing activities raise difficulties. In current research, manual tracing is supported with Information Retrieval (IR) or Machine Learning [148], leading to semi-automatic tracing activities. However, Salome et al. [146] remark skepticism with respect to these approaches due to the high "chance that incorrect links are generated or links are missing". The interaction between automated traceability capturing and manual post-processing is also controversially discussed in [149, 150]. As more promising approach, smart maintenance capabilities are proposed by [146]. Different research tries to overcome the human factors concerning traceability and the lack of motivation with gamification approaches [151].

Some research projects address post-creation assessment of trace links. For example, Rempel describes an approach, in which a guideline traceability model is derived from standards, the project traceability is recovered, and compared with this model [14, 152].

In the context of the process at hand, traceability had to be maintained between the server-based requirements management Polarion and SL/SF. It spans multiple tools, with different APIs and different workflows. With the *Polarion Connector for Simulink* plugin (version 2.3, March 2019)[23], Polarion only offers a lightweight solution to establish traces between Polarion work items and SL/SF models based on the Simulink Requirement Management Interface (RMI [153, pp.5-2ff.]) leaving many questions, like change or configuration management, unanswered. It only addresses creation of bidirectional traces. Allocation of requirements is not addressed. No possibility exists to detect unidirectional, broken, or invalid links and to repair them. Furthermore, the linking workflow is not user-friendly requiring stepping through various context menus and dialogs for a single link. The linking is also limited to SL/SF, data in data dictionaries or SL test cases cannot be linked. Impact analysis or requirements coverage are manual workflows and thus traceability becomes an end in itself.

In release 2017b, MathWorks introduced new requirement management capabilities with SL Requirements [153], but an integrated solution to link Polarion work items was not included. In addition, MathWorks follows a different workflow in current releases. Starting with release 2018a[24], it is proposed to export Polarion work item into the ReqIF file format [154] and importing them into SL Requirements. However, this approach is not considered as bidirectional and buries synchronization challenges.

[23] http://extensions.polarion.com/extensions/173-polarion-connector-for-simulink [Accessed on: Sep. 07 2019]

[24] https://de.mathworks.com/help/releases/R2018a/slrequirements/ug/import-requirements-from-third-party-tools.html [Accessed on: Sep. 07 2019]

Beside those two approaches, no further tool is known to the author for traceability between Polarion and SL/SF. In the work of this thesis, a standalone tool has been implemented in MATLAB adding the important, missing features and allowing smooth traceability between Polarion and SL/SF as well as verification tools.

Beside the tooling, measures for required trace granularity from HLRs to elements of the Design Model is a very unaddressed topic. The question, which model elements have to be linked, and how many elements of a Design Model can be covered by a single requirement is undefined. Cleland-Huang et al. [46] note that standards like DO-178C “fail to […] specify the granularity of the links and the optimal trace path (i.e., the path through the graph from source to target artifacts).” A case study performed in [144] lists a not clearly defined trace granularity as one of the major problems of traceability in safety critical software projects and also Rierson emphasizes the importance of a consistent granularity [45, p. 121].

6.3 Structure

Section 6.4 describes the novel tool *SimPol*, which allows scalable requirement linking and implements solutions for the traceability workflows outlined in DR 24. The set of features reaches beyond capabilities of existing tools.

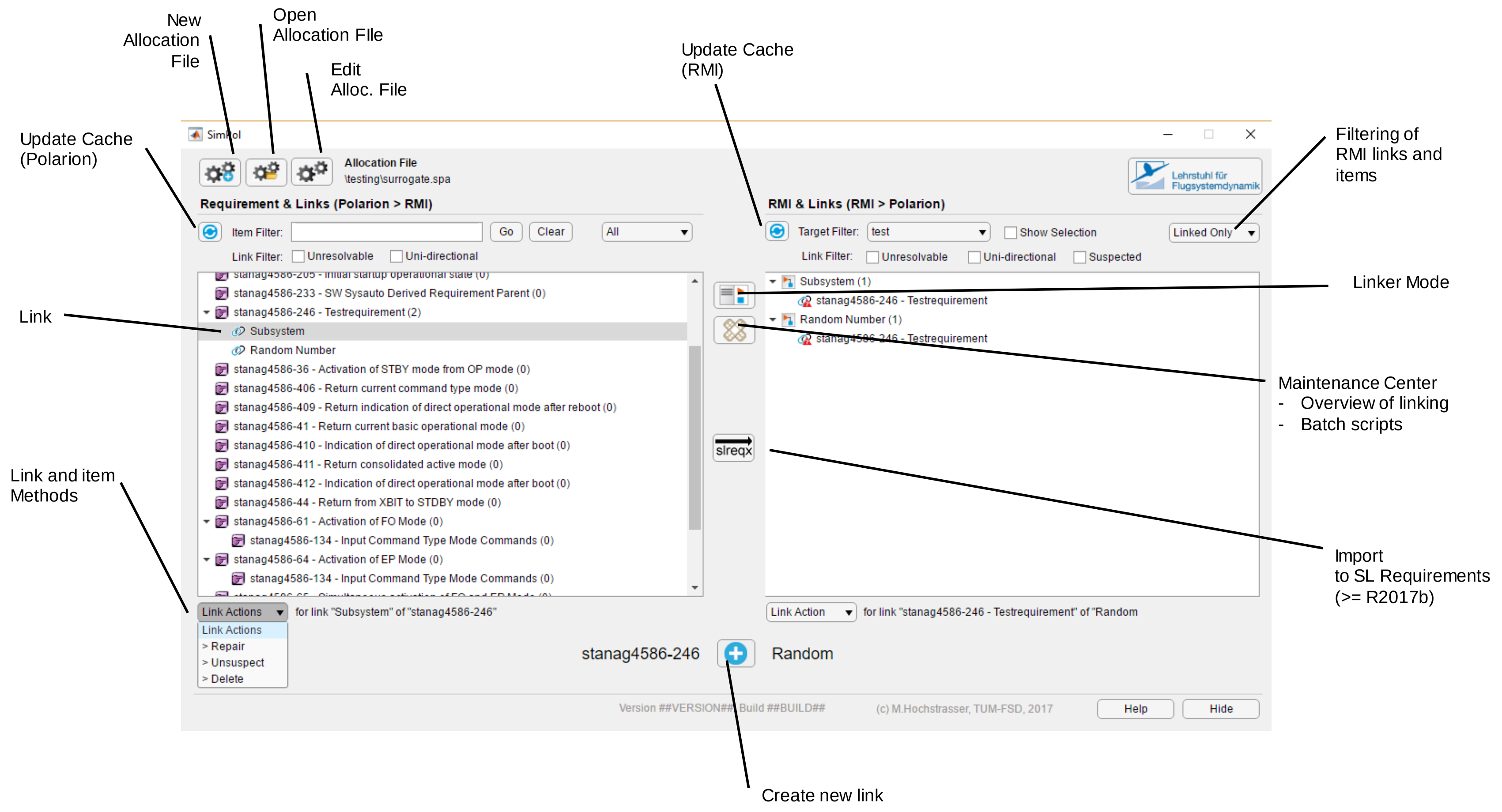
Sections 6.5.1 and 6.5.4 discuss established traceability rules for the presented process. Section 6.5.5 specifically addresses derived LLRs and demonstrates a new way of documenting them with *SimPol*, which perfectly fits into existing processes.

6.4 SimPol

Contribution 14: The author has developed a new, publically available tool called *SimPol* to manage traceability between Polarion and SL artifacts. Compared to existing solutions, *SimPol* supports all artifacts in SL relevant for the process (SL models, SL tests, SL data dictionaries) by an extendable, pluggable software architecture. In addition, the management effort is reduced by loading bidirectional traces into an integrated, abstract data model, which leverages automatic identification of missing, corrupted, or outdated traces and their resolution, bending of trace links to other artifact revisions, as well as impact analysis.

SimPol is a MATLAB application written by the author. It provides a bunch of features to manage traceability. Figure 66 shows a screenshot of the *SimPol* user interface. The window is divided into a Polarion side (left), and a SL/SF side (right). The Polarion side lists allocated Polarion work items and incoming links, whereas the SL/SF side summarizes model elements and their outgoing links. The divided view allows identification of dangling links and unlinked items.

For the actual linking workflow, the right-hand side can be hidden, and Polarion work items can be shown side-by side with a SL model.



stanag4586-205 - Initial startup operational state (0)
stanag4586-233 - SW Sysauto Derived Requirement Parent (0)
stanag4586-246 - Testrequirement (2)
Subsystem
Random Number
stanag4586-36 - Activation of STBY mode from OP mode (0)
stanag4586-406 - Return current command type mode (0)
stanag4586-409 - Return indication of direct operational mode after reboot (0)
stanag4586-41 - Return current basic operational mode (0)
stanag4586-410 - Indication of direct operational mode after boot (0)
stanag4586-411 - Return consolidated active mode (0)
stanag4586-412 - Indication of direct operational mode after boot (0)
stanag4586-44 - Return from XBIT to STDBY mode (0)
stanag4586-61 - Activation of FO Mode (0)
stanag4586-134 - Input Command Type Mode Commands (0)
stanag4586-64 - Activation of EP Mode (0)
stanag4586-134 - Input Command Type Mode Commands (0)

for link "Subsystem" of "stanag4586-246"

Subsystem (1)
stanag4586-246 - Testrequirement
Random Number (1)
stanag4586-246 - Testrequirement

for link "stanag4586-246 - Testrequirement" of "Random

Version ##VERSION## Build ##BUILD##
(c) M.Hochstrasser, TUM-FSD, 2017

Figure 66: Screenshot of *SimPol*

On Polarion side, the tool uses the provided Java Web Service API[25]. In MATLAB, it exchanges data with the Requirement Management Interface (RMI) (cf. Figure 67). The RMI is the common gateway used across various MATLAB toolboxes to attach requirement links to various types of elements (model elements, test cases,…) without altering the content of the artifact itself.

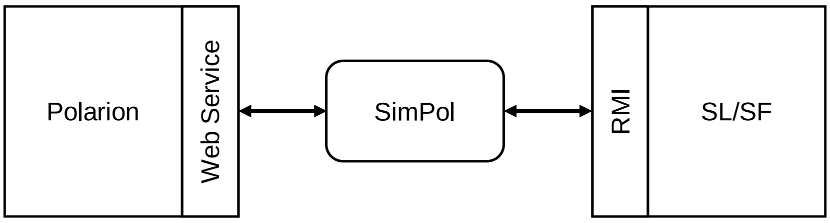


Figure 67: Basic *SimPol* infrastructure

Requirement allocation with *SimPol*

Central element of the *SimPol* workflow, which also distinguishes it from other approaches, is the allocation step at the beginning. The user has to select the work items in Polarion as well as the components in MATLAB, which he plans to link. Work items are container elements that can represent any kind of information in Polarion, here typically requirements. The selection is stored in a so-called *Allocation File* and must be loaded prior to any linking. The subset of allocated work items is specified by a flexible query string, which supports all query capabilities of Polarion and any work item type. On MATLAB side, file artifacts must be chosen. Supported are SL models, SL test files, and SL data dictionaries. The process is illustrated in Figure 68.

[25] https://almdemo.polarion.com/polarion/sdk/doc/javadoc/com/polarion/alm/ws/client/WebServiceFactory.html

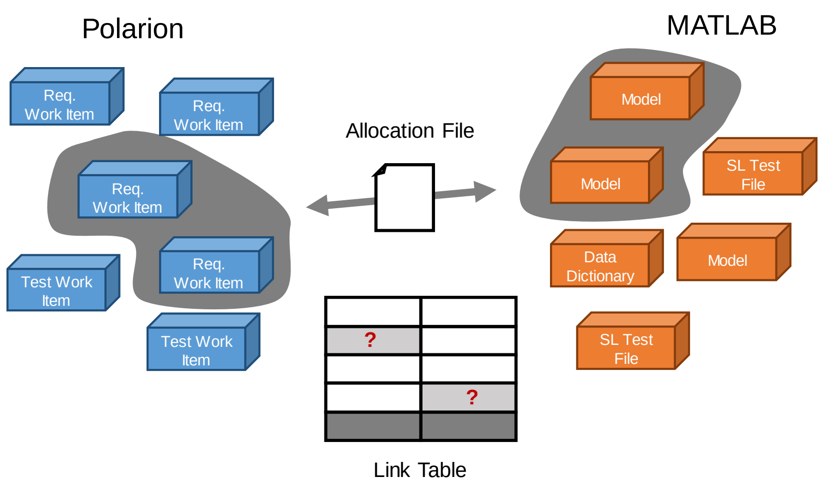


Figure 68: *SimPol* allocation process

Knowing the cohesive set on both sides enables the tool to determine unlinked elements, calculate coverage, and decide whether links are dangling or unidirectional. Therefore, *SimPol* internally builds a Link Table as shown in Figure 68 in an exemplary way. The allocation and precise selection of work items leverages a modular process and splitting requirements amongst modules. Allocation files can be created in advance and be provided to the respective modules.

Link realization and storage with *SimPol*

To create links, elements must be uniquely identifiable in both Polarion and MATLAB. In Polarion, every work item has a unique ID composed of the project name and an immutable, running number (e.g., `proj-75`). Work items are accessible via a Unique Resource Identifier (URI), e.g., `https://.../polarion/#/project/proj/workitem?id=proj-75`.

In MATLAB, different concepts are used. SL/SF model elements provide a Simulink ID (SID) [124, p, 1-18], SL Test works with a Universally Unique Identifier (UUID)[26], and SL data dictionary items with a unique, global name.

In the simplest case, links directly point to the unique ID of the linked item. Due to their bidirectional character, links are created and stored in both Polarion and MATLAB. The RMI link holds the URI of the work item, and the Polarion work item the unique ID of the element in MATLAB (as hyperlink added to the work item). The trace is in consequence navigable in both directions and each side holds the full trace information.

However, it is good practice that traces are separately stored from the linked artifacts, so that they don’t change the revision. For example, HLRs are often already verified and baselined before they are implemented. If any established link increased the revision of the work item, it would disrupt configuration management.

[26] http://www.boost.org/doc/libs/1_65_0/libs/uuid/

The RMI provides features to store links independently of the model (cf. section 6.5.1). However, hyperlinks written into Polarion work items always modify the revision.

Solution is the so-called *Surrogate Linking.* The method inserts a *surrogate work item* created in Polarion in-between the original Polarion work item and MATLAB elements. Surrogate work items mirror the actual linked MATLAB elements. The approach is outlined in Figure 69. Title of the surrogate work item is the unique ID of the respective MATLAB element, in this case the SID of the model element.

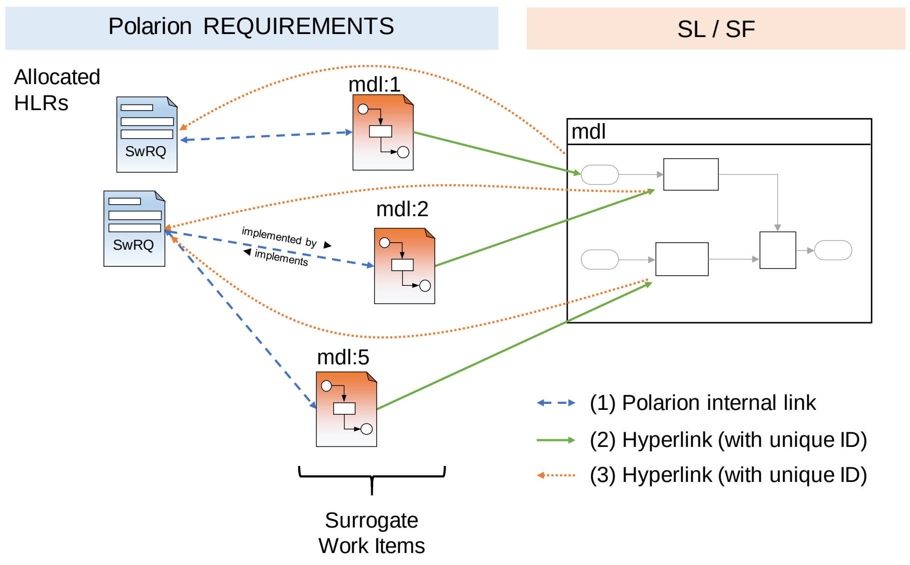
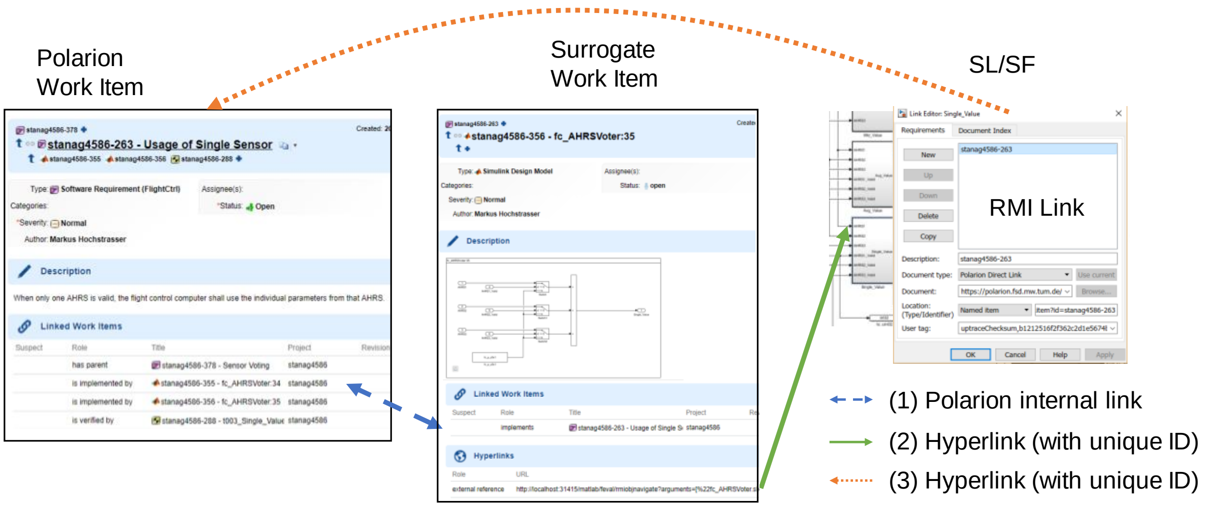


Figure 69: Surrogate Linking

With a surrogate work item, the bidirectional trace consists of three parts as highlighted in the example of Figure 70:

1. A bidirectional work item link in Polarion of type “implements” from Surrogate Work Item to HLR work item.
2. An unidirectional hyperlink in the surrogate work item pointing to the model primitive and containing its SID.
3. An unidirectional work item ID and a hyperlink wrapped in a RMI link of type `Polarion Link` attached to a model.

Links from surrogate work items to HLR work items (1) do not impact the revision of HLR work items, since the link information is only stored in the source work item. The reverse link is implicit. The directly placed or modified hyperlink to SL/SF (2) only impacts the revision index of the surrogate work item.


Polarion Work Item
Surrogate Work Item
SL/SF
RMI Link
(1) Polarion internal link
(2) Hyperlink (with unique ID)
(3) Hyperlink (with unique ID)

stanag4586-378
stanag4586-263 - Usage of Single Sensor
stanag4586-355 stanag4586-356 stanag4586-288
Created: 2

Type: Software Requirement (FlightCtrl)
Assignee(s):
Categories:
Status: Open
Severity: Normal
Author: Markus Hochstrasser

Description

When only one AHRS is valid, the flight control computer shall use the individual parameters from that AHRS.

Linked Work Items

Suspect	Role	Title	Project	Revision
	has parent	stanag4586-378 - Sensor Voting	stanag4586	
	is implemented by	stanag4586-355 - fc_AHRSVoter:34	stanag4586	
	is implemented by	stanag4586-356 - fc_AHRSVoter:35	stanag4586	
	is verified by	stanag4586-288 - t003_Single_Value	stanag4586	

stanag4586-263
stanag4586-356 - fc_AHRSVoter:35
Create

Type: Simulink Design Model
Assignee(s):
Categories:
Status: open
Severity: Normal
Author: Markus Hochstrasser

Description

Linked Work Items

Suspect	Role	Title	Project	Re
	implements	stanag4586-263 - Usage of Single S	stanag4586	

Hyperlinks

Role	URL
external reference	http://localhost:31415/matlab/feval/rmiobjnavigate?arguments=[%22fc_AHRSVoter.s

Link Editor: Single_Value
Requirements | Document Index
stanag4586-263
Description: stanag4586-263
Document type: Polarion Direct Link
Document: https://polarion.fsd.mw.tum.de/
Location: (Type/Identifier) Named item item?id=stanag4586-263
User tag: uptraceChecksum,b1212516f2f362c2d1e5674
New | Up | Down | Delete | Copy | Use current | Browse... | OK | Cancel | Help | Apply

Figure 70: Trace realization (Screenshots)

Having the model parts in a requirement tool also allows the adoption of Polarion features like adding an artifact status to work items, handing them over according to a defined process, attaching them to tasks, or assigning them to user roles. The embedded image provides a quick view of the model or subsystem without opening SL/SF.

This surrogate model approach is not new and already featured by the RMI for IBM Doors [155, pp. 7-13ff]. Also the *Polarion Connector for Simulink* plugin[27] supports the creation of surrogate items. However, it cannot handle the unneglectable disadvantage of the approach, which is an increased effort in checking the validity of links and synchronizing the surrogate work items. In contrast, *SimPol* can detect dangling links across the surrogate work items, update their content upon changes, and identify obsolete items. These are capabilities, which are necessary to successfully apply Surrogate Linking.

Configuration Management and *SimPol*

Another challenge related to configuration management addresses versioning and restoration. In reality, work items as well as MATLAB elements are not only identified by their unique ID, but also by their revision in the respective version control system. For example, a HLR may evolve over time and even be deleted. Different states are saved in different revisions. Ultimately, a bidirectional link has to address a revision on both sides, too.

Here, MATLAB elements are committed to the GIT version control system. Every commit is a revision with a unique ID. Revisions can be labeled with so-called *tags* [156, pp. 54ff]. GIT can handle different, parallel development branches, into which changes can be committed. “Branching means you diverge from the main line of development and continue to do work without messing with that main line” [156, p. 62]. The latest commit of each branch is called the “HEAD” commit.

[27] http://extensions.polarion.com/extensions/173-polarion-connector-for-simulink

Also Polarion (release 2015) bases on a version control system. Each change in any work item updates the revision of the underlying version control system. Changes can only be made to the latest revision ("HEAD"). Working with different branches is not supported in the used release. However, the latest revision can be baselined at any time under a custom tag. Baselines can be accessed at any time, but only in a read-only mode.

If an element in MATLAB is linked to a Polarion work item, it can either be linked to the HEAD in Polarion, which always points to the latest revision, or to a baseline, which points to a fix revision in the past. The information, where it points to, could be directly stored in RMI links. A slightly different URI references work items in baselines, e.g., `https://.../polarion/#/`**`baseline/82`**`/project/proj/workitem?id=proj-75`. However, storing revision information in RMI links requires an update of all links each time a new baseline is linked. This taints the link files unnecessarily, since during smaller changes, the majority of links remains untouched. Instead, *SimPol* stores the revision in the Allocation File. The RMI link only holds the relative path. The baseline, to which links direct, can be changed by only modifying the Allocation File. Broken links due to missing work items in the new revision or new unlinked work items are indicated in *SimPol*.

In the opposite direction, i.e., in hyperlinks placed in Polarion work items, a GIT revision is currently not automatically encoded. Since GIT is not a server-based application, revisions must be checked out on the file system. The revision information must thus be stored by other means.

Another problem exists on Polarion side. In a perfect workflow, requirements are baselined. These requirements are passed to the developer, which implements the design and creates links to the baselined requirements. However, baselines in Polarion are read-only and linking to a baseline is not possible.

The clean solution would be using different, linked Polarion projects (one for requirements and one for the surrogate model), but *SimPol* does currently not support cross-project linking.

Hence, another pragmatic approach has been chosen. The HLR project in Polarion and the SW Design in GIT are considered as one configuration item as depicted in Figure 71. This is reasonable, since both are modified by the same developer, too. The HLR project in Polarion references other, higher-level Polarion requirement projects (e.g., with system requirements). The Allocation File references the HLR project.

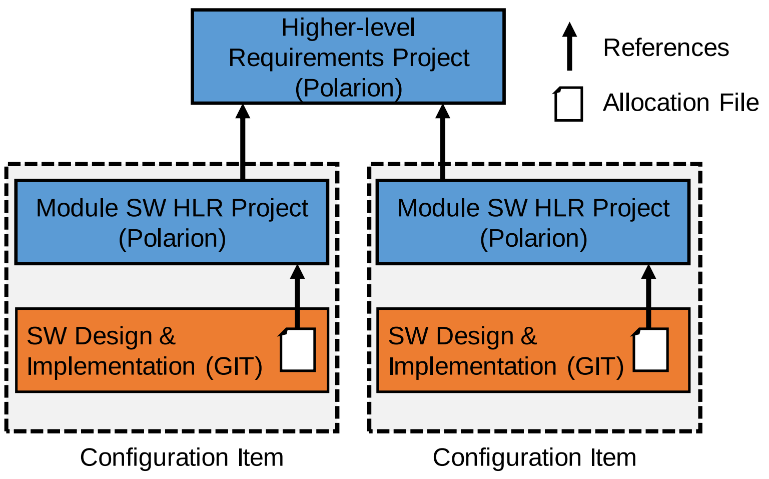

Higher-level Requirements Project (Polarion)
References
Allocation File
Module SW HLR Project (Polarion)
SW Design & Implementation (GIT)
Module SW HLR Project (Polarion)
SW Design & Implementation (GIT)
Configuration Item
Configuration Item

Figure 71: *SimPol* configuration items

The resulting workflow for a requirement change and a subsequent design change is outlined in Figure 72. The left blue line always represents important Polarion revisions, and the right orange line represents revisions of the SW Design versioned in GIT.

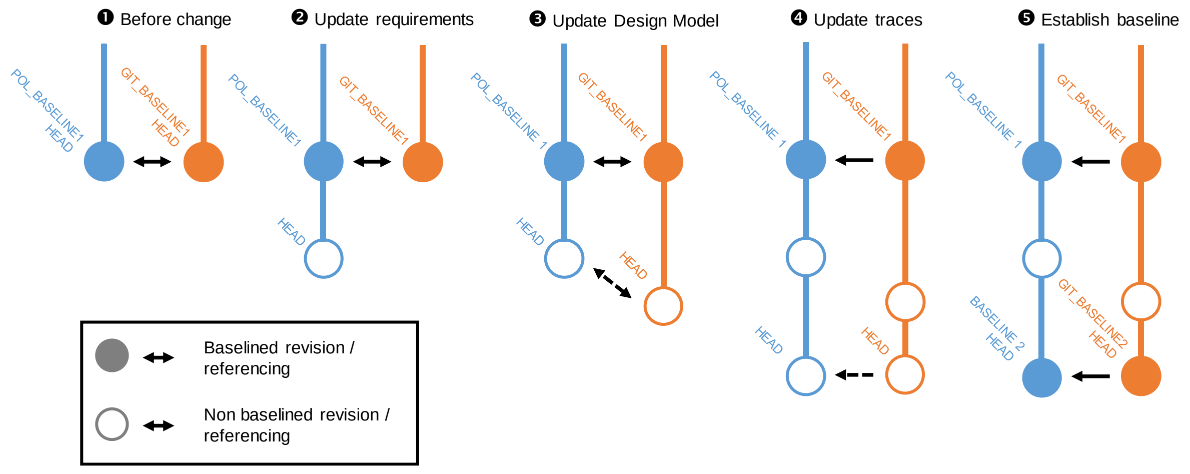


Figure 72: *SimPol* configuration management workflow for requirement changes

1. It is assumed that a *SimPol* Allocation File, which is baselined in GIT (tagged with GIT_BASELINE1), points to a Polarion baseline (POL_BASELINE1). POL_BASE-LINE1 is also equal to the HEAD revision of the Polarion project.

2. From the existing baseline, the requirements in Polarion are updated. This moves the HEAD revision in Polarion forward. At this point of time, design and implementation are not current with the requirements.

3. The developer now switches to design. At first, the *SimPol* Allocation File is changed to point to the HEAD revision in Polarion. After that, *SimPol* highlights model elements, which base on a changed requirement, as well as new requirements and broken links. The developer updates the design and may commit its changes.

4. After updating the design, traces are updated. Every bidirectional trace update is a separate project revision in Polarion and moves the HEAD revision forward on both sides, since work items and RMI link files are modified.

5. As soon as all links are up-to-date, a final baseline is established in Polarion (POL_BASELINE2) for the latest revision. The Allocation File in *SimPol* is updated to point to this new baseline. Allocation File and Design Model changes are afterwards committed to GIT and a tag in GIT is created (GIT_BASELINE2).

If a baseline in GIT is restored at a later point of time, the Allocation File and thus all links loaded with *SimPol* automatically direct to the correct baseline in Polarion.

6.5 Traceability rules

Contribution 15: A new set of traceability rules supports the adoption and consistent usage of *SimPol*, for example by clarifying, what kind of artifacts have to be traced to requirements and to which granularity or how derived low-level requirements are handled. The rules outline a directly usable traceability solution, which covers many existing use cases and lowers the adoption effort for small companies, which are not familiar with traceability so far.

6.5.1 Summary of rules

6.5.2 RMI settings

The RMI allows various settings for linking and report generation [153, pp. 1-26ff.]. The settings are stored for the MATLAB instance on the host computer and not with the model (except the option, whether links are stored externally or internally and the corresponding link file name). Thus, it is inevitable to restore the required preferences for linking every time before the Design Model is modified or reports are generated. All settings are independent of *SimPol*.

TR 1 – RMI settings

A. RMI information shall be stored externally (`StoreDataExternally` set to `on`).

B. If a model primitive is copied, requirement links shall not be duplicated automatically, but discarded for the new element (`DuplicateOnCopy` set to `off`).

RMI links can be stored internally and externally to models dependent on the setting `StoreDataExternally`. Internally stored links are embedded in the model file, externally stored links are saved in a `.slmx` file, which has the same name as the model by default. The advantage of externally stored links is their independence of the model. Adding, removing, or modifying links does not change the model version. From a process and configuration management point of view, external storage is favored.

In addition, the settings `DuplicateOnCopy` should be disabled. A copy of a model element would otherwise keep its links to requirements, but would not establish the reverse links in Polarion. Additionally, a copy of a model element does typically not link to the same requirements.

6.5.3 Requirement allocation to modules

The allocation defines, which requirements shall be implemented in which module.

TR 2 – Requirement allocations to modules

Requirements in Polarion shall be allocated to a module in three *SimPol* Allocation Files:

- an Allocation File for design implementation in SL/SF (`<module-ID>_design.spa`),
- an Allocation File with PDI HLRs for the public Simulink data dictionary, if PDIs exist (`<module-ID>_data.spa`),
- an Allocation File for test implementation in Simulink Test with the design and PDI requirements (`<module-ID_test.spa`).

The allocation mechanism of *SimPol* and the Allocation File has been introduced in section 6.4. Since *SimPol* supports only one type of link (e.g., SL/SF, Simulink data dictionary,…) per Allocation File, three files are necessary.

6.5.4 Traceability to higher-level requirements

The previous section answered the question, how to link requirements. This section answers the question, what to link. Figure 73 illustrates the complete traceability model for SW Design and simulation cases. Simulation case traceability is further discussed in simulation rule SR 4 of Appendix G.

TR 3 – Surrogate Linking

Trace links from Design Model or simulation case to higher-level requirements shall be established with *SimPol* via a surrogate model as depicted in Figure 73.

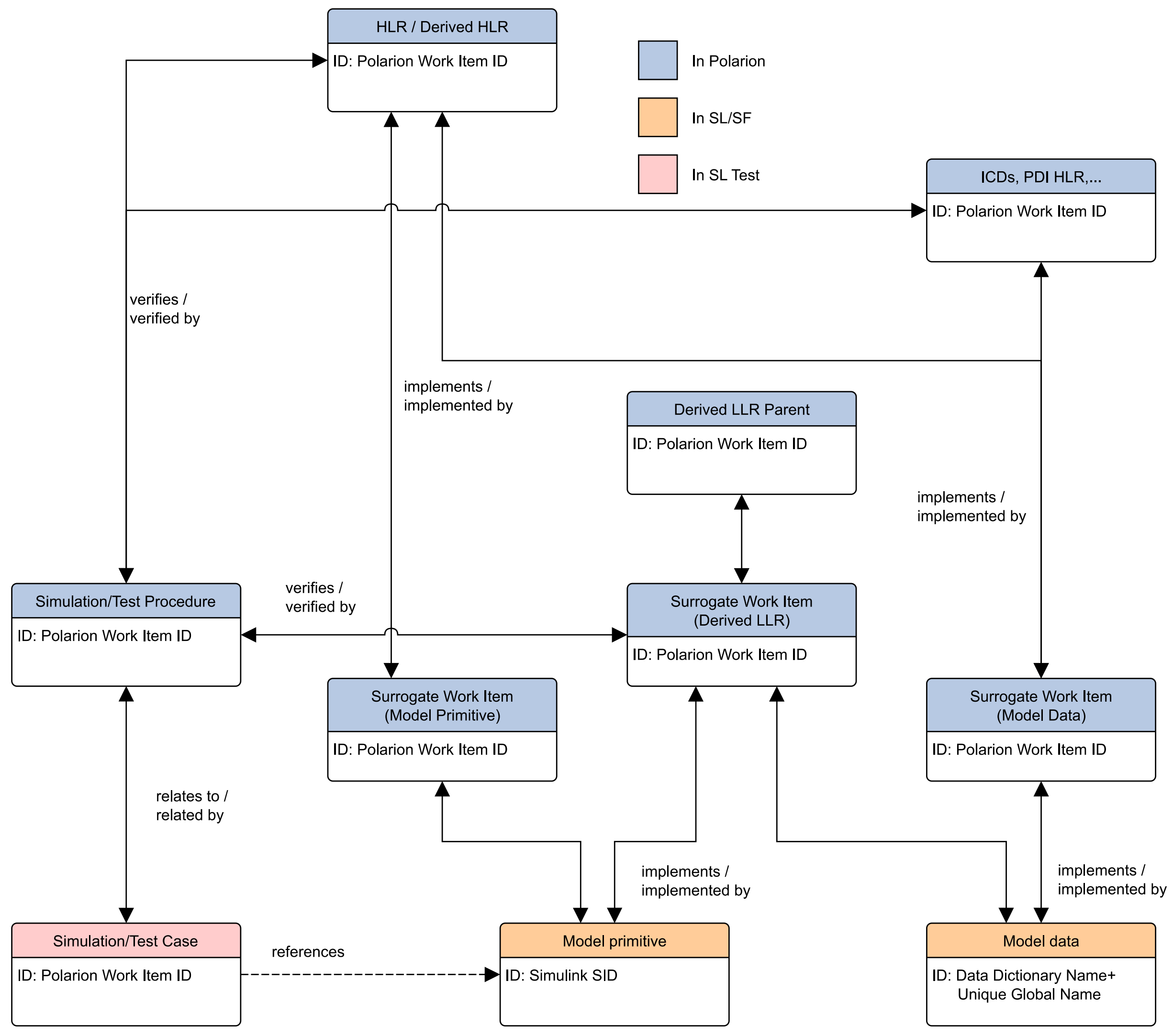


Figure 73: Traceability of SW Design and simulation cases

In MB.11.23(e), DO-331 requires a mechanism to limit the requirements implemented in a model and in MB.11.21 states that the granularity must be sufficient to demonstrate traceability to HLRs.

In R2017b, MathWorks added requirement considerations to the high-integrity guidelines. In [103, p. 8–2], “hisl_0070” states:

A. “Apply requirement links to the lowest level component of model elements. Model elements that do not impact the model's behavior or the generated code are exempt from requirement linking.”

B. “At the project level, define the maximum number of unique requirement links associated with each component. A minimum of one requirement link is required.”

C. “At the project level, define the maximum number of child model elements for each linked component.”

Components are equivalent to container primitives (cf. 5.6.4.2) plus area annotations.

TR 4 – Maximum number of traces to requirements on graphical primitives

A. Only model elements contributing to the Design Model shall be linked.

B. The maximum number of unique requirement links on a container primitive is 4, the minimum is 1.

C. Atomic primitives should not be linked. If they are linked, the maximum number of unique requirement links on an atomic primitive shall not exceed 1.

(B) resembles the requirement of “hisl_0070” (B). With (C), the option is introduced to link single atomic primitives. This might be necessary for blocks with higher complexity, like legacy code blocks or Lookup Table blocks. It is unreasonable that an atomic primitive links to more than one high-level requirement. It is a strong indication that the primitive is too complex, or the requirements are formulated too granular.

The previously formulated rules prohibit *too fine-grained linking*, but not *coarse-grained linking*. Therefore “hisl_0070” is followed.

TR 5 – Maximum content of linked container primitives

The block count of the content of linked container primitive shall not exceed 30.

The approach in “hisl_0070” (C) has some weaknesses. Table 32 lists two edge cases, which highlight the challenge. Some subsystems are used for extensive routing and bus-reordering as in 1). They have a high block count, but typically no or few requirements. Other subsystems have a low block count, but extremely high complexity as in 2). In these cases, the granularity guidelines may require significant redesign of the model.

One way to overcome this would be more sophisticated granularity criteria, e.g., by just counting the non-virtual blocks or additionally taking a complexity measure into account (e.g., number of condition points). This is future work.

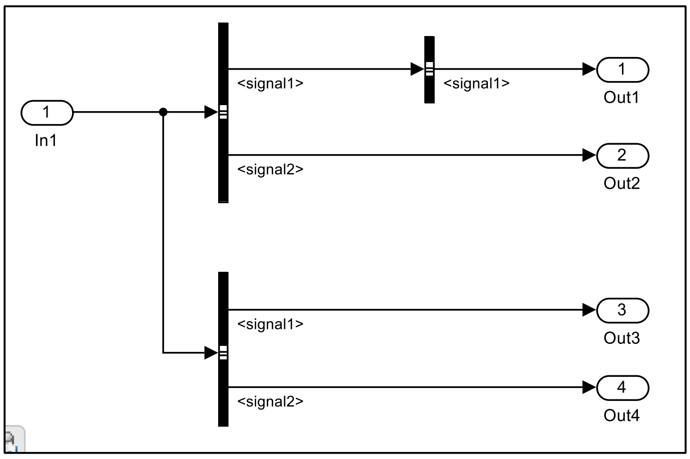
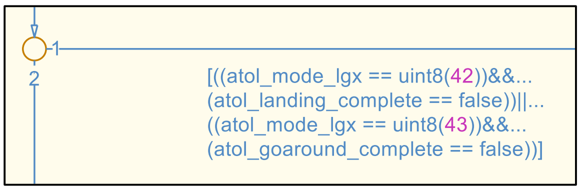
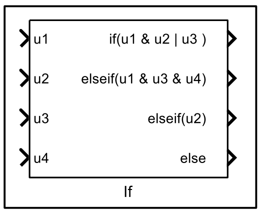
1) Many model elements with low complexity  1 In1 <signal1> <signal1> 1 Out1 <signal2> 2 Out2 <signal1> 3 Out3 <signal2> 4 Out4	Model element count: High Non-virtual element count: Low Cyclomatic Complexity: Low Condition count: Low
2) Single model elements with high complexity  1 2 [((atol_mode_lgx == uint8(42))&&... (atol_landing_complete == false))||... ((atol_mode_lgx == uint8(43))&&... (atol_goaround_complete == false))]  u1 if(u1 & u2 | u3) u2 elseif(u1 & u3 & u4) u3 elseif(u2) u4 else If	Model element count: Low Non-virtual element count: N/A Cyclomatic Complexity: Low Condition count: High

Table 32: Traceability granularity edge cases

TR 6 – Linking of Private Library content

Requirement links inside SL libraries shall be avoided.

Rationale is that links must be directly placed in the SL library, instances in SL Models cannot be linked internally using the RMI. The links in consequence do not trace to the place, where the block pattern is really implemented and where it appears in code. It is not possible to follow traces from Polarion requirements via a SL model element to code easily, which was one reason to prohibit linking of SL libraries.

The policy in combination with TR 5 also implicitly limits the size of Private Library subsystems and promotes the use of REUSABLE MODEL references.

TR 7 – Linking of model data

A. Model data in the data dictionary shall be linked, if it is the Simulink.Parameter for a PDI (cf. MR 36).

B. Model data in the data dictionary shall be linked, if it is the public model data (cf. MR 12).

Traceability of PDI implementations aligns with the discussions in DR 17. Public model data, which can be used in another module, requires further specification like design ranges (cf. DR 11).

6.5.5 Derived LLRs

DR 4 defines the requirements for handling derived LLRs. This section describes a feasible approach to handle derived LLRs in SL/SF with *SimPol*.

TR 8 – Derived LLRs

A. Derived LLRs shall be surrogate work items in Polarion and be clearly identifiable as such by linking them to a common parent work item.

B. The surrogate work item shall document the derived LLR as required by DR 4.

C. The surrogate work item of the derived LLR may link to other HLRs.

Derived LLRs can be considered as dangling implementation without traceability in most cases. They violate the traceability granularity constraints and are discovered thereby. Derived LLRs may be reasonable or unavoidable, but must be identifiable and documented (cf. DR 4).

The given, new approach documents derived requirements and patches granularity gaps. Latter is a major benefit, since otherwise the granularity measures must be manually compared with existing derived LLRs.

The solution for derived LLRs is depicted in Figure 74. All model elements, which are considered as derived, are linked to a single *Derived Parent* work item in Polarion. This work item has no further content and is not considered as requirement. For every linked derived LLR, *SimPol* creates a uniquely identifiable surrogate work item, containing a picture of the respective model element in SL/SF. This work item is enriched by the developer with a short description of the functionality, a rationale and justification as required by DR 4. If necessary, traceability can be established to other requirements, since derived LLRs may or may not be traceable to higher-level requirements.

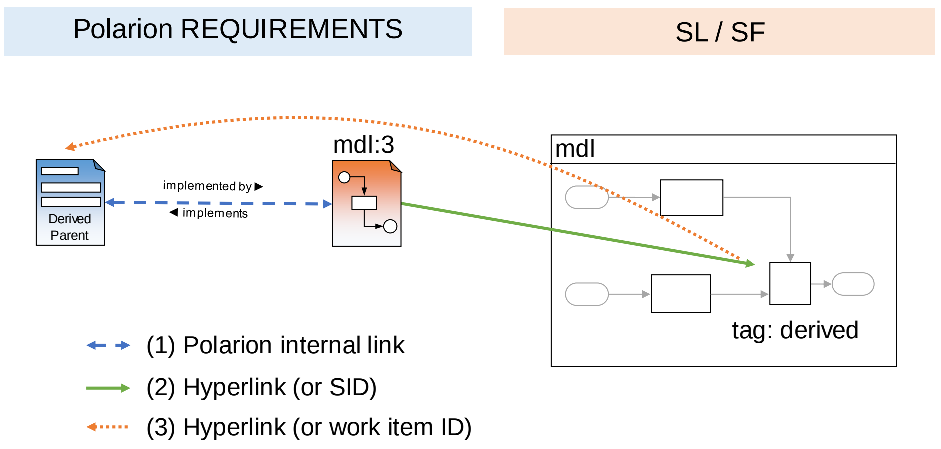


Figure 74: Documentation of derived requirements

Documenting derived LLRs in Polarion as separate work items comes along with a couple of advantages. Rationales, as required by DR 4, can be easily documented in the work item. The derived requirements can be fed to the safety process with the same workflow as for other requirements. In addition, traceability is established and the granularity criteria are met. No separate workflow to check traceability granularity requirements to needed. Traceability to test cases and procedures can be established as for any other requirement. And finally, derived LLRs can easily be distinguished from ordinary requirements and made visible by showing all children of the Derived Parent.

6.6 Summary and outlook

Central achievement is the implementation of *SimPol* presented in section 6.4. The new tooling allows to establish and maintain bidirectional linking between Polarion and various kinds of Simulink artifacts.

In connection with *SimPol*, a workflow to overcome configuration management challenges across tool and repository boundaries has been specified for the particular use case. In addition, a new approach for documenting derived LLRs has been defined in section 6.5.5, which smoothly integrates with existing requirement activities. Also granularity requirements for model-based development have been refined (cf. section 6.5.4).

SimPol itself is a new tool, which needs continuous improvement. One important point is the better support of the new Simulink Requirements product, which enables tighter integration with SL/SF (e.g., drag and drop of requirement links).

The enhanced set of traceability rules gives developers simple criteria for granularity evaluation, but, as pointed out, edge cases still exist, in which these rules are (wrongly) violated. More accurate metrics are necessary to define granularity requirements. In addition to the block count, also other measures like complexity should be regarded.

The rules presented focus on the SL side. In Polarion, additional rules to organize surrogate work items and links consistently across teams may be needed as well. Also the discussed shortcomings of the configuration management workflow should be followed up. Polarion is a fast developing tool and may come with new configuration management solutions in future releases. For example, the missing branching mechanism is currently one of the biggest shortcomings.

After publishing *SimPol* in the internet, many small- and medium-size companies contacted the author for support and adaptions. This emphasized the need for such a tool. With *SimPol* and the traceability rules, developers get both guidance and tooling to address the traceability objectives imposed by the standards. The work done as part of this thesis closes many of the urgent, large inter-tool gaps between Polarion and the SL ecosystem in a new way and is an important brick for a consistent and complete MBSwD process involving Polarion.

7 Process-oriented build tool and process automation

7.1 Objective

This section introduces a new tool solution implemented as part of this work to highly automate the development and verification process. The tool generates and consumes traceability information and extends the features of classical build tools with functionality relevant for development and verification tasks in a safety-critical process. The tool is used as platform to fully represent the MBSwD process at hand.

7.2 State-of-the-art

A study commissioned by the Nation Research Council of the U.S. (NRC) in 2010 examining the current state of the art and future paths for development of software-intensive systems states in Finding 4-2 [157]:

> *"Assurance is facilitated by advances in diverse aspects of software engineering and technology, including modelling, analysis, tools and environments, traceability programming languages, and process support. Advances focused on simultaneous creation of assurance-related evidence with ongoing development effort have high potential to improve the overall assurance of systems."*

This goal as barely been reached. Although the Open-DO initiative sensitized for the "big-freeze" problem, retaining a fully "ready-to-certify" state in software projects is still widely unachieved for DO projects due to the significant number of verification activities to be done and the difficulties to identify the impact of changes due to missing traceability.

Nowadays, two principle approaches can be observed, which help to keep artifacts in a "ready-to-certify" state. The first approach is summarized under the term *continuous integration* (CI). As many verification activities as possible are automated and re-executed after each change on a remote server. Only if they pass, a change is accepted and becomes available for other users.

The second approach is to evaluate development and verification artifacts and their relations after they have been generated. Desired outcome is an overview of the process completeness status with the artifacts at hand and a list of action items to restore a compliant state. These tools are called *integrated assessment platforms* in the following.

Integrated assessment platforms

The wish for integrated assessment platforms managing and interpreting all process-relevant artifacts and showing process compliance goes back several decades. In 2004, Aldrich et al. presented their “System Verification Manager”, a formal framework for collecting heterogeneous verification and development data [158]. The main research goal was automatic reasoning based on ontologies [159, 160].

Today, there are a couple of tools on the market following a similar direction. In February 2019, LDRA announced a partnership with Jama to connect artifact management in Jama with the verification methods and tools of LDRA[28]. SQUORE provides software analytics and accumulates software status information in dashboards[29]. VeroTrace is a cloud-based application lifecycle management with focus on certification standards[30]. Becker et al. [161] outline a toolchain that uses MES Quality Commander[31] to collect, evaluate and display the development status at a central place. Other tools are BTC Embedded Platform for dSPACE TargetLink[32], PTC Integrity[33], or AVL Lab Management[34].

None of these tools have a deep integration with SL/SF and most of them cannot manage SL models and projects to the required granularity, i.e., they cannot analyze traceability into SL models or the breakdown of SL models. Most of the tools mainly focus on displaying data. They rarely have a direct support to re-execute analyses or redo reviews. Almost none of the tools can assess up-to-dateness of artifacts (e.g., if the generated code still fits to the model or if a test case has to be re-executed after the model has been changed), since they do not save information about the input of activities at the time when they are executed.

Continuous integration solutions

The quickly increasing computation performance led to the adoption of continuous integration methods. More tests or analysis are executed more frequently and on separate servers. Permanent testing keeps the software in a working state at all times and counteracts process corrosion, i.e., a slow but steady, unconscious deviation from the standards and plans. This is a “brute-force” method to keep a software project in good shape.

[28] https://ldra.com/technology-partners/jama-connect/ [Accessed on: Aug. 04 2019]

[29] A product of SQUORING Technologies, https://www.squoring.com [Accessed on: Aug. 04 2019]

[30] A product of Verocel Inc., https://www.verocel.com/tools/lifecycle-management/ [Accessed on: Aug. 04 2019]

[31] A product of Model Engineering Solutions GmbH, https://model-engineers.com/de/quality-tools/mqc/ [Accessed on: Nov. 24 2019]

[32] A product of BTC Embedded Systems AG, https://www.btc-es.de/de/produkte/btc-embeddedplatform/ [Accessed on: Aug. 04 2019]

[33] A product of PTC Inc., https://www.net-online.de/ptc-integrity/ [Accessed on: Aug. 04 2019]

[34] A product of AVL List GmbH, https://www.avl.com/de-DE/web/guest/lab-management [Accessed on: Aug. 04 2019]

CI is realized by a couple of interacting tools. Rahman distinguishes build automation, continuous integration, infrastructure as code, and version control tools [162, p. 21]. Continuous integration tools are mainly server applications, that "integrate software changes into the shared mainline" [162, p. 21]. They are fully automated and mostly run on a separate server (e.g., Jenkins[35]).

Continuous integration tools execute build tools, which drive the automation and execute the tasks. Traditionally, they compile software changes into executable binaries [162]. Nowadays, this view may be short-sighted, since their functionality reaches far beyond this scope, which is also expressed by their new naming as "build management tools". They "calculate, how to reach the goal you specify by executing tasks in the correct order, running each task that your goal depends on exactly once" [11]. To a certain degree, they also check up-to-dateness of tasks depending on the input and manage output artifacts. Build tools are directly pluggable into CI tools, which integrate software changes.

However, traceability and build automation tools rarely work together deeply, although the potential is amazing. Build automation tools know about input and output dependencies and can contribute important traceability information. This comes close to the concept of "ubiquitous traceability" of Gotel [163] and Cleland-Huang [46], which "is achieved automatically, as a result of collecting, analyzing, and processing every piece of evidence from which trace data can be inferred and managed" [46, p. 2].

Furthermore, when it comes to setting up a CI system with a build automation tool, the differences between MBSwD and traditional software development raise challenges. Existing, traditional build tools, like Maven[36], Gradle[37], or Apache Ant[38], origin from the Java domain. They are not easily pluggable to SL/SF. These build tools can do up-to-dateness checking to some amount, but mainly work on file basis. The dependencies in the SL/SF environment are more complex and cannot be resolved on file basis, only. Seibel describes such relationships as inherent dependencies [164]. Many artifacts are also only deserializable in MATLAB. Using any of these build tools would require an external connection to a running MATLAB instance. In addition, the dependency graphs, which the build tools assemble, are not accessible for impact analysis and do not include manually established traceability.

The focus of CI tooling is on automation, not on displaying or guiding developers. Many companies have automated the execution of tasks, but face challenges reading back and interpreting the results. CI tools offer some plugins to visualize results in a rudimentary way, but showing the compliance status of a full process is not supported. Even if sophisticated plugins would be programmed, CI tools like Jenkins are server-side solutions. Evaluating the status of a local project on the host computer and directly acting on it would not be possible.

[35] Open-source software project led by Software in Public Interest, Inc., https://jenkins.io/ [Accessed on: Nov. 24 2019]

[36] A software of the Apache Foundation, https://maven.apache.org/ [Accessed on: Aug. 04 2019]

[37] A product of Gradle Inc., https://gradle.org/ [Accessed on: Aug. 04 2019]

[38] Apache Ant project, https://ant.apache.org/ [Accessed on: Aug. 04 2019]

Traditional build tools mainly address automatable activities with a clear pass/fail output. However, such a distinct output is not always present. Although full automation is desirable, not all activities in a DO-178C process actually allow it. Many analyses require manual review afterwards (e.g., missing coverage) and hand-written justifications. A significant amount of time is also spent with model or code reviews, which cannot be automated. Classical build tools do not support these processes.

Finally, both integrated assessment platforms and continuous integration solutions have their contribution in maintaining a “ready-to-certify” status. However, they are mostly independent, solve just parts of the problem, and have their difficulties with model-based development. The following sections present a tool solution, which combines the best of both worlds and is specifically aligned to process activities.

7.3 Structure

In section 7.4, the new process-oriented build tool *mrails*, which has been implemented by the author, is briefly presented. Since already documented in various publications, technical details are omitted.

Section 7.5 introduces a new way of standardizing task execution and review leveraged by the process-oriented build tool. The presented general workflow is valid for all tasks of the process at hand and consequently applied in section 8.

7.4 Process-oriented build tool

Contribution 16: A new type of build tool specifically designed for model-based development in a process has been implemented. The tool provides a common framework to bundle the implementation of process tasks and automate their execution based on dependencies. The tool specifically addresses the needs of a safety-critical process by providing solutions for review workflows as well. Especially review workflows are not well covered by traditional build tools.

Contribution 17: An innovative approach to couple traceability with build dependencies collected during the build has been implemented. The new symbiosis generates fine-granular traceability of sub-file level in local and CI workflows and facilitates enhanced process analysis like checking of up-to-dateness, completeness, or cleanliness of development and verification artifacts.

Contribution 18: The process-oriented build tool leverages novel automated completeness assessment of the certification artifacts, i.e., whether all activities have been performed on all artifacts and the necessary certification evidence exists. This allows to keep the process in a certification-ready state and helps developers to identify upcoming work.

Contribution 19: The process-oriented build tool leverages novel automated consistency assessment to check whether artifacts are outdated and have to be generated or reviewed again. This allows to keep the process in a certification-ready state at all times.

Contribution 20: The process-oriented build tool has been equipped with easy-to-use web-based UIs. They summarize the status of the project at a central point, provide status information for each task and direct links to open the relevant artifacts. This overview helps developers keeping track of the project status from a single view, which leads to higher software quality and more up-to-date artifacts.

7.4.1 Application life cycle

The process-oriented build tool, called *mrails,* is, in its core, a build tool that eases automatic execution of custom, sometimes dependent jobs in organized manner. As any other build tool, it leverages process automation in a local and server-based workflow, i.e., it can be executed in a CI environment and on the desktop.

Figure 75 illustrates the simplified data flow in a continuous integration environment during the implementation of a software change. Gray boxes indicate those tasks supported by the build tool. The separation between *source files* and *derived files* and their archiving in different repositories is a core setup decision. Source files (or source artifacts) are mainly manually modified artifacts, like requirements, models, or files with justifications and review lists. Auto-generated code is stepping out of the line, but is also considered as source artifact. Derived files or artifacts are all other auto-generated artifacts, like verification reports, cache files, or internal files of the build tool. They are most often binaries and not stored in the version control repository for source files to save resources. Sometimes it is even sufficient to store them on a network drive for a limited time. Those files are optional, since they are always reproducible from the source files, if the same tool chain is used. However, they allow a quick start on the desktop and contain important information about the current revision. Only if a baseline is created, for which certification credit is sought, a copy must be archived under the requirements imposed by DO-178C.

At the beginning, the developer clones a revision of the project from the MASTER branch of the source file repository onto his desktop (1). Here, the source file repository is assumed to be under GIT version control (cf. AS 14). The cloned files only contain source artifacts. The corresponding derived files must be obtained from a separate repository. The source control commit contains a reference to the location or revision of the derived files.

After retrieving all artifacts, the developer assesses the impact of the change or directly implements the change (3). The process-oriented build tool supports both steps with a couple of features. After that, the developer can pre-qualify the change (4), i.e., the developer runs verification tasks locally on his machine and checks the output. The process-oriented build tool helps to identify tasks and artifacts, which are impacted by the change and provides automation to re-execute those. Final step is to submit the changed sources to a temporary GIT branch aside the MASTER branch (5).

The submission triggers a CI run. The CI server performs a full rebuild of all tasks and evaluates the outcome automatically (6). If no errors or violations are detected (7), derived files are pushed into the derived file repository (8) and the change set is merged into the MASTER branch of the source file repository (9). Otherwise, the developer has to rework and submit a new version.

Whether a full rebuild is required in CI (6) depends on the robustness of change detection of the build tool. Partial builds based on impact analysis are drastically faster, but are only reliable, if really all dependencies are known. Since this can hardly be proven in a complex environment like SL/SF, a full rebuild should at least be performed before baselining or shipping a revision.

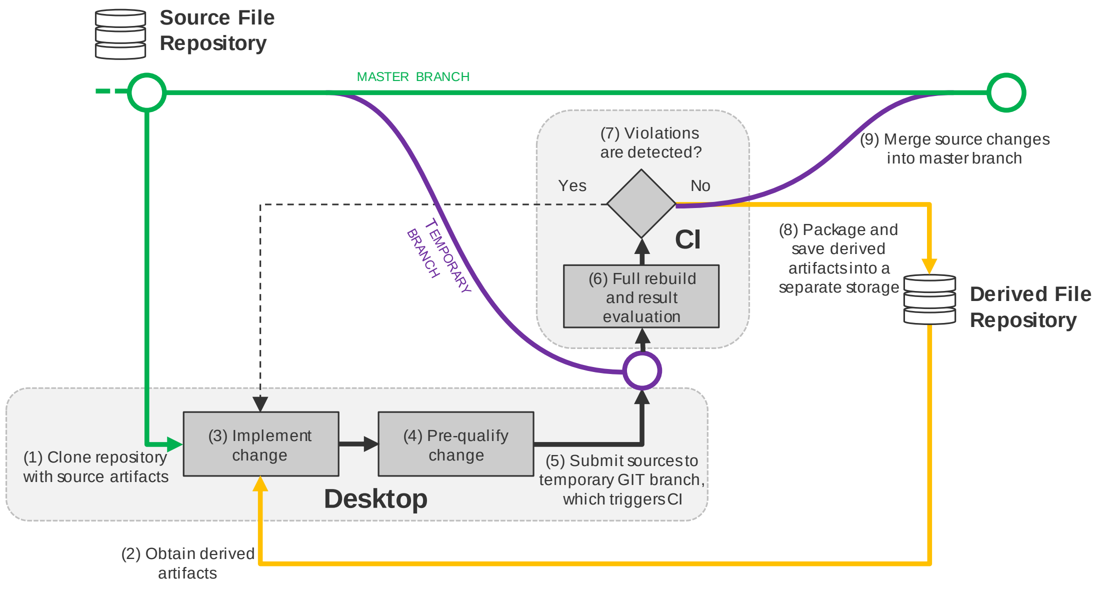

Source File Repository
MASTER BRANCH
(7) Violations are detected?
Yes
No
CI
(9) Merge source changes into master branch
TEMPORARY BRANCH
(8) Package and save derived artifacts into a separate storage
(6) Full rebuild and result evaluation
Derived File Repository
(1) Clone repository with source artifacts
(3) Implement change
(4) Pre-qualify change
(5) Submit sources to temporary GIT branch, which triggers CI
Desktop
(2) Obtain derived artifacts

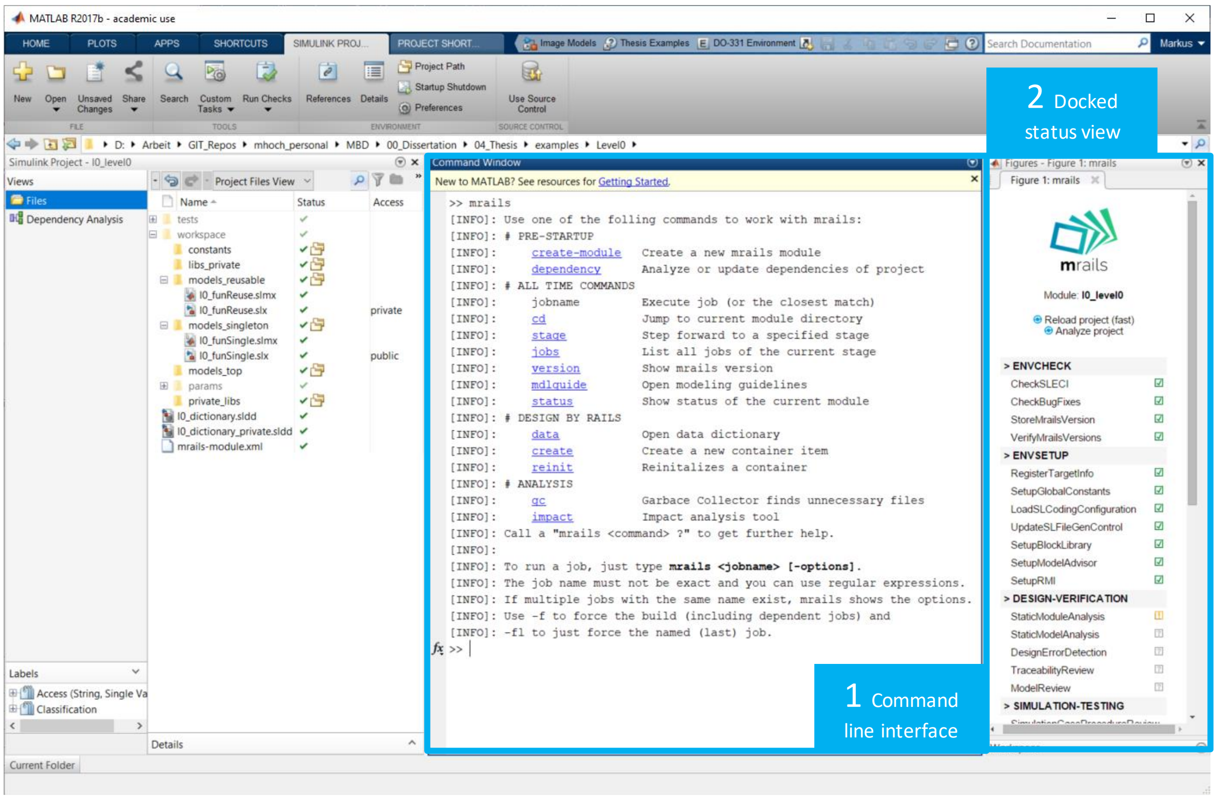
Figure 75: Interplay between build automation tools

At first appearance, the workflow does not differentiate drastically from any other build tool workflow. However, this workflow is, in most companies, only implemented for a small subset of verification tasks, for example testing, for which the result is easy to access and interpret. Goal of the process-oriented build tool is to achieve full process coverage and also include review-intensive tasks. Additionally, the process-oriented build tool adds various supporting features along the workflow as outlined in the following.

Implement change

mrails supports developers in implementing changes into their project in various ways. It supports the integration of an externally defined modeling environment and setup of the development environment. It provides methods to automatically create new modules and integrate dependent modules as well as methods to quickly create different kinds of model elements. Any created item conforms with the standards by construction (e.g., new empty SL models, Simulink.Parameter or Simulink.Bus objects). Last but not least, review of the compliance status is possible in a web frontend.

Figure 76 shows a screenshot of the command line interface and a docked status overview.



```
New to MATLAB? See resources for Getting Started.

>> mrails
[INFO]: Use one of the folling commands to work with mrails:
[INFO]: # PRE-STARTUP
[INFO]:     create-module   Create a new mrails module
[INFO]:     dependency      Analyze or update dependencies of project
[INFO]: # ALL TIME COMMANDS
[INFO]:     jobname         Execute job (or the closest match)
[INFO]:     cd              Jump to current module directory
[INFO]:     stage           Step forward to a specified stage
[INFO]:     jobs            List all jobs of the current stage
[INFO]:     version         Show mrails version
[INFO]:     mdlguide        Open modeling guidelines
[INFO]:     status          Show status of the current module
[INFO]: # DESIGN BY RAILS
[INFO]:     data            Open data dictionary
[INFO]:     create          Create a new container item
[INFO]:     reinit          Reinitalizes a container
[INFO]: # ANALYSIS
[INFO]:     gc              Garbace Collector finds unnecessary files
[INFO]:     impact          Impact analysis tool
[INFO]: Call a "mrails <command> ?" to get further help.
[INFO]:
[INFO]: To run a job, just type mrails <jobname> [-options].
[INFO]: The job name must not be exact and you can use regular expressions.
[INFO]: If multiple jobs with the same name exist, mrails shows the options.
[INFO]: Use -f to force the build (including dependent jobs) and
[INFO]: -fl to just force the named (last) job.
fx >>
```

Figure 76: Command line interface of the build tool

Pre-qualify change

To pre-qualify a change, *mrails* comes along with several novelties.

The tool allows the integration of a full build workflow with *jobs* and job dependencies, which can be triggered from a standard interface. Jobs are contained, executable scripts in the build tool. A job can but must not map to one process task

The tool supports in situ dependency analysis of artifacts along with any build job. The dependency analysis has sub-file granularity, e.g., it recognizes dependences within SL models or Simulink Test files. This is far below the granularity of traditional build tools.

One of the core features is detection of staleness of jobs and artifacts (incl. missing artifacts) using the analyzed sub-file dependencies. Only stale jobs are executed by default, which significantly reduces pre-qualification effort and time.

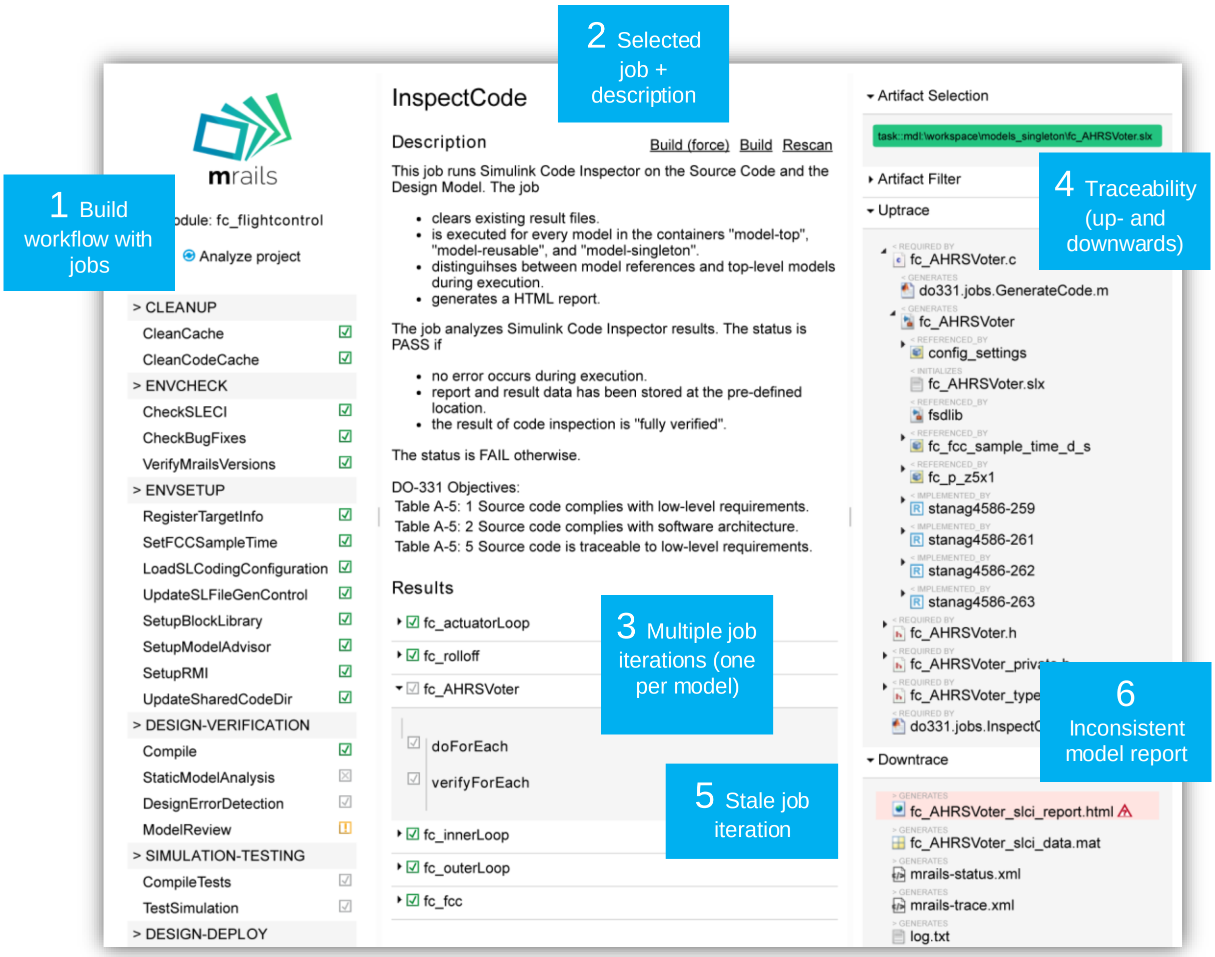
The build tool supports the dynamic generation of review lists. Review lists can be generated depending on the content of artifacts. Review items, which are not relevant, can be dropped. For example, a review of a modeling rule requires the inspection of a specific block type. If this block type never occurs, the respective review item needs not to be included in the review list of the respective SL model.

Review lists support staleness detection like any other job. The tool recognizes review items to be re-evaluated based on a change of the reviewed artifact. Depending on the type of artifact, staleness can be assessed with sub-file granularity.

Compliance status review is possible in a web frontend. The frontend summarizes all relevant artifacts of each task and provides quick navigation to them (if MATLAB is running in the background). Reviews can directly be performed in the frontend as well.

Figure 77 shows a screenshot of the web frontend. The left panel displays the status of the build workflow and each job (1). By clicking on a job, further details are exposed in the middle panel (2). Below the description, results for so-called *job iterations* are displayed (3). Each job can be automatically executed multiple times on different artifacts.

The panel on the right shows inputs, on which the currently selected iteration depends, as well as the produced outputs (4). In the displayed example, the selected iteration `fc_AHRSVoter` is stale (indicated by gray icons) (5), since the output report `fc_AHRSVoter_slci_report.html` does not match to the existing model anymore (6).


1 Build workflow with jobs
2 Selected job + description
3 Multiple job iterations (one per model)
4 Traceability (up- and downwards)
5 Stale job iteration
6 Inconsistent model report
mrails
Build (force)
Build
Rescan
Artifact Selection
Artifact Filter
Uptrace
Downtrace

odule: fc_flightcontrol

Analyze project

> CLEANUP
CleanCache
CleanCodeCache
> ENVCHECK
CheckSLECI
CheckBugFixes
VerifyMrailsVersions
> ENVSETUP
RegisterTargetInfo
SetFCCSampleTime
LoadSLCodingConfiguration
UpdateSLFileGenControl
SetupBlockLibrary
SetupModelAdvisor
SetupRMI
UpdateSharedCodeDir
> DESIGN-VERIFICATION
Compile
StaticModelAnalysis
DesignErrorDetection
ModelReview
> SIMULATION-TESTING
CompileTests
TestSimulation
> DESIGN-DEPLOY

InspectCode

Description

This job runs Simulink Code Inspector on the Source Code and the Design Model. The job

- clears existing result files.
- is executed for every model in the containers "model-top", "model-reusable", and "model-singleton".
- distinguihses between model references and top-level models during execution.
- generates a HTML report.

The job analyzes Simulink Code Inspector results. The status is PASS if

- no error occurs during execution.
- report and result data has been stored at the pre-defined location.
- the result of code inspection is "fully verified".

The status is FAIL otherwise.

DO-331 Objectives:
Table A-5: 1 Source code complies with low-level requirements.
Table A-5: 2 Source code complies with software architecture.
Table A-5: 5 Source code is traceable to low-level requirements.

Results

fc_actuatorLoop
fc_rolloff
fc_AHRSVoter
doForEach
verifyForEach
fc_innerLoop
fc_outerLoop
fc_fcc

task::mdl:\workspace\models_singleton\fc_AHRSVoter.slx

< REQUIRED BY fc_AHRSVoter.c
< GENERATES do331.jobs.GenerateCode.m
< GENERATES fc_AHRSVoter
< REFERENCED_BY config_settings
< INITIALIZES fc_AHRSVoter.slx
< REFERENCED_BY fsdlib
< REFERENCED_BY fc_fcc_sample_time_d_s
< REFERENCED_BY fc_p_z5x1
< IMPLEMENTED_BY stanag4586-259
< IMPLEMENTED_BY stanag4586-261
< IMPLEMENTED_BY stanag4586-262
< IMPLEMENTED_BY stanag4586-263
< REQUIRED BY fc_AHRSVoter.h
< REQUIRED BY fc_AHRSVoter_priva
< REQUIRED BY fc_AHRSVoter_type
< REQUIRED BY do331.jobs.Inspect

> GENERATES fc_AHRSVoter_slci_report.html
> GENERATES fc_AHRSVoter_slci_data.mat
> GENERATES mrails-status.xml
> GENERATES mrails-trace.xml
> GENERATES log.txt

Figure 77: Graphical user interface of the build tool (adapted from [39])

CI integration

The tool provides a sophisticated command line interface, which can be used by the CI system to execute tasks and assess pass or fail criteria. Since the whole UI is web-based, it is technically possible to show it directly for data on the CI system. The developer can get an overview of the status in CI without checking out the whole repository locally.

7.4.2 Implementation Overview

The following is a summary from [33], which describes the implementation of the process-oriented build tool.

Figure 78 illustrates the main architectural components of the build tool: the workflow management, the artifact graph, and the status interface.

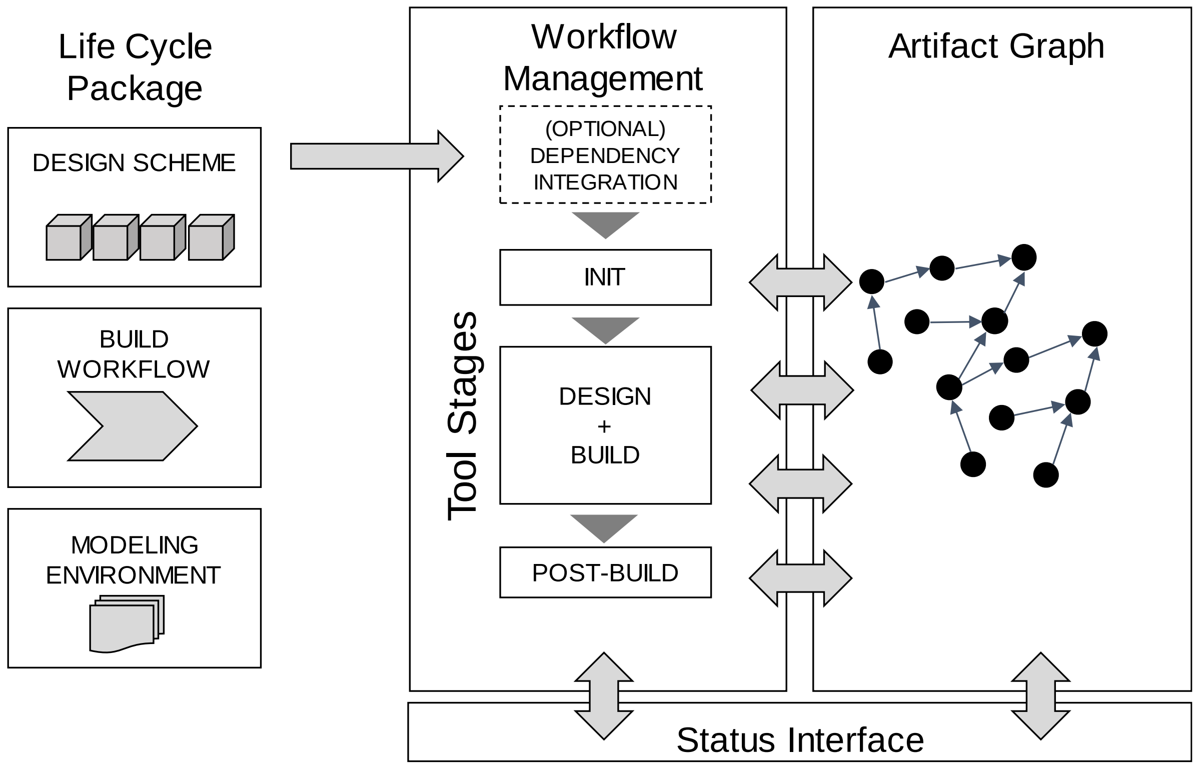


Figure 78: Components of the process-oriented build tool (adapted from [33])

Workflow management

The workflow management controls the build workflow. It manages a couple of fix steps, so-called stages, which must be executed sequentially. Jobs of the build workflow can be hooked into the different stages. The user has to bring the tool into a specific stage. After that, the jobs of the stage can be executed in arbitrary order. For example, before DESIGN AND BUILD jobs can be executed, the INIT stage must be passed, which means that all jobs of the INIT stage must be executed.

The workflow management holds the job execution dependency tree, i.e., it computes the order in which dependent tasks have to be executed (e.g., generate code before code can be analyzed).

The DEPENDENCY INTEGRATION STAGE interacts with the version control system. Dependent modules are downloaded or updated, versions are analyzed and conflicts caused by transitive dependencies reported. The INIT Stage addresses the loading and setup of the environment. In the DESIGN AND BUILD stage, the developer actually implements the model, simulation test cases, and requirement linking. Workflows with tools are not constrained, however model scaffolding capabilities are provided to accelerate development. The majority of verification jobs is hooked into this stage, too. The POST BUILD stage presupposes that all build jobs have been executed at least once and traceability has been established as far as possible. It is reserved for jobs requiring the full artifact graph (like the detection of temporary, unused artifacts).

Artifact graph

The main technical innovation of the tool is that sub-file traceability and dependencies are analyzed in-situ with the build and integrated into a common dependency graph. Extensive traceability is thus created without any effort for the developer. The combined source of relationships enables impact analysis, but is also directly consumed to evaluate staleness of build tasks. This symbiosis is unique.

The relationships are managed in artifact graphs. Technical details of the implementation of the artifact graph are given in [39]. Figure 79 shows a screenshot of an artifact graph generated from an example project.

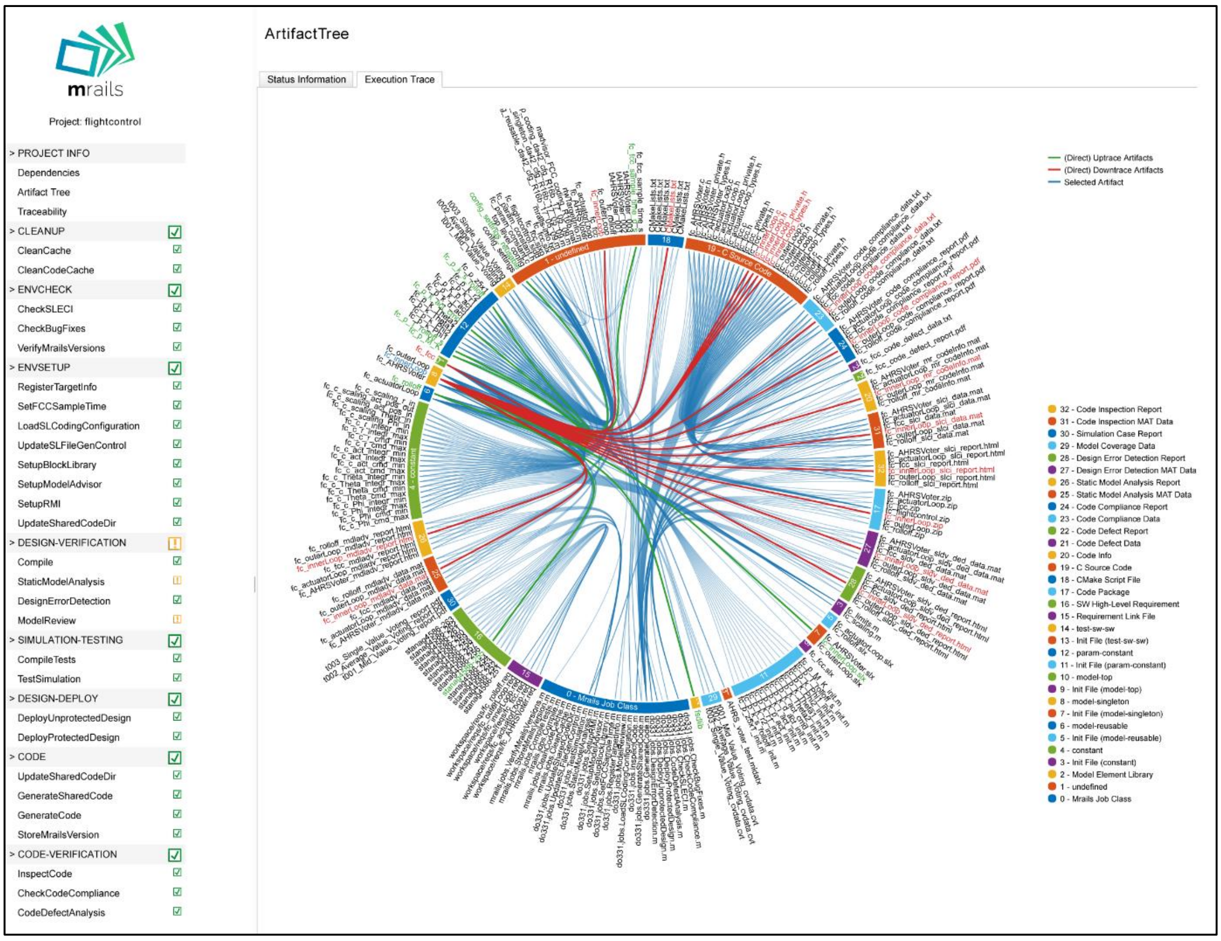


Figure 79: Visualization of an example artifact graph (from [39])

Life cycle package

Another important concept are life cycle packages. Life cycle packages fully control the behavior of the build tool.

The life cycle package consists of a

- design scheme
- build workflow
- modeling environment

The three parts are highly coupled and cannot be used in isolation. The *design scheme* defines classes of design artifacts including rules, how they can be created, identified, validated, initialized, or decomposed. In the MBSwD process, the design scheme describes the specializations of the safe modeling subset introduced in section 5.6.4. The design scheme serves as source of information for the artifact graph, but also enables *model scaffolding*. Model scaffolding provides methods to quickly create artifacts based on the principle “Convention over Configuration” [39]. Many tasks, like writing initialization scripts or creating model data objects according to the modeling rules, are time-consuming, error-prone, and regularly repeated by every developer. Iterating to compliant models and model data with checks after implementation is both a reverse and time-wasting workflow. Model scaffolding supports developers doing the right thing from the beginning.

The build workflow describes the jobs to be executed in a formalized way, their inputs, outputs, “iterations” to be performed on multiple artifacts, as well as their execution order. Jobs mainly map to tasks declared in section 4. The structure of build jobs is explained in detail in section 7.5. How each task has been implemented as job is described in section 8.

The third part is the *modeling environment* as introduced in section 5.8.

Status interface

The status interface is the access point for information. It provides an MATLAB API interface to query the status for jobs in different formats (e.g., JUnit test format[39]) and is for example usable in CI systems.

The status has been standardized. Each element in the build workflow has a status according to Figure 80. In the hierarchical build workflow, the status is aggregated using a “worst-case” rule. The priority is also depicted in Figure 80. A status on the top-right has a higher priority than a status on the lower-left.

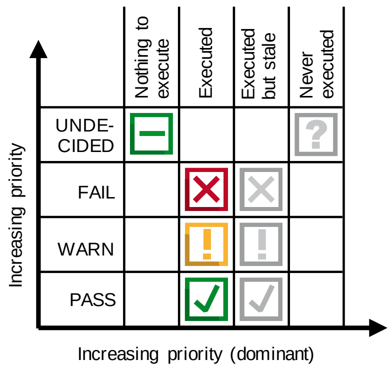


Figure 80: Standardized status types based from [39]

The build tool also comes along with a graphical, web-based user interface. An important design goal was to technically support read-only access without requiring a MATLAB session as well as a scenario, in which the web interface is fully interactive and closely connected to MATLAB. [33]

The underlying architecture is depicted in Figure 81. Central part is the status interface in MATLAB. Upon job execution, status and dependency data is saved in XML files for each job. A client browser can request this data, e.g., via a running Jetty HTTP server[40]. The client page is programmed with JavaScript. Calls and callbacks between the HTTP server and MATLAB are realized by a Jetty servlet extension and the MATLAB Engine API for Java [165, pp. 12-2ff].

[39] Java testing framework, https://junit.org/junit5/ [Accessed on: Jan. 19 2020]

[40] https://www.eclipse.org/jetty [Accessed on: Aug. 04 2019]

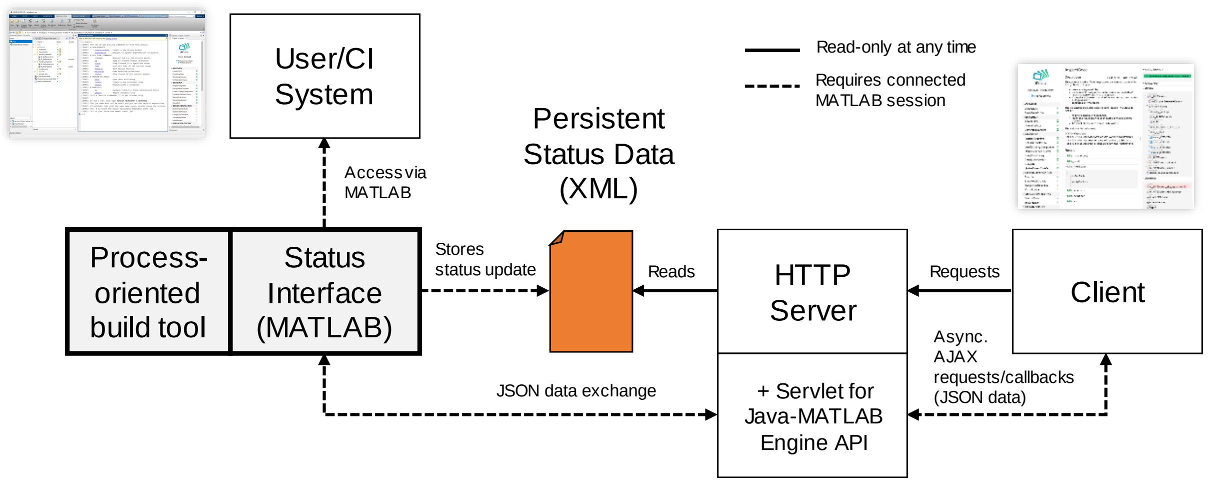

User/CI System
Access via MATLAB
Persistent Status Data (XML)
Read-only at any time
Requires connected MATLAB session
Process-oriented build tool
Status Interface (MATLAB)
Stores status update
Reads
HTTP Server
Requests
Client
Async. AJAX requests/callbacks (JSON data)
JSON data exchange
+ Servlet for Java-MATLAB Engine API

Figure 81: Architecture of status and web-based interface adapted from [33]

7.5 Standardized implementation of build jobs

The build tool provides a novel framework to implement tasks in a standardized manner as build jobs, which is presented in this section.

Contribution 21: A novel framework for standardizing task execution, evaluation, review, and report generation has been established. Generalized status types with consistent consequences or actions for all tool outputs have recognition value for developers. Core contribution are two different, generic justification workflows that either fully embed justifications into the process-oriented build tool or are tightly linked to external tool workflows. They offer up-to-dateness checking of review lists to a certain granularity. Since safety-critical processes always require reviews, these workflows significantly reduce the review effort.

7.5.1 Process notation

To describe processes in the following, a notation as given in Figure 82 is used. The process elements are straight-forward and do not need further explanation. Artifacts are distinguished in source and derived artifacts as introduced in section 7.4.1.

Furthermore, artifacts are grouped on an abstract-level (colors in Figure 82). Supporting material (blue artifacts 🟦) is part of the modelling environment, like execution scripts, configurations or templates. The process-oriented build tool writes various artifacts to manage staleness or justifications (orange artifacts 🟧). Evidence artifacts (green artifact 🟩) are those artifacts, which have to be archived and shown to authorities, if requested. These are mainly reports. All other artifacts are white (⬜).

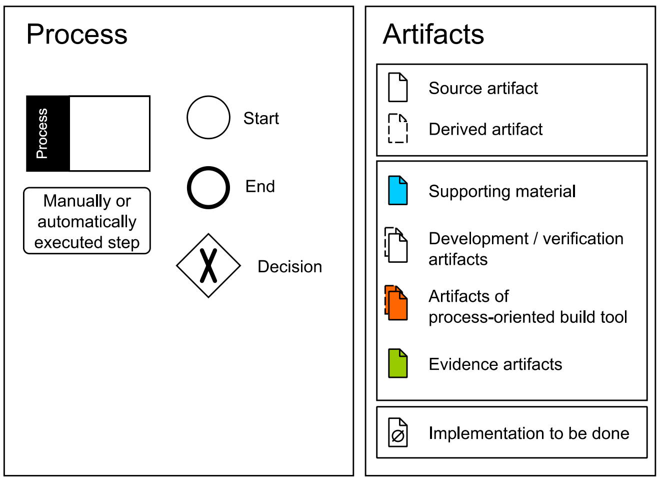

Process
Process
Manually or automatically executed step
Start
End
Decision
Artifacts
Source artifact
Derived artifact
Supporting material
Development / verification artifacts
Artifacts of process-oriented build tool
Evidence artifacts
Implementation to be done

Figure 82: Process notation

7.5.2 Job execution standardization

Each task declared in section 4 has been wrapped into a job of the process-oriented build tool. Figure 83 illustrates the standardized execution process. Each job always has development/verification input and output artifacts. Supporting material customizes the execution of tools. The job implementation class itself is supporting material. The trace file (T) stores checksums of the input and output dependencies as well as traceability information. The log (L) is a dump of the MATLAB command window.

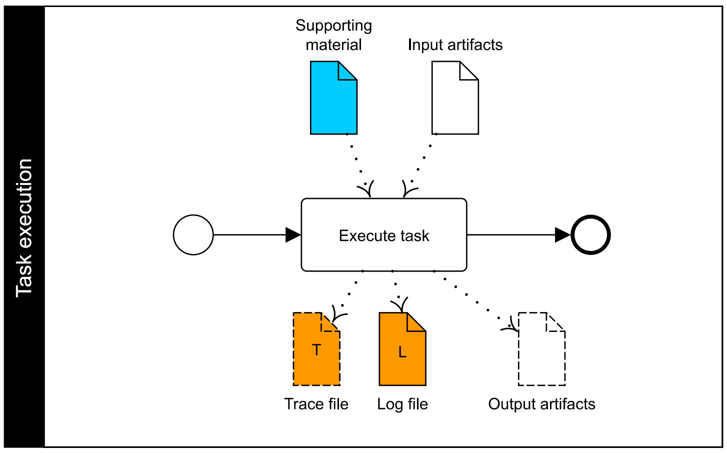


Figure 83: Standardized process execution

Manual review tasks follow the same process. Those jobs do not execute a tool in the execution step. However, they create checklists.

Each job is wrapped in a MATLAB class derived from a provided abstract class as outlined in Listing 10. Listing 10 implements the static model analysis. The `select` method returns the artifacts from the artifact graph, for which the job shall automatically be repeated (in this case for every model). `requireForEeach` defines the inputs and `expectForEach` the expected outputs of the job. The actual execution happens in `executeForEach`. After that, `verifyForEach` is called and assesses the results.

```
classdef StaticModelAnalysis < mrails.jobs.Job

    methods(Access=public)

        % Return jobs this job depends on

        function depedentJobs = dependsOn(job)
             ...
        end

        % Return artifacts to iterate on
        function selArts = select(job)

            t = job.getSourceTree();

            % Gets the files artifacts of the contains
            indices = [t.selectByGroup('model-top');
                t.selectByGroup('model-reusable');
                t.selectByGroup('model-singleton')];

            % Return artifact objects
            selArts = t.select(indices);

        end

        % Return artifacts that are input
        function at = requireForEach(job, artifact)
            % Depends on all dependencies of the model
            at = job.getSourceTree().getParentTree(artifact);

            % Depends on further artifacts like custom checks,
            % check configuration file,...
            ...
        End

        % Return artifacts that must be there as output
        function at = expectForEach(job, artifact)
             ...
        end

        % Execute iteration of job and return whether everything went right
        function status = doForEach(job, artifact)
            ...
            status = mrails.status.StatusT.PASS;
        end

        % Analyze output and return standardized status (result standardization)
        function status = verifyForEach(job, artifact)
            ...
            status = mrails.status.StatusT.PASS;
        end
end
```

Listing 10: Example job implementation

7.5.3 Job result standardization

Each execution returns an output, which must somehow be standardized. Figure 80 already showed the different possible, normed status types. This section explains, how they are interpreted in the scope of the process.

Each verification task returns a set of output artifacts with *result items.* What a result item is, depends on the type of task output. For example, the check status of every performed SL Model Advisor check or SL Test results. A result item can indicate that the result is approved or it can indicate a *finding*, i.e., if a MISRA C rule violation has been detected.

After job execution, result items are mapped to the standardized status types. Again, the mapping is specific for the task and tool. As already introduced, the standardized status distinguishes the states PASS, WARN, and FAIL. Those are handled as follows:

- PASS indicates that no action is required.
- FAIL must always be resolved by rework of the requirement, design, or code.
- WARN either requires rework or justification.
- MISSING and STALE status types require rework or re-execution.

The MISSING and STALE status types are evaluated by the build tool. A mapping for the other status types has been documented as part of this work for each task in section 8.

One benefit of this method is that the mapping can easily be adapted for a desired software readiness level step by step. In some projects, it makes sense to start with less strict process requirements, which are stepped up with increasing maturity of the software modules.

An example mapping is provided in Table 33. A tool returns different execution information, which is mapped to a standardized status. The standardized status for an “Execution with warning” changes depending on the needed software level. In this thesis, all provided mapping tables reflect the software readiness level necessary for full DO compliance.

Tool-Specific Result Item Status	**Standardized Status (Level 2)**	**Standardized Status (Level 1)**
Execution error	☒ (FAIL)	☒ (FAIL)
Execution with warning	! (WARN)	☒ (FAIL)
Execution without warning	☑ (PASS)	☑ (PASS)

Table 33: Example status mapping

7.5.4 Justification workflows

Some findings with status WARN can be justified, others require rework. Justified findings including the justification are called *deviations*.

Concerning deviations, the MISRA C approach is followed (cf. MISRA C 5.4). It classifies deviations as *project deviations* and *specific deviations*. Project deviations cover classes of findings and are handled centrally for the whole project while setting up or improving the process. They can be considered as well-documented and regarded under safety assessment considerations from a developer’s point of view. Provided documentation has project-wide validity. A good example is a repeatedly thrown warning for a code rule violation that is not critical in combination with the used compiler, but cannot be turned off.

In contrast, specific deviations require a case by case assessment including a formal analysis of the impact on safety, which must be documented for each occurrence. The content of a deviation record, independent of a project or specific deviation, is given in MISRA C 5.4.

For each task with project deviations, a supplemental document with all deviation records is provided. Specific deviations are documented in records similar to MISRA C Appendix I. Since the justification capabilities of many tools are limited, deviations are separately documented and just referenced in the justifications.

The central question is now, how justifications are stored. The process-oriented build tool has been implemented to support an *external* and an *embedded justification workflow*. The differences are explained in the following.

Embedded justification workflow

In this workflow, justifications are added using the UI of the process-oriented build tool and are fully controlled by the build tool. It is illustrated in Figure 84. The embedded workflow makes sense, if no or insufficient tool support for justifications exists and for custom review checklists. Examples are model reviews or the justification of SL Model Advisor results.

Contribution 22: Functionality for the generation of dynamic review lists has been integrated into the process-oriented build tool. They can be auto-generated based on the context. Thereby, review work is significantly reduced and a higher level of consistency is achieved, since review lists are managed by the tool at a central place and the review status is directly indicated as for any other automated task.

First step is the standardization of results. The machine-readable outputs of the execution are parsed and translated to standardized results according to the mapping table. This process writes a so-called status file (S) and is implemented in the `verifyForEach` function of Listing 10.

The standardized results are displayed in the UI of the process-oriented built tool and the user can directly mitigate the status and add references to deviations in the UI as illustrated in Figure 85. The first column indicates the standardized status. The second column provides a short title. The third column shows a description (in this case a link to the respective rule, since it is the checklist of a model review). Column four shows the findings and justifications, which can be added using the controls on the right-hand side. Each row can be separately justified and approved.

Input for review and justification are project deviations and the status file. Justifications are stored in a separate justification file (J) by the process-oriented build tool. Justification files are source files, so they are submitted into version control. Justifications are automatically imported and overlaid if the UI is opened. If any dependency has changed, the respective items of the view list are set to STALE and repeated approval is necessary. Specific deviations must be collected by the reviewer in a separate output document.

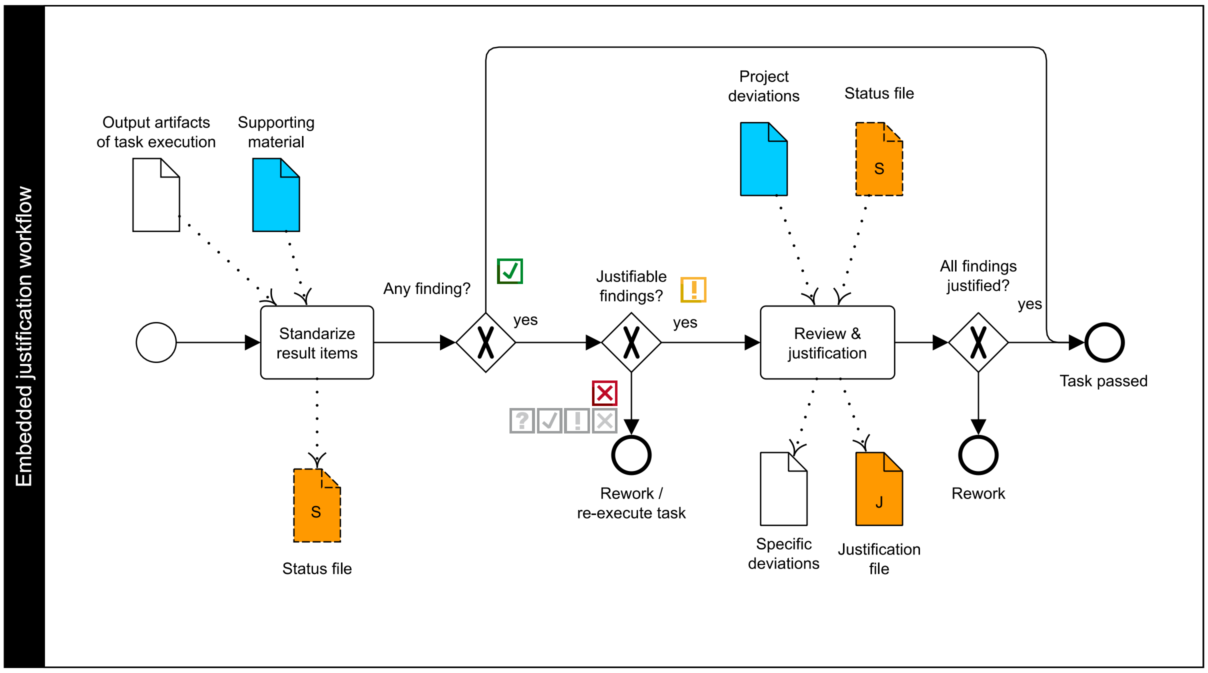
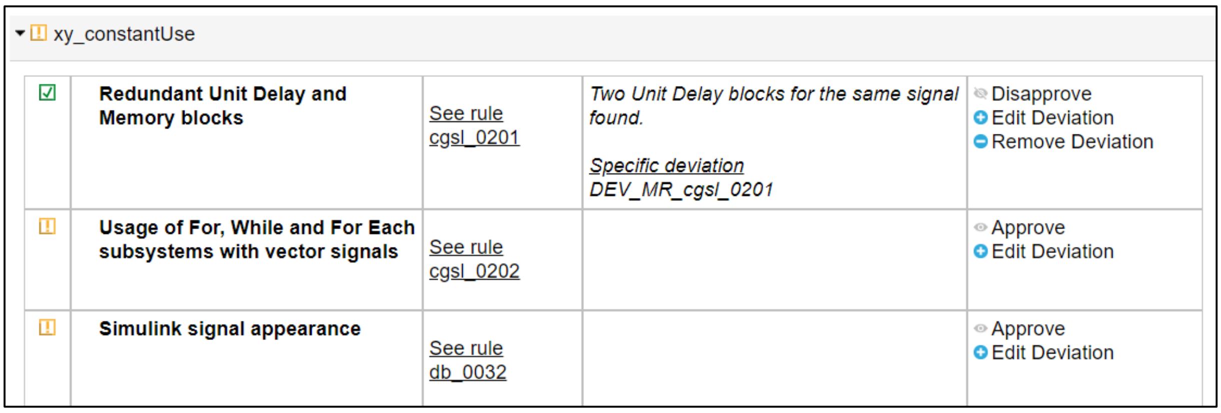


Figure 84: Embedded justification workflow



▾ ! xy_constantUse

☑	**Redundant Unit Delay and Memory blocks**	See rule cgsl_0201	*Two Unit Delay blocks for the same signal found.* *Specific deviation DEV_MR_cgsl_0201*	Disapprove Edit Deviation Remove Deviation
!	**Usage of For, While and For Each subsystems with vector signals**	See rule cgsl_0202		Approve Edit Deviation
!	**Simulink signal appearance**	See rule db_0032		Approve Edit Deviation

Figure 85: Embedded review list in process-oriented build tool

External justification workflow

In general, however, the analysis tools have own, deeper integrated mechanisms to review results and attach justifications to findings. One example is model coverage or code analysis results. For these cases, an external justification workflow exists.

The workflow is illustrated in Figure 86. The results are reviewed in the analysis tool right after executing the tool and exported to a tool-specific justification file (as source artifact). After that, the justifications are added on top of the results during the standardization process. Embedded justifications in the process-oriented build tool are thus not allowed and only a PASS state is accepted.

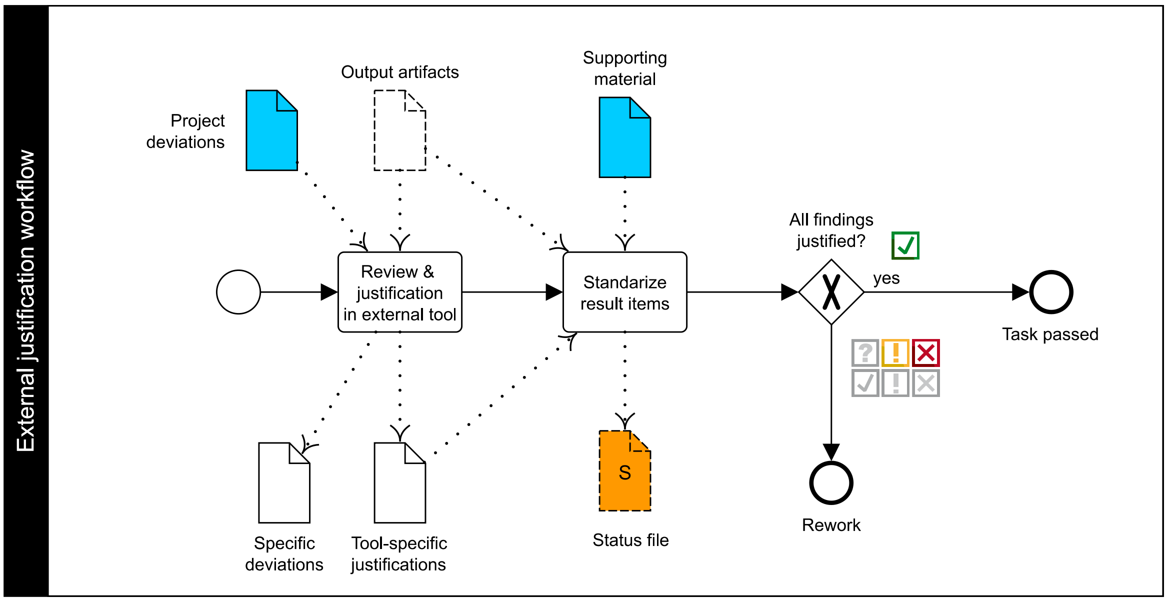


Figure 86: External justification workflow

7.5.5 Evidence generation

The final artifacts for certification evidence are printable PDF reports. Report generation is normally the last step after execution and review of a task. In some cases, it is also part of the execution step.

Report generation for all tasks has been implemented in a separate job, which requires that all other tasks have already been executed.

As depicted in Figure 87, inputs are supporting material (like report templates or job implementation), the original results, the status file, and the tool-specific justifications (in case of the external review workflow).

Outputs are printable evidence documents (green artifacts 🟩).

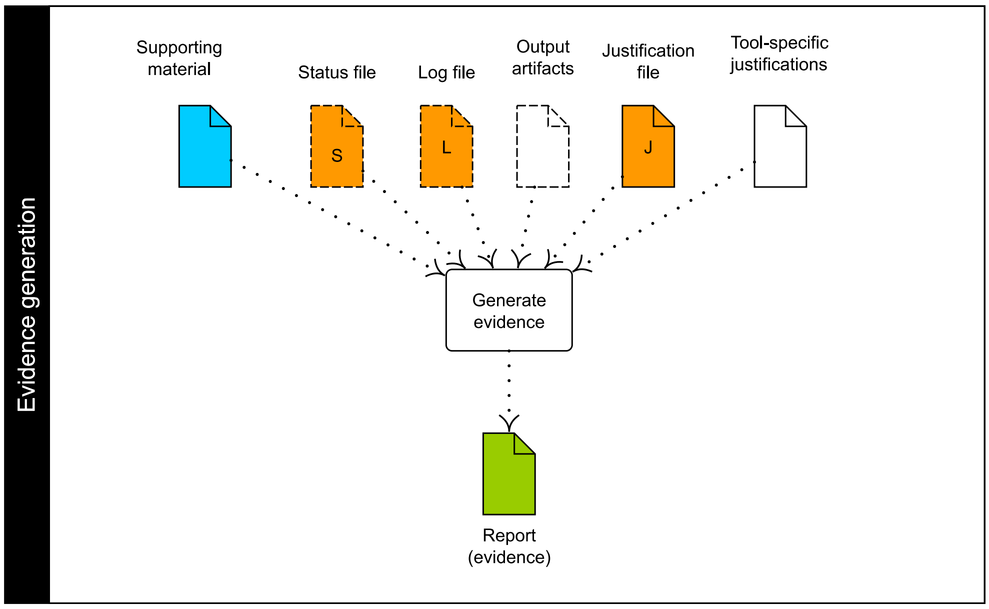


Figure 87: Report generation workflow and artifacts

7.6 Summary and outlook

With the process-oriented build tool, a framework has been created, which has the capability to automate a majority of tasks from the MBSwD process, including often neglected review activities, without interfering with existing workflows.

The backend of the build tool comes with several innovative concepts. The idea of collecting build dependencies in situ with build automation and combining it with other, manually established traceability information is new. This quickly leads to extensive traceability/artifact graphs empowering impact analysis or staleness detection. With build capabilities like sub-file staleness detection, both local pre-qualification and CI runs are significantly accelerated.

The central web interface allows to get a holistic view onto a project from the perspective of process compliance. It indicates stale and missing artifacts or tasks and provides quick navigation to source and derived artifacts.

Finally, the unique standardized job implementation framework provides a way to plug in almost any tool and present outcomes in a way, which are easy to understand and imply consistent actions. A lot of effort has been spent to include review and justification activities in smooth workflows and also support those with a maximum degree of automation. The external and embedded justification workflows suffice these requirements, since they fully or tightly integrate tool-specific review activities, allow dynamic adaption or generation of review lists and leverage staleness detection.

The build tool is still a prototype and may need further enhancement in the backend, like performance improvement or better integration of nested modules, as well as in the frontend (e.g., usability improvements). Although the front end is laid out for standalone usage without MATLAB, this workflow has never been tested in combination with CI. Settling online-access of results may be a future goal. Other future aspects are leveraging MATLAB parallel computation capabilities to execute jobs in parallel, if possible, or the automatic generation of summary reports.

In addition, the complexity of the artifact graph is unneglectable. Defining queries to fetch the right inputs is not straight-forward and may require several adjustments in future. The risk can be mitigated by performing a full-build in continuous integration as outlined in section 7.4.1.

With the process-oriented build tool, a powerful, innovative framework tailored to safety-critical MBSwD has been created establishing a centralized, unified way to implement and execute all kinds of process tasks.

8 Modular development process (part 2)

This section is a continuation of section 4, which introduced the tasks of Development and Verification Processes in the modular MBSwD process on an abstract level and explained the relevance for DO-178C/DO-331. This section explains, how the detailed implementation has been realized using the process-oriented build tool. Features and tool settings have been carefully selected to support the process goals and various artifacts have been created to guarantee appropriate tool configuration and execution (cf. Appendix H).

8.1 Development tasks

8.1.1 SwDP-DP-MB 6 – Assembly of Design Description

MathWorks proposes the creation of a printable Design Description document (PDF) of the SL models [32]. Beside the Design Model, the Design Description also contains requirement traceability information (Trace Data). This may be reasonable for long-term archiving, since it is readable without SL/SF and describes the compiled SL model.

Experience showed that reviewing the Design Model based on the Design Description is impracticable due to its organization and the massive amount of data, which is structured in a flat document. Whether a textual Design Description is required should be discussed with the authorities.

Here, it has been decided to not export the Design Description in every build. The task has been listed for completeness, only.

8.1.2 SwDP-CP-MB 1 –Modular source code

As defined in section 4.4.3, each module process generates and partially verifies code separately, which is novel and not an existing feature of Embedded Coder.

Contribution 23: A process to generate modular code with Embedded Coder has been realized, which fulfills configuration management requirements. Additional scripts and resources have been created to allow generation of modular code and safe integration into higher-level SW modules afterwards. Such a workflow is not supported by the MathWorks tool chain natively. Central advance is the handling of shared code. Generation of modular code enables reusability of code as well as early and separate verification.

In SL/SF, Embedded Coder generates code per model with the `slbuild` command [127, pp. 2-856ff.]. It distinguishes between a top-level build and a model reference build differing in the generated code interface. A top-level build uses `slbuild` without an additional build specification, a model reference build needs the option `ModelReferenceRTWTargetOnly`. Code of the latter is callable from another model. Models, which are not integrated into other models should perform a top-level build. A top-level build considers the modeling as finished and assumes that all necessary information is available. The code is optimized accordingly.

Independent of the build mode, a build is always greedy, i.e., it builds the whole nested model hierarchy. In the example of Figure 88, code generation of A1 would always trigger code generation of A2, A3, B1, and B2 – even in a model reference build. The model reference build allows a build with the model reference interface, but cannot just build a single model if it has a nested model hierarchy. All results of one code generation run are placed in one folder hierarchy.

As consequence, only public models must be built as model reference for a module. Private units are thereby implicitly built, unless they are dead design. Only the top-level model is built with a top-level interface.

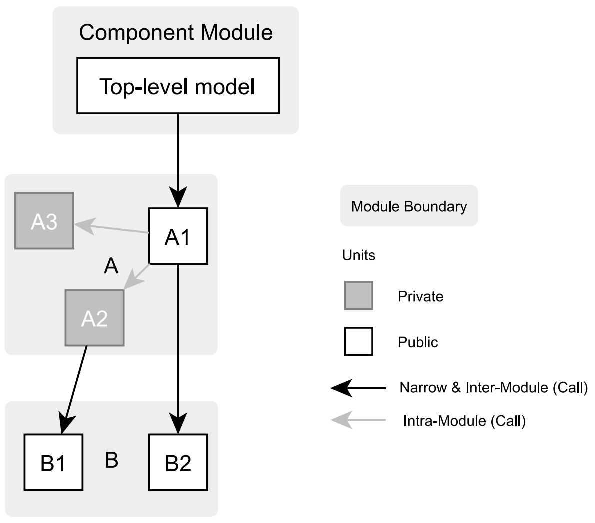


Figure 88: Example model hierarchy for code generation

If code for a model already exists, it is checked for up-to-dateness and just code of changed models is regenerated. For example, if A1 is built and, in a subsequent run, B1 is built as model reference, Embedded Coder would not regenerate code for B1 again.

8.1.2.1 ***Reuse of existing code from other modules***

During one build, Embedded Coder can work with exactly one code folder into which it generates new code and looks for existing code. However, due to source control, modules are in separate folders. In consequence, each module needs its own code folder as depicted in Figure 89.

In the example above, if B1 and B2 are built in module B and integrated into module A, Embedded Coder will regenerate B1 and B2, if A1 is built in a new code folder. Ideally, code folders could be distributed, but this is not possible with Embedded Coder.

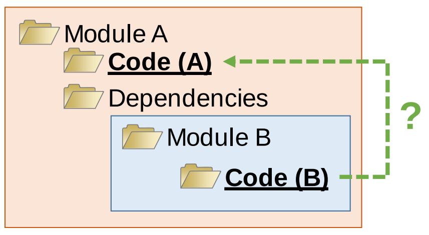


Figure 89: Distributed code folders

Thus, different workarounds have been tested to import already verified code into the code folder of the current module *instead of rebuilding it*.

- Copy existing code. This approach copies code to the current code generation folder prior to the build.

- Protected models. The original intention of model protection was to protect intellectual property by bundling a model simulation target and generated code into a single encrypted model. Protected models can be simulated, and they can expand the pre-generated code, but their content can be hidden [124, pp. 8–95ff]. The idea is to generate code as part of protected models for all SL models with public interfaces in a model. The protected models are then provided to other modules. Their models can reference the protected models. If code is generated for a higher-level model, protected models just expand the pre-generated code.

- Cross release export and import. Since a few releases, MathWorks provides cross-release export and import functionality. Existing code can be imported from a repository to the current code folder and a SIL block is generated. This SIL block is used instead of model references. [135, pp. 20-85ff]

None of these workarounds proved to be perfect. All of them have their strengths and weaknesses as shortly summarized in Table 34 and lead to a lot of scripting overhead and restrictions. It is doable, however the robustness of the code generation process and the final benefit is questionable. In other words, no practical workaround could be achieved with these approaches to avoid regeneration of code from models in other modules.

Regardless this limitation, *modular code could be generated*. It was not possible to avoid rebuilding completely, but the code could be structured in a way so that it is independent from code of other modules. As consequence, although the same code is technically rebuilt in other modules, it can be shown that the code is always equivalent and the previously performed verification activities remain valid.

The implemented solution is as follows:

- Code is generated for all public models in a module in `ModelReferenceRTWTargetOnly` mode (except for the top-level model, for which the top-level code generation mode is used). This also generates code of models in other modules (no import is made).

- After code generation, for each generated code file belonging to the current module, a checksum is calculated and stored in a text file. This is triggered in an Embedded Coder exit hook [96, pp. 70-40f].
- The checksums of files, which belong to regenerated code of other modules, are compared to already existing, stored checksums. If they don't match, an error is thrown.

Thereby it is ensured that code generation in one module does not change code files of other modules, i.e., the code remains modular. Figure 90 depicts the idea with the previous example. In module B, code is generated for B1 and B2. In addition, checksums are prepared for every code file. In module A, code is generated for A1 to A3, but also for B1 and B2 as well. This is the part, which cannot be avoided. However, in order to make sure that the code did not change, all code files of B1 und B2 are compared against the previously stored checksums.

Keeping the code independent is achieved by the various modeling rules. However, there is still some code, which is reused across modules, so-called shared code. The following sections address this topic.

Workaround	Advantages	Disadvantages
Copy existing code	• No compatibility issues with other tools or features.	• Manually scripted copying of code from all nested modules into the current code folder. • Experiments showed that if existing code was only copied to an empty code folder, Embedded Coder would fail to recognize it as up-to-date and already existing code. A "dummy build" had to be performed to create the basic code folder infrastructure. After that the existing code could be integrated.
Protected models	• Integration of already generated module code is completely handled by Embedded Coder. • Mismatching configurations are detected • Protected models without encryption are ordinary zip archives. • If just public models were built, private models would not be indepentently accessible by other modules by construction. • Simple switching between referenced protected and unprotected models by simply putting the relevant models on the search path without changing the referencing model.	• Model protection does not work on models, which reference protected models themselves. So a full rebuild of all models is required during code generation. • A protected model always includes code for the whole underlying model hierarchy. The deeper the model hierarchy, the more models are embedded in one protected model. In consequence, multiple version of the same code may exist in different protected models, if the same model is referenced multiple times. • The majority of verification tools does not support referenced protected models. • Shared code files are not pre-generated, but created upon-expansion. So they are not modularly generated.
Cross-release export and import	• Code does not have to be regenerated • Pre-compiled simulation target using the real code • Feature, which represents the intended workflow at closest	• Referencing only by a SIL block, which is not supported by many verification tools. A SIL block cannot be easily exchanged by a model reference block (as it is the case for protected models), so it is a black box for debugging. • It is unclear, how shared code is handled. • Slow cross-release import functions • New feature in R2017b with several incompatibilities [96, pp. 33-88ff]

Table 34: Modular code generation workarounds

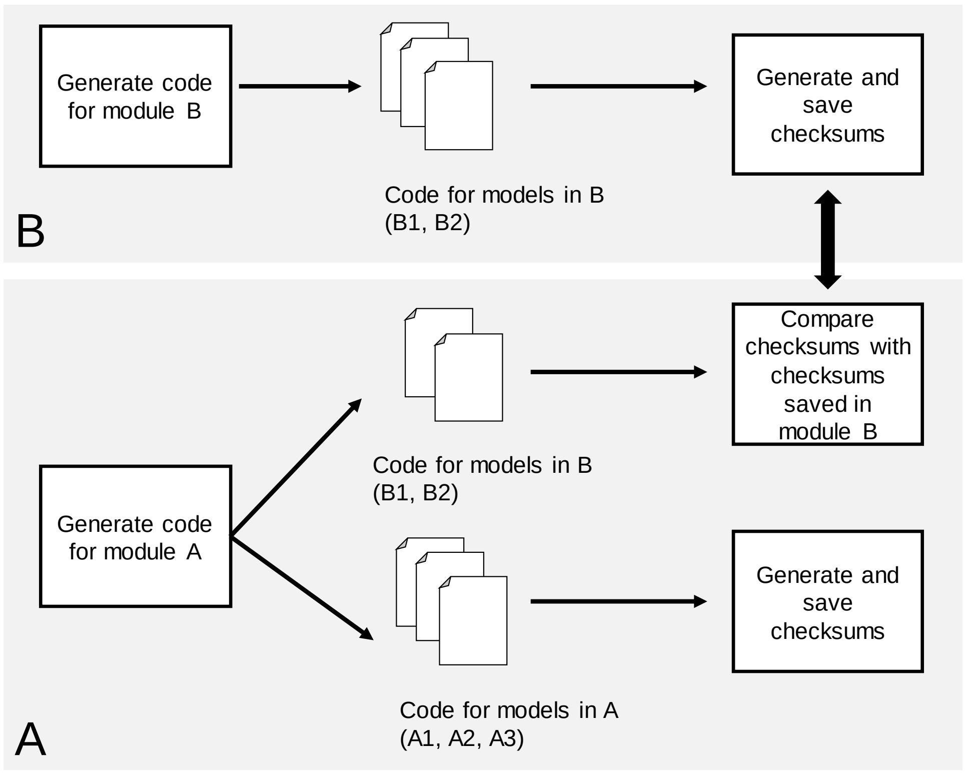

Generate code for module B
Code for models in B (B1, B2)
Generate and save checksums
B
Compare checksums with checksums saved in module B
Code for models in B (B1, B2)
Generate code for module A
Generate and save checksums
Code for models in A (A1, A2, A3)
A

Figure 90: Example for generation of modular code

8.1.2.2 ***Shared code***

Shared code is generated code containing declarations and definitions for variables and types, but also larger code functions for some blocks, which are shared amongst multiple SL models (and in consequence also amongst multiple modules). In the Embedded Coder documentation, it is called *shared utility code* [135, pp. 9–71ff.] and always placed in a folder called `_sharedutils`. Figure 91 illustrates the position of shared code folder.

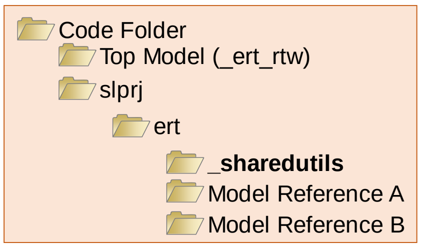


Figure 91: Code folder structure

From a process point of view, a solution had to be found for two “sources” of shared code:

1. Design independent shared code
2. Design induced shared code

Design independent shared code

This group encapsulates all code, whose structure is independent of the SL model and which can exist prior to a model build. Here, it is the Cert C Standard library, which will be detailed in section 8.1.2.5, but also so-called *canonical shared code*.

Canonical shared code is generated code, which is independent of the Design Model. It mainly represents basic type definitions and complex functionality, which is not explicitly graphically modeled. A popular example is the code generated from a Lookup Table block (see also section 5.6.5.5). Its functions and files have canonical names, which means that there is a unique, reproducible name for every variation of the block parameters. Figuratively, Embedded Coder maintains a repository of pre-generated code for special blocks and operations, from which the canonical functions are copied.

For example, for every combination of algorithm block settings of the Lookup Table block, a separate canonical function name exists. A **bin**ary search with **l**inear interpolation and **c**lipping extrapolation for double data types results in a function called `look1_`**`binlc`**`a`.

For canonical shared code, no specification exists in the tool documentation and the SL block itself cannot be considered as sufficient low-level requirement. The code must either be reengineered or replaced by code developed under a traditional DO-178C process. Embedded Coder must thus be instructed to omit regeneration of shared utility code and import reengineered code (for example with requirement traceability).

Embedded Coder can be instructed to copy the canonical shared code from an existing folder prior to coding, and skip the generation process (option `ExistingSharedCode`) [96, p. 33-82]. The process setup can contain preassembled canonical shared code, which has been developed and verified under DO-178C and serves as repository for Embedded Coder. This method has been chosen for all design independent shared code, since it is very convenient and can be configured to throw an error, if an unsupported canonical function is required during code generation.

It is important to note that the code is not executed in normal simulation runs, i.e., it must be taken care that simulation matches the code behavior. However, any mismatch will be caught by equivalence testing.

Design induced shared code

In addition, shared utility code may contain designed declarations and definitions of buses, types, and data, which are referenced in multiple models (e.g., of a bus, which is used in more than one model). This truly generated code shall be called *design induced shared code*.

Design induced shared code is typically generated in situ with code for a model hierarchy, but has various disadvantages for modularization:

- By default, Embedded Coder "reuses" shared code files and expands them incrementally. For example, if a new shared parameter is discovered during code generation, it is appended to existing code files (even to code files of previous builds). In the standard workflow, the same shared files would look differently in each module depending on what SL models are referenced and in which order. This must be strictly avoided in generation of modular code, since shared code files of other modules are already verified.

- Depending on the storage class, some definitions of Simulink.Parameter objects are only generated with a top-level build, but sub-level modules do not make a top-level build (no top-level model). But for code verification in the module, these parameters are already required.

- The header of shared code files is generated with information of the first SL model (i.e., model name, model version,…), in which a shared variable or type definition occurs. This cannot be controlled using the code generation templates. This information is misleading and may change the revision, although no actual change of the parameter happened.

To modularize design induced shared code, considerations during modeling and independent share code generation were necessary.

In code generation settings, shared code placement is controlled with the option `UtilityFuncGeneration` [130, p. 9-15]. It has been set to shared location to instruct Embedded Coder to place utility code into the `_sharedutils` under all circumstances. In addition, specific modeling rules and naming conventions, mainly for storage classes of model data, ensure that shared code remains modular (cf. section 5.6.4.7). For example, if a Simulink.Bus is specified as exported with a named header file, the definition will be generated into a shared location and can be used in the code of different models. If the type definition was not shared, Embedded Coder would generate a definition "guarded" with macros (`#ifndef`) into the code of every model using the bus. This, for example, raises MISRA violations (MISRA C Rule 5.6 "A typedef name shall be a unique identifier").

Since modeling rules were not sufficient to solve all issues, the code generation approach had to be modified. An independent *shared code generation* step generates shared code separately from the Design Model. This works as follows:

- A temporary model, which contains all model data of the current module leading to design induced shared code (`EXPORTED BUS`, `SAFE ENUMERATION`, `PARAMETER CONSTANT`) is generated.

- Code generation settings are swapped from imported to exported storage class (for example, cf. MR 36).

- Code for the temporary model is generated once per module in a top-level build prior to any other model build. Only the generated shared code is archived.

Since all of the model data, for which shared code is generated, must be used in the actual Design Model (cf. MR 40), these shared files would be generated anyway. The step just anticipates the process to get a reproducible result. As consequence, SLCI theoretically verifies design induced shared code at least once. Anyway, verification is added to the manual code review, since the classical code generation process has been modified and it is not clear, how deep SLCI analyzes data with imported storage class.

8.1.2.3 ***Final code generation workflow***

The new code generation workflow is illustrated in Figure 92. In a first step, design induced shared code is generated for the current module (“Generate module shared code”).

Prior to model code generation, the composition of a so-called *shared code repository* is executed (“Make shared code repository”). Its generation is hooked into the starting sequence of code generation and executed prior to the actual build of Embedded Coder. Thereby, prepared canonical shared code is integrated into the shared code repository together with existing, design induced shared code of all nested modules. Any model build triggered makes a new clone of the composed shared code repository.

The actual model code generation references the shared code repository and builds code from the model (“Generate model code”). After code generation, all (regenerated) code of nested modules, shared or not, is verified against existing checksums, to ensure that the code generation process did not taint them. For new files, checksums are created and saved (cf. section 8.1.2.1).

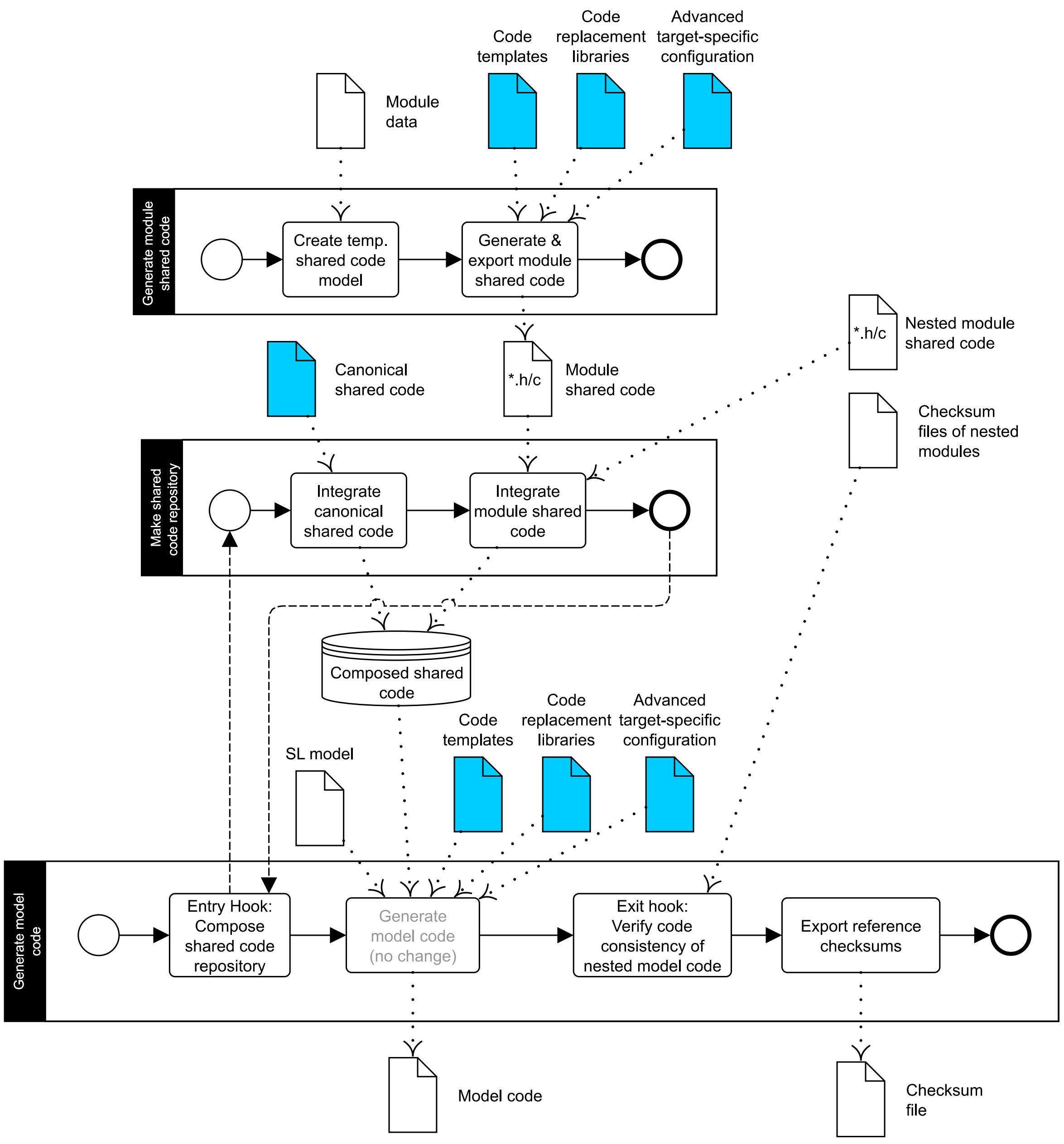

Code templates
Code replacement libraries
Advanced target-specific configuration
Module data
Generate module shared code
Create temp. shared code model
Generate & export module shared code
*.h/c
Nested module shared code
Canonical shared code
*.h/c
Module shared code
Checksum files of nested modules
Make shared code repository
Integrate canonical shared code
Integrate module shared code
Composed shared code
Code templates
Code replacement libraries
Advanced target-specific configuration
SL model
Generate model code
Entry Hook: Compose shared code repository
Generate model code (no change)
Exit hook: Verify code consistency of nested model code
Export reference checksums
Model code
Checksum file

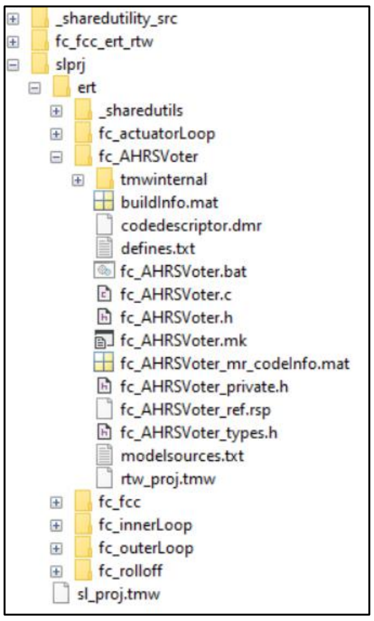
Figure 92: Workflow for generation of modular code

8.1.2.4 *Code archival*

Although auto-generated, the Source Code must be treated differently than other generated artifacts, like reports. According to DO-178C Table A-2, Source Code is of Control Category 1 for all DAL levels A to C, so it must be similarly treated like requirements or the Design Model.

Code generation with Embedded Coder produces a couple of artifacts beside the code as shown in the screenshot of Figure 93. Strictly spoken, only source and header files (*.c/*.h) are Source Code. All other files are supporting information for Embedded Coder itself, but also for many verification tools like SLCI, Polyspace, or SIL simulation. Without this information, those tools do not work correctly. In addition, these files document the code generation process, e.g., by the structural checksum of the model or a copy of the code generation settings.

As consequence, the chosen approach is to put all files in the code generation folder under source control. The report generation has been disabled, it is only generated on demand. Then the generated artifacts are easier to predict. Important is to ensure that a clean rebuild is performed before the source code is submitted.



```
⊞ _sharedutility_src
⊞ fc_fcc_ert_rtw
⊟ slprj
    ⊟ ert
        ⊞ _sharedutils
        ⊞ fc_actuatorLoop
        ⊟ fc_AHRSVoter
            ⊞ tmwinternal
              buildInfo.mat
              codedescriptor.dmr
              defines.txt
              fc_AHRSVoter.bat
              fc_AHRSVoter.c
              fc_AHRSVoter.h
              fc_AHRSVoter.mk
              fc_AHRSVoter_mr_codeInfo.mat
              fc_AHRSVoter_private.h
              fc_AHRSVoter_ref.rsp
              fc_AHRSVoter_types.h
              modelsources.txt
              rtw_proj.tmw
        ⊞ fc_fcc
        ⊞ fc_innerLoop
        ⊞ fc_outerLoop
        ⊞ fc_rolloff
      sl_proj.tmw
```

Figure 93: Example code generation output

8.1.2.5 *Standard C Library*

A Standard C Library implements various functions, for which the C standard only provides function prototypes. Embedded Coder by default assumes the availability of a Standard C Library and swaps some calculations by just calling the respective function prototypes. A typical example is the `TrigonometricFunction` block, which is transformed to a function call directing into the Standard C Library (e.g., `sin(…)`).

For safety-critical applications, the standard library is typically replaced, since it must be developed under DO-178C as well. CR 10 specifies the integration of a Cert Standard C Library with available function signatures. Theses signatures deviate from the C Standard Library specified by C99.

If C Standard Library functions are named or called differently, Embedded Coder must be instructed to bend the relevant function calls. In principle, two practical solutions exist. Either a so-called Code Replacement Library (`crtool`) is registered, which replaces code fragments during the code generation process, reaching from operators to functions calls [96, pp. 51-1ff]. Or pre-compiled S-Function blocks with code generation assets to replace the relevant functions in simulation and inject custom code during the code generation process with the Legacy Code Tool [166, pp. 4-44ff] are provided.

The two concepts have been investigated concerning their applicability for the given process. The results are listed in Appendix D. Under the assumption that the C Standard Library implements equal behavior as the replaced SL functions, Code Replacement Libraries are chosen, since they are the more flexible approach.

Only functions explicitly involving special quantities, like `isnan`, use the Legacy Code approach. Reason is that the original blocks checking the special quantities are not compliant with SLCI.

8.1.2.6 *Code generation settings*

The code generation settings are manifold. Discussing each chosen setting would clearly go beyond the scope of this thesis. Anyway, a few settings required some special considerations when it comes to certification-ready code generation and are explained in Appendix E.

8.2 Verification tasks

8.2.1 SwVP-DP-MB 1 – Static model analysis

Main objective of this task is to verify conformance of the detailed design with the Software Model Standard, in particular the design, traceability, module design, and fundamental modeling rules. SL Model Advisor is a tool to execute independent checks in the MATLAB and SL/SF environment in batch. Therefore, built-in checks but also own checks can be used. For own checks, SL Model Advisor provides an infrastructure to traverse the data model of SL models and libraries. Using SL Model Advisor in a process context required special consideration and adaptions.

Contribution 24: A task for static model analysis has been defined and implemented. Resources have been created that allow the automated execution and result assessment of SL Model Advisor checks for models. Checks have been implemented for new and adapted modeling rules. A new categorization for the criticality of check violations has been introduced. The SL Model Advisor results have been integrated into the process-oriented build tool.

Application

Figure 94 illustrates the workflow and the created supporting material. Input for execution of the analysis are a configuration set, which defines the checks to run, and the implementations of custom checks.

The approach is to run all available checks on all SL models *and* on SL libraries in a module. The decision, whether a check is applicable for the library, is made by the check itself (e.g., since libraries do not have configuration settings, related checks cannot be executed). Primary output are intermediate result data for each analyzed model.

In this task, the embedded justification workflow is applied. The result data are imported into the process-oriented build tool and mapped to the standardized results. Findings can be assed in the build tool and turned into deviations by adding justifications.

Finally, two PDF reports are generated. One is the standard SL Model Advisor report containing information about executed checks and the result. The second report mirrors the justifications of the embedded review.

This task is a module-level task (D^M), i.e., it has to be executed for every module including the component module. However, the check behavior itself may differ for a top-level and nested models, but the application remains the same from the user’s point of view.

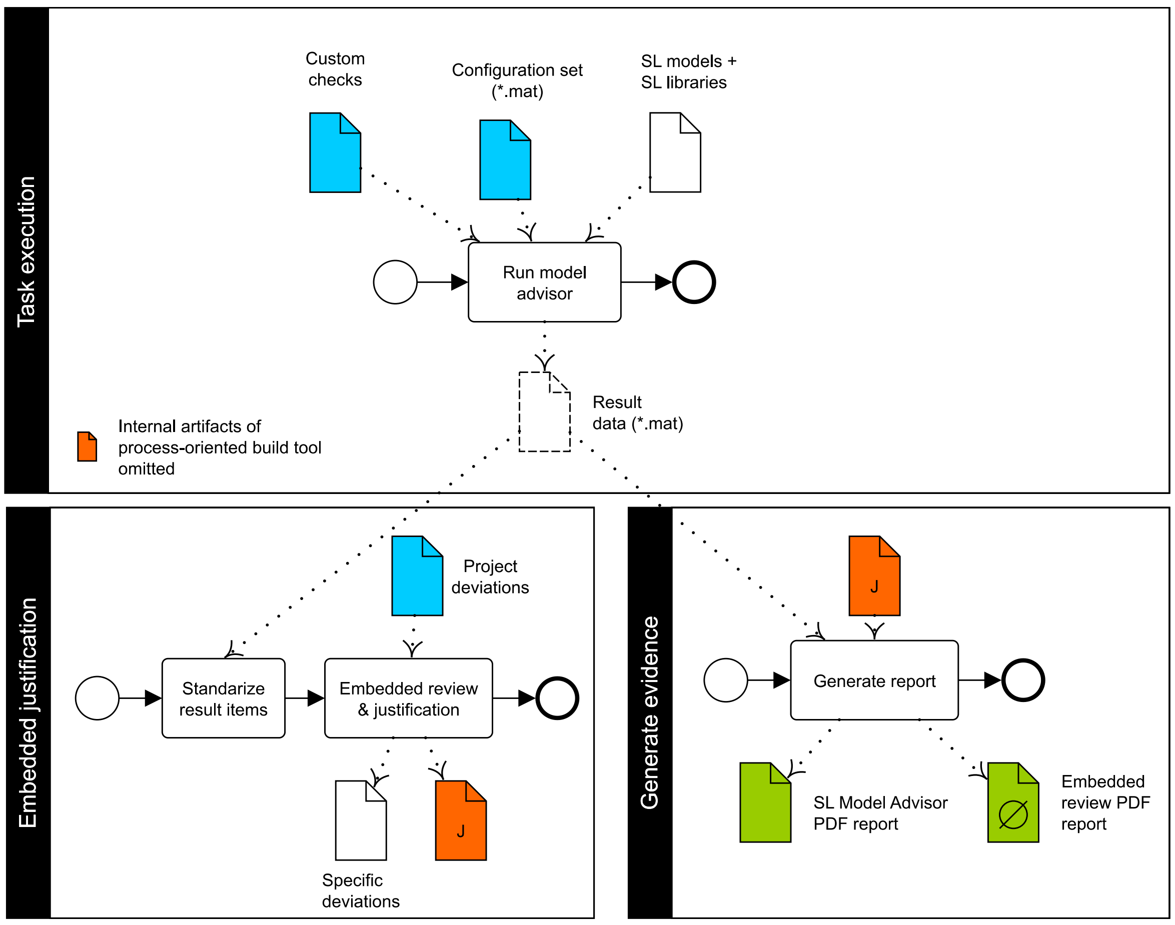


Figure 94: Workflow and artifacts for SwVP-DP-MB 1

It is important to execute all checks from the root level of the respective SL model, never from subsystems. Technically, the tool allows the execution starting at model-level, but also at subsystem level. However, the impact of a change in the model is typically not clearly isolatable to a subsystem. Some checks are even not assessable on subsystem level. Custom checks are also much easier to construct, if they do not have to distinguish between root- and sub-level entry point. To ensure this, a check has been added verifying that the entry point of the analysis is the root-level of the SL model.

Prerequisites

In order to prepare SL Model Advisor for the task, prerequisites had to be created and deployed with the modeling environment:

- Custom checks implemented in MATLAB for module design, traceability rules, and fundamental modeling rules
- Configuration file selecting and grouping relevant checks
- Traceability from rules to checks
- Detailed job implementation for execution and result standardization
- Detailed job implementation for PDF report generation

The set of checks is a mixture of checks created in this work and checks shipped with Simulink Check for DO-178/DO-331 compliance [167, pp. 3-41ff.], MISRA C compliance [167, pp. 3-63ff], MAAB compliance [167, pp. 3-56ff], or SLCI compatibility [135, pp. 2-5ff]. Since the fundamental modeling rules are derived from these general sets, numerous rules are directly covered by them. For own modeling rules and adapted rules, new checks have been created where feasible.

Since modeling rules vary significantly in their safety impact, they have been assigned to different categories of criticality, in particular safety, compatibility, and quality. This mitigates the necessary effort for verification and tool qualification. The influence of and reason for categorization has been presented by Hochstrasser [36] and is summarized in Table 35.

	Safety	**Compatibility**	**Quality**
Impact	If not respected, hard to find and/or critical errors may be introduced. Negative impact possibly undetected in verification activities.	If not respected, rework is required. Safety relevant impact is detected in verification activities.	No direct influence on safety, but on readability, usability, or maintainability of the model.
Justification of Finding	No justification possible, rework required.	Justification possible, rework required if not justifiable.	Rework required if specified thresholds are violated.
Tool Qual. Verification of Checks	Required	Not required	Not required

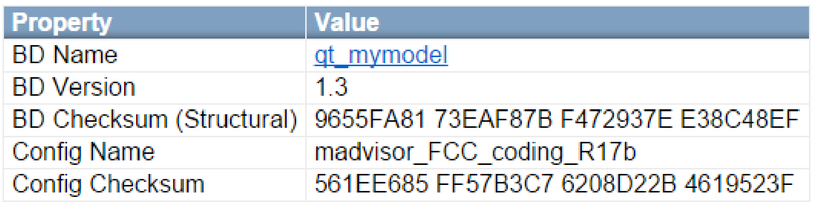
Table 35: Criticality of modeling rules and checks

Authoring custom check requires some experience. Some guidelines are given by Jaffry [168]. The following list provides additional guidelines gathered by the author during check implementation:

1. SIDs are used for identification of blocks wherever possible. There are different APIs for Stateflow and Simulink model primitives as well as different IDs (Simulink ID, Block path, Stateflow ID), but SIDs are the most consistent identification method.
2. The assumption that checks are only executed from the root-level model and not from subsystems has been respected in check development. This drastically decreased complexity and implementation effort.
3. Compile checks must follow and analyze library links. Non-compile checks don't have to follow library links, since they are executed on SL libraries, too. Checks requiring a compilation cannot be executed on SL libraries, so they must be analyzed from the context they are referenced in. However, returning results of referenced libraries is not the default setting for various API functions (e.g., `find_system`) and must be activated.
4. The content of pseudo primitives (i.e., subsystems in the DO-332 Foundation Library, cf. section 5.6.5.5) is not analyzed. They cannot be changed by the developer and are assumed to be sufficiently verified.

5. All checks look under all type of masks to avoid that a user can exclude a subsystem from checks by adding a mask (e.g., with dialogs).

In order to improve configuration management and traceability, a check has been created to embed the structural checksum of models and file checksum of library files into the report, as well as a checksum of the check configuration set. The report by default does only include the model version. This is an extended version of check “Display model version information” [169, pp. 2-79f].



Property	Value
BD Name	qt_mymodel
BD Version	1.3
BD Checksum (Structural)	9655FA81 73EAF87B F472937E E38C48EF
Config Name	madvisor_FCC_coding_R17b
Config Checksum	561EE685 FF57B3C7 6208D22B 4619523F

Figure 95: Extended configuration information in SL Model Advisor Report (BD = Block Diagram)

Result evaluation and justification

SL Model Advisor provides the capability to exclude parts of the model from analysis [167, pp. 3-28ff]. This workflow is considered as dangerous from a safety point of view, since exclusions are persistent model overlays and not clearly visible. If a subsystem is excluded from a check, future changes will not be analyzed by the check anymore. In addition, exclusions and justifications should be kept separate from the Design Model. Checks have been implemented to make sure that no exclusions exist. The embedded justification workflow of the process-oriented build tool has been leveraged instead.

Prior to review, the results of the checks are standardized as in Table 36. Model Advisor itself knows the execution states “pass”, “warn”, and “fail”. Checks only fail, if an abnormal condition in check execution occurs (e.g., a model, which cannot be compiled, or a bug in the check). A check throws a “warn”, if the checked rule is violated, and a “pass” if not.

How warnings are treated depends on the criticality of the check, which is derived from the criticality of the connected modeling rules. From all justifiable result items, a review checklist is auto-generated (one per SL model/library).

SL Model Advisor Check Result	Check Criticality	Resulting Standardized Status	Comment
Fail	Any	☒ (FAIL)	A failing SL Model Advisor check is not justifiable and must be followed up. It is caused by an abnormal condition, e.g., by a model, which cannot be compiled, missing dependencies, or a bug in the check itself.
Warn	Safety	☒ (FAIL)	Violation of checks categorized under “safety” are not justifiable, since a failure may have a severe impact on safety of the software.
	Compatibility	! (WARN)	Compatibility checks can be justified, since a rationale exists that a tool downstream of the workflow detects possibly caused safety issues.
	Quality	✓ (PASS) or ☒ (FAIL) if number of violations > 2% of number of model elements	The sum of quality check violations must lie below a certain threshold. If this threshold is exceeded, a non-justifiable failure is thrown.
Pass	Any	✓ (PASS)	

Table 36: Status mapping for SwVP-DP-MB 1

Limitations

Since executed per model/library, this task just knows the model scope. Module-level assessments, like checking for unused model data, is not covered by this task.

Although configuration settings are globally shared in the module, they are checked for each model. They should be moved to the Static Module Analysis in future.

At the time when this thesis is written, not all rules are covered by checks. Uncovered rules are subject to review (cf. SwVP-DP-MB 3). Also only one of the two PDF reports is implemented.

The strategy proposed by MathWorks is to qualify the checks in tool qualification, not SL Model Advisor itself. The majority of DO-178C checks (cf. [167, pp. 3-41ff]) is covered by the DO Qualification Kit (cf. Simulink Check Tool Operational Requirements [170]). For custom checks, the provided tool qualification has to be extended (at least for those categorized under “safety”). The execution complies with [170].

The automatic generation of review lists after analysis is also not part of the existing tool qualification, but the DO Qualification Kit does not cover the built-in Model Advisor exclusion feature either ([170] p. 2-1).

8.2.2 **SwVP-DP-MB 2 –** Static module analysis

Main objective of this task is to verify conformance with the Software Model Standard on module-level, i.e., check modeling rules, which span more than one model and have project scope. These rules are mainly part of the module design and traceability rule sets.

Contribution 25: A task for static module analysis has been defined and implemented. Resources have been created that allow the automated execution and result assessment of SL Model Advisor checks for whole modules. Checks on module level are not part of any check set shipped by MathWorks and therefore represent an innovation. They are special for the modular process at hand.

Application

This task uses the SL Model Advisor framework. In contrast to SwVP-DP-MB 1, the task is not executed for a specific model, but once per module. Since SL Model Advisor requires a target model, the check is executed on the built-in Simulink block library, but it analyzes the whole project.

Some examples for checks are:

- Conformance of model data in the data dictionaries with the safe subset
- Detection of dead units and data (e.g., unreferenced private SL models)
- SL Project sanity analysis (e.g., files that have not been added to the project)
- Encapsulation requirements check (e.g., usage of private SL models of other modules)
- Top-down requirement traceability analysis (e.g., verify that all requirements have a trace to the Design Model)

This task is a module scope task (D^M) and a component scope task ($D2^C$). A couple of checks have to be executed for every module including the component module and some checks, like detection of unused, public SL models, can just be executed on component level.

Prerequisites

The described types of checks are not part of any existing check set. All of them had to be implemented. For check implementation, the aspects described in SwVP-DP-MB 1 hold.

Result evaluation and justification

Result evaluation is equal to SwVP-DP-MB 1.

Limitations

The limitations concerning tool qualification for custom checks are equal to SwVP-DP-MB 1.

Not all module design rules have already been covered with checks. Many rules still require manual review. Additional effort should be spent in future to implement them.

8.2.3 SwVP-DP-MB 3 – Model review

The main objective of the task is to verify the conformance to the Software Model Standard for those rules, which are not automatically checkable in tasks SwVP-DP-MB 1 and SwVP-DP-MB 2. Examples are the implementation of interfaces compliant to the specification or use of units.

The following presents an approach that leverages automatic, incremental review list generation from the Software Model Standard leveraging the embedded justification workflow. Incremental means that existing review lists are updated, deprecated justifications are highlighted and new findings are appended. This drastically reduces review effort.

Contribution 26: Dynamic check lists have been implemented for model review tightly integrated in the process-oriented build tool. The dynamic check lists are auto-generated and inherit the full feature set of the process-oriented build tool, like automated evaluation of up-to-dateness. This significantly reduces the review effort.

Application

During task execution, for each container primitive, a status file is created, containing all relevant review items. The checklist is applicable for the respective layer of the container primitive, only. Up-to-dateness, i.e., whether the content of the container primitive has changed and review is required again, is checked for each SL model or atomic subsystem by the structural checksum and for virtual subsystems with the structural checksum of the closest parent.

The machine-readable checklist is imported into the process-oriented build tool during standardization of result items. Figure 85 illustrates the embedded review workflow and Figure 85 shows a screenshot of the review list.

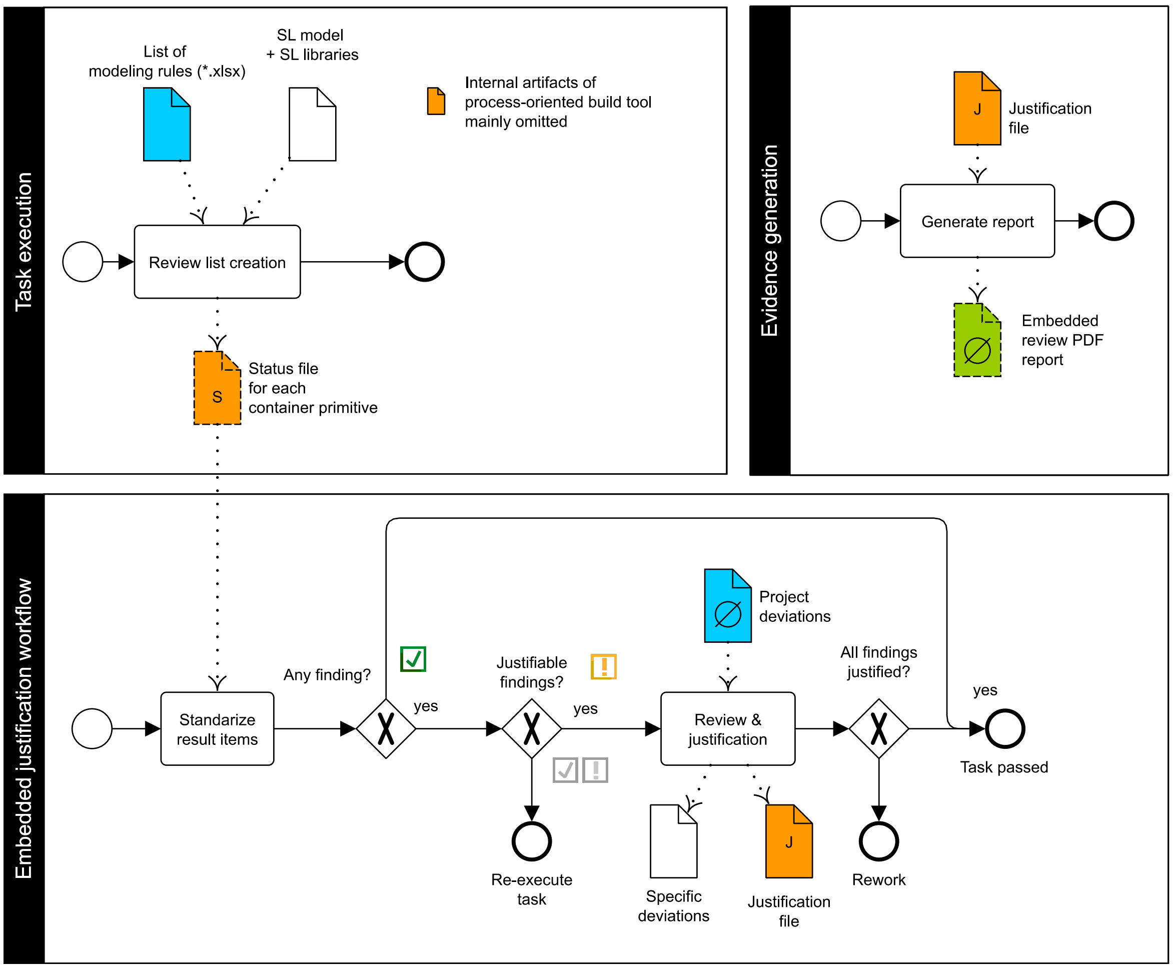


Figure 96: Workflow and artifacts for SwVP-DP-MB 3

Prerequisites

In order to identify rules, which are subject to model review, traceability to checks of tasks SwVP-DP-MB 1 and SwVP-DP-MB 2 has been established. Scripts have been implemented to transform and replicate the manually written checklists.

Result evaluation and justification

By default, all review items have the standardized status WARN. If no findings are found, the status is manually changed to PASS. If findings are found, a justification has to be added and the status is also changed to PASS.

Review Item Status	Standardized Status	Comment
Unreviewed	! (WARN)	Manual review and potential justification.
Reviewed	✓ (PASS)	No finding or approved deviation.

Table 37: Status mapping for SwVP-DP-MB 3

Limitations

A structural checksum is the only way to assess up-to-dateness of a subsystem. A structural checksum only reflects functional changes, not, for example, the appearance of the diagram. Those changes would not affect the up-to-dateness of the model. I.e., modeling rules addressing appearance are not correctly covered. However, the resulting risk is low, since these are quality rules.

Furthermore, the structural checksum also includes the checksum of all children. There is no checksum per layer, i.e., for deep subsystem hierarchies, the structural checksum of higher-level subsystems is a significant over-approximation.

The automatic generation of review lists is also not part of any existing tool qualification.

In the current implementation, task execution only identifies rules, which are not automatically verified, and copies them into the review list. In future, this process might also involve certain logic and dynamic. Not every rule is applicable for the content of every container primitive. For example, some rules target specific SL blocks, which might not be there. Enriching the review list creation with such logic would significantly decrease review work. The given implementation is a first step into this direction.

Generation of a PDF report of the embedded checklist has not been implemented yet. Project deviations have not been collected yet.

8.2.4 SwVP-DP-MB 4 – Traceability review and analysis

Objective of this task is to review traceability between Design Model and HLRs and verify, if the design complies with the requirements and if traces are correct. This refers to the traceability rules of section 6.5, which could not be analyzed with automated checks.

Contribution 27: Dynamic check lists have been implemented for traceability review. They are tightly integrated in the process-oriented build tool. The dynamic check lists are auto-generated and inherit the full feature set of the process-oriented build tool, like automated evaluation of up-to-dateness. This significantly reduces the review effort.

Application

The application aligns with SwVP-DP-MB 3. Instead of model review lists, bottom-up review lists are auto-generated for every model and a top-down review list is generated once per module. Templates for these checklists have been developed in this work.

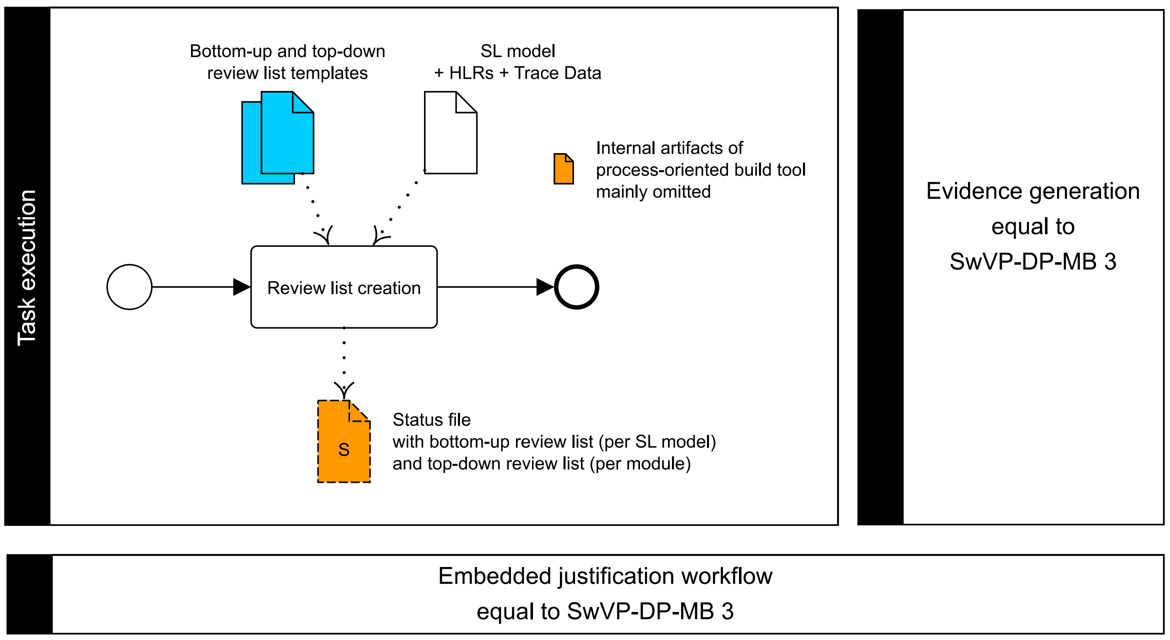

Task execution
Bottom-up and top-down review list templates
SL model + HLRs + Trace Data
Internal artifacts of process-oriented build tool mainly omitted
Review list creation
S
Status file with bottom-up review list (per SL model) and top-down review list (per module)
Evidence generation equal to SwVP-DP-MB 3
Embedded justification workflow equal to SwVP-DP-MB 3

Figure 97: Workflow and artifacts for SwVP-DP-MB 4

Prerequisites

Scripts have been implemented to transform and replicate the manually written checklists. Both checklists are also unique for the given process. Common traceability review checklists from traditional DO-178C reviews cannot directly be used, since they have been assembled for textual requirements. The MathWorks DO Workflow [32] does not include checklists for model-based software development, either. The created checklists address the specific scenario, in which traceability from SL models to Polarion is established with *SimPol*.

In addition to the checklists, some checks have been implemented to replace at least some of the review items. Those are executed as part of SwVP-DP-MB 1 and SwVP-DP-MB 2. They supplement the check for high-integrity guideline “hisl_0070” [103, pp. 8-2ff], which does not cover the special use of libraries or model data (TR 6 and TR 7). Also, the review of certain rules is supported by analysis with *SimPol*.

Result evaluation and justification

Equal to SwVP-DP-MB 3.

Limitations

Equal to SwVP-DP-MB 3, but in contrast to SwVP-DP-MB 3, review lists are not generated for every atomic subsystem, but for whole models, only.

For supporting tooling with *SimPol*, no tool qualification exists.

Generation of a PDF report of the embedded checklist has not been implemented yet.

8.2.5 SwVP-DP-MB 5 – Design error detection

This task has two main objectives:

- Assurance-guarantee analysis
- Run-time error analysis (i.e., division by zero, index out of bounds, integer overflow)

In addition, dead logic detection is performed, but without seeking certification credit for it.

The assurance-guarantee analysis shall prove that the signal ranges specified at narrow interfaces of modules are satisfied under all circumstances. Signal ranges are not tested for robustness. It is thus mandatory to prove their correctness. Detection of run-time errors and dead logic are based on the correctness of the assurance-guarantee analysis.

Contribution 28: Simulink Design Verifier has been adapted to perform assurance-guarantee analysis, run-time error analysis, and dead-logic detection. Resources have been implemented that allow the automated execution and result assessment. Additional functionality has been added to support the modular approach of the process at hand.

Application

All analyses are performed with SL Design Verifier. Assurance-guarantee and run-time error analyses are performed in one execution, whereas the dead logic analysis is a separate run.

Design Verifier analyses are executed on a model and are greedy, i.e., they always analyze the whole nested model hierarchy by default, i.e., it is sufficient to execute them on all public SL models[41].

Figure 98 illustrates workflow and artifacts.

[41] The Test Generation Advisor can analyze subcomponents, but for dead-logic detection and test case generation only [123, pp. 7-21ff].

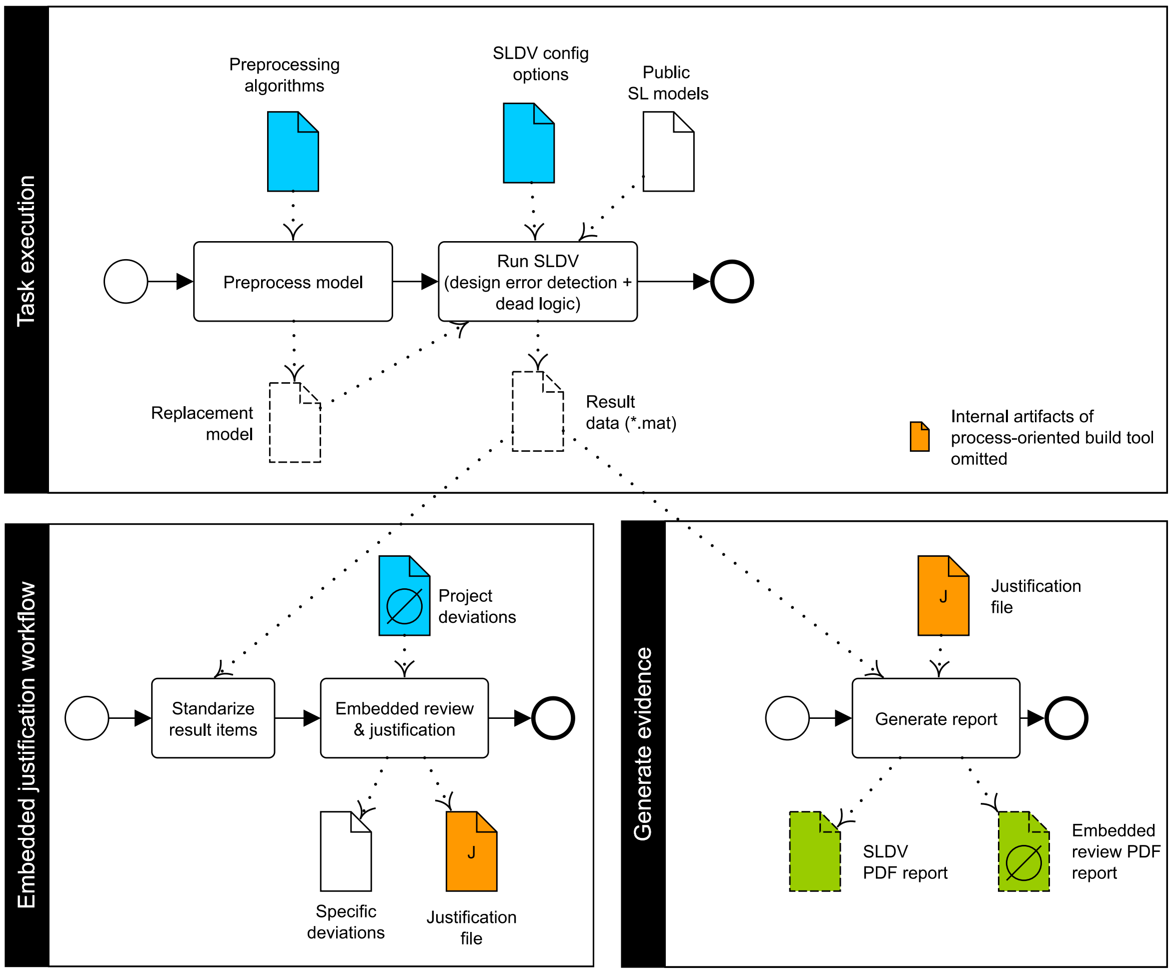


Figure 98: Workflow and artifacts for SwVP-DP-MB 5

Tool overview

For further understanding, a short discussion about Design Verifier and the applied methods is required. According to the tool qualification plan, Design Verifier internally applies *abstract interpretation* and *Satisfiability(SAT)-based model checking*. For abstract interpretation it uses the engine of Polyspace Code Prover, for model checking the Prover Plug-In of Prover Technoloy[42]. [171, p. 4-4]

These types of analyses are called formal methods and they have to be *sound*. An analysis is *sound* for a specific property, if it does not produce false positive results, or in the words of DO-333, it “never asserts a property to be true when it is not true (DO-333 p.4)”. However, results can remain undecided or return false negative results, i.e., raise a failure if there is actually no failure.

[42] https://www.prover.com/software-solutions-rail-control/

Abstract interpretation is a method of program analysis, which computes an over-approximation of value ranges possible during program execution in a suitable abstract domain. The abstraction is designed to preserve the validity of certain properties. One example is the derivation of value ranges. If divisor $x \in [lb, ub]$, then abstract interpretation can prove division by zero for larger derived ranges $[lb_a, ub_a]$ with $[lb, ub] \in [lb_a, ub_a]$. Disadvantage is that during the abstraction process, values ranges can get wide pretty soon and the method may return false negative results. Abstract interpretation performed by the Polyspace Code Prover engine is, compared to model checking, fast and bit-precise (i.e., supports floating-point values according to IEEE-748 [172]). Further information can be found in the description of the theoretical foundation as part of the DO Qualification Kit [173].

Model checking is a fully automated technique to check properties expressed in computational tree logic expressions (so-called CTL*). Most tools, like Design Verifier, work on a special subset of CTL*, the linear temporal logic (LTL). In LTL, only safety and liveness properties can be specified (cf. Table 38). Reachability, fairness and the freedom of dead lock cannot be expressed. Design Verifier applies symbolic model checking with a SAT and a LP ("linear programming")-based solver representing the model as a state transition system (an automaton with a single state) (cf. [171], which provides further references for the theoretical background).

Reachability	Safety	Liveness	Fairness	Dead Lock Freedom
A particular situation can be reached.	Under a certain condition, an event never occurs.	Under a certain condition, an event will ultimately occur.	Under a certain condition, an event will occur (or will fail to occur) infinitely often.	The system can never be in a situation in which no progress is possible

Table 38: Property categorization from [35] and [174]

Both tools apply *over- and under-approximations*. Over-approximations do not have an impact on the validity of results, under-approximations may have. The Code Prover documentation understands over-approximation under the term approximation, which includes abstraction and preserves exhaustiveness [175, p 4-4]. If Design Verifier reports an approximation, it is an under-approximation. The analysis is not sound in this case [123, pp. 2-15ff]. Most limiting under-approximation necessary for model checking is the approximation of floating-point values with rational numbers, which does not represent rounding and finite precision.

Another method used by Code Prover and Design Verifier to simplify the analysis is *stubbing*, which replaces unsupported blocks with stubs that only have an identical interface, but no or also over-approximated behavior. Design Verifier performs automatic stubbing, which is sound, and supports custom stubbing.

In case of auto stubbing, if Design Verifier reports an objective as valid, it is valid independent of over-approximation due to stubbing. Design Verifier performs stubbing automatically, if enabled, for supported incompatible blocks [123, pp. 2-8ff], but flags the model as "partially compatible" during compatibility checks. Automatic stubbing is not covered by tool qualification kit [176, p. 2-5]. Automatic stubbing works for built-in blocks and is not customizable. In some cases, it can introduce a significant over-approximation, although deeper knowledge about the stubbed system or block is known.

To register custom stubbing, Design Verifier provides the *block replacement* mechanism [123, pp. 4-1ff]. Block replacement is a customizable preprocessing step of the model. If block replacement is applied, the user must ensure that the replacements are valid over-approximations.

Prerequisites

In R2017b, Design Verifier comes along with several limitations and incompatibilities for the chosen safe modeling subset by default. To support the process and modeling rules at hand, significant preprocessing algorithms for the models had to be implemented. They are documented in Appendix F.

In addition, a Design Verifier configuration has been chosen. Table 39 maps the previously introduced task steps to the used Design Verifier technique.

Task Step	**Design Verifier Technique**	**Applied Formal Method**
Assurance-guarantee analysis	Design range checks [123, pp. 6–29ff]	Abstract interpretation
Run-time error analysis	Static run-time error detection [123, pp. 6-24ff] • Division by zero • Index out of bounds • Integer under- and overflow	Combination of model checking and abstract interpretation
Dead logic detection	Quick dead logic detection [123, pp. 6-10ff]	Abstract interpretation

Table 39: Applied SL Design Verifier techniques

Design Verifier performs “Design Range Checks", i.e., based on provided range information, further signal ranges are formally derived and checked. This allows Design Verifier to decide whether specified intermediate ranges are satisfied. Design Verifier static run-time error detection bases both on abstract interpretation and model checking and reveals the listed design flaws. For example, division by zero is either decided using Polyspace or Prover.

Dead logic detection requires some additional explanations. Design Verifier offers two options: Dead logic detection without active logic detection (called “Quick Dead Logic Detection” or “Reduced Capability” [176, p. 2-1]) or with active logic detection (“Full capability” [176, p. 2-1]).

Quick Dead Logic detection bases on Polyspace Code Prover abstract interpretation, only. However, dead logic is a reachability property, which cannot be preserved with value range abstraction. Larger, over-approximated signal ranges may lead to false positive results. Anyway, this kind of analysis is fast, can reveal certain dead logic patterns, and requires fewer under-approximations. It is just sound for some known patterns, but not overall [123, p. 6-10]. Active logic checks decision and condition coverage on the returned results of the quick dead logic check. This method is much slower, but is sound, if it doesn’t require under-approximations (like floating-point to rational approximation). Only this approach is covered by tool qualification [176, p. 2-5].

The proposed approach is to take the quick dead logic analysis without seeking certification credit for it, since reachability is thoroughly satisfied by model coverage afterwards.

Result evaluation and justification

Prior to each execution, Design Verifier performs a compatibility check. The interpretation of the outcome is listed in Table 40.

Design Verifier Compatibility check Status	**Standardized Status**	**Comment**
Compatible	☑ (PASS)	
Partially compatible	☑ (PASS)	Indicates incompatible blocks, which are automatically stubbed, but is sound.
Not compatible	☒ (FAIL)	Design Verifier cannot be executed. The model must be reworked.

Table 40: Status mapping for SwVP-DP-MB 5 (compatibility checks)

The results of the actual analysis are standardized and subject to the embedded justification workflow in the process-oriented build tool.

The result states for assurance-guarantee and run-time error analysis are explained in the Design Verifier documentation [123, pp. 13-34ff], but an interpretation is not given. Thus the mapping shown in Table 41 has been assembled. Only fully valid objectives are not subject to review. Just in this case, the analysis is sound. Whenever an approximation is involved, or if the objective is undecided, manual review and justification is required.

Design Verifier Objective Status	Result of Generated Counter Example (simulation case)	Standardized Status	Comment
Valid	No counter example needed	☑ (PASS)	A valid objective has been produced in a sound analysis.
Valid under approximation	No counter example needed	⚠ (WARN)	If an under-approximation is applied, the method is not sound. Cases might exist, in which the objective is not valid. The objectives must be manually reviewed and justified.
Undecided due to stubbing	No counter example needed	⚠ (WARN)	Due to technical or mathematical reasons, Design Verifier cannot provide a result. Manual review and justification of the objectives is required. In case of a timeout, a rework of the model should be done to reduce verification times.
Undecided due to nonlinearities	No counter example needed	⚠ (WARN)	
Undecided due to division by zero	No counter example needed	⚠ (WARN)	
Undecided due to timeout	No counter example needed	⚠ (WARN)	
Falsified – needs simulation	Simulation test case returns valid result	⚠ (WARN)	If an objective is falsified, Design Verifier normally can provide a test case. If approximations are present, these test cases can produce a valid result in rare cases. Then the objective shall be reviewed and justified. If the test result is invalid, it is the proof for a design error.
	Simulation test case returns invalid result	☒ (FAIL)	
Falsified – no counterexample	No counter example available	☒ (FAIL)	In some cases, Design Verifier fails to find a counterexample (e.g., if the model has no inputs). Since this could be observed in edge cases only, where the design was questionable, no possibility of justification is provided.

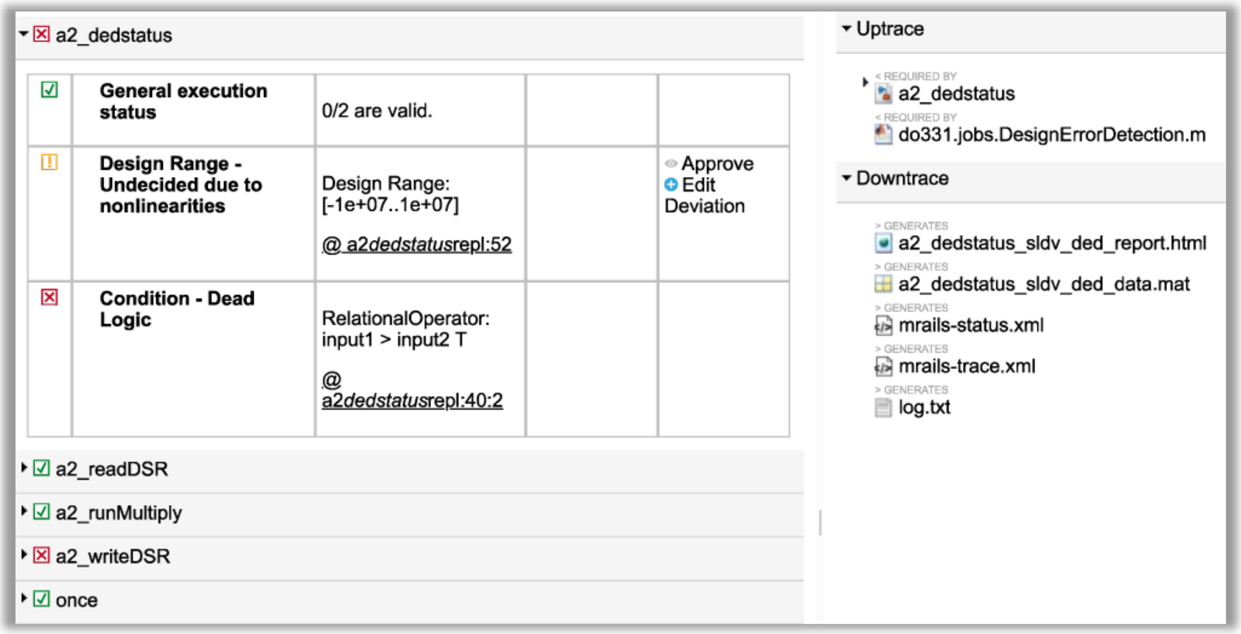
Table 41: Status mapping for SwVP-DP-MB 5 (assurance-guarantee and run-time error)

For dead logic, the relaxed mapping of Table 42 applies. This is argued with the fact that reachability is sufficiently covered by model coverage. Reviewing and justifying undecidable objectives would be duplicate work. In addition, quick dead logic analysis does not return undecidable objectives.

Design Verifier Objective Status	Standardized Status	Comment
No dead logic	✓ (PASS)	Directly maps to the result states of the quick dead logic analysis, which does not return undecidable items.
Dead logic	☒ (FAIL)	

Table 42: Status mapping for SwVP-DP-MB 5 (quick dead logic)

Figure 99 displays the review checklist generated after the analyses in the process-oriented build tool. It allows jumping into the model quickly and overlays the produced results (Design Verifier result view). Setting a deviation is enabled according to the standardized status in the mapping tables.


☒ a2_dedstatus

✓	**General execution status**	0/2 are valid.		
!	**Design Range - Undecided due to nonlinearities**	Design Range: [-1e+07..1e+07] @ a2dedstatusrepl:52		Approve Edit Deviation
☒	**Condition - Dead Logic**	RelationalOperator: input1 > input2 T @ a2dedstatusrepl:40:2		

✓ a2_readDSR
✓ a2_runMultiply
☒ a2_writeDSR
✓ once

Uptrace
REQUIRED BY a2_dedstatus
REQUIRED BY do331.jobs.DesignErrorDetection.m

Downtrace
GENERATES a2_dedstatus_sldv_ded_report.html
GENERATES a2_dedstatus_sldv_ded_data.mat
GENERATES mrails-status.xml
GENERATES mrails-trace.xml
GENERATES log.txt

Figure 99: Example of auto-generated review checklist for SL Design Verifier

Limitations

The presented analysis configuration significantly deviates from the tool operational requirements and needs major enhancements of the tool qualification kit.

In order to make Design Verifier usable, significant transformation of the model is required. This would require significant custom tool qualification enhancements.

Experience also showed, that, if the model is not of logical nature, approximations have a large impact on the results and many objectives have to be reviewed [35]. However, if no analysis were applied, these objectives would have to be reviewed anyway. A weaker application would be to let Design Verifier generate the objectives, transform them to review lists and manually check those. The objective generation would be fully covered by the tool qualification kit, since it is not affected by approximations and auto-stubbing and does not require the custom extensions made here.

The automatic generation of review lists after analysis is not part of the existing tool qualification.

The project deviations have not been collected in the scope of the thesis.

8.2.6 SwVP-DP-MB 6 – Simulation / test procedure and case development

The objective of this task is to derive test cases from HLRs. Implementing simulation cases from requirements underlies considerable freedom, since many different methods exist to create simulation cases. DO-178C requires the development of normal and robustness simulation cases. DO-178C 6.4.2.1 and 6.4.2.2 give a list of examples, which shall be covered. A further discussion of test strategies is out of scope of this thesis.

Contribution 29: Rules for simulation test procedure and case development with Simulink Test have been developed. These rules help organizing simulation test in a module, ensure efficient reusability for simulation and tests, and support the modular test and coverage collection system.

Application

In this task, normal and robustness simulation procedures and cases are manually derived from HLRs and derived requirements according to simulation rules (SR) documented in Appendix G. SL Test Manager [132, pp. 5-2ff] is used to author and manage these tests.

The derivation of simulation procedures and cases should follow known testing strategies, such as boundary value testing [45, p. 200]. Not used shall be Design Verifier test case generation to auto-generate test cases, since this is not requirements-based testing, but structural testing (cf. DO-248C FAQ #44).

Prerequisites

Typically, for test case development, a test plan is developed as proposed in [45, p. 206]. The rules specified therein are then checked in the test case review (SwVP-DP-MB 7). For this plan, a minimum set of testing rules has been assembled in Appendix G.

Furthermore, *SimPol* has been enhanced to allow linking of simulation cases in SL Test.

8.2.7 SwVP-DP-MB 7 – Simulation / test case and procedure review

Objective of this task is to check, if the simulation procedures comply with the rules specified in the test plan (simulation procedure review) and to verify that requirements-based coverage is given for HLRs (simulation case review).

Contribution 30: The jobs for dynamic checklists to review simulation/test cases and procedures have been set up. The dynamic check lists are auto-generated and inherit the full feature set of the process-oriented build tool, like automated evaluation of up-to-dateness. This significantly reduces the review effort.

Application

In its execution, this task is similar to SwVP-DP-MB 3 and SwVP-DP-MB 4.

A bottom-up review list is auto-generated for every simulation procedure/case. One top-down review list is generated per module. The main checklist is the bottom-up review list, which contains most review items for test rules applicable for the simulation procedures, but also items to review requirements-based coverage with the traceability from simulation cases to HLRs. The top-down review checklist is mainly to review requirements-based coverage with the traceability from HLRs and derived requirements to simulation cases.

Prerequisites

Templates for bottom-up and top-down review lists are required. The review list generation and result standardization has been implemented.

Result evaluation and justification

Equal to SwVP-DP-MB 3.

Limitations

Equal to SwVP-DP-MB 3 and SwVP-DP-MB 4.

The simulation rules are not yet translated to review lists.

8.2.8 **SwVP-DP-MB 8 –** Simulation testing & result review

In this task, all simulation cases specified in the module are executed and the results are assessed.

Contribution 31: Resources have been created to automatically execute simulation tests and assess the results. Dynamic check lists have been implemented for result review. They are tightly integrated in the process-oriented build tool. The dynamic check lists are auto-generated and inherit the full feature set of the process-oriented build tool, like automated evaluation of up-to-dateness. This significantly reduces the review effort.

Application

Each simulation case in SL Test Manager is executed separately and the results are also stored separately. I.e., simulations are not executed from higher hierarchical elements like a SL test suite or a whole file (cf. [132, pp. 5-2ff] for the test hierarchy in the SL Test Manager). This simplifies result artifact management, since otherwise any combination of result set could be stored. Recording of coverage happens in-situ with simulation, the coverage settings are not an execution parameter, but part of the test procedure. Coverage settings are discussed as part of SwVP-DP-MB 9 in detail.

After the execution, the raw simulation result is exported to the `.mldatx` format [177, pp. 1-90ff], which includes coverage and full result data for equivalence testing per simulation case. The results are imported into the process-oriented build tool. They are displayed, but cannot be justified. Finally, the results are exported to a PDF test report. The whole workflow is depicted in Figure 100.

Prerequisites

Scripts to execute and assess the test case results have been implemented as well as templates and scripts to exports the reports.

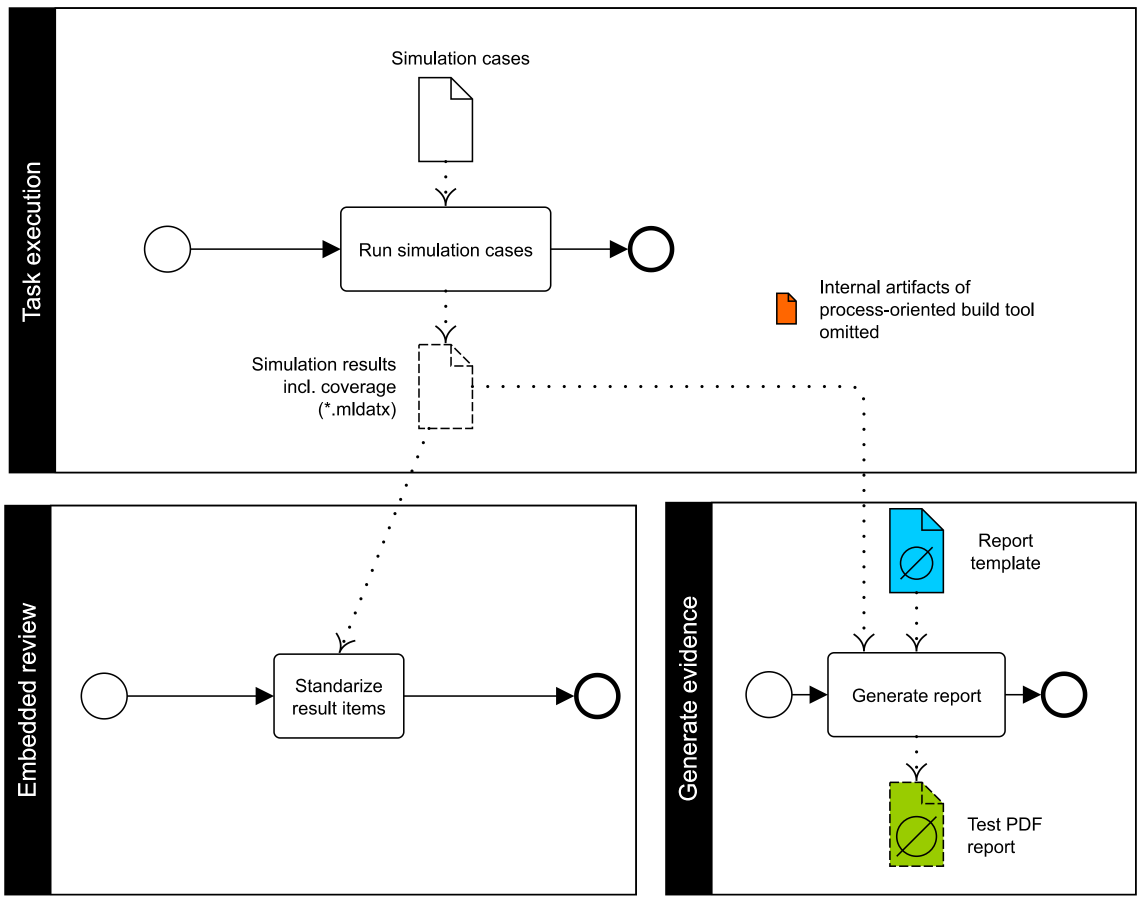


Figure 100: Workflow and artifacts for SwVP-DP-MB 8

Result evaluation and justification

The mapping of result standardization is given in Table 43 and is straight-forward. A justification is not possible in any case.

Review Item Status	Standardized Status	Comment
Pass	✓ (PASS)	
Fail	☒ (FAIL)	The pass/fail criteria falsified or the simulation has not been executed properly. Rework of either the Design Model or the simulation case is required.
Not executed	☒ (FAIL)	There shall be no simulation cases, which are not executed.

Table 43: Status mapping for SwVP-DP-MB 8

Limitations

This task has no specific limitations beyond tool qualification considerations.

The report generation has not been implemented and project deviations have not been documented yet.

8.2.9 **SwVP-DP-MB 9 –** Model coverage assessment

Model coverage has been introduced in section 4.6.3.2. As repetition, in this process, model coverage has two goals on module-level (D^M). It shall show that simulation cases cover LLRs. As side-product, model coverage shall provide an estimate for structural code coverage. In the component process ($D2^C$), it measures the control coupling achieved by simulation cases of the component interface.

Contribution 32: Criteria for model coverage assessment have been constituted. Resources have been created to aggregate model coverage per module and across modules. Such an automated coverage aggregation and assessment is not natively supported by MathWorks, but inevitable in a modular process. Model execution coverage is also applied in a new way to asses coupling between models.

Application

Model coverage analysis is executed on the results generated in task SwVP-DP-MB 8. Model coverage data has already been collected and stored as part of the simulation results. The settings for coverage are stored in the simulation procedure and have already been verified in SwVP-DP-MB 7.

In D^M, model coverage originating from the execution of *all simulation cases in the module* is assessed for every model *of the current module*. Coverage of nested models from other modules is assumed to have 100%.

In $D2^C$, model coverage originating from the execution of *simulation cases testing the component interface* is assessed for every model *of all modules,* except models in library modules.

As input to this task, all stored simulation case results are taken as illustrated in Figure 101. Each result only contains a portion of the final coverage, since one model is typically executed by many simulation cases for different execution paths. The coverage inside one simulation result is thus called a *coverage fragment* in the following. One coverage fragment may also contribute coverage to multiple models.

To obtain the complete coverage, all coverage fragments are aggregated. SL Coverage provides the necessary functionality [101, p. 3-21], since adding up coverage fragments is not straight-forward. Then, coverage for each model is separately stored as `.cvt` file.

In addition, a coverage filter file (`.cvf` file) is generated. Coverage filter files are a file format of SL Coverage to store exclusions and justifications attached to coverage [101, pp. 7-4ff]. The basic filter file contains exclusions for model references, which are not in the current module, since they are separately tested and coverage is assessed separately. Otherwise, a model would only achieve full coverage, if also all nested models achieved 100% coverage.

The review has been realized with the external justification workflow. Depending on the coverage result, justifications can be added to the filter file for particular blocks. Therefore, the SL Coverage graphical user interface should be used. During result standardization, the build tool applies the filter file on the coverage and analyzes the outcome. After all findings are resolved, a report is generated.

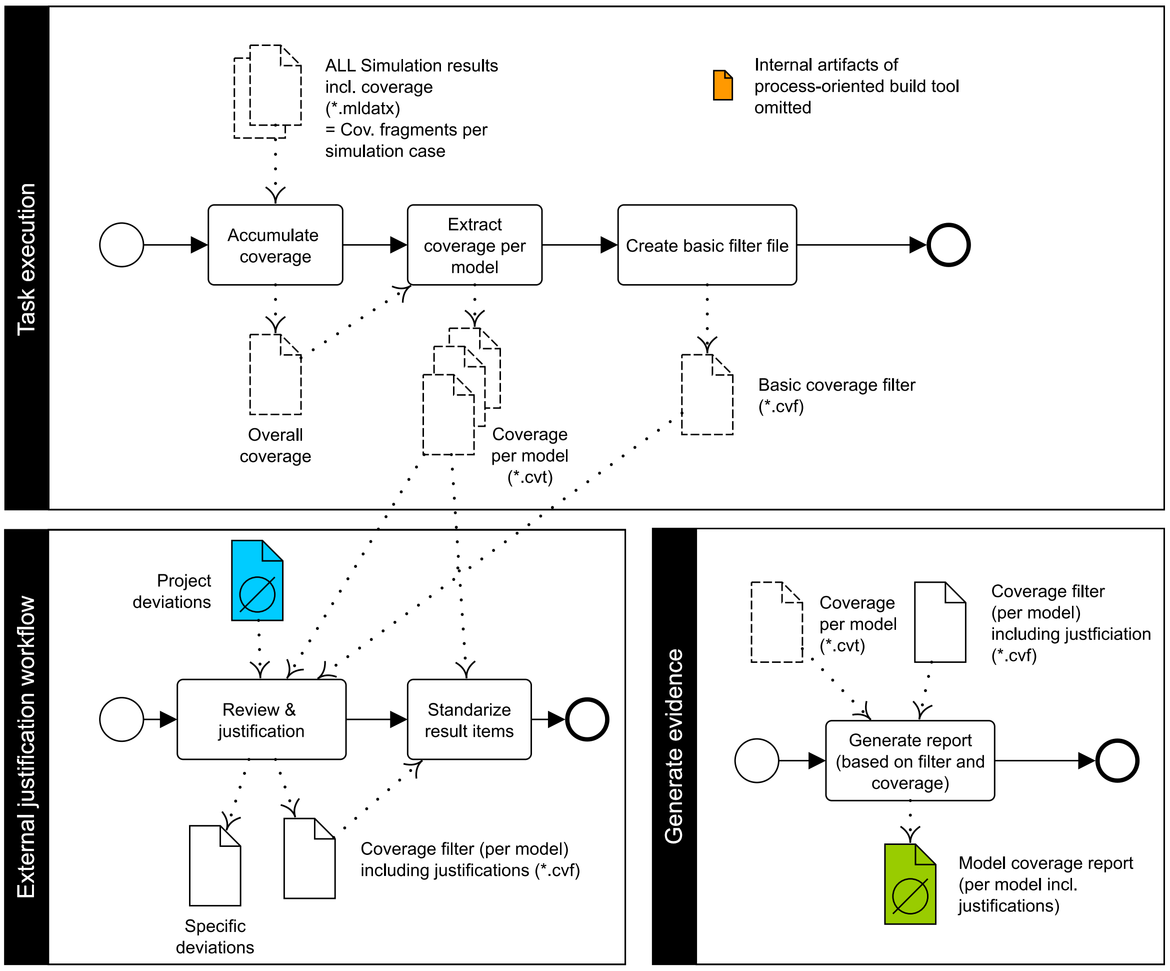

Task execution
ALL Simulation results incl. coverage (*.mldatx) = Cov. fragments per simulation case
Internal artifacts of process-oriented build tool omitted
Accumulate coverage
Extract coverage per model
Create basic filter file
Overall coverage
Coverage per model (*.cvt)
Basic coverage filter (*.cvf)
External justification workflow
Project deviations
Review & justification
Standarize result items
Specific deviations
Coverage filter (per model) including justifications (*.cvf)
Generate evidence
Coverage per model (*.cvt)
Coverage filter (per model) including justficiation (*.cvf)
Generate report (based on filter and coverage)
Model coverage report (per model incl. justifications)

Figure 101: Workflow and artifacts for SwVP-DP-MB 9

Prerequisites

The necessary algorithms for Figure 101 have been implemented in the build tool. The external justification workflow has been tightly coupled with SL coverage.

Furthermore, coverage settings had to be chosen providing the best outcome for the given process. Thereby, central question was, what kind of model coverage measures fulfill DO-331 criteria. DO-331 requirements on model coverage are minimalistic. Only a few general and not very helpful examples are provided in Table MB.6.1. Main requirement is that coverage is recorded with normal range and robustness simulation cases.

More interesting is the objective apart from DO-331, that it shall help to estimate structural code coverage and provide sufficient coverage preservation capabilities towards code coverage. Preservation means that, if full model coverage is achieved, it is very likely, that full structural coverage can be achieved with the same simulation / test cases on the auto-generated code as well.

The structural code coverage objectives for DAL B software are statement and decision coverage (cf. DO-178C Table A-7:6,7). In classical software engineering, decision coverage is equal to branch point coverage. In the sense of DO-178C, branch point coverage is only a subset of decision coverage as clarified by the CAST position paper [178]. The DO-178C understanding of decision coverage is (from [178] p.3):

- Every point of entry and exit in the program has been invoked at least once.
- Every control statement (i.e., branch point) in the program has taken all possible outcomes (i.e., branches) at least once.
- Every non-constant *boolean expression* in the program has evaluated to both a True and a False result,

A boolean expression is any expression resulting into `True` or `False` (DO-178C Glossary and DO-248C DP #13). In consequence, also an operation like `D=A&&B` is considered as boolean expression, regardless whether it controls a branch point, like `if(…)` (cf. [179] and [180]). Whether bitwise operations are Boolean expressions is under controversial discussion [181]. According to DO-248C DP #13, they are not.

SL Coverage supports eleven different model coverage measures in R2017b [101, pp. 1-3ff]. It is standing to reason to choose model decision coverage as best match, but this type of coverage bases on branch points as clearly visible in the list of covered blocks [101, pp. 2-2ff]. For example, a Rational Operator block is certainly translated to a boolean expression in code, however, the block does not have decision model coverage.

The closest match to DO-178C decision coverage, and the criteria chosen here, is assessing execution, decision, and condition model coverage. A condition is a boolean expression without boolean operator, for which a `True` or `False` result must be achieved. In `D=A&&B`, this would be `A=True`, `A=False`, `B=True`, and `B=False`. This is certainly more than code decision coverage, but in the opinion of the author, the better match than decision model coverage, only.

Beside coverage criteria selection, a couple other factors like coverage recording settings, simulation settings, or modeling guidelines have an impact on coverage. They shall be discussed in the following. Again, coverage preservation is the main driver for these decisions, thus some more thoughts on this topic shall be made at first.

Preservation of coverage should be considered as nice to have, not as a requirement. As [182] outlines, formulating preservation criteria between model coverage and code coverage is difficult due the modeling language style (e.g., control flow vs. data flow representation), higher abstractions of model elements, or behavior of code generation, which can perform inlining or optimizations. [183] states that the problems with coverage preservation from model to code are similar to those from source code to object code and names inlining, reuse, and dead code as potential issues. However, there is an undeniable correlation between model coverage and structural coverage as investigated in various assessments [184, 185] and it is reasonable to choose modeling guidelines and document best practices to improve preservation quality.

There are a couple of simulation settings, which drastically change the behavior of coverage and may mitigate the effect and applicability of the measures. In particular, MathWorks lists conditional branch execution, block reduction, and inlined parameters [101, pp. 1-11f]. In MR 3 and MR 35, those settings have already been discussed in other contexts. The therein chosen values comply with the requirements for model coverage preservation.

Another interesting topic is *short-circuiting*, which is a known coverage dilemma for code. In ANSI C99, the boolean expression `D=A||B` contains hidden control flow and is a so-called short-circuit expression. If `A` evaluates to true, `B` is never evaluated. For code coverage collection, a study conducted by the FAA proposes to expand such expressions with the underlying control flow [180, p. 17], i.e., for the given example into Listing 11.

```
if(A)
  D = A;
else
  if(B)
    D = B;
  else
    D = false;
  end
end
```

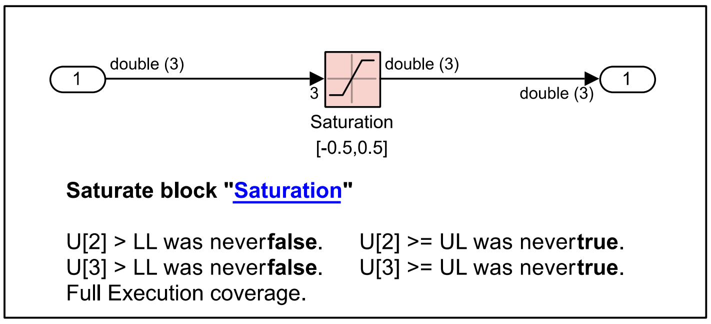
Listing 11: Example for expansion of short-circuiting (pseudo code)

This is thus the strategy, which should be followed in model condition coverage by activating the option “Treat Simulink Logic blocks as short-circuited” (`CovLogicBlockShortCircuit`) for model coverage collection [123, p. 2-23]. Otherwise the coverage would not consider short-circuit expressions and return full coverage, although not the full statement has been evaluated. This typically requires more simulation cases. The setting is only relevant for SL blocks. Logical expressions in MATLAB or SF action language are treated as short-circuit expressions, anyway.

There are also a couple of situations, in which the higher-level abstraction may rise preservation problems. Of course, this problem arises for almost all functions, which generate canonical shared code. However, these functions are separately verified. But besides, also small block settings have an impact.

One example is the implicit saturation on integer overflow. Enumerations for example provide the option of safe casting, which generates addition robustness code (cf. MR 34 and Listing 7). This leads to additional decisions and conditions, which negatively impacts preservation of coverage. Model coverage analysis addresses “Saturate on overflow” in a separate metrics, but according to the documentation only requires a case, in which the saturations comes into effect and a case, in which it doesn’t. The implementations in C often show more complex behavior concerning execution paths, decisions, and conditions. Since full coverage of “Saturate on overflow” metrics seldom maps to full code coverage, this coverage is not assessed here. Moreover, fundamental modeling guidelines have been established limiting the use of block options deteriorating coverage preservation.

In other cases, model coverage tries to cover all situations and is more sensitive than required. A typical example are operations on multi-dimensional signals, like in Figure 102. The model saturates a multi-dimensional signal and model decision coverage is required for each element of the signal.



Saturate block "Saturation"

U[2] > LL was never**false**.   U[2] >= UL was never**true**.
U[3] > LL was never**false**.   U[3] >= UL was never**true**.
Full Execution coverage.

Figure 102: Example for multi-dimensional decision points

This may seem unnecessary, since in C code, the whole operation would be nested in a loop iterating through an array. Code coverage would just require the decisions to be `True` or `False`, once. However, how those operations are actually translated into code is determined by the Embedded Coder configuration. Embedded Coder can either use the loop syntax, or it can repeat the same code pattern multiple times (so-called “loop unrolling”). The translated code variants for Figure 102 are shown in Listing 12. In the latter case, indeed, decision coverage for each element would be required in C code, too. Thus, model coverage just makes a conservative assumption. In consequence, depending on the signal length, achieving full coverage may be impossible and project deviations should be provided for this case.

```
 /* LOOP UNROLLING THRESHOLD <= 3 */
for (i = 0; i < 3; i = i + 1) {
  u0 = xy_multiDimDecision_U.In1[i];
  if (u0 > 0.5) {
    y = 0.5;
  } else if (u0 < (-0.5)) {
    y = (-0.5);
  } else {
    y = u0;
  }

  rtb_Saturation[i] = y;
}
```

```
/* LOOP UNROLLING THRESHOLD > 3*/
u0 = xy_multiDimDecision_U.In1[0];
if (u0 > 0.5) {
  rtb_Saturation_idx_0 = 0.5;
} else if (u0 < (-0.5)) {
  rtb_Saturation_idx_0 = (-0.5);
} else {
  rtb_Saturation_idx_0 = u0;
}

u0_0 = xy_multiDimDecision_U.In1[1];
if (u0_0 > 0.5) {
  rtb_Saturation_idx_1 = 0.5;
} else if (u0_0 < (-0.5)) {
  rtb_Saturation_idx_1 = (-0.5);
} else {
  rtb_Saturation_idx_1 = u0_0;
}
...
```

Listing 12: Example for loop unrolling of vector operations

The difficulties of inlining and reuse of generated code for subsystems have been discussed in section MR 19. REUSABLE MODELS are, in terms of model coverage, the only reasonable approach, since they have clearly defined interfaces and behavior, and the coverage aggregates over multiple instances.

Another challenge, which may arise, is model coverage for invariant operations. In Figure 103, the Relational Operator block is invariant, since `m_var1` and `m_var2` are non-tunable MATLAB variables (cf. MR 35). However, the block receives condition points (which are never achievable, since the values are constant). Pre-calculation of invariant expressions is not part of the block reduction and cannot be disabled. In consequence, these situations can occur and project deviations must be provided to reference for justification. In most cases, however, these implementations are also not compatible with SLCI, anyway, and should be replaced with pre-calculated values.

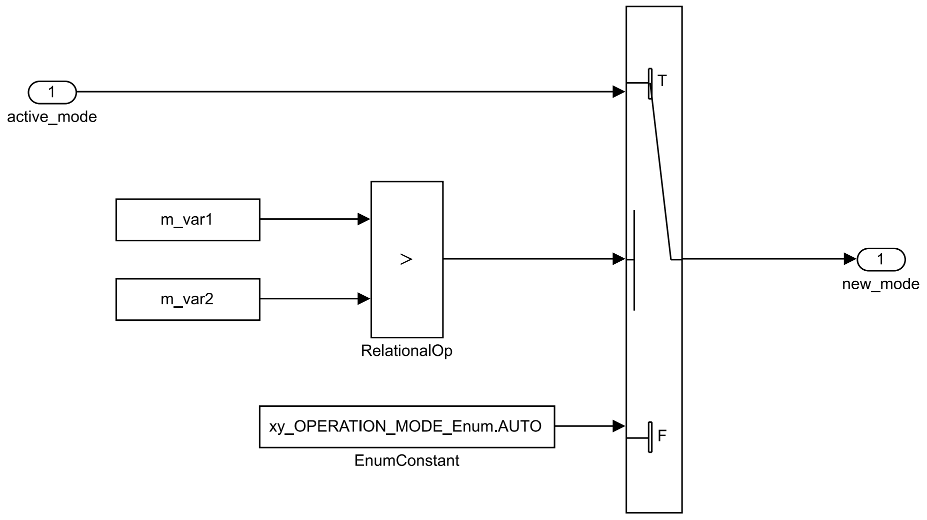


Figure 103: Example of an invariant operation

Result evaluation and justification

The recorded coverage is evaluated differently for D^M and $D2^C$. In D^M, the coverage results for each model of the current module are assessed separately and further processed according to Table 44. The chosen coverage objective is that each SL model shall receive 100% model coverage (execution, decision, and condition) with the simulation cases derived from the requirements allocated to the module, refined in the module, or derived requirements. Referenced SL models from other modules are assumed to have 100% coverage.

Resolution strategies for noncovered design and unreachable design have already been provided in DR 30.

Coverage (Execution, Decision, and Condition)	**Standardized Status**	**Comment**
100%	✓ (PASS)	Note: If a model does not have decisions or conditions, the respective coverage measure is interpreted as 100%. However it must be checked in advance, whether the coverage measurement was activated.
< 100%	! (WARN)	Missing coverage can be justified if not resolvable otherwise (DR 30). Project deviations can be referenced.
Not executed or missing coverage fragment	☒ (FAIL)	Each SL model must at least once be executed in a test, so a coverage fragment must exist.

Table 44: Status mapping for SwVP-DP-MB 9 (D^M)

In $D2^C$, Table 45 applies. To check whether all models are in the call hierarchy, it is sufficient to evaluate, whether execution coverage exists and is greater than 0% for each SL model in all modules. Exceptions are SL models in library modules, since they don't have to be called.

For this kind of coverage, only component simulation test cases are used (cf. Table 12).

Coverage (Execution)	**Standardized Status**	**Comment**
> 0% or in library module	✓ (PASS)	
0%, not executed, or missing coverage fragment	✗ (FAIL)	Uncalled models are a design flaw and require rework.

Table 45: Status mapping for SwVP-DP-MB 9 (D2^C)

Model coverage does not reveal all kinds of unused model elements (cf. MR 40). Thus, essential part of the verification are the static module checks and all efforts made for data coupling and control analysis (cf. SwVP-DP-MB 2 and section 4.6.4).

Limitations

The aggregation process is complex and not supported by the existing tool qualification kit of MathWorks.

Currently, no possibility exists to detect whether deviations inserted through coverage filters are still valid after modifying parts of a SL model, changing the simulation case, or just re-executing the simulation case. Thus, existing annotations are dropped in any of these cases, which requires a lot of additional effort.

Model coverage does not provide any means to evaluate data coupling coverage. The so-called “Signal Range Coverage” goes into the right direction, but does not support Simulink.Bus objects.

It is not very clear, where coverage settings are stored and which have precedence. SL Test has some coverage settings like coverage types, however more detailed settings are not possible. They are taken from the model under test. Thus, it is important that a harness model is used, which links to a compatible test configuration set (cf. SR 10).

The report generation has not been implemented yet and project deviations have been collected informally, only.

8.2.10 SwVP-CP-MB 1 – Automatic code review

This task is executed with Simulink Code Inspector (SLCI). The purpose is, on the one hand, to generate traceability between Design Model and generated Source Code and verify it. On the other hand, the step verifies the compliance of Source Code with the Design Model. It is an independent backward verification of the Embedded Coder code generation process. Code inspection largely, but not completely, replaces code review of the auto-generated code.

The step combines trace data generation of activity SwDP-CP-MB 2 and its verification. Trace data is under control category 1 according to DO-178C Table A-2 in this process, whereas verification results for the respective objective are under control category 2 (DO-178C Table A-5:5). Since the resulting report combines both, it must be treated as control category 1 artifact.

Contribution 33: A task has been defined and resources have been created to automatically execute automated code review with Simulink Code Inspector and assess the returned results.

Tool overview

In general, SLCI generates two intermediate representations in form of abstract syntax trees (ASTs), one from the Design Model and one from the generated Source Code. These ASTs are normalized and compared with each other [89, 186]. If the matching fails, code inspection fails. In case of a fail in one point, SLCI skips analysis of the nested elements. In fact, each intermediate representation consists of multiple independent ASTs reversely built from outputs or test points. These ASTs can be separately verified.

The generation of an AST and the matching procedure just works for a limited set of SL/SF features and just if the Source Code fulfills some requirements. If the AST cannot be built or matched due to incompatibilities, SLCI is designed to fail, i.e., *it is sound*.

To predict the success of the inspection, SLCI provides a set of compatibility checks. These checks test the Design Model and code generation settings for incompatibilities prior to the actual analysis. The results are categorized as FATAL, for which an inspection will not even run, and NON-FATAL, for which SLCI at least tries to map the ASTs. Compatibility checks are implemented in a conservative manner. If a setting may lead to an incompatibility, although just under some specific conditions, the setting is flagged as NON-FATAL. The code inspection might succeed nevertheless. In other words, it is ok to have NON-FATAL incompatibility flags, if the analysis succeeds. Some of these cases are provided in the documentation [134, pp. 1-4ff]. All compatibility checks have been added to the static model analysis (SwVP-DP-MB 1). They are categorized as compatibility checks, which allow a deviation.

Application

Code inspection has to be performed separately for every SL model and its generated code. Code inspection is limited to the scope of one model. SLCI is instructed to perform a top-level analysis for a top-level model.

According to Figure 104, for each SL model, a separate report is generated and stored as trace data under control category 1. The report is reviewed and directly considered as evidence artifact.

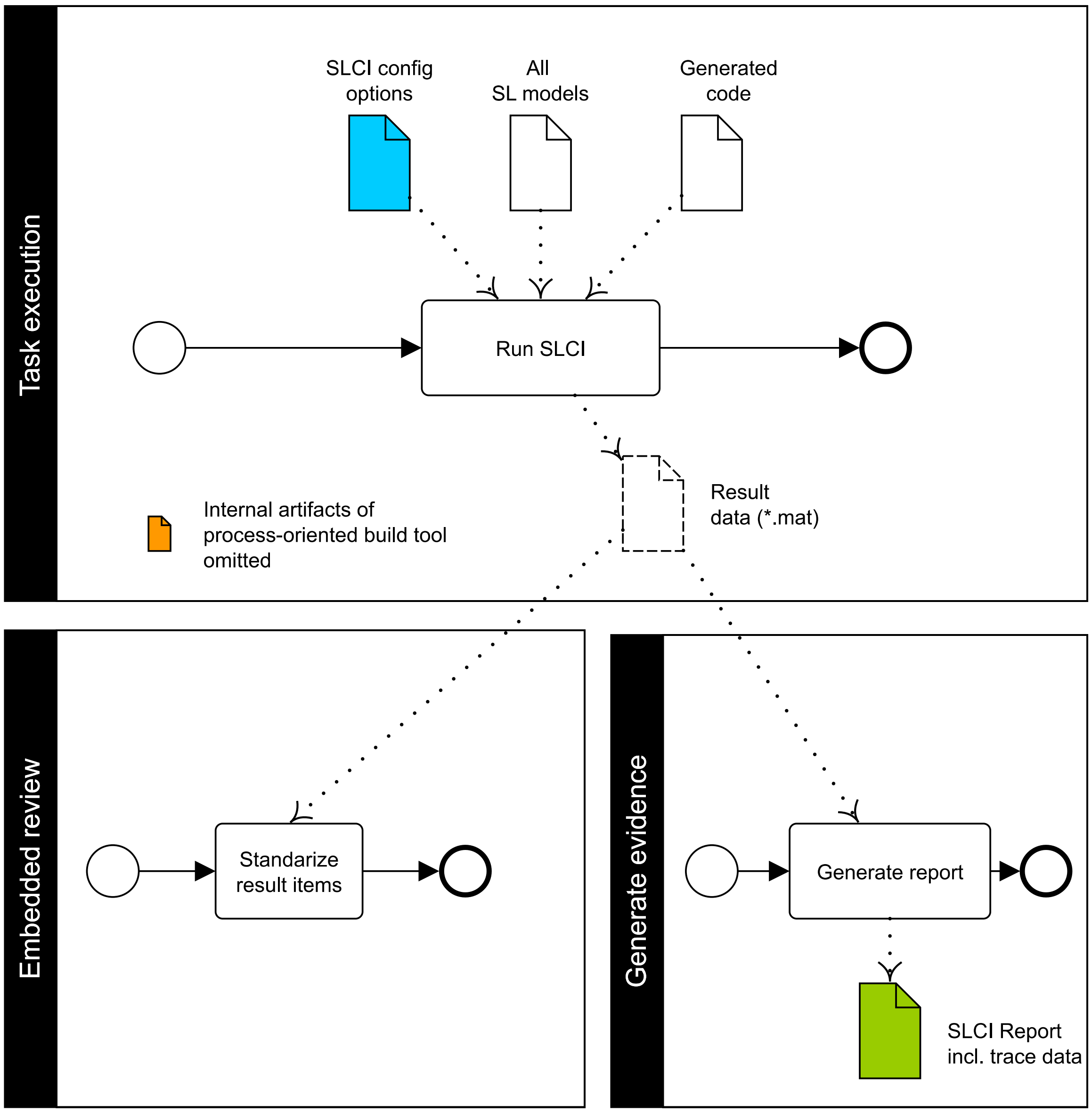


Figure 104: Workflow and artifacts for SwVP-CP-MB 1

Prerequisites

The correct SLCI options had to be chosen. Execution and result standardization have been implemented.

Result evaluation and justification

Output of the task is an inspection report only. SLCI has the ability to generate a traceability matrix, too [134, p. 3-29]. However, it was not considered necessary, since the report contains the trace data as well. Furthermore, the possibility to add justifications into the traceability matrix is not required, since the decision here is to follow a fairly conservative approach.

In this process, only a fully verified overall analysis result, without partially or failed sub results, is acceptable (cf. Table 46). Rationale for this decision was the difficulty to identify the origin for partially traceable/verifiable elements as well as the impact. Partial results may also be quite extensive, e.g., if SLCI cancels the analysis after it finds a failure.

SLCI Status	Standardized Status	Comment
Verified	✓ (PASS)	
Partially traced / verified	☒ (FAIL)	Partially traced / verified results can have different reasons. It is hard to identify the origin. Being able to create deviations would be an error-prone process. Rework of the model is required.
Failed to traced / verify	☒ (FAIL)	This is a true traceability or compatibility issue between Source Code and Design Model. Rework of the model is required.

Table 46: Status mapping for SwVP-CP-MB 1

Limitations

Identifying the origin of issues is cumbersome, although the documentation provides some hints [134, pp. 3-19ff]. The report not clearly identifies the reason for failed or partial verification. Finding the reason resulted in time-consuming work. With the safe subset, the scope could be drastically limited and it is more likely to create a compliant model “out-of-the box”.

Shared code is just partially verified. Canonical shared code is excluded from analysis. This is acceptable, since existing, pre-verified shared code is used (cf. section 8.1.2.2). Also the separately generated module shared code is not or just partially verified by SLCI and is subject to review.

Code inspection can only be executed on SL models, which can generate code. SL libraries cannot be checked. SLCI does not detect SL models and SL library elements, which are not used.

8.2.11 SwVP-CP-MB 2 – Static code analysis for standard compliance

In this task, Polyspace Bug Finder is used to verify the compliance to the code standard. The checks mainly cover MISRA C rules. The tool is considered as static code analyzer, not as formal method tool, since it does not prove that certain violations do not exist. The tool searches for known patterns.

Contribution 34: Resources have been created to configure Polyspace BugFinder, so that it checks the selected coding rules and design errors. The analysis and result evaluation has been fully automated. Different configurations have been developed for module- and component-level analysis in the modular process.

Application

In C^M, a selected set of MISRA C checks is executed on the Source Code of each SL model (public and private) separately without analyzing nested models.

In C^C, all applicable MISRA C checks are executed on the whole Source Code of the application starting from the top-level model.

Figure 105 roughly illustrates the automated processes. At first, analysis options are prepared and a Polyspace project is generated with these options from the SL model (entry point of analysis). Then the analysis is executed. Output of the analysis is a machine readable database of result data.

Review and justification follows the external workflow. The results are loaded into Polyspace, reviewed, and justified (annotations). The result standardization overlays result with annotation data and displays the outcome in the process-oriented build tool. In the final step, a report is generated as evidence artifact.

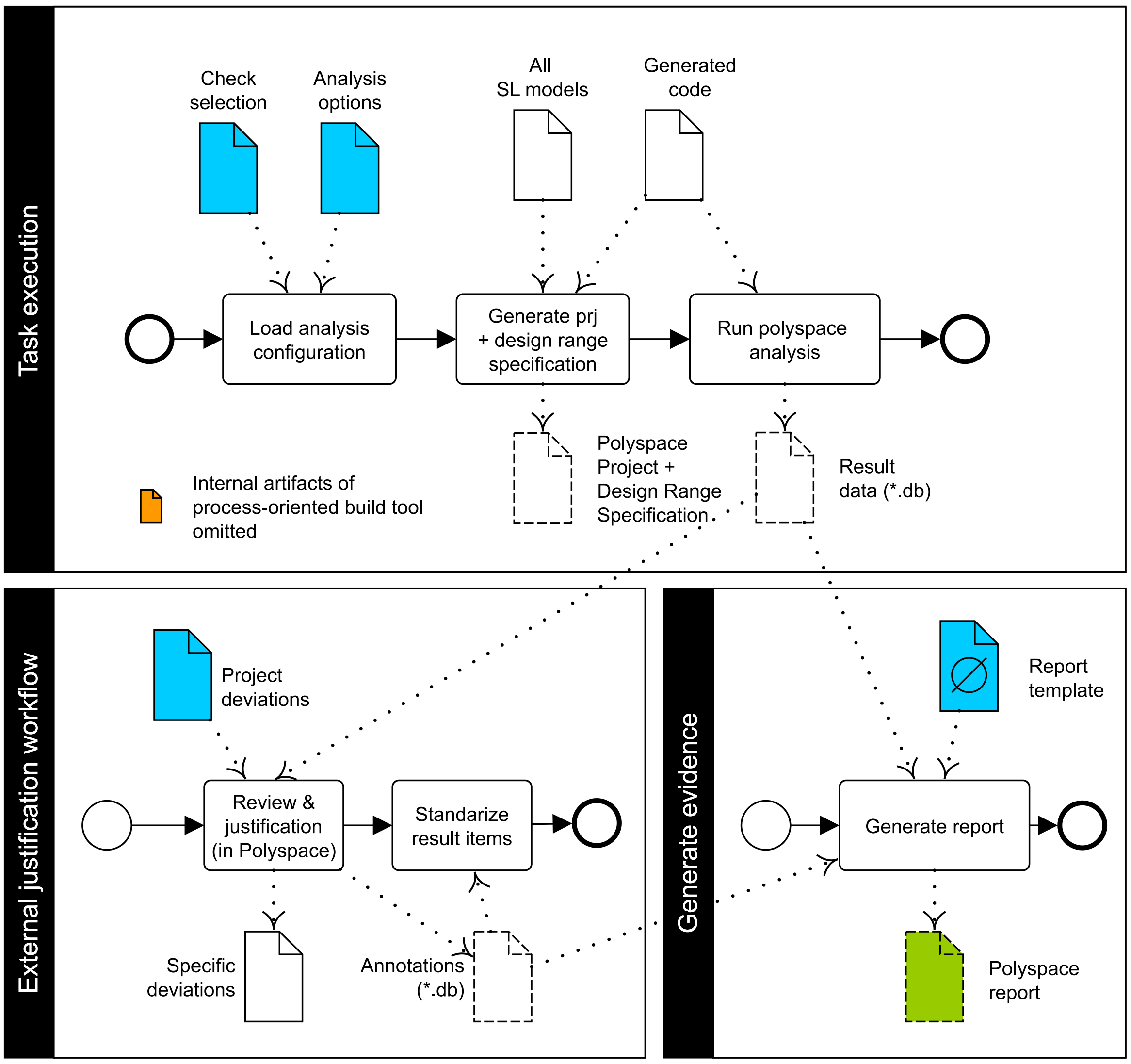


Figure 105: Workflow and artifacts for SwVP-CP-MB 2

Prerequisites

From the large amount of Polyspace settings, analysis options have been selected. Analysis options define, for example, the type of analysis being executed (like standard checking or run-time error detection) or target- and compiler-specific settings (`polyspace.ModelLinkOptions` [187, pp. 4-121ff]). The analysis options are independent of the analyzed Source Code, i.e., they do not define, which files to analyze.

The check selection lists all the checks, which shall be executed during analysis. Bug Finder is shipped with a set of checks for all MISRA C rules and numerous MISRA C directives[43]. However, certain customization and argumentation was necessary:

- Bug Finder provides checks for all MISRA C rules, also for those categorized as *undecidable* by static code analysis tools in MISRA C. It had to be evaluated, whether the introduced restrictions and existing limitations are acceptable.
- Not all MISRA rules and directives are evaluable for the code of a secluded model. Many of them raise false positives, if executed on a part of the whole software application.
- For a few rules, documenting *project deviations* (MISRA C 5.4) cannot be avoided.
- Directives are just partially covered by checks.

MISRA C explicitly supports the use of static code analyzers to verify rules. All rules labelled as *decidable* in MIRSA C are theoretically checkable by a static code analysis tool (MISRA C 6.5). However, the property is very conservative. By interpreting the rules in a stricter manner, Bug Finder also provides checks for, according to MISRA, *undecidable* rules.

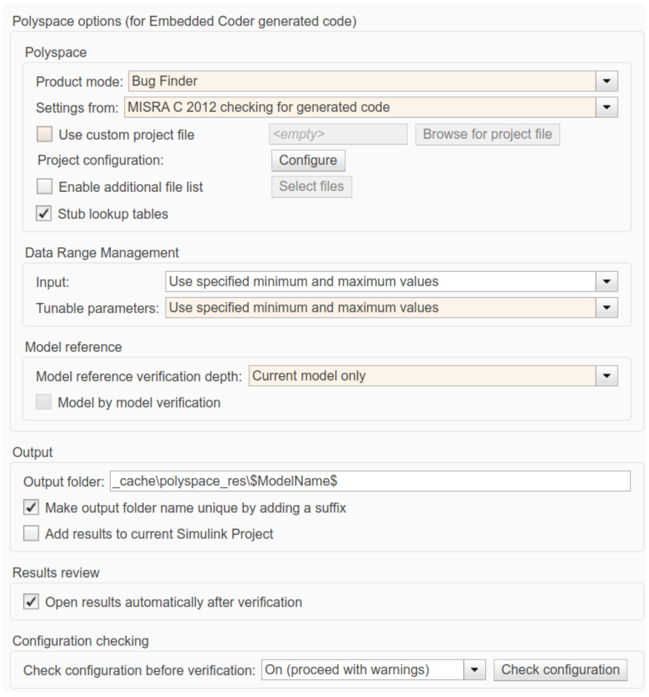
For example, MISRA C Rule 13.1 states that initializer lists shall not contain persistent side effects. A persistent side effect changes the execution state [126, p. 224]. Due to the unlimited complexity of side effect constructs, e.g., by function calls, evaluation by static code analysis might be impossible. The rule is thus categorized as undecidable. By specifying that variables shall be constants in initializers (and no forwarding function calls), Bug Finder makes the rule decidable. If such a constraint applies, a *Polyspace Specification* is given and documented in the tool requirements [188]. Whether an extended Polyspace Specification is acceptable, has been documented with the check selection. If a rule (or directive) cannot be checked sufficiently, other means of verification have been specified.

In order to perform a separate analysis for each model reference code and a holistic analysis from the top-level model, two lists of checks have been assembled. That not all rules are fully assessable for parts of the software is also considered in MISRA C. It distinguishes between rules, which can be verified in the scope of a *single translation unit*, and those, which require an analysis of the whole code (MISRA C 6.6). The checks of the first group have been selected for C^M and C^C analysis. All other checks are only verified as part of code analysis in C^C.

[43] https://de.mathworks.com/help/releases/R2017b/bugfinder/misra-c-2012-reference.html [Accessed on: Jul. 21 2019]

If Embedded Coder is used, Polyspace provides a convenient way to generate a runnable project based on a SL model and the generated code (`pslinkrun` [187, pp. 4-18ff]). This project generation derives additional information, like the location of the analyzable code files, the root level function calls, and a *design range specification* (DRS) automatically. The DRS is a XML file, which contains minimum and maximum ranges specified in the model in a readable format for Polyspace. To automate the project generation, a script has been created.

The project is generated prior to the analysis. The generation itself can be configured with `pslinkoptions` [187, pp. 4-10ff]. The settings are shown in the screenshot of the Polyspace configuration settings in SL (Figure 106). Some of them are discussed in more detail in the following, since they have a significant impact in this process.



Polyspace options (for Embedded Coder generated code)

Polyspace

Product mode: Bug Finder

Settings from: MISRA C 2012 checking for generated code

☐ Use custom project file <empty> Browse for project file

Project configuration: Configure

☐ Enable additional file list Select files

✓ Stub lookup tables

Data Range Management

Input: Use specified minimum and maximum values

Tunable parameters: Use specified minimum and maximum values

Model reference

Model reference verification depth: Current model only

☐ Model by model verification

Output

Output folder: _cache\polyspace_res\$ModelName$

✓ Make output folder name unique by adding a suffix

☐ Add results to current Simulink Project

Results review

✓ Open results automatically after verification

Configuration checking

Check configuration before verification: On (proceed with warnings) Check configuration

Figure 106: Polyspace options for project derivation from auto-generated code of Embedded Coder (in SL model configuration settings)

Especially the model reference verification depth plays an important role. In C^M, the option has been set to “Current model only”. Bug Finder searches for MISRA violations and defects in a limited scope, since most of them are directly decidable (single translation unit) and it does not make extensive abstract interpretation to propagate signal ranges. Model code can thus be analyzed in isolation pretty well. For C^C, the depth has been expanded to the full model hierarchy and “Model by model verification” has been disabled in order to respect coupling effects and verify the rules with a scope beyond a translation unit.

Other relevant settings in Figure 106 relate to “Data Range Management”. The options control, whether and if signal ranges from the model shall be considered in the analysis. In the process at hand, the options `InputRangeMode`, `ParamRangeMode`, and `OutputRangeMode` (Code Prover only) [187, p. 9-12ff] have all been set to respect the specified minimum and maximum, since the signal ranges specified at public interfaces (DR 11) shall be incorporated in the analysis where possible. Especially checking the output range is an important task of Code Prover, which uses the same DRS.

However, the DRS in combination with the chosen setting required some further considerations for parameters. `ParamRangeMode` either defines that the specified ranges or the calibration data (i.e., the default value, which has been set in the SL model) are used. Latter is a fix value without range. This is a global analysis setting. However, for the parameter types defined in MR 36, calibration data setting is required for PARAMETER CONSTANT and the range setting for PARAMETER DATA ITEM specializations. By default, just one or the other is possible.

The found workaround here was to activate the use of signal ranges for all parameters and create a hook in the project generation process, which modifies the exported DRS. During this step, range entries for PARAMETER CONSTANT specializations are removed from the DRS. This was not possible with the public Polyspace API, since `pslinkrun` directly starts the analysis after creating the DRS file. It required support from MathWorks.

Result evaluation and justification

The assessment, interpretation, and justification workflow depends on the criticality of the rule or directive. MISRA C distinguishes between mandatory, required, and advisory. Deviations from rules marked as mandatory are not permitted. Deviations from required rules shall be formally justified and the risks be evaluated. Deviations from advisory rules do not necessarily underlie a formal process, but should be documented. Interpretation of MISRA violations based on their category and available justification options are listed in Table 47.

Polyspace knows different ways to store deviations. Annotations can be added to the results data [189, pp. 1-46ff], to code [189, pp. 1–48ff], or also to the model [189, pp. 8-12ff]. Annotations for blocks are handed over to Polyspace, but are stored in the model. But since they are verification artifacts, they should not taint the model (no way to store annotations externally in R2017b is known to the author). Code annotations are for manual workflows and, although they are not identified as code change by Embedded Coder, overwritten if new code is generated. Here, it has been chosen to store annotations with the results, which creates a separate annotation file. Polyspace automatically imports annotations into new runs [189, pp. 5-41f].

Check Execution Status	Standardized Status	Comment
Mandatory	☒ (fail)	Violation requires rework of model and update of generated code.
Required	! (warn)	To justify, review Polyspace annotation, set status in Polyspace to “Justified” and add comment. Reference to specific deviation.
Advisory / Readability	✓ (pass)	Advisory violations should be reviewed and are automatically documented.

Table 47: Status mapping for SwVP-CP-MB 2

Limitations

The tool qualification does not cover the functionality of `pslinkrun` (i.e., the project generation) [190]. In consequence, the project is considered as input of the task and must be reviewed. This is especially critical for the DRS, which can have a significant impact on the results. In addition, a customized report is not provided yet.

8.2.12 **SwVP-CP-MB 3 –** Static code analysis for error detection

The objective of the task is to check, whether runtime errors can occur in the code (cf. MISRA C Dir 4.1). The analysis is performed with Polyspace Bug Finder, and searches for defects. It is not a formal method.

Application

The workflow for this task is similar as for SwVP-CP-MB 2. The analysis is separately executed in C^M for the code of all SL models and in C^C for the top-level model (with full verification depth).

Prerequisites

This task has in principle the same prerequisites as SwVP-CP-MB 2 and the main infrastructure is reused, except that a selection of activated defect checks [189, pp. 4-2ff] has been created instead of a list of MISRA C checks and the result standardization process slightly differs.

Result evaluation and justification

Review is similar to SwVP-CP-MB 2, however for a detected defect, a deviation cannot be provided as stated in Table 48.

Defect Status	**Standardized Status**	**Comment**
Defect found	☒ (fail)	Any defect requires rework of model and update of generated code.
No defect	☑ (pass)	

Table 48: Status mapping for SwVP-CP-MB 3

8.2.13 **SwVP-CP-MB 4 –** Code review

In this task, coding rules, which cannot be checked by analysis with SLCI or Polyspace, are reviewed. For example, the independence of the algorithm of byte-ordering (CR 9) or design-induced shared code.

Code reviews for the MBSwD have not been further addressed in this thesis, since they do not significantly differ from code reviews performed in the industry in traditional software development processes.

However, the process-oriented build tool supports auto-generation of interactive review lists for each code file as well.

8.2.14 **SwVP-CP-MB 5 –** Code proving

This task has two objectives and operates on C code:

- Assurance-guarantee analysis
- Run-time error analysis

This task has not been implemented yet in the scope of this work. Anyway, some considerations have been collected.

The task is similar to SwVP-DP-MB 5, just for Source Code. Polyspace Code Prover performs, in contrast to Polyspace Bug Finder, a sound analysis and proves objectives using abstract interpretation. Over-approximated design ranges will again lead to various undecidable objectives.

As in SwVP-DP-MB 5, dead code detection can identify some dead code, but not prove the absence of all dead code. Since reachability is covered by testing anyway, it is not further regarded in this task.

Since code proving is quite low compared to Polyspace Bug Finder analysis and abstract interpretation may end up in largely over-approximated derived signal ranges, the tool should be executed on module-level.

The implementation aligns with SwVP-CP-MB 2, since the same Polyspace infrastructure for configuration, project generation, execution, and result justification is used.

However, workarounds for tool incompatibilities in R2017b must be found. For example, the tool cannot prove intermediate outputs of SL models. A possible workaround is exporting the data to global variables, for which assertions to prove can be provided to Polyspace. This has already been prepared with MR 25.

Another challenge is limiting the analysis depth. A similar approach as for Design Verifier seems feasible, where stubs have been auto-generated.

8.2.15 **SwVP-CP-MB 6** – SIL testing & result review

Purpose of this activity is to execute the simulation cases on the Source Code in SIL in order to record structural coverage. In the SIL mode, Source Code is compiled for the host computer and executed on the host.

Contribution 35: Resources have been implemented that allow the automated execution and result assessment of simulation/test cases in software-in-the-loop mode and perform equivalence comparison with previously recorded model simulation results.

Application

Prior to testing, the Source Code of the current and nested modules is generated (if not yet there) and copied to temporary testing location. This temporary testing location is set as new code repository during PIL testing.

All simulation cases of the module are executed in SIL mode. Embedded Coder automatically adds the missing interface and instrumentation code and compiles it to an executable, which is called in SIL testing.

The results are checked for equivalence with the simulation results (cf. SwVP-DP-MB 8) in the subsequent process. Verifying the equivalence of the SIL results with the model simulation results is an optional step (cf. Table 12). However, it improves confidence before hardware testing without additional costs.

The results are reviewed, but cannot be justified. If a test case fails, investigation and rework is required. Finally, a report of the SIL test results is generated per simulation case.

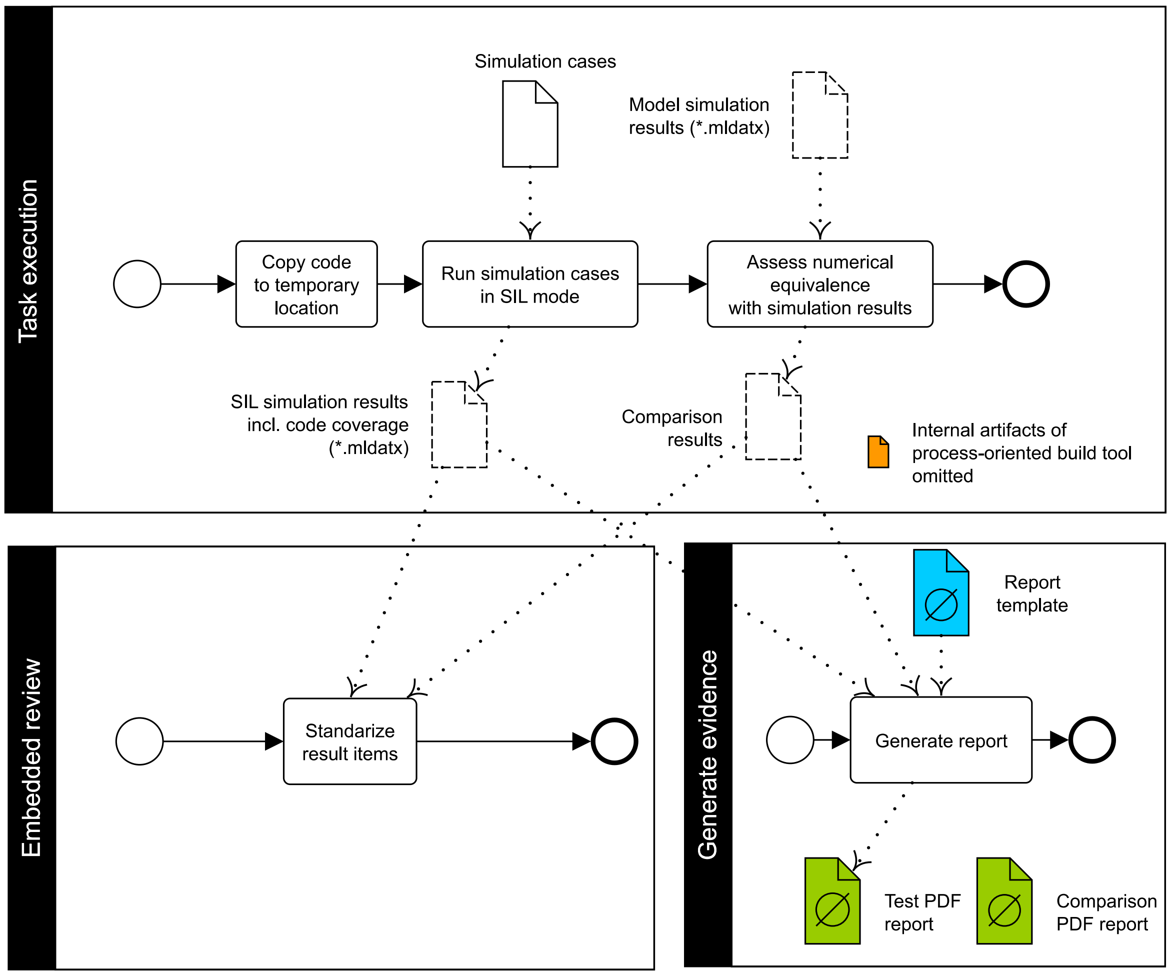

Task execution
Simulation cases
Model simulation results (*.mldatx)
Copy code to temporary location
Run simulation cases in SIL mode
Assess numerical equivalence with simulation results
SIL simulation results incl. code coverage (*.mldatx)
Comparison results
Internal artifacts of process-oriented build tool omitted
Embedded review
Standarize result items
Generate evidence
Report template
Generate report
Test PDF report
Comparison PDF report

Figure 107: Workflow and artifacts for SwVP-CP-MB 6

Prerequisites

Although quite similar to SwVP-DP-MB 8, a couple of problems had to be solved. At first, the default compiler had to be changed, since the code has some compiler specific language constructs (cf. MR 32). The CompCert compiler cannot be used, since it is a cross-compiler for the target. Since the GNU preprocessor, assembler and linker are part of the target tool chain (cf. AS 11), MinGW[44], a GNU compiler for Windows, is the obvious choice.

[44] MinGW is a GNU compiler for Windows, http://www.mingw.org/ [Accessed on: Aug. 20 2019]

In SIL mode, Embedded Coder generates additional code for coverage instrumentation and data exchange. This code does not change the actual Source Code, but is distributed all over the code generation folder. Since it is just temporary, it should not be added to source control. The only option to keep these additional artifacts apart, was copying the Source Code to a new location, switching the SL code generation path, and deleting the folder after SIL execution. To prohibit any modification of the code while adding instrumentation, the documentation proposes to set the configuration parameter `UpdateModelReferenceTargets` to `AssumeUpToDate`, which never allows a rebuilt [96, pp. 64-23ff, 128, pp. 15-5ff]. However, since this setting always threw an error in R2017b and its correct execution is hard to prove, additional checks had to be implemented. After SIL, the code files in the temporary folder and those in the original code folder are checked for equivalence.

There were also workflow problems related to switching from Normal to SIL mode, since the simulation mode cannot be passed to SL Test from outside, but is stored in the test case. On top, the options provided in SL Test are not sufficient to fully control SIL execution. Further details and a recommended solution are given in SR 10.

Using Code Replacement Libraries was no problem. They were correctly picked from the modeling environment, separately compiled, and linked into the final host executable object code.

In order to make code executable on both the host and target environment, the Embedded Coder option `PortableWordSizes` [131, pp. 12-15f] had to be activated during Source Code generation and the code must be independent of the byte-ordering (cf. CR 9). Especially the last is a requirement, which can significantly impact the modeling and cannot be solved with configuration settings.

The equivalence assessment compares simulation with SIL results. This does not mean the pass/fail status, but signal recordings with a value for each time step. Minimum signal recording requirements are provided in SR 8 and SR 9. The comparison has been implemented with the function `Simulink.sdi.compareRuns` [127, pp. 2-701ff], since the results for normal simulation and SIL are produced in separate runs. SL Test provides an equivalence test, however this test scenario just compares numerical equivalence between two runs, and not pass fail criteria. The user would have to duplicate simulation case (baseline + equivalence test type in SL Test) to achieve the same result.

Result evaluation and justification

Table 49 summarizes, how the SIL testing result is handled.

Review Item Status	**Standardized Status**	**Comment**
Test result status “Pass”	☑ (PASS)	
Test result status “Fail”	☒ (FAIL)	Either the pass/fail criteria falsified or the simulation has not been executed properly. Rework of either the Design Model or the simulation case is required.
Equivalence/numerical mismatch	☒ (FAIL)	Any mismatch produces are failure and must be investigated.
Source Code changed	☒ (FAIL)	If any Source code file has been modified, the SIL results are invalid.

Table 49: Status mapping for SwVP-CP-MB 6

Limitations

The different workflow problems required workarounds. Especially the additional requirements on how to structure test cases in SR 10 are not optimal and lead to unnecessary work.

8.2.16 **SwVP-CP-MB 7** – SIL structural coverage assessment

In this task, the structural code coverage, which has been recorded during SIL testing, is assessed.

Contribution 36: Criteria for code coverage assessment have been constituted. Resources have been created to aggregate code coverage per module and across modules. Such an automated coverage aggregation and assessment is not natively supported by MathWorks, but inevitable in a modular process.

Application

The application is, in principle, similar to the workflow described in SwVP-DP-MB 9, since SL Coverage is used as well. The code coverage fragments of the SIL test cases are accumulated and extracted per code file (instead of per SL model). The coverage is assessed per code file.

Shared canonical code, for which coverage is automatically recorded, is excluded from the analysis.

Coverage types

DO-178C requires statement and decision coverage for DAL B software. As in SwVP-DP-MB 9 for model coverage, the combination of decision and condition coverage matches DO-178C decision coverage. DO-178C statement coverage is exposed under execution coverage. For code coverage, this is also stated in the documentation [101, p. 4-3]. Execution, decision, and condition coverage is evaluated in the C^M process.

In C^C, to asses control coupling coverage (cf. section 4.6.4), function coverage is evaluated [101, p. 4-6]. Function coverage indicates, whether all functions of a code file are called at least once. Therefore, only the coverage generated by test cases for component interface are relevant.

Prerequisites

The prerequisites for model and code coverage are mainly identical (cf. SwVP-DP-MB 9).

Result evaluation and justification

The recorded coverage is evaluated differently for C^M and C^C.

In C^M, the coverage results for each code file of the current module are evaluated and further processed according to Table 50. The different types of unreachable code are described in CR 14, CR 15, and CR 16. Resolution strategies for code coverage gaps are state-of-the-art and not further discussed here (for example, cf. [45, p. 387]).

Coverage (Execution, Decision, and Condition)	**Standardized Status**	**Comment**
100%	☑ (PASS)	Note: If a code file does not have decisions or conditions, the respective coverage measure is interpreted as 100%. However it must be checked in advance, whether the coverage measurement was activated.
< 100%	! (WARN)	Missing coverage shall be justified. Project deviations can be referenced. Deviations shall be made using SL Coverage “Justifications” and be stored in a coverage filter file. Each deviation requires a rationale.
Not executed or missing coverage fragment	☒ (FAIL)	Each code file must contain a function at least once be executed in a test, so a coverage fragment must exist.

Table 50: Status mapping for SwVP-CP-MB 7 (C^M)

In C^C, Table 51 applies. To check whether all models are in the call hierarchy, a function coverage for every code file of 100% is required. This must be evaluated for all SL models in all modules, which are not in a library module.

Coverage (Function, Function call)	**Standardized Status**	**Comment**
100% or in library module	☑ (PASS)	
< 100% and not in library	☒ (FAIL)	Uncalled functions are a design flaw and require rework.

Table 51: Status mapping for SwVP-CP-MB 7 (C^C)

Limitations

Equal to SwVP-DP-MB 9.

The algorithm for assessing call coverage has not been implemented in the scope of this work.

8.3 Summary and outlook

Modular development process (part 1)

In section 4.4, the foundation for a modular software life cycle has been laid. The important distinction of component modules and processes has been introduced in section 4.4, which became the central concept throughout the thesis. All relevant objectives of DO-331 have been broken down into dedicated tasks and assigned to the new process categories.

As central point, section 4.6.3 specified a consistent testing strategy involving new concepts of DO-331, like model simulation, model coverage, or SIL testing. This testing strategy has been optimized for a maximum of test case reuse and early identification of testing issues, like missing coverage with dead logic detection, model coverage and optimized preservation of coverage. The testing strategy also adopted the paradigm of modularization by separating component- and module-level tests, coverage goals or coupling verification.

The MBSwD in this thesis just roughly describes the directly connected Requirement and Integration Processes. This pretends more independence than really exists. Upstream workflows, in which the Design Model replaces high-level or even system requirements, are tightly coupled. This coupling has not been documented in detail. But also downstream tasks are not independent. For example, hardware/software integration testing, Worst-Case-Execution-Time, or Stack analysis need to run simulation cases as well and come along with compatibility requirements, which have not been respected in the given process.

And finally, concepts like independence or the standards for tool qualification and for formal methods (DO-331) have not been considered in detail, although, for example, formal methods are applied. Those topics definitely need further investigation and the additional objectives have to be integrated.

The presented MBSwD is new and unique in the way it is defined and goes beyond the existing industry standard by applying innovative workflows combined with modularization.

Modular development process (part 2)

In part 2, the new approach for generation of modular code has been realized. Section 8.1.2 outlined, how the code is generated and how the different types of shared code are treated.

The presented solution supports generation of modular code, although it became clear that real modular code generation is just partially possible with the available features of Embedded Coder. Shared code is completely modularly generated, all other code is regenerated in each module. A post-processing workflow was necessary to ensure equality of regenerated code with the original code. Being able to skip code regeneration with Embedded Coder would significantly reduce code generation times. Especially for shared code, solutions concerning creation, inclusion into code generation, and the treatment in a modular development had to be found.

For each verification activity, the discussion outlined the chosen application workflow as well as involved artifacts. The standardized status mapping has been consistently applied. It reduces the different outputs of all tasks to a common denominator. The status is directly intuitive for any developer and has the same consequence or leads to the same actions throughout the process.

For many tasks, extensive automation enhancements and workarounds have been programmed. For example, for design error detection, model coverage aggregation and assessment, Polyspace project configuration, or SIL testing.

Finally, for almost each task, an implementation as part of the process-oriented build tool could be provided. The outcome is a workflow configuration for the build tool, which spans the whole process and provides the maximum possible automation for every task. Even the review and justification workflows are tightly coupled or even fully integrated into the build tool.

Nevertheless, future improvements are possible and necessary. At first, there are several unaddressed topics. Code proving has not been implemented yet and the report generation has been skipped for several tasks.

Contracting of value ranges makes software development easier and leads to more efficient code. But contracting is only reasonable, if contracts can be verified with formal methods. However, here the respective verification tools are not that far yet. On the one hand, the tools are primarily designed for running them from a root-level entry point. Verifying intermediate ranges is not supported in the way it should be. But on the other hand, formal methods deliver highly over-approximated results for large models/code, so that a breakdown is inevitable. A couple of scenarios with Design Verifier could be solved by implementing pre-processing steps of the model.

Migration to newer tool releases certainly solves existing problems and makes various workarounds obsolete. For example, later versions of Design Verifier support signal ranges in buses or the exclusion of objectives. Significant improvements can also be expected concerning SIL and SIL coverage, which both were fairly new features in R2017b.

With respect to certification, also a gap analysis for tool qualification may be a next step. Several functions are used that are not part of the DO Qualification Kit of MathWorks in R2017b. Some of them have been listed as limitations, however these remarks are certainly not complete.

Also in detail, improvements are possible. The static model and module checks do still not cover all rules, which leads to more review effort. And the possibility to dynamically create review items based on the reviewed content should finally be used.

To sum up, with the defined tasks, a majority of the process can be covered. The implementation as build tool configuration offers high automation and the user-interface of the build tool turns into a core platform to assess the health and compliance status and navigate to all kinds of source and derived artifacts from a central platform.

9 Conclusions

Sections 5.9, 6.6, 7.6, and 8.3 already provided a summary and outlook for the individual topics. This section evaluates the overall achievements of the work with respect to the originally formulated objectives stated in section 1.2:

1. Lower the adoption risk of a MBSwD for smaller companies by providing a MBSwD process and tooling, which is out-of-the box applicable, bridges tool gaps, and is consistent.

2. Improve development efficiency, ensure scalability, and support for agile development by tool usage and the introduction of a high degree of software modularization and reuse whilst adhering to process requirements and safety.

3. Provide a solution for exhaustive process automation and keeping a “ready-to-certify” state of software artifacts.

During the implementation of the model-based process, the challenges, which beginners face, became visible. There is a huge gap between the general objectives in the process standards and the final implementation guidelines and tool usage for development and verification. Getting lost is a common danger. The thesis addresses this gap in an organized way and covers the breath of the process. A holistic picture is revealed starting with objectives mapped to tasks in section 4, the definition of rules in section 5, up to detailed procedures in section 8. No publication is known to the author offering a similar scope.

In the timeframe of the thesis, it was not possible to address tool qualification, formal methods, or the other integral processes required by DO-178C to the depth they deserve. Therefore, additional work is required and some adaptions may be necessary.

The dependencies between standards, tool chain capabilities, and design, model, or coding rules are complex and only resolvable in an iterative process. Although processes differ by nature, the given rules and tasks significantly shorten the iterative effort in any adoption process and help to avoid common pitfalls. The best practice to componentize models with SL models instead of SL libraries is one example. The solutions have been discussed with various experts and contain collected best practices from many projects. Assumptions and context have been worked out and clearly organized, so that the considerations made for the software life cycle can be easily adapted to any process. Furthermore, the created artifacts (Appendix H) are a practical entry-point, in contrast to the generic and broadly applicable tool qualification kits.

One major tool gap, which has been identified, was related to traceability. Since no satisfying solution existed to establish and maintain the needed inter-tool traceability between Simulink and Polarion, *SimPol* has been developed. After publishing the tool, different companies reached out for feature requests or bug fixes. The interest shows that *SimPol* satisfies an existing demand.

Another objective was to optimize the process for efficiency and scalability. Several introduced tasks go beyond the state-of-the-art in safety-critical workflows, like the different scenarios for model usage, SIL and PIL approaches, or the application of formal methods. This thesis embeds them into a full safety-critical process. For example, SIL in combination with modularization allows to evaluate code coverage in an early stage.

Nothing more than consequent modularization paves the way for parallel development, reuse of certification evidence, and the application of agile methods or DevOps. Thus, modular development and verification from design to code has been introduced. The process at hand shows that a large number of verification tasks, like static model analysis, testing, formal verification, or coverage can be performed independently to a certain degree. Component-level tasks have been set up to finally close the gap, which arises, if verification is performed on module-level. With generation of modular code, all benefits of modularization have been leveraged for code development and verification tasks as well, although true modular code generation, which avoids regeneration of code, was not realizable.

The prize for modularization are more rules, e.g., for interface specifications, and a more sophisticated planning of verification. The thesis showed that the optimization towards modularization also reaches and sometimes crosses the limit of what is possible with today’s tools. Various workarounds were necessary to achieve the goal, which would finally need additional tool qualification in a certification project.

The objective of lowering the adoption risks certainly conflicts with the aim of innovating the process. There are many innovative parts, like the SIL workflow, contracting, or deactivated functionality that differ from common practice by intention. Companies new to MBSwD should discuss these new approaches with authorities in advance and in detail.

Finally, the thesis contributed an innovative solution for the “big-freeze” problem. The software tool *mrails* brings together manually established traceability and information, which is inherent of the build process. Although the tool is a prototype and not simply certifiable under DO-330, it proves the feasibility of a central software platform serving as framework for process automation and staleness detection of derived artifacts. The considerations made for review workflows, which are a fundamental part of safety-critical processes, also distinguish it from existing solutions. Full automation and containing the impact of a change, also throughout reviews, is a first step towards continuous software delivery in a safety-critical context.

The thesis has presented novel concepts for modularization and automation, which deliver a foundation to cope with growing software on the one hand and the necessary certification rigor in safety critical applications on the other hand.

10 References

[1] SAE International, “ARP-4754A -Guidelines for Development of Civil Aircraft and Systems: Aerospace Recommended Practice (ARP),” ARP-4754A, Dec. 2010.

[2] RTCA, “DO-178C - Software Considerations in Airborne Systems and Equipment Certification,” DO-178C, 2011.

[3] RTCA, “DO-248C - Supporting Information for DO-178C and DO-278A,” DO-248C, 2011.

[4] RTCA, “DO-330 - Software Tool Qualification Considerations,” DO-330, 2011.

[5] RTCA, “DO-331 - Model-Based Development and Verification Supplement to DO-178C and DO-278A,” DO-331, 2011.

[6] RTCA, “DO-333 Formal Methods Supplement to DO-178C and DO-278A,” DO-333, 2011.

[7] ISO, “ISO 26262-6:2018(en) - Road vehicles - Functional safety - Part 6: Product development at the software level: 43.040.10,” ISO 26262-6:2018(en), Dec. 2018.

[8] J. D. Kennedy and M. Towhidnejad, “Innovation and certification in aviation software,” in *Proc. of Integrated Communications, Navigation and Surveillance Conference (ICNS)*, Herndon, VA, USA, Apr. 2017 - Apr. 2017, 3D3-1-3D3-15.

[9] Project manager (Thomas Heimann), “Studie IT-Trends 2019: Intelligente Technologien,” Capgemini, 2019. Accessed: Sep. 7 2019. [Online]. Available: https://www.capgemini.com/de-de/wp-content/uploads/sites/5/2019/02/IT-Trends-Studie-2019.pdf

[10] I. Sommerville, *Software engineering,* 9th ed. Boston: Pearson Education Limited, 2011.

[11] J. Humble and D. Farley, *Continuous delivery: Reliable software releases through build, test, and deployment automation*. Upper Saddle River, NJ: Addison-Wesley, 2015.

[12] J. Marsden *et al.,* “ED-12C/DO-178C vs. Agile Manifesto: A Solution to Agile Development of Certifiable Avionics Systems,” in *Proc. of Embedded Real Time Software and Systems (ERTSS 2018)*, Toulouse, France, 2018.

[13] D. J. Coe and J. H. Kulick, “A Model-Based Agile Process for DO-178C Certification,” in *Proc. of 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing*, Las Vegas, US, 2013.

[14] P. Rempel, P. Mäder, T. Kuschke, and J. Cleland-Huang, “Mind the gap: Assessing the conformance of software traceability to relevant guidelines,” in *Proc. of the 36th International Conference on Software Engineering - ICSE 2014*, Hyderabad, India, 2014, pp. 943–954.

[15] P. Lars, “DA42 MNG FBW Research Aircraft,” in *In-flight simulators and fly-by-wire/light demonstrators: A historical account of international aeronautical research*, P. G. Hamel and R. V. Jategaonkar, Eds., Cham, Switzerland: Springer, 2017, pp. 146–148.

[16] C. Krause and F. Holzapfel, “Designing a System Automation for a novel UAV Demonstrator (submitted for publication),” in *Proc. of 14th International Conference on Control, Automation, Robotics and Vision: ICARCV 2016*, Phuket, Thailand, 2016.

[17] C. Krause and F. Holzapfel, “Implementing a multi-level finite state machine with MATLAB Simulink and Stateflow in the environment of high-integrity aircraft controller software,” in *Proc. of International Conference on Control, Automation and Robotics 2018 (ICCAR)*, Auckland, 2018, pp. 147–151.

[18] S. P. Schatz and F. Holzapfel, “Modular trajectory / path following controller using nonlinear error dynamics,” in *Proc. of Aerospace Electronics and Remote Sensing Technology (ICARES), 2014 IEEE International*, Yogyakarta, Indonesia, 2014, pp. 157–163.

[19] S. Schatz and F. Holzapfel, “Nonlinear Modular 3D Trajectory Control of a General Aviation Aircraft,” in *Advances in Aerospace Guidance, Navigation and Control: Selected Papers of the Fourth CEAS Specialist Conference on Guidance, Navigation and Control Held in Warsaw, Poland, April 2017*, B. Dołęga, R. Głębocki, D. Kordos, and M. Żugaj, Eds.: Springer Cham, 2017, pp. 163–183.

[20] S. Schatz, “Development and flight-testing of a trajectory controller employing full nonlinear kinematics,” Dissertation, Technische Universität München, München, 2019.

[21] Simon Schatz, “Development and Flight-Testing of a Trajectory Controller Employing Full Nonlinear Kinematics,” Dissertation, Lehrstuhl für Flugsystemdynamik, Technische Universität München, München, 2018.

[22] E. Karlsson *et al.,* “Automatic Flight Path Control of an Experimental DA42 General Aviation Aircraft,” in *Proc. of 14th International Conference on Control, Automation, Robotics and Vision: ICARCV 2016*, Phuket, Thailand, 2016.

[23] E. Karlsson, A. Gabrys, S. P. Schatz, and F. Holzapfel, “Dynamic Flight Path Control Coupling for Energy and Maneuvering Integrity: (submitted for publication),” in *Proc. of 14th International Conference on Control, Automation, Robotics and Vision: ICARCV 2016*, Phuket, Thailand, 2016.

[24] E. Karlsson *et al.,* “Development of an Automatic Flight Path Controller for a DA42 General Aviation Aircraft,” in *Advances in Aerospace Guidance, Navigation and Control: Selected Papers of the Fourth CEAS Specialist Conference on Guidance, Navigation and Control Held in Warsaw, Poland, April 2017*, B. Dołęga, R. Głębocki, D. Kordos, and M. Żugaj, Eds.: Springer Cham, 2017, pp. 121–139.

[25] A. W. Zollitsch *et al.,* “Automatic takeoff of a general aviation research aircraft,” in *Proc. of 2017 Asian Control Conference Gold Coast, Australia: Gold Coast Convention and Exhibition Centre, 17th-20th December 2017*, 2017.

[26] V. Schneider *et al.,* “Online trajectory generation using clothoid segments,” in *Proc. of 14th International Conference on Control, Automation, Robotics and Vision: ICARCV 2016*, Phuket, Thailand, 2016.

[27] V. Schneider, N. Mumm, and F. Holzapfel, “Trajectory generation for an integrated mission management system,” in *Proc. of Aerospace Electronics and Remote Sensing Technology (ICARES), 2014 IEEE International*, Kuta, Bali, 2015.

[28] V. Schneider and F. Holzapfel, “Modular Trajectory Generation Test Platform for Real Flight Systems,” in *Advances in Aerospace Guidance, Navigation and Control: Selected Papers of the Fourth CEAS Specialist Conference on Guidance, Navigation and Control Held in Warsaw, Poland, April 2017*, B. Dołęga, R. Głębocki, D. Kordos, and M. Żugaj, Eds.: Springer Cham, 2017.

[29] S. P. Schatz *et al.,* “Flightplan Flight Tests of an Experimental DA42 Generation Aviation Aircraft,” in *Proc. of 14th International Conference on Control, Automation, Robotics and Vision: ICARCV 2016*, Phuket, Thailand, 2016.

[30] V. Schneider, “Trajectory generation for integrated flight guidance,” Dissertation, Technische Universität München, München, 2018.

[31] M. Hochstrasser, C. Krause, V. Schneider, and F. Holzapfel, “Model-based Implementation of an Onboard STANAG 4586 Vehicle Specific Module for an Air Vehicle,” in *Proc. of AIAA Modeling and Simulation Technologies Conference*, Grapewine, Texas, US, 2017.

[32] The MathWorks Inc., “DO Qualification R2017b: Model-Based Design Workflow for DO-178C,” Natick, MA, USA, Sep. 2017.

[33] M. Hochstrasser, S. Myschik, and F. Holzapfel, “Application of a Process-Oriented Build Tool for Flight Controller Development Along a DO-178C/DO-331 Process,” in *Model-Driven Engineering and Software Development: 6th International Conference, MODELSWARD 2018, Funchal, Madeira, Portugal, January 22–24, 2018, Revised Selected Papers*, S. Hammoudi, L. F. Pires, and B. Selic, Eds.: SPRINGER NATURE, 2019, pp. 380–405.

[34] M. Hochstrasser, S. Myschik, and F. Holzapfel, “A modular model-based DO-178C software life cycle - Planning, realization, and preservation,” München, Oct. 10 2018. Accessed: Dec. 25 2018. [Online]. Available: https://www.dglr.de/fileadmin/inhalte/dglr/fb/q3/veranstaltungen/Q34_2018_Agile_SW/DGLR_Q34_2018_Hochstrasser_TUM_Agile.pdf

[35] M. Hochstrasser, M. Hornauer, and F. Holzapfel, “Formal Verification of Flight Control Applications along a Model-Based Development Process: A Case Study,” München, Oct. 5 2016. Accessed: Nov. 9 2016. [Online]. Available: http://www.dglr.de/fileadmin/inhalte/dglr/fb/q3/veranstaltungen/L63_Q34_2016_Software_Safety/2016_DGLR_Workshop_TUM_samoconsult.pdf

[36] M. Hochstrasser, S. P. Schatz, K. Nürnberger, M. Hornauer, S. Myschik, and F. Holzapfel, “Aspects of a Consistent Modeling Environment for DO-331 Design Model Development of Flight Control Algorithms,” in *Advances in Aerospace Guidance, Navigation and Control: Selected Papers of the Fourth CEAS Specialist Conference on Guidance, Navigation and Control Held in Warsaw, Poland, April 2017*, B. Dołęga, R. Głębocki, D. Kordos, and M. Żugaj, Eds.: Springer Cham, 2017.

[37] K. Nürnberger, M. Hochstrasser, and F. Holzapfel, “Execution time analysis and optimisation techniques in the model-based development of a flight control software,” *IET Cyber-Physical Systems: Theory & Applications*, Volume 2, Issue 2, pp. 57–64, Jul. 2017, doi: 10.1049/iet-cps.2016.0046.

[38] K. Schmiechen, M. Hochstrasser, J. Rhein, C. Schropp, and F. Holzapfel, “Traceable and Model-Based Requirements Derivation, Simulation, and Validation Using MATLAB Simulink and Polarion Requirements,” in *AIAA Scitech 2019 Forum*, p. 334.

[39] M. Hochstrasser, S. Myschik, and F. Holzapfel, “A Process-Oriented Build Tool for Safety-Critical Model-Based Software Development,” in *Proc. of the 6th International Conference on Model-Driven Engineering and Software Development*, Funchal, Madeira, Portugal, 2018, pp. 191–202.

[40] European Aviation Safety Agency EASA, “AMC 20-115D - Airborne Software Development Assurance Using EUROCAE ED-12 and RTCA DO-178: Acceptable Means of Compliance,” AMC 20-115D, Oct. 2017. [Online]. Available: https://www.easa.europa.eu/document-library/certification-specifications/amc-20-amendment-14

[41] Federal Aviation Administration FAA, “AC 20-115D - Airborne Software Development Assurance Using EUROCAE ED-12 ( ) and RTCA DO-178( ),” AC 20-115D, Jul. 2017.

[42] European Aviation Safety Agency EASA, “CM-SWCEH-002 - Software Aspects of Certification,” Mar. 2012.

[43] European Aviation Safety Agency EASA, “CM-SWAEH-002 - Software Aspects of Certification: Notification of a Proposal to Issue a Certification Memorandum,” CM-SWAEH-002, Oct. 2013.

[44] RTCA, “DO-332 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A,” DO-332, 2011.

[45] L. Rierson, *Developing safety-critical software: A practical guide for aviation software and DO-178C compliance*. Boca Raton, FL: CRC Press LLC, 2013.

[46] J. Cleland-Huang, O. C. Z. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, “Software traceability: Trends and future directions,” in *Proc. of the Conference on Future of Software Engineering - FOSE 2014*, Hyderabad, India, 2014, pp. 55–69.

[47] Esterel Technologies SA, “Efficient Development of Safe Avionics Software with DO-178C Objectives Using SCADE Suite: Methodology Handbook,” Elancourt, France, Jun. 2015.

[48] The MathWorks Inc., “DO Qualification Kit - Polyspace Code Prover Tool Qualification Plan: R2017b.,” Natick, MA, USA, Sep. 2017.

[49] The MathWorks Inc., “DO Qualification Kit - Simulink Test Tool Qualification Plan: R2017b,” Natick, MA, USA, Sep. 2017.

[50] M. Hornauer, F. Schuck, and F. Holzapfel, “Wechselwirkungen zwischen GNC Algorithmus und Software,” München, 2013.

[51] RTCA, “DO-160G - Environmental Conditions and Test Procedures for Airborne Equipment,” DO-160G, 2010.

[52] M. Hornauer and F. Holzapfel, “Model Based Testing for CS-23 Avionic and UAV Applications: DGLR Workshop 2011,” München, 2011.

[53] RTCA, “DO-200B - Standards for Processing Aeronautical Data,” DO-200B, Jun. 2015.

[54] Federal Aviation Administration FAA, “AC 20-153B - Acceptance of Aeronautical Data Processes and Associated Databases: Advisory Circular,” AC 20-153B, Apr. 2016. Accessed: Mar. 27 2017. [Online]. Available: https://www.faa.gov/documentLibrary/media/Advisory_Circular/AC_20-153B.pdf

[55] J. R. Levine, *Linkers and loaders*. San Francisco, Calif. et al.: Morgan Kaufmann, 2010.

[56] R. M. Stallman and Z. Weinberg, “The C Preprocessor: For gcc version 4.2.4,” 2005.

[57] AbsInt Angewandte Informatik GmbH, Ed., “CompCert User Documentation,” Jul. 2016.

[58] R. M. Stallman, “Using the GNU Compiler Collection: For gcc version 4.2.4,” GNU Press, Boston, MA, USA, 2005.

[59] J. Schumann and E. Denney, “Customer Survey on Code Generators in Safety-critical Applications,” Mar. 2006.

[60] K. Goseva-Popstojanova, T. Kahsai, M. Knudson, T. Kyanko, N. Nkwocha, and J. Schumann, “Survey on Model-Based Software Engineering and Auto-Generated Code,” NASA/TM-2016-219443, Oct. 2016.

[61] G. Reith, *Adoption of Model-Driven Engineering in Small Workgroups and in Large Organisations.* [Online]. Available: https://nmi.org.uk/wp-content/uploads/2015/06/MathWorks-Adoption-of-Model-Driven-Engineering.pdf (accessed: Oct. 27 2017).

[62] M. Broy, S. Kirstan, H. Krcmar, and B. Schätz, “What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry?,” in *Software Design and Development*, I. R. Management Association, Ed.: IGI Global, 2014, pp. 310–334.

[63] D. Bhatt, G. Madl, D. Oglesby, and K. Schloegel, “Towards Scalable Verification of Commercial Avionics Software,” in *Proc. of AIAA Infotech@Aerospace 2010*, 2010.

[64] M. Reke, “Modellbasierte Entwicklung automobiler Steuerungssysteme in kleinen und mittelständischen Unternehmen,” Department of Computer Science, RWTH Aachen, Aachen, 2013.

[65] T. Erkinnen and B. Potter, *Model-Based Design for DO-178B with Qualified Tools: AIAA Modeling and Simulation Technologies Conference and Exhibit*. Hyatt Regency McCormick Place, Chicago Illinois: American Institute of Aeronautics and Astronautics Inc, 2009.

[66] B. Potter, *Complying with DO-178C and DO-331 using Model-Based Design*, *SAE Paper*.

[67] M. Conrad, J. Friedman, and G. Sandmann, “Verification and Validation According to IEC 61508: A Workflow to Facilitate the Development of High-Integrity Applications,” *SAE Int. J. Commer. Veh.*, vol. 2, no. 2, pp. 272–279, 2009, doi: 10.4271/2009-01-2929.
[68] M. Conrad, “Verification and Validation according to ISO 26262: A Workflow to Facilitate the Development of High-Integrity Software,” in *Proc. of Embedded Real Time Software and Systems (ERTSS) 2012*, 2012.
[69] A. Paz and G. El Boussaidi, “On the Exploration of Model-Based Support for DO-178C-Compliant Avionics Software Development and Certification,” in *Proc. of 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Ottawa, ON, Canada, 2016, pp. 229–236.
[70] J. C. Marques, S. M. H. Yelisetty, L. A. V. Dias, and A. M. da Cunha, “Using Model-Based Development as Software Low-Level Requirements to Achieve Airborne Software Certification,” in *Proc. of 2012 Ninth International Conference on Information Technology - New Generations*, Las Vegas, NV, USA, 2012, pp. 431–436.
[71] W. Wu, *Model-based design for effective control system development*. Hershey, PA: Information Science Reference, 2017.
[72] S. Basagiannis, “Software certification of airborne cyber-physical systems under DO-178C,” in *Proc. of 2016 International Workshop on Symbolic and Numerical Methods for Reachability Analysis (SNR)*, Vienna, Austria, 2016, pp. 1–6.
[73] E. Dillaber, L. Kendrick, W. Jin, and V. Reddy, “Pragmatic Strategies for Adopting Model-Based Design for Embedded Applications,” 2010.
[74] P. F. Smith, S. M. Prabhu, and J. H. Friedmann, “Best Practices for Establishing a Model-Based Design Culture,” *SAE Paper*, 2007-01-0777, 2007.
[75] R. Aarenstrup, *Managing Model-Based Design*. Natick, MA, USA: The MathWorks Inc., 2015.
[76] Bill Potter, *DO178_case_study (on MathWorks File Exchange): Case Study for DO-178 using MathWorks tools.* [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/56056-do178-case-study (accessed: May 18 2016).
[77] J. Abraham, “Verification and Validation Spanning Models to Code,” in *Proc. of AIAA Modeling and Simulation Technologies Conference 2015*, Kissimmee, Florida, 2015.
[78] M. A. Beine, “A Model-Based Reference Workflow for the Development of Safety-Critical Software,” in *Proc. of SAE Convergence Conference 2010*, Detroit, Michigan, USA, 2010.
[79] U. Eisemann, “Applying Model-Based Techniques for Aerospace Projects in Accordance with DO-178C, DO-331 and DO-333,” in *Proc. of Embedded Real Time Software and Systems (ERTSS) 2016*, Toulouse, 2016.
[80] S. Mahapatra, J. Ghidella, and G. Walker, “Team-Based Collaboration in Model-Based Design,” in *Proc. of AIAA Modeling and Simulation Technologies Conference*, Minneapolis, Minnesota, 2012.
[81] K. Grand, V. Reddy, G. Sasaki, and E. Dillaber, “Large-Scale Modeling for Embedded Applications,” *SAE Paper*, 2010-01-0938, 2010.
[82] K. A. Saleh, *Software engineering*: J. Ross Publishing, 2009.
[83] ARINC, “Avionics Application Standard Software Interface, Part 0, Overview of ARINC 653,” 653, Aug. 2015.
[84] G. K. Hanssen, G. Wedzinga, and M. Stuip, “An Assessment of Avionics Software Development Practice: Justifications for an Agile Development Process,” in *Lecture Notes in Business Information Processing, Agile Processes in Software Engineering and Extreme Programming*, H. Baumeister, H. Lichter, and M. Riebisch, Eds., Cham: Springer International Publishing, 2017, pp. 217–231.

[85] J. Marques and A. M. da Cunha, “Tailoring Traditional Software Life Cycles to Ensure Compliance of RTCA DO-178C and DO-331 with Model-Driven Design,” in *Proc. of 2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, London, Sep. 2018 - Sep. 2018, pp. 1–8.

[86] American Institute of Aeronautics and Astronautics, Inc., Ed., *The Liebherr Fully Integrated FCS Design – a Case Study*. Los Angeles, CA: American Institute of Aeronautics and Astronautics, Inc., 2013.

[87] Position Paper CAST-15 - Certification Authorities Software Team (CAST), “Merging High-Level and Low-Level Requirements: Position Paper,” CAST-15, Feb. 2003.

[88] A. J. Kornecki and J. Zalewski, “The Qualification of Software Development Tools From the DO-178B Certification Perspective,” *CrossTalk - The Journal of Defense Software Engineering*, Apr. 2006.

[89] M. Conrad *et al.,* “Automating Code Reviews with Simulink Code Inspector,” in *Proc. of VIII Dagstuhl-Workshop: Model-Based Development of Embedded Systems (MBEES)*, Dagstuhl, Germany, 2012.

[90] The MathWorks Inc., “DO Qualification Kit - Simulink Code Inspector Tool Qualification Plan: R2017b,” Natick, MA, USA, Sep. 2017.

[91] The MathWorks Inc., *Simulink Code Inspector and Polyspace Qualified under DO-330: Qualified Code Generation with MathWorks Embedded Coder*. Natick, MA, USA, 2015. Accessed: May 3 2016. [Online]. Available: http://de.mathworks.com/company/newsroom/simulink-code-inspector-and-polyspace-qualified-under-do-330.html

[92] Certification Authorities Software Team (CAST), “Position Paper CAST-17 - Structural Coverage of Object Code: Position Paper,” CAST-17, Jun. 2003.

[93] M. Bordin, C. Comar, T. Gingold, J. Guitton, O. Hainque, and T. Quinot, “Object and Source Coverage for Critical Applications with the COUVERTURE Open Analysis Framework,” in *Proc. of Embedded Real Time Software and Systems (ERTSS 2010)*, 2010.

[94] The MathWorks Inc., “DO Qualification Kit - Simulink Coverage Tool Qualification Plan: R2017b,” Natick, MA, USA, Sep. 2017.

[95] W. Wong, “George Romanski of Verocel Explains DO-178C Certification for Airborne Equipment,” Electronic Design, Jul. 2014. Accessed: Jan. 5 2018. [Online]. Available: http://www.electronicdesign.com/embedded/george-romanski-verocel-explains-do-178c-certification-airborne-equipment

[96] The MathWorks Inc., “Embedded Coder User's Guide: R2017b,” Natick, MA, USA, Sep. 2017. Accessed: Jul. 13 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/ecoder/ecoder_ug.pdf

[97] Lauterbach, “Integration Trace32 - Simulink,” 2011. Accessed: Jan. 6 2018. [Online]. Available: http://www.lauterbach.com/publications/integration_trace32_simulink_d.pdf

[98] T. Maia and M. Souza, “A Practical Methodology for DO-178C Data and Control Coupling Objective Compliance,” in *Proc. of the International Conference on Software Engineering Research and Practice (SERP)*, The Steering Committee of The World Congress in Computer Science, Ed., 2018, pp. 236–240. Accessed: 1st Dec. 2019. [Online]. Available: https://csce.ucmss.com/cr/books/2018/LFS/CSREA2018/SER4278.pdf

[99] J. J. Chilenski and J. L. Kurtz, “Object-Oriented Technology Verification Phase 2 Handbook: Data Coupling and Control Coupling,” DOT/FAA/AR-07/19, Aug. 2007.

[100] Certification Authorities Software Team (CAST), “Position Paper CAST-19 - Clarification of Structural Coverage Analyses of Data Coupling and Control Coupling: Position Paper,” CAST-19, Jan. 2004.

[101] The MathWorks Inc., “Simulink Coverage User's Guide: R2017b,” Natick, MA, USA, Sep. 2017. Accessed: Jul. 13 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/slcoverage/slcoverage_ug.pdf

[102] S. Bhattacharya, “Introduction to Data Coupling and Control Coupling (LDRA Technology, Inc.),” Orlando, Florida, US, Tutorials at the 34th International System Safety Conference 2016, Aug. 2016. Accessed: 1st Dec. 2019. [Online]. Available: https://system-safety.org/issc2016/T11_Data_Coupling.pdf

[103] The MathWorks Inc., “Modeling Guidelines for High-Integrity Systems: R2017b,” Natick, MA, USA, Sep. 2017.

[104] The MathWorks Inc., “Guidelines and factors to consider for code generation: R2017b,” Natick, MA, USA, Sep. 2017.

[105] The Motor Industry Software Reliability Association, “MISRA AC GMG - Generic modelling design and style guidelines,” Nuneaton, UK, May. 2009.

[106] The Motor Industry Software Reliability Association, “MISRA AC SLSF - Modelling design and style guidelines for the application of Simulink and Stateflow,” Nuneaton, UK, May. 2009.

[107] The Motor Industry Software Reliability Association, “MISRA AC TL -Modelling style guidelines for the application of TargetLink in the context of automatic code generation,” 2007.

[108] The MathWorks Automotive Advisory Board (MAAB), “MathWorks Automotive Advisory Board Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow: R2017b,” Natick, MA, USA, Sep. 2017.

[109] Japan MBD Automotive Advisory Board (JMAAB), Ed., “Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow,” Jun. 2015.

[110] J. Henry, “Orion GN&C MATLAB/Simulink Standards: FltDyn-CEV-08-148,” CEV Flight Dynamics Team, Oct. 2011.

[111] Ford Motor Company, “Structured analysis and design using Matlab/Simulink/Stateflow modeling style guidelines.,” 1999.

[112] A. Ferrari and A. Fantechi, “Modeling Guidelines for Code Generation in the Railway Signaling Context,” in *Proc. of the First NASA Formal Methods Symposium*, Florence, Italy, 2009.

[113] G. Walde and R. Luckner, “Bridging the tool gap for model-based design from flight control function design in Simulink to software design in SCADE,” in *Proc. of 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, Sacramento, CA, USA, 2016, pp. 1–10.

[114] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi, “Defining and Translating a "Safe" Subset of Simulink/Stateflow into Lustre,” in *Proc. of EMSOFT 2004*, Pisa, Italy, 2004.

[115] G. Hamon and J. Rushby, “An Operational Semantics for Stateflow,” in *Lecture notes in computer science, Fundamental Approaches to Software Engineering*, G. Goos, J. Hartmanis, J. van Leeuwen, M. Wermelinger, and T. Margaria-Steffen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 229–243.

[116] D. L. Lempia and S. P. Miller, “Requirements Engineering Management Handbook,” Federal Aviation Administration FAA, U.S. Department of Transportation, Springfield, Virginia, US DOT/FAA/AR-08/32, Jun. 2009. Accessed: Jul. 14 2019. [Online]. Available: https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/ar-08-32.pdf

[117] Bureau International des Poids et Mesures, “The International System of Units (SI),” 2006.

[118] D. Monniaux, *The pitfalls of verifying floating-point computations*: HAL archives ouvertes, 2008.
[119] D. Goldberg, “What Every Computer Scientist Should Know About Floating-Point Arithmetic,” in Vol 23, No 1, *Computer Surveys*, ACM, Ed., 1991.
[120] K. Nürnberger, *Development of Elementary Mathematics Functions in an Avionics Context*. München, 2019.
[121] The MathWorks Inc., *Simulink Code Inspector Reference: R2017b*. Natick, MA, USA, 2017.
[122] The MathWorks Inc., “Simulink Design Verifier Reference: R2017b,” Natick, MA, USA, Sep. 2017.
[123] The MathWorks Inc., “Simulink Design Verifier User's Guide: R2017b,” Natick, MA, USA, Sep. 2017.
[124] The MathWorks Inc., “Simulink User's Guide: R2017b,” Natick, MA, USA, Sep. 2017. Accessed: Jul. 13 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/simulink/sl_using.pdf
[125] Freescale Semiconductor Inc., Ed., “e300 Power Architecture ™ Core Family - Reference Manual,” e300CORERM Rev. 4, Dec. 2007.
[126] The Motor Industry Software Reliability Association, “MISRA-C:2012 - Guidelines for the use of C language in critical systems,” Nuneaton, UK, Mar. 2013.
[127] The MathWorks Inc., “Simulink Reference,” Natick, MA, USA, Sep. 2017. Accessed: Jul. 13 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/simulink/sl_using.pdf
[128] The MathWorks Inc., “Simulink Graphical User Interface: R2017b,” Natick, MA, USA, Sep. 2017. Accessed: Jul. 13 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/simulink/slgui.pdf
[129] P. A. Darnell and P. E. Margolis, *C, a software engineering approach,* 3rd ed. New York, London: Springer-Verlag, 1996.
[130] The MathWorks Inc., “Simulink Coder Reference: R2017b,” Natick, MA, USA, Sep. 2017. Accessed: Jul. 13 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/rtw/rtw_ref.pdf
[131] The MathWorks Inc., “Embedded Coder Reference: R2017b,” Natick, MA, USA, Sep. 2017. Accessed: Jul. 13 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/ecoder/ecoder_ref.pdf
[132] The MathWorks Inc., “Simulink Test User's Guide: R2017b.,” Natick, MA, USA, Sep. 2017. Accessed: Nov. 29 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/sltest/sltest_ug.pdf
[133] J. Moore and J. Lee, “11 Best Practices for Developing ISO 26262 Applications with Simulink: White Paper,” 2019. Accessed: May 9 2020. [Online]. Available: https://de.mathworks.com/campaigns/offers/iso-26262-functional-safety-best-practices.html
[134] The MathWorks Inc., “Simulink Code Inspector User's Guide: R2017b,” Natick, MA, USA, Sep. 2017.
[135] The MathWorks Inc., “Simulink Coder User's Guide: R2017b,” Natick, MA, USA, Sep. 2017. Accessed: Jul. 13 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/rtw/rtw_ug.pdf
[136] M. Olszewska, Y. Dajsuren, H. Altinger, A. Serebrenik, M. Waldén, and M. G. J. van den Brand, “Tailoring complexity metrics for simulink models,” in *Proccedings of the 10th European Conference on Software Architecture Workshops - ECSAW '16*, Copenhagen, Denmark, 2016, pp. 1–7.

[137] Y. Dajsuren, “On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems,” Dissertation, Technische Universiteit Eindhoven, Eindhoven, 2015.

[138] I. Stürmer, H. Pohlheim, and T. Rogier, “Berechnung und Visualisierung der Modellkomplexität bei der modellbasierten Entwicklung,” in *Proc. of Automotive - Safety & Security 2010*, 2010, pp. 69–82.

[139] M. Olszewska, “Simulink-Specific Design Quality Metrics: TUCS Technical Report No 1002,” Abo Akademi University, Department of Computer Science, Turku, Finnland, Mar. 2011.

[140] J. Scheible, “Automatisierte Qualitätsbewertung am Beispiel von MATLAB Simulink-Modellen in der Automobil-Domäne,” Dissertation, Faculty of Science, Eberhard Karls Universität Tübingen, Tübingen, 2012. Accessed: Sep. 4 2016. [Online]. Available: https://publikationen.uni-tuebingen.de/xmlui/handle/10900/49708

[141] L. Mäurer, T. Hebecker, T. Stolte, M. Lipaczewski, U. Möhrstädt, and F. Ortmeier, “On Bringing Object-Oriented Software Metrics into the Model-Based World – Verifying ISO 26262 Compliance in Simulink,” in *Proc. of System Analysis and Modeling Conference (SAM) 2014: Models and Reusability*, Valencia, Spain, 2014, pp. 207–222.

[142] G. Walker, J. Friedman, and R. Aberg, “Configuration Management of the Model-Based Design Process,” *SAE Paper*, 2007-01-1775, 2007.

[143] O. Gotel *et al.,* “Traceability Fundamentals,” in *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel, and A. Zisman, Eds., London: Springer London, 2012, pp. 3–22.

[144] P. Mäder, P. L. Jones, Y. Zhang, and J. Cleland-Huang, “Strategic Traceability for Safety-Critical Projects,” *IEEE Softw.*, vol. 30, no. 3, pp. 58–66, 2013, doi: 10.1109/MS.2013.60.

[145] P. Rempel, P. Mader, and T. Kuschke, “An empirical study on project-specific traceability strategies,” in *Proc. of 2013 21st IEEE International Requirements Engineering Conference (RE)*, Rio de Janeiro-RJ, Brazil, 2013, pp. 195–204.

[146] M. Salome, M. Staron, and J.-P. Steghöfer, “Challenges of Establishing Traceability in the Automotive Domain,” in *Proc. of International Conference on Software Quality. Complexity and Challenges of Software Engineering in Emerging Technologies*, Wien, Austria, 2017, pp. 153–172. [Online]. Available: https://books.google.de/books?id=tC3XDQAAQBAJ

[147] M. Eyl, C. Reichmann, and K. Müller-Glaser, “Traceability in a Fine Grained Software Configuration Management System,” in *Proc. of International Conference on Software Quality. Complexity and Challenges of Software Engineering in Emerging Technologies*, Wien, Austria, 2017, pp. 15–29.

[148] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, “Software traceability with topic modeling,” in *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, Cape Town, South Africa, 2010, p. 95.

[149] D. Cuddeback, A. Dekhtyar, and J. Hayes, “Automated Requirements Traceability: The Study of Human Analysts,” in *Proc. of 2010 18th IEEE International Requirements Engineering Conference*, Sydney, Australia, 2010, pp. 231–240.

[150] J. H. Hayes, A. Dekhtyar, and S. Sundaram, “Text mining for software engineering,” *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, p. 1, 2005, doi: 10.1145/1082983.1083153.

[151] R. M. Parizi, A. Kasem, and A. Abdullah, “Towards gamification in software traceability: Between test and code artifacts,” in *Proc. of the 10th International Joint Conference on Software Technologies : Colmar, Alsace, France, 20 - 22 July, 2015*, 2015.

[152] P. Rempel and P. Mäder, “A quality model for the systematic assessment of requirements traceability,” in *Proc. of 2015 IEEE 23rd International Requirements Engineering Conference (RE)*, Ottawa, ON, Canada, 2015, pp. 176–185.

[153] The MathWorks Inc., “Simulink Requirements Reference: R2017b,” Natick, MA, USA, Sep. 2017. Accessed: Jul. 13 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/slrequirements/slrequirements_ref.pdf

[154] OMG Object Management Group, “Requirements Interchange Format (ReqIF),” formal/2016-07-01, Jul. 2016. Accessed: Feb. 17 2019. [Online]. Available: https://www.omg.org/spec/ReqIF/1.2

[155] The MathWorks Inc., “Simulink Requirements User's Guide: R2017b,” Natick, MA, USA, Sep. 2017. Accessed: Jul. 13 2019. [Online]. Available: https://de.mathworks.com/help/pdf_doc/slrequirements/slrequirements_ug.pdf

[156] S. Chacon and B. Straub, *Pro Git,* 2nd ed. Berkeley, CA: Apress, 2014.

[157] National Research Council (U.S.); National Academies Press (U.S.), *Critical code: Software producibility for defense*. Washington, D.C: National Academies Press, 2010. [Online]. Available: http://lib.myilibrary.com/detail.asp?id=291712

[158] B. Aldrich *et al.,* “Managing Verification Activities Using SVM,” in *Lecture notes in computer science, Formal Methods and Software Engineering*, D. Hutchison et al., Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 61–75.

[159] R. Kumar and B. H. Krogh, “Heterogeneous verification of embedded control systems,” in *Proc. of 2006 American Control Conference*, Minneapolis, MN, USA, Jun. 2006 - Jun. 2006, 6 pp.

[160] R. Kumar, B. H. Krogh, and P. Feiler, “An Ontology-Based Approach to Heterogeneous Verification of Embedded Control Systems,” in *Lecture notes in computer science, Hybrid Systems: Computation and Control*, D. Hutchison et al., Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 370–385.

[161] J. S. Becker, V. Bertram, T. Bienmüller, U. Brockmeyer, T. Peikenkamp, and T. Teige, “Interoperable Toolchain for Requirements-Driven Model-Based Development,” in *Proc. of Embedded Real Time Software and Systems (ERTSS 2018)*, Toulouse, France, 2018.

[162] A. Rahman, A. Partho, D. Meder, and L. Williams, “Which Factors Influence Practitioners' Usage of Build Automation Tools?,” in *Proc. of 2017 IEEE/ACM 3rd International Workshop on Rapid Continuous Software Engineering (RCoSE)*, Buenos Aires, Argentina, 2017, pp. 20–26.

[163] O. Gotel *et al.,* “The quest for Ubiquity: A roadmap for software and systems traceability research,” in *2012 20th IEEE International Requirements Engineering Conference (RE): Proceedings : September 24-28, 2012 : Chicago, Illinois, USA*, Chicago, IL, USA, 2012, pp. 71–80.

[164] A. Seibel, S. Neumann, and H. Giese, “Dynamic hierarchical mega models: Comprehensive traceability and its efficient maintenance,” *Softw Syst Model*, vol. 9, no. 4, pp. 493–528, 2010, doi: 10.1007/s10270-009-0146-z.

[165] The MathWorks Inc., “MATLAB External Interfaces: R2017b,” Natick, MA, USA, Sep. 2017.

[166] The MathWorks Inc., “Developing S-Functions: R2017b,” Natick, MA, USA, Sep. 2017. Accessed: Aug. 4 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/simulink/sfunctions.pdf

[167] The MathWorks Inc., “Simulink Check User's Guide: R2017b,” Natick, MA, USA, Sep. 2017.

[168] D. Jaffry, *Best Practices for Implementing Modeling Guidelines in Simulink.* [Online]. Available: http://de.mathworks.com/company/newsletters/articles/best-practices-for-implementing-modeling-guidelines-in-simulink.html (accessed: Feb. 3 2016).

[169] The MathWorks Inc., "Simulink Check Reference: R2017b," Natick, MA, USA, Sep. 2017. Accessed: Aug. 7 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/slcheck/slcheck_ref.pdf

[170] The MathWorks Inc., "DO Qualification Kit - Simulink Check Tool Operational Requirements: R2017b," Sep. 2017.

[171] The MathWorks Inc., "DO Qualificaton Kit - Simulink Design Verifier Tool Qualification Plan: R2017b," Natick, MA, USA, Sep. 2017.

[172] The MathWorks Inc., "DO Qualification Kit - Polyspace Code Prover Tool Requirements: R2017b," Natick, MA, USA, Sep. 2017.

[173] The MathWorks Inc., "DO Qualification Kit - Polyspace Code Prover Theoretical Foundation: R2017b.," Natick, MA, USA, Sep. 2017.

[174] J. F. Monin and M. G. Hinchey, *Understanding formal methods*. London, New York: Springer, 2003.

[175] The MathWorks Inc., "Polyspace Code Prover Reference: R2017b," Natick, MA, USA, Sep. 2017.

[176] The MathWorks Inc., "DO Qualification Kit - Simulink Design Verifier Tool Operational Requirements: R2017b," Natick, MA, USA, Sep. 2017.

[177] The MathWorks Inc., "Simulink Test Reference: R2017b," Natick, MA, USA, Sep. 2017. Accessed: Nov. 29 2019. [Online]. Available: https://de.mathworks.com/help/releases/R2017b/pdf_doc/sltest/sltest_ref.pdf

[178] Certification Authorities Software Team (CAST), "Position Paper CAST-10 - What is a "Decision" in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)?: Position Paper," CAST-10, Jun. 2002.

[179] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A Practical Tutorial on Modified Condition / Decision Coverage," NASA Langley Research Center, Hampton, Virginia NASA / TM-2001-210876, May. 2001.

[180] Federal Aviation Administration FAA, *Software Verification Tools Assessment Study: Final Report*. Springfield, Virginia, US: National Technical Information Service (NTIS), 2007.

[181] V. P. Kozyrev and M. A. Saburov, "Satisfying DO-178C Structural Coverage Objectives," *Programming and Computer Software*, vol. 44, no. 1, pp. 43–50, 2018, doi: 10.1134/S0361768818010048.

[182] R. Kirner, "Towards Preserving Model Coverage and Code Coverage," in vol. 2009, *EURASIP Journal on Embedded Systems*: Hindawi Publishing Corporation, 2009.

[183] W. Aldrich, "Using Model Coverage Analysis to Improve the Control Development Process," in *Proc. of AIAA Modeling and Simulation Technologies Conference and Exhibit*, Monterey, California, 2002.

[184] A. Baresel, M. Conrad, S. Sadeghipour, and J. Wegener, "Interplay between Model Coverage and Code Coverage," in *Proc. of EUROCAST Computer Aided Systems Theory*, Las Palmas, Gran Canaria, Spain, 2003.

[185] M. Conrad and S. Sadeghipour, "Einsatz von Überdeckungskriterien auf Modellebene–Erfahrungsbericht und experimentelle Ergebnisse," *Softwaretechnik Trends*, 2002.

[186] The MathWorks Inc., "DO Qualification Kit - Simulink Code Inspector Tool Operational Requirements: R2017b," Natick, MA, USA, Sep. 2017.

[187] The MathWorks Inc., “Polyspace Bug Finder Reference: R2017b,” Natick, MA, USA, Sep. 2017.

[188] The MathWorks Inc., “DO Qualification Kit - Polyspace Bug Finder Tool Requirements: R2017b,” Natick, MA, USA, Sep. 2017.

[189] The MathWorks Inc., “Polyspace Bug Finder User's Guide: R2017b,” Natick, MA, USA, Sep. 2017.

[190] The MathWorks Inc., “DO Qualification Kit - Polyspace Bug Finder Tool Operational Requirements: R2017b,” Natick, MA, USA, Sep. 2017.

[191] L. Hatton, *Safer C: Developing software in high-integrity and safety-critical systems*: McGraw-Hill, 1994.

[192] ISO/IEC, “Programming Languages C - 2nd Edition,” ISO/IEC 9899:1999, Dec. 1999.

[193] The Motor Industry Software Reliability Association, “MISRA C:2012 Amendment 1 - Additional security guidelines for MISRA C:2012,” Nuneaton, UK, Apr. 2016.

[194] The MathWorks Inc., *Embedded Coder MISRA C:2012 Compliance Considerations*. Natick, MA, USA.

[195] ISO/IEC, “Information technology - Language independent arithmetic - Part 1: Integer and floating point arithmetic,” ISO/IEC 10967-1:2012(E), Jul. 2012.

[196] Certification Authorities Software Team (CAST), “Position Paper CAST-21 - Compiler-supplied libraries: Position Paper,” CAST-21, Jan. 2004.

[197] B. W. Kernighan and D. M. Ritchie, *The C programming language,* 2nd ed. Englewood Cliffs, N.J.: Prentice Hall, 1988.

[198] J. Nellen, T. Rambow, M. T. B. Waez, E. Ábrahám, and J.-P. Katoen, “Formal Verification of Automotive Simulink Controller Models: Empirical Technical Challenges, Evaluation and Recommendations,” in *Lecture notes in computer science, Formal Methods*, K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, Eds., Cham: Springer International Publishing, 2018, pp. 382–398.

[199] The MathWorks Inc., “DO Qualification Kit - Simulink Test Tool Operational Requirements: R2017b,” Natick, MA, USA, Sep. 2017.

Appendix A Review and analysis of data coupling and control coupling

Task	Sequencing	Timing	Control Coupling	Data Coupling
SwVP-DP-MB 1 – Static model analysis		Execution rate is correct and execution dependencies are correctly implemented.	Call interface settings for simulation and code generation of Design Model conform to standard. Coupling and cohesion is assessed with quality measures.	Data interface settings for simulation and code generation of Design Model conform to standard. Code generation settings minimize data coupling. Coupling and cohesion is assessed with quality measures.
SwVP-DP-MB 2 – Static module analysis			Call coupling is limited by encapsulation and not violated. No unused units / models exist. Coupling and cohesion assessed with quality measures.	Data coupling is limited by encapsulation and not violated. No unused data exists. Coupling and cohesion assessed with quality measures.
SwVP-DP-MB 3 – Model review	Order of execution is identified and correct (review call tree in Simulink). Execution dependencie s are identified and correct (review call tree in Simulink).		Call interface of the Design Model complies with HLRs and ICDs. Inter-module call interfaces are defined, correct, and consistent with Software Architecture.	All external inputs and outputs are defined in the Design Model and comply with HLRs and ICDs. Inter-module data interfaces are defined, correct, and consistent with Software Architecture. Units are consistent and comply with specification.
SwVP-DP-MB 5 – Design error detection			No dead logic and thus uncalled functions exist.	Signal ranges of external outputs and inter-module interfaces are not violated (formally proven). Robust data is passed. Run-time errors do not occur (index out of bounds, division by zero, under- and overflow, unexpected floating-point values, formally proven).

SwVP-DP-MB 7 – Simulation / test case and procedure review **SwVP-DP-MB 8 – Simulation testing & result review** **SwVP-DP-MB 9 – Model coverage assessment**			Control coupling is correct and consistent (passing tests).	Interfaces comply to specification. Data is typed correctly and consistently (compilation, simulation, testing). Data corruption is prevented and detected (robustness simulation testing). Signal ranges of external outputs and inter-module interfaces are not violated (run-time diagnostics). Run-time errors do not occur (index out of bounds, division by zero, under- and overflow, unexpected floating-point values, by robustness simulation testing).
SwVP-CP-MB 1 – Automatic code review			Code is consistent with the design (especially Software Architecture).	Data interfaces on model and code level comply and are traceable.
SwVP-CP-MB 2 – Static code analysis for standard compliance **SwVP-CP-MB 3 – Static code analysis for error detection**		[In case of multi-rate models, defect analysis can check further data flow, multitasking, and concurrency issues. Here, an application with a single rate is assumed.]		Unused parameters are detected. All variables are set (or initialized) before use. There is no read before write and no write without further read. Run-time errors do not occur (index out of bounds, division by zero, under- and overflow, unexpected floating-point values).
SwVP-CP-MB 4 – Code review	Order of execution is identified and correct (review of Polyspace call tree)			

SwVP-CP-MB 5 – Code proving				Signal ranges of external outputs and inter-module interfaces are not violated (formally proven). All variables are set (or initialized) before use. Run-time errors do not occur (index out of bounds, division by zero, under- and overflow, unexpected floating-point values, formally proven). [Code Prover can support the identification of unused global variables and functions, however, since it uses abstract interpretation, it may not find all.]
SwVP-CP-MB 6 – SIL testing & result review				Interfaces comply to specification. Data is typed correctly and consistently (compilation, simulation, testing). Data corruption is prevented and detected (robustness simulation testing).

Appendix B Coding rules for code generation

Appendix B.1.1 Conformance

The rules in this section describe the conformance to standards and the process.

CR 1 - Conformance to Software Code Standard

The Source Code shall follow the rules of this Software Code Standard. Deviations shall be handled according to the respective verification task.

The trivial rule results from DO-178C Table A-5:4.

CR 2 - Safe C subset

The automatically generated C source code shall conform to ANSI ISO/IEC 9899:1999 and MISRA C:2012 “Guidelines for the use of C language in critical systems” Appendix E.

Especially in safety-critical applications, the C language prevailed against many other languages like Ada over the last years due to lower implementation costs and vast tool support (cf. C vs. Ada [191]). In contrast, the C language bases on weak specifications and consists of a considerable number of derivations. In 1994, Hatton already stated, “that C is full of holes, but we know where just about all of them are and we know how to plug most of them […] ([191] p.xi)”.

Figure 108 roughly illustrates the approach taken to derive a safety-critical C subset for the given context. Basis is the international standard ANSI ISO/IEC 9899:1999 (C99) [192] specifying a large part of the language, but also containing gaps. The standard categorizes specification gaps in *undefined*, *unspecified*, and *implementation-defined behavior* and summarizes it in Annex J. Undefined behavior is behavior, for which the standard does not postulate requirements (C99 3.4.3). For unspecified behavior, the standard proposes different alternative implementation possibilities (C99 3.4.4). Implementation-defined behavior is unspecified behavior, for which a choice has been made (e.g., by the compiler) and documented (C99 3.4.1).

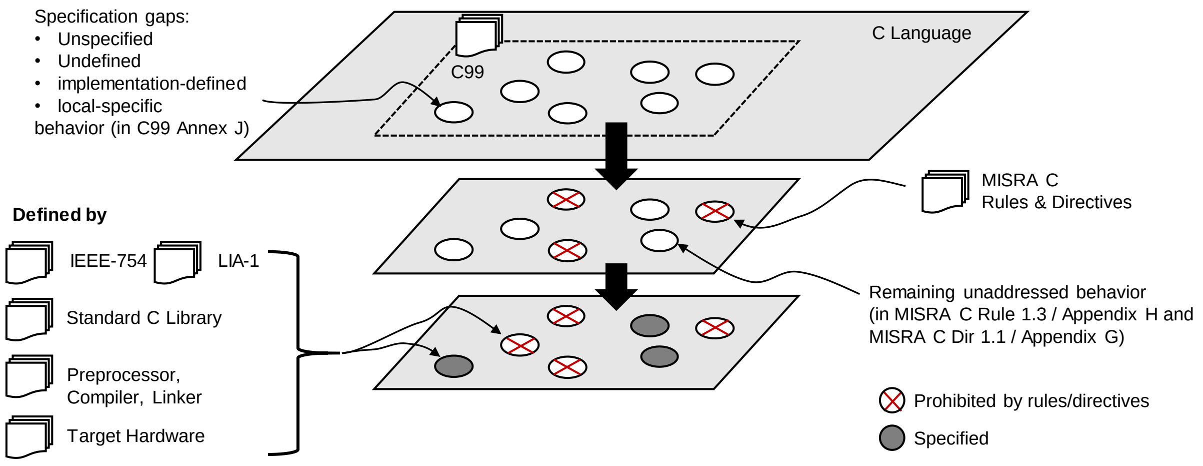


Figure 108: Derivation of a safe C subset

The standard MISRA C:2012 “Guidelines for the use of C language in critical systems” (MISRA C) addresses a large set of these deficiencies by defining *rules* as well as supporting *directives* to avoid undefined and unspecified behavior. MISRA C only covers most critical and frequently occurring undefined and unspecified behavior. It provides a compliance matrix against C99 Annex J in Appendix H. The remaining undefined and unspecified behavior must be avoided, as declared in MISRA C rule 1.3. Implementation-defined behavior of C99 Annex J must be documented and understood according to MISRA Directive 1.1. A checklist is provided in MISRA C Appendix G.

In preparation for MISRA C compliance,

- Implementation-defined behavior has been documented according to checklist MISRA C Appendix G.
- A set of applicable rules and directives has been identified by taking into account Appendix E of MISRA C. The appendix mitigates rule criticality for automatically generated code. For example, various rules are categorized as “readability” rules, which can be ignored in case of code generation. The additional security guidelines in Amendment 1 of MISRA C [193] have not been considered in this work.
- Verification activities have been assigned and a compliance matrix has been assembled.

According to MISRA C 5.3, compliance statements of automatic code generator vendors (like Embedded Coder MISRA C:2012 Compliance Considerations [194])“ can be used to significantly reduce the number of MISRA C guidelines that need to be checked by other means.” In some cases, this makes sense, for example if a code generator does not support dynamic memory allocation, there is no need to check for it. In other, subtler cases, such compliance statements have been considered with care as the code generator is not under tool qualification.

CR 3 - Configuration management

The code generation and the Source Code shall support the Configuration Management Process and considerations.

The Source Code artifacts must be generated, stored, and formatted in a way so that they can be handled according to the principles and processes of configuration management. This starts with unique names, checksums, and goes over change diffing or merging, up to annotating in a review process.

This certainly affects the code placement, but also the generated code itself. For example, assume the coder uses a running number for variables. If a single block is removed, the running number may change all over the model. Identifying the actual change when comparing Source Code of two different versions becomes extremely difficult. The code generation should be as robust as possible.

Appendix B.1.2 Compliance

Compliance requirements have been directly derived from the objectives DO-178C Table A-5:1,2 and the respective additional information in DO-178C 6.3.4.

CR 4 - Compliance with Design Model

A. The Source Code shall be accurate and complete with respect to the Design Model (LLRs).

B. Data- and control flow between Software Architecture and Source Code shall match.

C. The Source Code shall not implement functionality, for which no SW Design exists.

Appendix B.1.3 Accuracy and consistency

Accuracy and consistency touches different aspects. DO-178C 6.3.4f especially lists “correctness” as well as a couple of hardware requirements. Note that there is no objective for Source Code hardware compatibility in DO-178C Table A-5. “Unused variables” are handled as part of section Appendix B.1.5.

CR 5 - Readability

Source Code shall be readable and understandable.

Although auto-generated, Source Code should be readable to a certain extend. Some manual code review is still required, changes must be identified, and debugging or structural coverage analysis is performed with the raw Source Code.

For example, a coder should try to reuse naming from the Design Model for variables, so that the code remains understandable. Some coders allow memory optimizations, which reuse variables of the same data type wherever possible. In this case, the naming of variables does not necessarily fit to the function they fulfill.

CR 6 - Modular code

Generated code of a module shall be modular and independently verifiable on module-level as described by the process.

This is the basic requirement coming from the MBSwD.

CR 7 - Floating-point arithmetic

A. The generated code shall respect floating-point arithmetic according to IEEE 754-2008 / IEC 60559 “Standard for Floating Point Arithmetic” (IEEE 754) with the following floating-point setup:
   - binary representation
   - `Round-to-Nearest` and `Round-Ties-to-Even`
B. Floating-point exceptions shall neither be actively set nor be used in the algorithms to react on exceptions (cf. MISRA C Rule 21.12).
C. Supported data types shall have single (IEEE 754 32 total bits, 8 exponent bits) and double precision (IEE 754 64 total bits, 11 exponent bits).

Discussion Paper DP #17 of DO-248C provides technical considerations for the usage of floating-point arithmetic, other with the design and modeling rules.

The double-precision Floating-Point Unit (FPU) of the PowerPC e300 supports “a floating-point system as defined in the IEEE 754 standard, but requires software support to conform with that standard (PPC Manual p. 3-13).”

Floating-point calculations are controlled by the Floating-Point Status and Control Register (FPSCR), which is documented in the PPC Manual 2.1. The reset values are all zeros, which is the used configuration, too. In this configuration, all floating-point operations conform to the IEEE 754 standard (NI 0) and floating-point exceptions are disabled.

With `Round-to-Nearest` (RN 00), LIA-1 (ISO/IEC 10967-1:2012 “Language independent arithmetic” [195]) requirements concerning casting between floating-point numbers and from integer to floating-point numbers can be fulfilled (C99 H2.4 §4/5). `Round-to-Nearest` and `Round-Ties-to-Even` is also the default configuration of the used compiler (CompCert Manual 5 §6.5).

CR 8 - Supported data types

A. Only the supported data types by the compiler shall be used (see CompCert Manual 5 §5.2.4.2).

B. `long double` shall not be used.

C. Complex types shall not be used.

D. Bit-fields shall not be used.

E. The code shall respect the representation of signed integers as two’s complement.

`long double` is not supported by the selected compiler by default (§5.2.4.2, §6.2.5) and `double` precision is sufficient for the given application. Complex types are not supported by the selected compiler (§6.2.5).

Bit-fields as defined by C99 6.7.2.1 §8 raise various compatibility issues, especially at external software interfaces, and are not necessarily required by the presented application.

Two’s complement is the most common signed integer representation and used by CompCert (§6.2.6) for signed integers.

CR 9 - Byte-ordering compatibility

The generated code shall work independently of the byte-ordering.

Byte-ordering depends on the target computer and is big-endian for the used PPC (Power PC Manual [125] 3.1.2), but mostly little-endian for host desktop computers. In order to execute the same code in both the target (PIL testing) and host environment (SIL testing), the code must be independent of the byte-ordering.

Byte-ordering has significant impact on the design of algorithms. For example, it must be considered, if bit operations are applied or byte streams are decoded, since the host and target endianness may differ.

CR 10 - Cert Standard C Library

> The generated code shall only call external functions of the *Cert Standard C Library*.

Beside language aspects, C99 defines functional prototypes, macros and type declarations for a standard library addressing, for example, mathematical operations, I/O functionality, or memory management. C99 does not describe implementations. They are typically target- and compiler-specific and thus provided with the compiler. Since according to the CAST Position Paper [196], compiler-supplied libraries are not considered as part of the compiler, but separate airborne code requiring development as any other software function, a separate Cert Standard C Library is used.

The functions called by the generated code shall be limited to:

- cert_sin(f)
- cert_cos(f)
- cert_tan(f)
- cert_asin(f)
- cert_acos(f)
- cert_atan(f)
- cert_atan2(f)
- cert_floor(f)
- cert_ceil(f)
- cert_trunc(f)
- cert_pow(f)
- memset

Only for this set of functions, a DO-178C conform implementation can be provided (cf. the parallel work of Nürnberger [120]).

CR 11 - Header and indentation

> A. Any source or header file shall start with a comment header including information about source model version and name, coder version and settings, and a copyright statement.
>
> B. Indentation shall comply to K&R.

The listed information in the header helps identifying the source Design Model and the configuration of the coder. A project statement is required by the company or project.

C99 and MISRA C do not propose indentation rules, therefore indentation shall be according to K&R style [197].

CR 12 - Hardware resources compatibility

Worst-case estimations of required hardware resources of Source Code shall satisfy the hardware requirements. This shall include

- execution timing
- memory and stack

The Source Code shall respect the hardware limitations. Worst-case resource estimations shall be made to avoid significant rework later on.

Appendix B.1.4 Traceability

CR 13 - Tracing to Design Model

A. Bi-directional tracing from Source Code to identifiable parts of the Design Model and derived LLRs shall be in the generated code.

B. The trace shall be added as comment in front of the respective functional code.

C. Traces shall be navigable in both directions.

This rule overwrites MISRA C Dir 3.1 “Requirements Traceability”.

Appendix B.1.5 Verifiability

CR 14 - Unreachable code and data

There shall be no Extraneous or Dead Code in the Source Code or object code.

Any code, which cannot be covered by structural coverage analysis on the Executable Object Code, must be investigated. This code shall be called *unreachable code or data*.

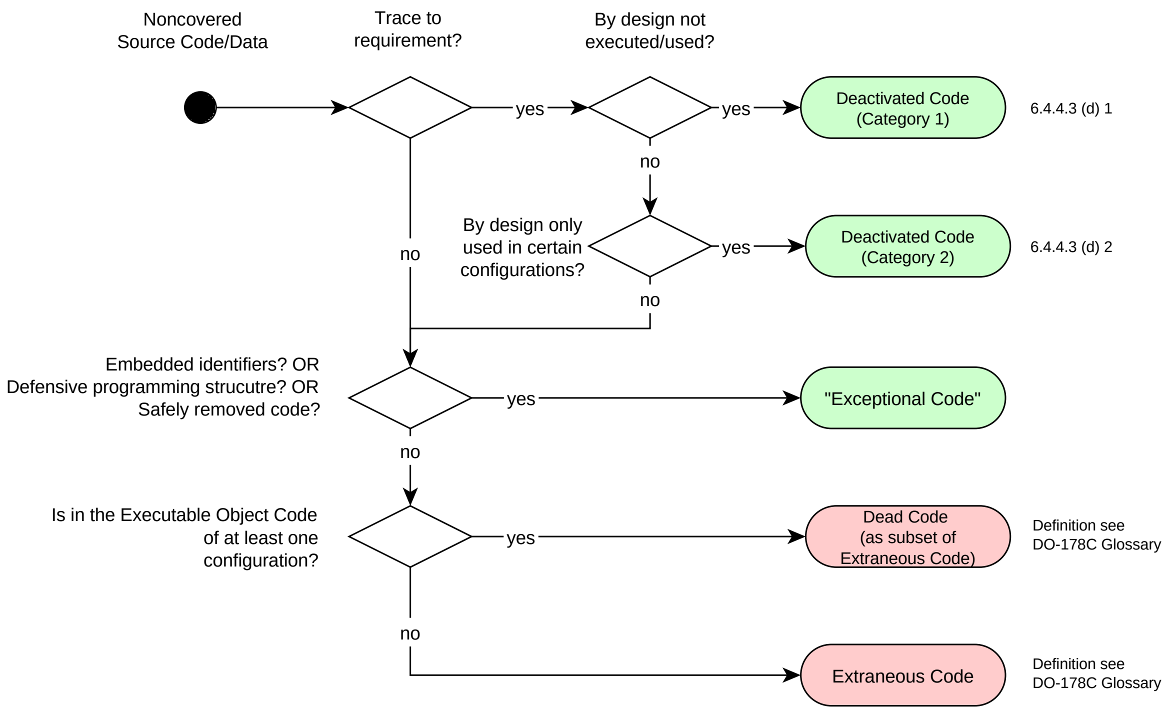
DO-178C further distinguishes *Deactivated Code*, *Extraneous Code*, and *Dead Code* (cf. DO-178C p.111f and [45]). In addition, DO-178C knows a few exceptions, here summarized under *Exceptional Code*. Figure 109 illustrates the classification logic.

Extraneous Code is all unreachable Source Code or object code, which is *not* traceable to a higher-level requirement and which does not satisfy the criteria for Exceptional Code. If Extraneous Code appears in the executable object code of at least one configuration, it is Dead Code. Extraneous and Dead code must be removed according to DO-178C 6.4.4.3(c).

Unreachable code is not regarded as Extraneous or Dead Code, if one of the following criteria is met. Either the code is Deactivated Code, which means it must be traceable to requirements and its deactivation must be planned, or it is Exceptional Code, if one of the following exceptions applies:

- Defensive programming structures
- Embedded identifiers
- Safely removed code

The exceptions are further explained in CR 16. Some of them are allowed in the MBSwD process, others are prohibited.



Noncovered Source Code/Data
Trace to requirement?
By design not executed/used?
yes
yes
Deactivated Code (Category 1)
6.4.4.3 (d) 1
no
By design only used in certain configurations?
no
yes
Deactivated Code (Category 2)
6.4.4.3 (d) 2
no
Embedded identifiers? OR Defensive programming strucutre? OR Safely removed code?
yes
"Exceptional Code"
no
Is in the Executable Object Code of at least one configuration?
yes
Dead Code (as subset of Extraneous Code)
Definition see DO-178C Glossary
no
Extraneous Code
Definition see DO-178C Glossary

Figure 109: Classification of noncovered code according to DO-178C and Rierson [45]

MISRA C uses similar terms, but with a totally different meaning. MISRA 8.2 describes the treatment of so-called *unused code*. Unused code is formally provable and it is *always* a code construction error. In contrast, DO-178C unreachable code bases on test coverage. Unreachable code may exist due to missing requirements, test cases, or the other reasons discussed before.

Different kinds of unused code exist. *MISRA unreachable code* (Rule 2.1), which is code that cannot be executed, is always part of the DO-178C unreachable code. *MISRA dead code* (Rule 2.2) describes code that is executed, but does not influence the program behavior (like two writes before a read). It cannot be reliably revealed by structural coverage analysis. MISRA also addresses unused type, tag, macro, and label declarations as well as unused parameters in functions, which are also not in the scope of classical structural coverage measures.

CR 15 - Deactivated Code

Code shall not be deactivated.

Deactivated Code is just executed in specific or no configurations of the software application. In contrast to Extraneous Code, this behavior is planned. The code must be traceable to requirements, and also the deactivation mechanism must be designed and verified (cf. DO-178C 5.2.4). Deactivated Code during structural coverage analysis is handled according to two categories as explained in DO-178C 6.4.4.3 (d). Hence, deactivation generates more effort than just not executing Source Code and specifying that it is not executed.

According to DO-178C 5.2.4, a typical example of deactivated code is unselected functionality or unused libraries. The deactivation mechanisms proposed by Rierson are profound, involving hardware switches (e.g., pins) or compilers, safely removing deactivated code from the Executable Object Code.

In model-based design, the concept of deactivated design, as introduced in DR 31, shall be leveraged instead of Deactivated Code. It safely removes deactivated design already during code generation.

PDIs shall also not cause Deactivated Code.

CR 16 - Exceptional code

A. The following defensive programming structures in the sense of DO-178C are considered as exceptional code:
   - `else` branch for every `if` construct.

B. Embedded identifiers in the sense of DO-178C shall not be used.

C. Safely removed code shall be documented and the noncovered code in the structural coverage analysis shall be justified accordingly. The documentation shall provide:
   - reason, why this code cannot be removed.
   - evidence that the code does not exist in the Executable Object Code anymore.
   - evidence that procedures are in place ensuring that this code is not inadvertently inserted in the future (e.g., build process with explicit settings and reasons).

Defensive programming structures

The rule may list accepted defensive programming structures, which are accepted if introduced by the code generator and justified as such in the coverage analysis. They are not considered as Extraneous Code. Defensive programming structures are considered as subset of defensive programming practices listed in DO-248C FAQ #32. Defensive programming not specified as defensive programming structure must base on robustness requirements, which are tested.

DO-178C details the scope of defensive programming structures in the glossary for “Deactivated Code” (p. 111) as “*compiler-inserted code* for range and array index checks, error or exception handling routines, bounds and reasonableness checking, queuing controls, and times stamps”.

If code-generator-inserted code for range and array index checks can be considered as defensive programming structure must be clarified with the authority. Here, a conservative approach is taken and the code is not considered as defensive programming structure. The only accepted defensive programming structures are those listed in (A).

Embedded identifiers

The rule may list accepted embedded identifiers, which can be justified as such if not covered by structural coverage analysis. According to Rierson [45, p. 384], for example checksums or part numbers. In the presented project, there is no need for embedded identifiers.

Safely removed code

Safely removed code is code, which is not in the Executable Object Code, but only in the Source Code or object code. The documentation objectives are from Rierson [45, p. 385].

A common example is generated code, which has a non-configurable code interface with functions, that are never called under some circumstances (cf. section MR 15).

Appendix C Code generation examples

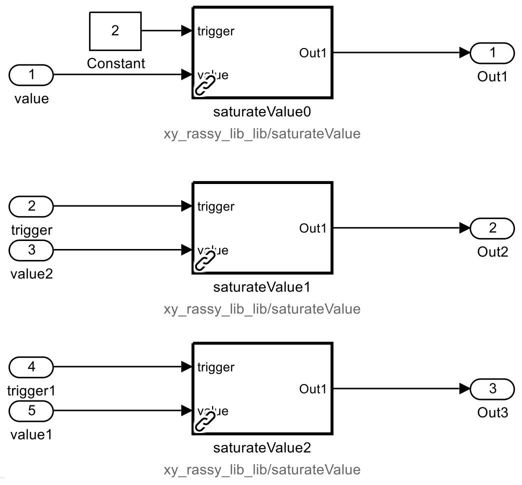
Appendix C.1 Example 1: Context-dependent reusable subsystems

This example shows, how the context, in which the atomic subsystem is embedded in, can influence code generation.

All three atomic subsystems are instances of the same library block, so they have the same settings. Inport and outport blocks have fix, not inherited values.

The first subsystem is connected to a constant input. The coder recognizes this and generates a function with constant input. For the second and third atomic subsystem, the input is connected to a model inport. The coder recognizes that the value is a true argument and generates the reusable function.

For the second and third atomic subsystems, the structural checksum is equal and a reusable function is generated. For the first, a separate function is generated.

Model with reusable subsystems	**MODEL**
 2 Constant 1 value trigger value Out1 saturateValue0 xy_rassy_lib_lib/saturateValue 1 Out1 2 trigger 3 value2 trigger value Out1 saturateValue1 xy_rassy_lib_lib/saturateValue 2 Out2 4 trigger1 5 value1 trigger value Out1 saturateValue2 xy_rassy_lib_lib/saturateValue 3 Out3	

saturateValue.c (first function) C CODE

```c
/* Output and update for atomic system: '<Root>/saturateValue0' */
void xy_rassy4_saturateValue(real_T rtu_value, B_saturateValue_xy_rassy4_T
  *localB, const ConstB_saturateValue_xy_rassy_T *localC)
{
    /* Switch: '<S2>/Switch' incorporates:
   *  Constant: '<S2>/Constant'
   */
  if (localC->Compare) {
    localB->Switch = rtu_value;
  } else {
    localB->Switch = 0.0;
  }

  /* End of Switch: '<S2>/Switch' */
}
```

xy_rassy4_saturateValue1.c (second function) C CODE

```c
/* Output and update for atomic system:
 *    '<Root>/saturateValue1'
 *    '<Root>/saturateValue2' */
void xy_rassy4_saturateValue1(real_T rtu_trigger, real_T rtu_value,
  B_saturateValue_xy_rassy_p1u2_T *localB)
{
  boolean_T b_Compare;

  /* RelationalOperator: '<S6>/Compare' incorporates:
   *  Constant: '<S6>/Constant'    */
  b_Compare = (rtu_trigger <= 3.0);

  /* Switch: '<S3>/Switch' incorporates:
   *  Constant: '<S3>/Constant'    */
  if (b_Compare) {
    localB->Switch = rtu_value;
  } else {
    localB->Switch = 0.0;
  }

  /* End of Switch: '<S3>/Switch' */
}
```

Appendix C.2 Example 2: Global signals in reusable models

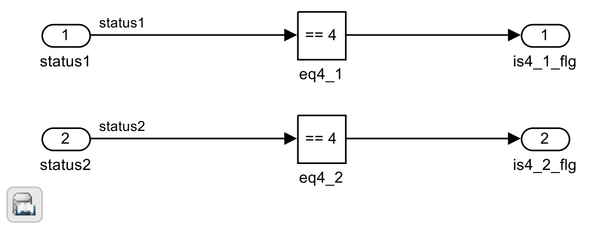
This example shall hilight the naming problem with signals in REUSABLE MODELS. The REUSABLE MODEL `xy_sig_reusable` has two inports with explicitly assigned signal names and storage class `SimulinkGlobal` to expose the variable for testing. External storage classes are not allowed, since they cause conflicting identifiers.

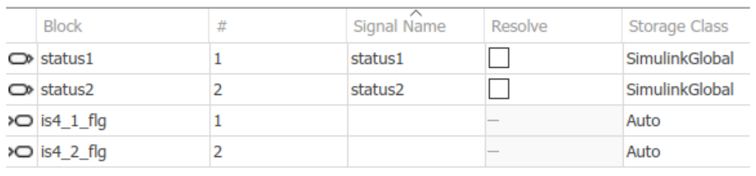
Model `xy_integr_sig_reusable` integrates three instances of `xy_sig_reusable`. The first observation in the generated code is that the signals may be placed in different structures. As can be seen in line 7 and 12, variable `status1` is a field of the input structure (`_U`). Since `status2` cannot be taken directly from the input, but is multiplied with two, the signal `status2` is generated in the block IO (`_B`) structure (line 4).

The second observation is that signals with the same name are extended by a mangle. Since `status2` in line 26 already exists in the `_B` structure, a postfix has been generated.

As a consequence, the naming and placement of a signal with non-external storage class depends on various factors.

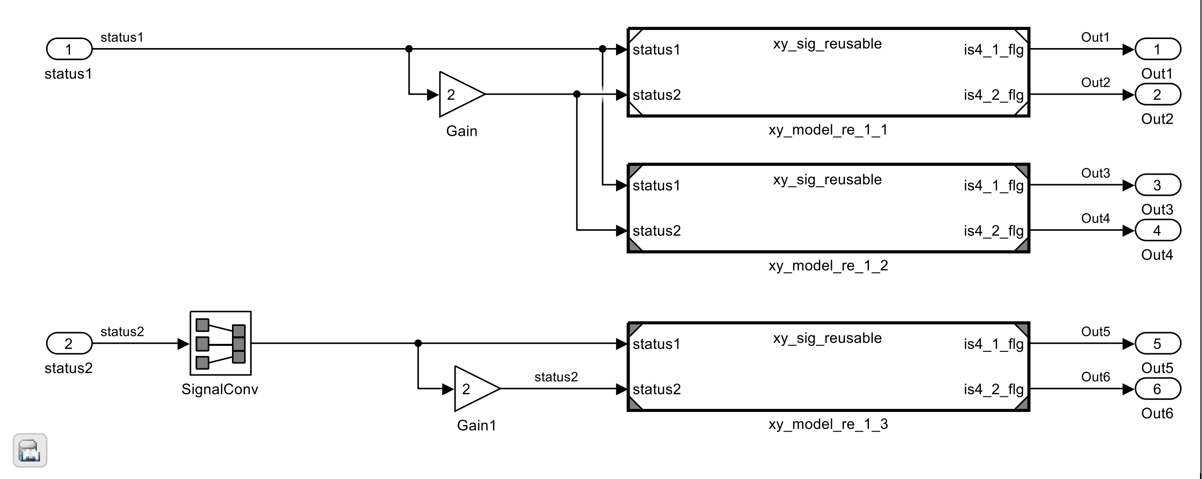
Signals > xy_sig_reusable	**MODEL**





	Block	#	Signal Name	Resolve	Storage Class
	status1	1	status1	☐	SimulinkGlobal
	status2	2	status2	☐	SimulinkGlobal
	is4_1_flg	1		–	Auto
	is4_2_flg	2		–	Auto

Signals > xy_integr_sig_reusable	MODEL



Xy_integr_sig_reusable.c	C CODE

```
        /* Gain: '<Root>/Gain' incorporates:
        *  Inport: '<Root>/status1'
        */
        xy_integr_sig_reusable_B.status2 = 2.0 * xy_integr_sig_reusable_U.status1;

        /* ModelReference: '<Root>/xy_model_re_1_1' */
        xy_sig_reusable(&xy_integr_sig_reusable_U.status1,
                &xy_integr_sig_reusable_B.status2,
                &xy_integr_sig_reusable_Y.Out1, &xy_integr_sig_reusable_Y.Out2);

        /* ModelReference: '<Root>/xy_model_re_1_2' */
        xy_sig_reusable(&xy_integr_sig_reusable_U.status1,
                &xy_integr_sig_reusable_B.status2,
                &xy_integr_sig_reusable_Y.Out3, &xy_integr_sig_reusable_Y.Out4);

        /* SignalConversion: '<Root>/SignalConv' incorporates:
        *  Inport: '<Root>/status2'
        */
        xy_integr_sig_reusable_B.status1 = xy_integr_sig_reusable_U.status2;

        /* Gain: '<Root>/Gain1' */
        xy_integr_sig_reusable_B.status2_l5ij = 2.0 * xy_integr_sig_reusable_B.sta-
        tus1;

        /* ModelReference: '<Root>/xy_model_re_1_3' */
        xy_sig_reusable(&xy_integr_sig_reusable_B.status1,
                &xy_integr_sig_reusable_B.status2_l5ij,
                &xy_integr_sig_reusable_Y.Out5, &xy_integr_sig_reusable_Y.Out6);
```

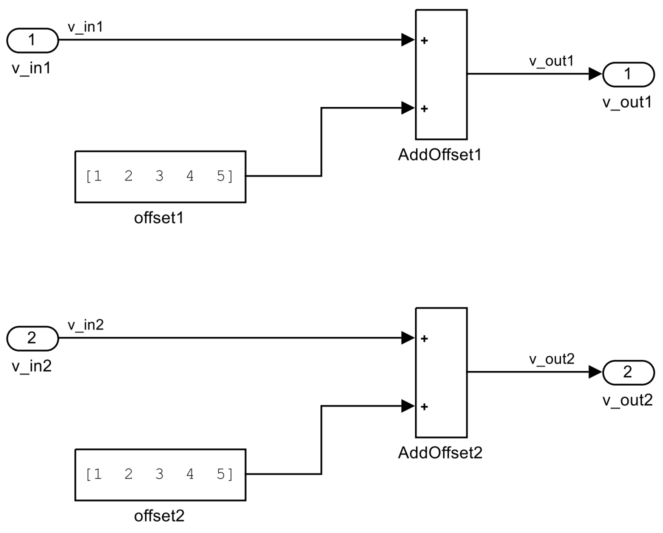
Appendix C.3 Example 3: Parameter pooling

This example shows a model, which uses the same offset vector twice in independent addition operations. The vector is directly defined in a Constant block as mask parameter.

Since arrays in C cannot directly be used in operations, a variable must be declared and defined. These non-inlined parameters are collected in a single structure named `ConstP_<model-name>_T`, like `ConstP_xy_calcOff1_T` in the first code snippet. The duplicate usage is recognized by Embedded Coder and a common, pooled variable is generated.

The name of the field in the structure is typically inherited from the context, in which the parameter is referred to. Is the parameter name used multiple times, Embedded Coder simply numbers it with `pooled<N>`.

Parameters > xy_calcOff | **MODEL**



xy_paramPool1.c | **C CODE**

```c
/* Constant parameters (auto storage) */
typedef struct {
  /* Pooled Parameter (Expression: [1 2 3 4 5])
   * Referenced by:
   *   '<Root>/offset1' (Parameter: Value)
   *   '<Root>/offset2' (Parameter: Value)
   */
  real_T pooled1[5];
} ConstP_xy_paramPool1_T;
```

xy_paramPool1_data.c | **C CODE**

```c
/* Constant parameters (auto storage) */
const ConstP_xy_paramPool1_T xy_paramPool1_ConstP = {
  /* Pooled Parameter (Expression: [1 2 3 4 5])
   * Referenced by:
   *   '<Root>/offset1' (Parameter: Value)
   *   '<Root>/offset2' (Parameter: Value)
   */
  { 1.0, 2.0, 3.0, 4.0, 5.0 }
...
```

xy_paramPool1.c	**C CODE**

```
/* Outport: '<Root>/v_out1' incorporates:
*  Constant: '<Root>/offset1'
*  Inport: '<Root>/v_in1'
*  Sum: '<Root>/AddOffset'
*/
for (i = 0; i < 5; i = i + 1) {
    xy_calcOff1_Y->v_out1[i] = xy_paramPool1_U->v_in1[i] +
    xy_ xy_paramPool1_ConstP.pooled1[(i)];
}
```

Appendix D Comparison of Standard C Library integrations

Criteria	Code Replacement Library	Legacy Code
Simulation Behavior	A Code Replacement Library replaces calls in the code generation process. Only SIL or PIL simulations modes, which compile the actual Source Code, use the Cert Standard C Library. Normal mode simulation behavior may differ depending the Code Replacement Library implementation.	All simulation modes already use the target code (S-Function).
Limitations on the Implementation of the Cert Standard C Library	If there are target dependencies (e.g., endianness may differ between host and target [31], or calls to RTOS, drivers,...), Normal mode simulation works in any case, since it does not use the replacements. SIL simulation might fail.	S-Functions can only be compiled, if there is no limiting target dependency and the S-Functions are executable on the host
Replacement Flexibility	The same block type can expand to numerous different replacements depending on its settings. For example, the Trigonometric Function block works for different input data types and dimensions of the incoming signal.	Legacy S-Function block have strongly typed interfaces. A variety of different blocks is required to cover all use cases. Calculations on multi-dimensional signals of different length must be implemented in wrapping loops.
	Only the actual function calls are replaced, but robustness code remains (e.g., an `asin` generates robustness code around the function). This may, depending on the verification strategy, lead to unnecessary or even dead code.	The S-Functions are independent blocks, which do not generate additional robustness code.
	Blocks with combinations of settings, for which no replacement exists, can be simulated and code can be generated. The call into the wrong library would be detected late during compilation. However, model checks can prevent invalid settings.	Only supported functionality can be used by developer by definition.
	Replacements affect the canonical shared code. However, since in the chosen approach canonical shared code is not generated, this is not a true benefit.	No replacement in canonical shared code (however, existing code is used anyway).
		Using S-Functions in SF blocks has not further been investigated, but should be checked for feasibility, if this is a necessary requirement.
Verification	SLCI: Various limitations, but most mathematical functions are covered [121, p. 2-26].	SLCI: No limitations, as long as integration into start and initialize functions is not required [121, pp. 3–49f.].
	SL Design Verifier: Code Replacement Libraries do not affect the analysis. Different behavior of the replaceable may lead to wrong design verifier results.	SL Design Verifier: Can analyze legacy code with limitations. If not analyzable, S-Functions must be manually stubbed. Respective stubbing functions must be provided.
	Polyspace Bug Finder / Code Prover: Works on generated code, and thus needs the Code Replacement Libraries for analysis.	Polyspace Bug Finder / Code Prover: Works on generated code, and thus needs the Code Replacement Libraries for analysis.

Appendix E Selected code generation settings

Setting Name and Purpose	Chosen Setting	Rationale
`ReqsInCode` If enabled, copies requirement information into code.	`Off`	If `On`, any minor change in the requirement would update the Source Code, although the Design Model itself may not require a change. This causes unnecessary overhead in terms of configuration management, verification, or build automation. In addition, direct traceability from code to HLRs is not necessary. The code traces to the design and the design traces to the HLRs, which is sufficient.
`ExpressionFolding` Eliminates superfluous local variables by creating compound expression.	`Off`	Reduces the number of local variables, but may result in long compound expressions. This may decrease traceability granularity (since all source block traces are just listed above the long compound expression). CompCert applies certain optimizations for the local variables, so turning off superfluous expression folding did not come along with heavy performance loses. In contrary, analysis for WCET analysis were more accurate [37].
`PortableWordSize` Generates code, which compiles on platforms with different word sizes.	`On`	Required by SIL.
`LocalBlockOutputs` Prefers local against global block outputs.	`On`	Although automatic code generators can better cope with global variables than manual programmers, their occurrence shall be minimized to reduce coupling. Local block outputs are subject to CompCert optimization.
`BufferReuse` Reuse local block output variables. If activated, local variables storing block outputs are reused for different blocks.	`Off`	Reuse of local block output may reduce the stack memory, but makes the code hard to read and decreases robustness of the generated code in case of changes. For example, a variable named by its first usage may be reused all over the code. If the first usage renamed, changes occur all over the code.
`GlobalBufferReuse` Reuse global variables.	`Off`	Cf. `BufferReuse`
`GlobalVariableUsage` Determines whether the code generator shall prefer local or global variables.	`Minimize global data access`	Although automatic code generators can better cope with global variables than manual programmers, their occurrence shall be minimized to reduce coupling.
`UseSpecifiedMinMax` Optimize code based on signal ranges.	`Off`	This option may automatically remove dead code and disguise design errors.
`CastingMode` Controls, how the coder makes castings.	`Standards`	Embedded Coder tries to generate MISRA C compliant casting.
`Loop unrolling`	`2`	The default value of 5 leads to partially unrolled loops in (3x3)x(3x1) matrix-vector multiplications, which is an often used operation in controllers. SLCI does not support partially unrolled loops.

Appendix F Simulink Design Verifier model preprocessing

There are some major challenges with using Design Verifier for the proposed safe modeling subset, since in R2017b, a significant number of unsupported SL features and limitations exist with Design Verifier [123, pp. 3-10ff].

The problems mainly involve constructs of the safe modeling subset, which allow the specification of a narrow contract with signal ranges. As repetition, according to DR 11, inter-module interfaces specifications can be wide, narrow and unconstrained (which is equal to wide concerning ranges). There are explicit data flow interfaces realized with model references and their Inport and Outport blocks in SL. And there are implicit data flow interfaces modeled with Data Store Read/Write blocks. Furthermore, PDIs are a special case of parameters, which have to be considered as non-constant with a narrow contract. MR 23 states, that only Simulink.Bus objects should have signal ranges and thus represent narrow contracts.

The only option to overcome the stated issues was a customization of model pre-processing. Design Verifier performs a pre-processing anyway, e.g., it replaces model references with subsystem blocks. So it is not a violation of the workflow.

Issue 1: In R2017b, the analysis always traverses a full model hierarchy. It cannot be limited to certain levels. Thus, Design Verifier analyses are difficult to apply in large models and no benefit is obtained from narrow interface specifications. A subsystem analysis as shown in [198] is not a feasible approach, since it requires intermediate signal ranges, which are not specified here (cf. MR 23).

The implemented solution stubs external units as depicted in Figure 110. Calls inside a unit are not exchanged. The stub models are auto-generated and exchanged during block replacement.

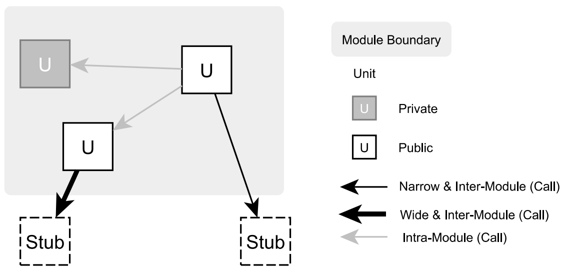


Figure 110: Stubbed unit

The stub models terminate all inputs and provide dummy output signals, which formally span the whole specified range. The termination of inputs is possible, since Design Verifier already proves properties, like the signal ranges, at the place, where the incoming signal is written.

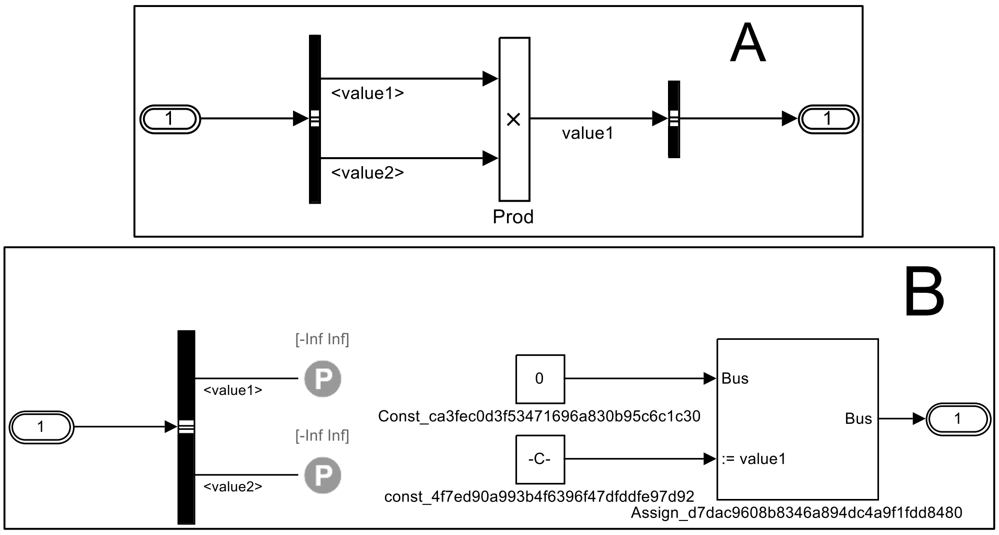

A
1
<value1>
<value2>
×
Prod
value1
1
B
1
<value1>
<value2>
[-Inf Inf]
P
[-Inf Inf]
P
0
Const_ca3fec0d3f53471696a830b95c6c1c30
-C-
const_4f7ed90a993b4f6396f47dfddfe97d92
Bus
:= value1
Bus
Assign_d7dac9608b8346a894dc4a9f1fdd8480
1

Figure 111: Original model (A) and auto-generated stub model (B)

For example, from model A in Figure 111, the stub model B is generated. The process is not so straight-forward as it seems to be due to the limited Simulink.Bus support of Design Verifier in R2017b. Another problem to overcome is that an incoming signal, which is terminated with the Terminator block, will be completely ignored in the Design Verifier analysis. This can be avoided by connecting the incoming signal with a Proof Objective block telling Design Verifier that the derived ranges are required. Note that property proving is a separate analysis mode in Design Verifier and not executed. The only purpose of the blocks is to retain the signal. Unfortunately, Proof Objective blocks need a scalar numeric input, and, in consequence, each bus must be unfolded. The output replacement is similar to the replacement performed for data stores and PDIs.

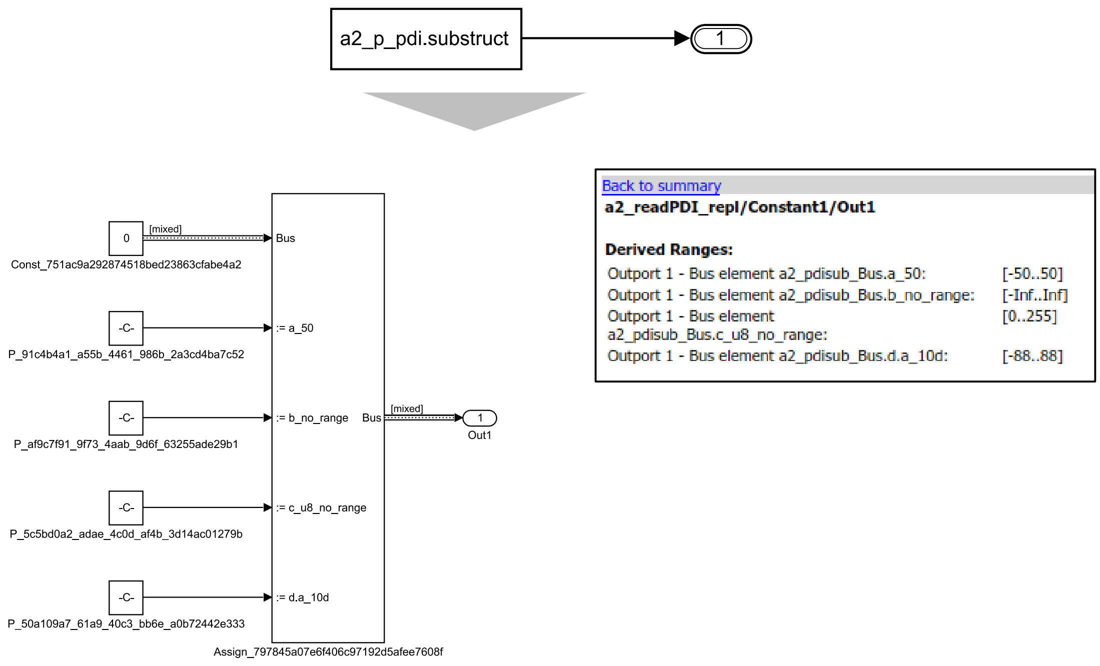
Issue 2: Tunable parameters are not considered as tunable by default in Design Verifier analysis. They are constant without a dynamic range, which may lead to incorrect results (here especially for PDIs).

Issue 3: Signal ranges specified in Simulink.Bus objects attached to Simulink.Parameter and Simulink.Signal objects (as required for Data Stores) are not considered. For example, a value read from a Data Store field does not have the signal range of the underlying bus data type.

For Parameter Constant objects, the default behavior of Design Verifier fits, which assumes that the values are invariant. For Constant blocks referencing a PDI and Data Store Read blocks, the signal range must be “artificially” expanded. In principle, Design Verifier provides a special configuration for tunable parameters [123, pp. 5-5f.]. In a separate file, the signal range can be specified for Simulink.Parameter objects, but only for parameters of numeric type. Unfortunately, Simulink.Parameter objects with Simulink.Bus data types are not supported in R2017b. And there is no feasible way for Simulink.Signals used as data store.

The preprocessing in the implemented workaround thus replaces Data Store Read and Constant blocks, which reference PDIs, with a block pattern supported by SLDV. Therefore, a temporary numeric Simulink.Parameter object is generated for every element of the bus that is read. If the output is of bus data type, a Bus Assignment block is used to compose these Simulink.Parameter objects. For the temporary parameters of numeric type, a parameter configuration file is created with the respective ranges.

Figure 112 shows an example, in which a simple Constant block that reads a nested structure from a PDI, is expanded to provide signal ranges for Design Verifier analysis. The respective parameter configuration file is shown in Listing 13.


a2_p_pdi.substruct
1
0
[mixed]
Const_751ac9a292874518bed23863cfabe4a2
Bus
-C-
P_91c4b4a1_a55b_4461_986b_2a3cd4ba7c52
:= a_50
-C-
P_af9c7f91_9f73_4aab_9d6f_63255ade29b1
:= b_no_range
Bus
[mixed]
1
Out1
-C-
P_5c5bd0a2_adae_4c0d_af4b_3d14ac01279b
:= c_u8_no_range
-C-
P_50a109a7_61a9_40c3_bb6e_a0b72442e333
:= d.a_10d
Assign_797845a07e6f406c97192d5afee7608f

Back to summary

a2_readPDI_repl/Constant1/Out1

Derived Ranges:

Outport 1 - Bus element a2_pdisub_Bus.a_50:	[-50..50]
Outport 1 - Bus element a2_pdisub_Bus.b_no_range:	[-Inf..Inf]
Outport 1 - Bus element a2_pdisub_Bus.c_u8_no_range:	[0..255]
Outport 1 - Bus element a2_pdisub_Bus.d.a_10d:	[-88..88]

Figure 112: Example of SL Design Verifier signal range expansion

```
function params = a2_readPDI_sldv_param_init()
params = [];

% Parameters for model a2_readPDI
params.P_91c4b4a1_a55b_4461_986b_2a3cd4ba7c52 = [-50 50];
params.P_af9c7f91_9f73_4aab_9d6f_63255ade29b1 = [-Inf Inf];
params.P_5c5bd0a2_adae_4c0d_af4b_3d14ac01279b = [-Inf Inf];
params.P_50a109a7_61a9_40c3_bb6e_a0b72442e333 = [-88 88];

end
```

Listing 13: Example of SL Design Verifier parameter configuration file

According to MR 26, data stores shall only be written once and completely. If the data store is of bus type, the written signal must be of bus type, too. In consequence, it must be built somewhere inside the model. For any writing of a bus, Design Verifier generates signal range objectives and proves them. A bus written into a data store can thus never violate the range constraints, except if an unconstrained top-level input is directly fed into the data store. Since unconstrained interfaces are subject to robustness testing, this will be revealed.

The transformations work for SL, but not for SF. Thus, PDIs and data stores shall not be used in SF (cf. MR 26).

Appendix G Simulation case development rules

In the following, simulation case development rules (SR) for SL Test in the sense of the MBSwD process are given.

SR 1 – Simulation procedures

Simulation procedures are documented in separate high-level documents.

Simulation and test procedures differ and must be exchangeable. In addition, the procedure is typically identical for all test cases. Rierson notices the possibility to write "separate high-level documents that explain, how to execute the test cases" [45, p. 208].

SR 2 – Simulation cases

A. Each simulation case shall consist of two parts:

   - SL test case object in SL Test Manager (simulation/test case and procedure).
   - Simulation/test case resources.

B. Simulink Test Manager files (SL test files) shall functionally group simulation cases. Simulink Test Manager suites (SL test suites) inside SL test files may be used for further hierarchical structuring.

Depending on the type of test, additional resources (like inputs, outputs, Simulink models,..) are required to run the test.

SR 3 – Simulation case documentation

Simulation cases shall be fully documented in SL test cases as in Table 52.

Test Case Revision History	In description of SL test case.
Test Author	In description of SL test case.
Test Procedure	Reference to applicable simulation/test procedure(s)
Identification of Software Under Test	By section “System under test” and connected test harnesses.
Test Description	In description of SL test case.
Requirements(s) Tested	Documented in section “Requirements”.
Target Environment Support	Add as tag FULL-IMAGE to indicate that this test can be executed on the full image of the whole software (beyond MBSwD) in the target environment.
Test Category	HIGH-LEVEL (derived from HLRs) LOW-LEVEL (derived from LLRs) or both In case of a low-level test, a rational for its necessity shall be provided.
Test Type	Add as tag (exclusive) NORMAL ROBUSTNESS
Test Inputs	Documented in section “Inputs”
Test Steps or Scenarios	Parameter overwrites, configuration overwrites, inputs and iterations describe the scenarios. In Simulation Tests, assessment blocks contain a detailed description of the steps.
Test Outputs	Documented in section “Outputs”
Expected Results	Simulation Test: Documented in the test model. Baseline Test: Documented in section “Baseline Criteria”. Equivalence Test: Documented by results of the compared simulation.
Pass/fail Criteria	Simulation Test: Defined by assessment blocks and implementation. Baseline Test: Defined by expected results and the tolerances specified in section “Baseline Criteria”.

Table 52: Simulation case documentation

The criteria of Table 52 have been taken from Rierson [45] 9.6.5.2 and mapped to the features of SL test cases.

Traceability is, as for Design Models, established with *SimPol* according to Figure 73.

SR 4 – Traceability to HLRs

A. The trace between a simulation cases and a simulation procedure (Polarion work item) shall be established via surrogate linking.

B. Only SL test cases shall trace to HLRs (not test suites).

C. Every HLR shall trace to at least one simulation case.

D. A sufficient number of robustness and normal range test cases shall be derived from HLRs.

SL Test can override a couple of simulation settings of the model under test, but not all. Unfortunately, SL Test does not distinguish between execution and test case settings.

SR 5 – Traceability to LLRs

A. Simulation cases derived from LLRs should be avoided. Simulation and testing objectives should be achieved with cases from higher-level requirements.

B. If a simulation case has to be derived from an LLR, the trace between a simulation cases and the part of the Design Model, from which it has been derived, shall be established with the RMI directly (without *SimPol*).

C. Only SL test cases shall trace to LLRs (not test suites).

In some cases, it is inevitable to develop a LLR simulation case. These simulation cases perform low-level testing.

SR 6 – Simulation case execution settings

A. Model coverage settings shall be set on SL test file level and be inherited from nested SL test suites or SL test cases.

   - Record coverage for system under test and for referenced models.
   - Selected coverage is execution, decision, and condition coverage.

B. Simulation mode shall be set to `Normal`.

The chosen model coverage criteria are discussed in section 8.2.9. Simulation cases must always be executed in Normal mode to allow coverage recording. In addition, only Normal mode provides availability of all diagnostics.

SR 7 – Testable units

A. Only testable units shall be selected as simulation targets.

B. Inputs, outputs, intermediate values, and states must be testable.

Testable units are discussed in MR 6.

SR 8 – Simulation signal recording

A. States, data store values, and outputs of the models under test shall be recorded for equivalence testing. If these signal values are sufficient for equivalence testing, additional signal logging shall be activated.

B. Recorded simulation outputs shall have identifiable and understandable names.

Test cases in SL Test do not record outputs by default. For example, a test harness model as depicted in Figure 113 verifies outputs implicitly in the Test Sequence block.

SR 9 – Simulation output recording

Test in SL Test shall comply with the tool operation requirements specified in the MathWorks tool qualification kit.

The tool qualification kit for Simulink Test imposes some restriction on the usage of SL Test, e.g., it only supports `Intersection` and `Union` modes for synchronizing time vectors and just the alignment of data vectors (`Interp`) in `Zoh` mode [199].

SR 10 – Recommended test harness models

A. The SL test harness feature should not be used. Test harnesses should be independent models.

B. Model variants should be used for switching between the different simulation modes (Normal, SIL, and PIL).

C. In SIL and PIL mode, the code interface of the model reference under test should be set to `Top Model` for top-level models, and `Model Reference` for all other models.

D. Test harness models should use the base workspace as global workspace.

E. Test harness models should have a separate configuration set. The configuration set should match the test case settings in SL Test, but also further detail e.g., coverage settings.

It is recommended to control the simulation mode not via the test case, but in the model block properties of the test harness, since in R2017b, the simulation mode cannot be switched without tainting the test case. To avoid tainting the test harness itself, the model block can be placed in a model variant subsystem (one for each simulation mode, Figure 113). The simulation mode can then be easily switched by changing a parameter in the workspace. Second argument for this implementation is the fact that the code interface must be specified (top model or model reference) for SIL (Figure 114). This cannot be done in the test case. If the code interface is not correctly set, code is regenerated and not the final product code is simulated.

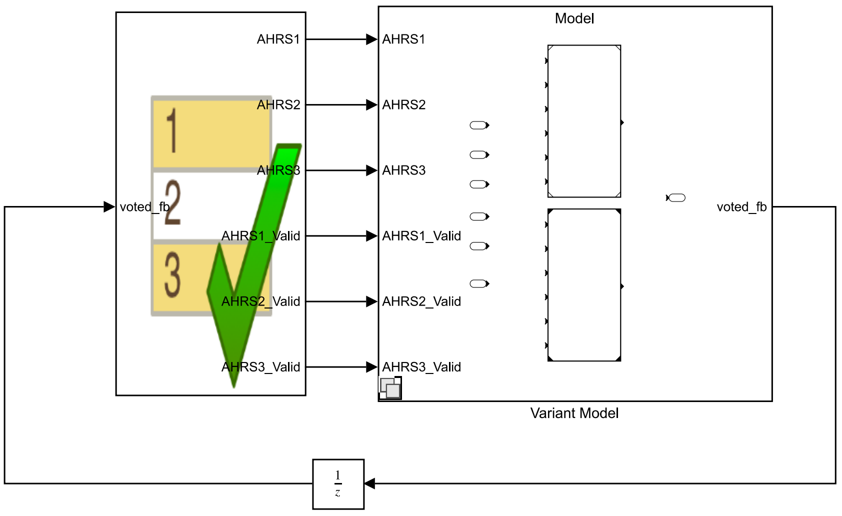
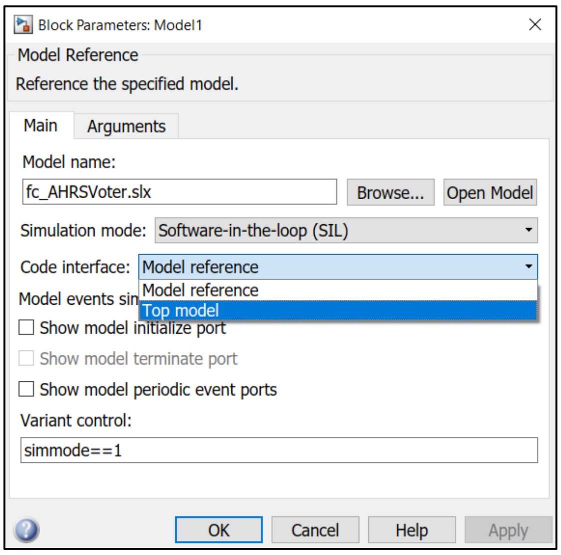


Figure 113: Test harness model with simulation mode variants


Block Parameters: Model1

Model Reference

Reference the specified model.

Main | Arguments

Model name: fc_AHRSVoter.slx [Browse...] [Open Model]

Simulation mode: Software-in-the-loop (SIL)

Code interface: Model reference
- Model reference
- Top model

Model events sin

☐ Show model initialize port

☐ Show model terminate port

☐ Show model periodic event ports

Variant control: simmode==1

[OK] [Cancel] [Help] [Apply]

Figure 114: SIL code interface setting of model block

Test harnesses are a practical feature, but are not compatible with the proposed model variant implementation in R2017b. Manually created test harness models have a little bit less automation in case of interface updates, but fulfill the same purpose.

In addition, it is also reasonable that a test harness model does not use one of the data dictionaries of the Design Model to avoid tainting them unnecessarily. It should reference the base workspace. In consequence, it also requires a separate testing configuration set. Here, the top-level configuration set has been used with additional settings concerning coverage (important is that the settings match the test case settings, otherwise compilation errors could be observed).

Appendix H List of artifacts

This section provides a list of artifacts, which the author has created during this work.

Process Documentation

- Software Verification Plan Template
- Code Standard Document Template
- Naming Convention Document
- Documented design rule set
- Documented module design rule set
- Documented fundamental modeling rules
- Documented traceability rules
- Documented simulation testing rules
- Traceability matrix from DO-178C to design rules to module design rules, traceability, and coding rules

Modeling environment

- SL model and code generation configuration settings for different model types
- DO-331 Foundation Library
- SECI List
- Code Generation Templates
- Prepared canonical shared code
- Code Replacement Libraries
- Legacy Code blocks for floating-point special quantity support

Jobs in the process-oriented build tool (MATLAB code) and related artifacts

- Jobs to set up the environment
  - Job to check SLECI
  - Job to check installed bug fixes
  - Job to register Embedded Coder target extensions
  - Job to setup DO-331 Foundation Library
  - Job to load configurations
  - Job to setup model advisor, default configuration, and custom checks
  - Job to setup RMI and preferences
  - Job to setup SL file generation control
- Job for generation of modular code and verification
- Job for shared code generation
  - Hooks for code generation
- Job to execute static model analysis, evaluate results, and create report
  - Custom model checks
  - Custom model check configuration
  - Project deviation document for static model analysis
- Job to execute static module analysis, evaluate results, and create report
  - Selection of custom module checks
  - Custom module check configuration
- Job for model review and review list generation
- Job for traceability review and review list generation
  - Templates for bottom-up and top-down traceability review lists
- Job to execute design error detection, evaluate results, and create report
  - Extended pre-processing algorithm for Simulink Design Verifier
- Job for simulation case & result review and review list generation
- Job to execute simulation cases and evaluate results
- Job to execute model coverage assessment (incl. aggregation) and evaluate results

- Job to execute automatic code review, evaluate results, and create report
- Job to execute static code analysis for compliance checking, evaluate results, and create report
    - Polyspace check selection
    - Polyspace analysis options
    - Project deviation document for static code analysis
    - DRS post-processing algorithm
- Job to execute static code analysis for error detection, evaluate results, and create report
- Job to assemble code review list
- Job to execute simulation cases, perform equivalence checking, and evaluate results in SIL
- Job to execute structural coverage assessment (incl. aggregation) and evaluate results

Complete software tools

- Tool to manage ICDs and auto-generate framework code for the MBSwD
- Contribution in PIL framework
- *SimPol*
- Process-oriented build tool *mrails*