A Process-oriented Build Tool for Safety-critical Model-based Software Development

Markus Hochstrasser[1], Stephan Myschik[2] and Florian Holzapfel[1]
[1]*Institute of Flight System Dynamics, TU München, Boltzmannstraße 15, 85748 Garching, Germany*
[2]*Chair of Flight Mechanics and Flight Control, Universität der Bundeswehr München, Werner-Heisenberg-Weg 39, 85577 Neubiberg, Germany*

Keywords: Build Automation, Workflow Management System, Component-based Software Engineering, Software Development Process, Model Standards, Model Scaffolding, Continuous Integration.

Abstract: By conquering new application areas, software complexity and size increases steadily. Development cycles must become faster to deliver critical updates in shorter time. Simultaneously, software takes over more and more safety-critical tasks, requiring strict software development processes. Up to today, these processes suffer from extensive manual review work and written, static documentation in form of standards, checklists, and procedures. This paper presents a monolithic, process-oriented build tool for model-based development in MATLAB, Simulink, and Stateflow. Beyond classical build automation functionality, it supports and accelerates process workflows. The tool provides infrastructure to formalize and ship workflows, checklists, and standards, but also features to assess completeness, consistency, compliance, and cleanliness with respect to them. Additionally, it allows definition of dynamic, incrementally updated checklists, and composes traceability in parallel with the build. The efficacy and achievable process coverage is demonstrated in an example application.

1 INTRODUCTION

With increasing requirements for automatic and autonomous functionality, software development faces new challenges concerning implementation and verification techniques. Algorithms rapidly grow in size and complexity (Basagiannis, 2016). At the same time, software has to fulfill higher safety and security standards due to the evolved tasks they take over. Besides, software development not only gets larger, it also becomes faster. With new software trends, like connected or cloud-based algorithms, software development life cycles have to accelerate drastically. Increased vulnerability of systems requires to deliver new software releases more frequently.

To handle complexity and accelerate development, the transition from classical to model-based or even model-driven software development (MBD/MDD) is a step, which many companies have already taken over the last years. It comes along with enormous cost and time savings, but requires elaborate model-based development "and an established development process" (Broy et al., 2014) as well as a high degree of scalability and automation of tasks to fulfill objectives (Bhatt et al., 2010).

By developing flight control algorithms for safety-critical applications for years, the Institute of Flight System Dynamics at TU München has gathered broad experience concerning MBD with MATLAB, Simulink, and Stateflow[1] (Hornauer and Holzapfel, 2011; Hornauer et al., 2013; Hochstrasser et al., 2016; Hochstrasser et al., 2017), especially in the context of DO-178C/DO-331 (RTCA, 2011a; RTCA, 2011c).

Thereby, a gap between process planning and its realization could be observed, contradicting the demanded high degree of automation in MBD.

Process engineering meta models like SPEM 2.0 (OMG Object Management Group, 2008) help to define a process, activities, and artefacts on a planning level (cf. (Gallina, 2014)), but engineers must manually identify the tasks to execute and how to execute them. If the planning documentation is too detailed, developers are overwhelmed by the details. If it is too abstract, the space for interpretation is too large. In both cases, deviations from the intended planning are very likely.

Furthermore, it is difficult to communicate a pro-

[1]Product of MathWorks Inc. for model-based simulation and software development

cess setup to developers in a way, in which they can efficiently apply it. In consequence, developers struggle with the tool complexity[2]. A process setup for MATLAB, Simulink, and Stateflow may contain a whole modeling environment with configuration settings and customizations (Hochstrasser et al., 2017; Estrada et al., 2013; Dillaber et al., 2010), but also more process-related information, like modeling rules, procedures for the analyses to execute and reviews to perform (cf. reference workflows addressed by (Marques et al., 2012; Potter, 2012; Erkinnen and Potter, 2009; Conrad et al., 2012; Conrad et al., 2009)).

Assessment of process compliance, consistency, and cleanliness of the software project, especially after changes, is still manual work and linked to enormous effort. Every tool has its own report format and justification workflow. Model reviews are intensively used by a majority of companies (Broy et al., 2014), but sophisticated tool support is missing.

Last but not least, the provided traceability in MBD approaches is too weak to perform serious impact analysis. Neither verification arefacts like reports are typically included, nor is it granular enough (Paz and El Boussaidi, 2016).

The proposed solution for these challenges is a *process-oriented build tool*, specifically designed for MBD in high-integrity applications. In general, build tools allow modeling of a dependency network of tasks and their execution in an ordered and optimized way (cf. (Humble and Farley, 2015; Berglund and McCullough, 2011; Sonatype Company, 2008)). Core idea and contribution of this paper is to enhance and adopt known basic build tool capabilities. The introduced term *process-oriented* emphasizes on the one hand the interaction with a traceability graph in contrast to task- and product-oriented tools (Humble and Farley, 2015). On the other hand, features are added to improve process compliance, guidance, and the resolution of the previously stated challenges in a safety-critical context.

Section 2 introduces the generic workflow supported by the tool, followed by a more detailed discussion of the tool capabilities in Section 3. Section 4 presents selected insights into traceability management and the underlying data model. Afterwards, Section 5 outlines an example implementation highlighting the reachable process coverage. In Section 6, existing tools and solutions are discussed and differences to the presented tool are revealed. Limitations and an outlook of future work are finally given in Section 7, before the results are summarized in Section 8.

The presented tool has originally been created to support a MBD workflow fulfilling objectives of DO-178C/DO-331 in accordance with the MathWorks reference workflow (The MathWorks Inc., 2016a) (Release 2016b). Anyway, it is customizable for every process in MATLAB and not limited to the use in a DO-178C process.

2 WORKFLOW OVERVIEW

Recent studies showed that build tools are well accepted, if they do not hinder usual work style, are customizable, and fit into workflows (Rahman et al., 2017). Thus, an important aspect of the presented tool is to accompany the whole workflow, but at the same time allow developers using existing, familiar routines.

Figure 1 depicts a typical MBD change workflow as it is often pursued. Involved are three roles, a Process Manager, a Developer, and an Assessor. Developer and Assessor may be the same person depending on the required independence of the process. Not considered for simplicity is independent testing.

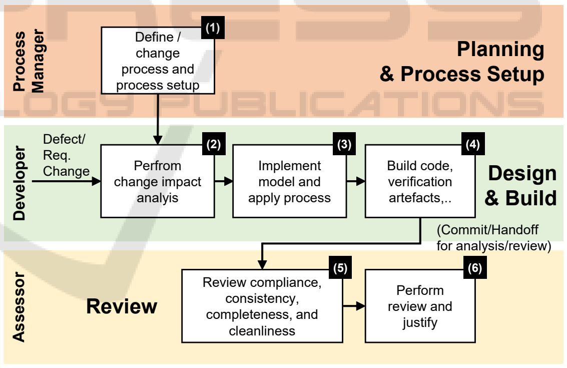

Process Manager
Define / change process and process setup
(1)
Planning & Process Setup
Developer
Defect/ Req. Change
Perfrom change impact analysis
(2)
Implement model and apply process
(3)
Build code, verification artefacts,..
(4)
Design & Build
(Commit/Handoff for analysis/review)
Assessor
Review
Review compliance, consistency, completeness, and cleanliness
(5)
Perform review and justify
(6)

Figure 1: Traditional change workflow.

Most processes start with an intensive planning phase, in which the Process Manager, who is responsible for the planning and tools, sets up and documents a tool chain, tool configuration, modeling rules, activities, and checklists to fulfill the objectives imposed by the respective development standard (1). The plans, standards, and setup are passed to a Developer, who reads and applies it during implementation (3). In case of a change, impact analysis may be performed prior to the actual modification of the design (2). As soon as the model has been updated, code generation and verification tasks are executed to create needed artefacts (4). In all cases, identification of impacted files is a manual and error-prone task, since

[2]Reith Graham, 2015, Adoption of Model-Driven Engineering in Small Workgroups and in Large Organisations, https://nmi.org.uk/wp-content/uploads/2015/06/MathWorks-Adoption-of-Model-Driven-Engineering.pdf [Accessed on 2017/27/10]

appropriate tools and traces are often missing. Finally, the module is handed over to an Assessor for analysis and review (5 and 6). Again, extensive manual work is required to check, if reports fit to the latest model version and checklists must be reevaluated step by step. Justifications are typically distributed over different reports or even embedded in the models, which often leads to problems with not matching revision numbers. All steps (1-6) require fully initialized models in a running MATLAB instance.

This very generic process is supported by the build tool in different ways and can be leveraged with a CI system as drafted in Figure 2. An environment to formalize and wrap procedures or checklists into a dynamic build workflow and to create rapid scaffolding plugins is provided to the Process Manager as further described in Section 3.1 (1). The formalized setup is passed to the Developer, who can integrate it into his software modules. After that, the tool exposes several automatic and semi-automatic methods to rapidly scaffold the software model according to standards and guidelines (3). This accelerates model creation and leads to models, which are compliant to standards by construction. Traceability information of a previous build can be accessed via a web interface and is useful to estimate the impact of a change in advance (2).

As soon as the design is finished, pre-defined build jobs (e.g., code generation or analysis jobs on model and code) can be executed incrementally based on modifications and their dependencies (4). Along with the build, the tool automatically creates and maintains detailed traceability. This close interaction between build steps and traceability is a key aspect of the tool, as it relieves developers from manually updating traceability information. The build not only supports fully automatic tasks, but can also be used to generate dynamic checklists, which benefit from incremental change detection and impact recognition in the background, too.

The changed software module is submitted to the version control system and a connected CI system automatically checks, if the minimum requirements for a successful build are fulfilled and the stored build information is up-to-date (5). If this is the case, a manual reviewer can access the build information via a web interface and review or justify the results without changing existing artefacts (6). Additional review information is also submitted and again checked by the CI system, whether it fulfills approval requirements (e.g., if all warnings are justified).

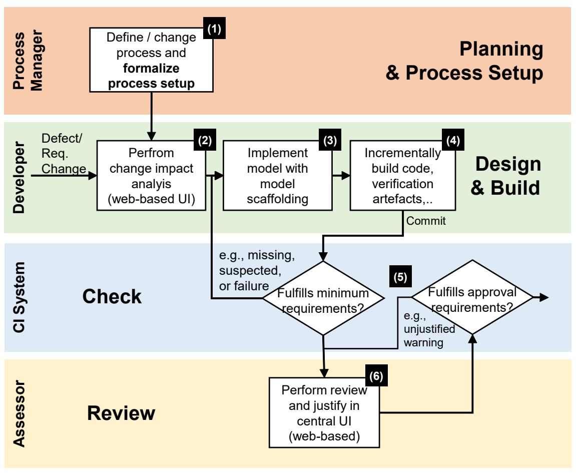

Process Manager
Define / change process and formalize process setup
(1)
Planning & Process Setup
Developer
Defect/ Req. Change
Perfrom change impact analyis (web-based UI)
(2)
Implement model with model scaffolding
(3)
Incrementally build code, verification artefacts,..
(4)
Design & Build
Commit
CI System
Check
e.g., missing, suspected, or failure
Fulfills minimum requirements?
(5)
Fulfills approval requirements?
e.g., unjustified warning
Assessor
Review
Perform review and justify in central UI (web-based)
(6)

Figure 2: Change workflow with build tool support and CI system.

3 SUPPORTING TOOL FEATURES

The previous section gave a brief overview of the process. In this section, the different phases are addressed in detail.

3.1 Planning

In the Planning Phase, the Process Manager transforms written rules and procedures into a so-called *Life Cycle Package*. The Life Cycle Package is a combination of XML specification, MATLAB classes, and resources of a full modeling environment (Hochstrasser et al., 2017). It provides the blueprint for model scaffolding (cf. Section 3.2.3) and the build workflow. Life Cycle Packages are separate repositories in the version control system and thus fully independent from the software modules managed by the Developer.

Software modules are self-contained parts of the software with artefacts passing the development and verification process together. Typically, different software modules are maintained by different developers. Here, any module at least consists of a single Simulink Projects[3] instance and a XML module description. The XML description is the basic information source of the module and defines a unique ID, a name of the module, other module dependencies that shall be loaded, and the Life Cycle Package that applies to

[3]Simulink Projects is a MathWorks toolbox for team collaboration and to manage models, https://nl.mathworks.com/discovery/simulink-projects.html [Accessed on 2017/07/28]

the module.

3.2 Design and Build

If a Life Cycle Package is linked to the software module, the Developer can use design and build functionality, which is organized along stages as depicted in Figure 3. These stages must be accessed sequentially, since, in contrast to common source code build tools, MATLAB and Simulink require initialized data, objects, and further artefacts in memory. The next sections traverse along the build workflow of Figure 3.

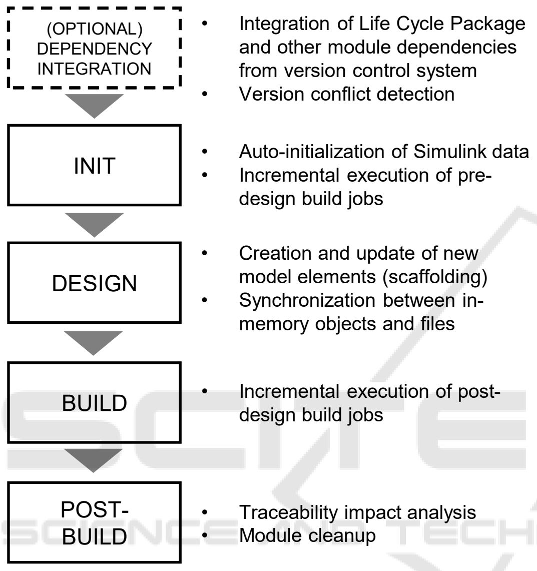

(OPTIONAL) DEPENDENCY INTEGRATION
• Integration of Life Cycle Package and other module dependencies from version control system
• Version conflict detection
INIT
• Auto-initialization of Simulink data
• Incremental execution of pre-design build jobs
DESIGN
• Creation and update of new model elements (scaffolding)
• Synchronization between in-memory objects and files
BUILD
• Incremental execution of post-design build jobs
POST-BUILD
• Traceability impact analysis
• Module cleanup

Figure 3: Fix workflow stages and provided functionality.

3.2.1 Life Cycle Package and Dependency Integration Stage

Prior to any implementation work, the tool supports integration of other modules and automatic detection of nested modules, so-called transitive dependency resolution. Thereby, version conflicts are detected and reported. Depending on the Life Cycle Package configuration, integrated dependencies may be initialized differently than the current working module, for example with regard to the set search path[4].

3.2.2 INIT Stage

The INIT stage loads the Life Cycle Packages, sets up search paths and auto-initializes all captured artefacts of the current and linked modules automatically. Build jobs, which must be executed prior to design stage can be hooked into this stage, too (cf. Section 3.2.4). This is beneficial, if, for example, the software environment shall be checked or customizations shall be registered in Simulink in advance and the result shall be documented.

3.2.3 DESIGN Stage

In the DESIGN stage, the Developer actually changes the model, simulation test cases, or requirement linking. The workflows are basically unconstrained in this stage, but support is given with the so-called *model scaffolding*.

Many tasks, like writing initialization scripts or saving objects as files, are time-consuming, error-prone, and regularly repeated by every developer. In addition, extensive modeling rules are most often handed over as sole textual descriptions, in the best case in combination with model checks that can evaluate, whether the rules have been followed or not[5]. But finding the textual guidelines of interest for the various configurations that the MATLAB and Simulink environment provides is cumbersome, and iterating to a compliant model using model checks is a reverse and time-wasting workflow.

A related problem appeared some years ago in the web development domain, where every developer built his web platform from the scratch. This led to significant overhead by repeating tasks and security issues due to bad configurations. In 2004, David Heinemeier Hansson presented "Ruby on Rails" (Ruby, 2016), a web development framework based on the principles "Don't Repeat Yourself" and "Convention over Configuration (CoC)". The idea behind the principle CoC is reducing a set of configurations to a smaller set of conventions. In most cases, it is sufficient for developers to learn the conventions without caring about the configuration behind. "The Convention over Configuration pattern rewards developers for adhering to naming conventions and enforces this in a stricter manner by building it into the framework"[6].

[4]The search path specifies a set of folders where MATLAB looks for files, classes, functions, and so on. Typically it is updated once during initialization of a model, but in some cases it is reasonable to update paths during build. The build tool provides features to do this in a controlled way and in connection with the build workflow.

[5]Jaffry, David, 2014, http://de.mathworks.com/company/ newsletters/articles/best-practices-for-implementing-modeling-guidelines-in-simulink.html [Accessed on 2017/27/07]

[6]Chen, Nicolas, 2006, http://softwareengineering.vazexqi.com/files/pattern.html [Accessed on 2017/12/07]

The build tool adopts this pattern to reduce manual configuration work with model scaffolding provided to the user as command-line tool. To define conventions, the Process Manager customizes pre-defined abstract classes and formalizes information about folder structure, modeling elements, and tool settings. The Developer is then supported in

- creating new software modules, their structure, and configured Simulink Projects as defined by guidelines,
- initialization of data from files (auto-initialization without custom init scripts),
- creating models and related data in compliance with guidelines and standards, as well as removing them, and
- synchronization between initialization scripts and in-memory data of Simulink (e.g., buses in Simulink workspaces).

For example, the Process Manager defines the configuration for a special type of Simulink Bus, which is imported from existing C header files and thus does not have to be auto-generated. The type of bus is registered in the build tool Life Cycle Package as `bus-imported` together with the information that any object name shall follow the expression `<module-id>_<busname>_Bus`. As a consequence, the tool automatically assumes that initialization scripts are stored in a folder called `bus/imported` with the filename pattern `<module-id>_<busname>_Bus_init.m` and registers helpful commands, like

```
create bus-imported <busname>
```

to create a fully configured parameter in the Simulink workspace, together with an initialization script.

The scaffolding interface imposes restrictions on the user, like storing files at a specific location or using pre-defined formats, but all modules created this way have the same look and feel and already comply with standards by construction for the most part.

3.2.4 BUILD Stage

After finishing the design change, the Developer enters the BUILD stage to generate code, run analysis, or generate review checklists.

The sequence of build jobs and their dependencies must have been defined within the Life Cycle Package XML specification in advance. Build jobs are derived from an abstract base class handling a wide range of operations silently in the background. They can be executed iteratively for a set of artefacts resulting from a user-defined, programmable selection rule. E.g., a static model analysis can be executed for every model in the software module.

Every job iteration is executed incrementally, meaning that inputs and outputs are checked, whether they are deprecated. Therefore, meta information of input and output artefacts is stored after each execution and compared to the currently available set of artefacts at a later point of time.

The build tool organizes jobs in an acyclic, directed dependency graph. This allows the user to define dependencies between jobs. Dependent jobs are checked and re-executed prior to the called job in an organized manner derived from the execution graph.

Beyond these functionalities, the interaction with MATLAB during the execution must be emphasized. A job can automatically record the command line output and document thrown exceptions as well as warnings during build.

3.2.5 POST-BUILD Stage

The POST-BUILD stage presupposes, that all build jobs have been executed at least once and traceability has been established as far as possible. Basic intention of the POST-BUILD is to compose a single, holistic artefact traceability tree and store it for future impact analysis.

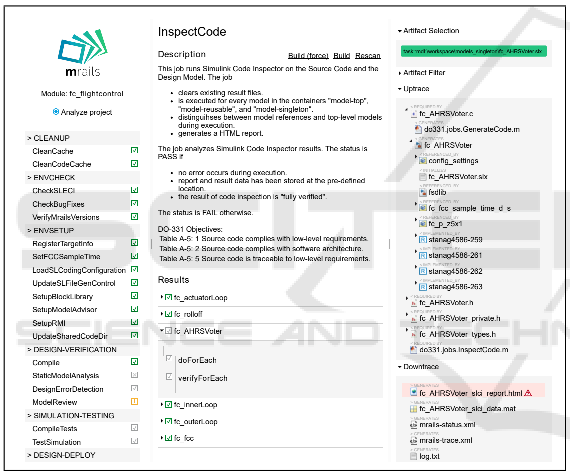
Besides, another addressed problem are dirty software projects. Orphaned files and objects are often the result of modifications, renaming, or remaining intermediate artefacts of the implementation. If projects are not regularly cleaned up, unneeded files accumulate and the question, whether a file is unnecessary and can be deleted or not, becomes harder and harder to answer. Example applications with the build tool showed that it is possible to generate a holistic traceability graph containing meta data for all files and objects that play any role in the module. Artefacts not appearing in this graph are suspected to have no functionality. With the *module cleanup* feature, the Developer can automatically identify orphaned files and objects and delete them. This provides the possibility to evaluate and enforce cleanliness.

3.3 Check and Review

The build tool can assess the status of a software module. Such a scan is "silent", it does not modify any artefact of the module, but detects changes and outdated artefacts. The returned result is structured and can easily be post-processed into the required format, e.g., for CI Systems to accept or deny a commit into version control.

The status of the build, job dependencies and the whole traceability tree is stored in XML files and is fully accessible without MATLAB and Simulink. This independence may be the foundation for online review processes, fully performed in the browser.

To display the results and go a first step into this direction, a web-based graphical user interface has been created on top of the build tool as shown in Figure 4. It can directly interact with MATLAB and fetch the latest status, but can also just read previously updated status and traceability information. On the left-hand side of the view in Figure 4, the status of the build workflow is displayed for each phase and job. Icons indicate the status. By clicking on the jobs, further details are exposed on the middle panel. Beneath a general description, results are displayed for each iteration of the job (task). The tool bar on the right shows the dependencies and impact of the selected task. In the given example, the selected task `fc_AHRSVoter` is deprecated, since the output report does not match the underlying model anymore.


mrails

Module: 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 — Build (force) Build Rescan

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

Artifact Selection

task::mdl:\workspace\models_singleton\fc_AHRSVoter.slx

Artifact Filter

Uptrace

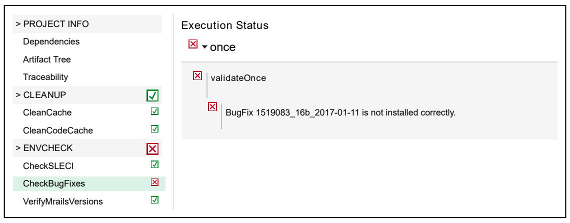
- < REQUIRED BY fc_AHRSVoter.c
  - < GENERATES do331.jobs.GenerateCode.m
  - < GENERATES fc_AHRSVoter
    - < REFERENCED_BY config_settings
    - < INITIALIZES fc_AHRSVoter.slx
    - < REFERENCED_BY fsdlib
    - < REFERENCED_BY fc_fcc_sample_time_d_s
    - < REFERENCED_BY fc_p_z5x1
    - < IMPLEMENTED_BY stanag4586-259
    - < IMPLEMENTED_BY stanag4586-261
    - < IMPLEMENTED_BY stanag4586-262
    - < IMPLEMENTED_BY stanag4586-263
- < REQUIRED BY fc_AHRSVoter.h
- < REQUIRED BY fc_AHRSVoter_private.h
- < REQUIRED BY fc_AHRSVoter_types.h
- < REQUIRED BY do331.jobs.InspectCode.m

Downtrace

- > GENERATES fc_AHRSVoter_slci_report.html ⚠
- > GENERATES fc_AHRSVoter_slci_data.mat
- > GENERATES mrails-status.xml
- > GENERATES mrails-trace.xml
- > GENERATES log.txt

Figure 4: Graphical user interface of the build tool.

The user can drill into any job and get further information about the failure state or the nested status structure. In Figure 5, for example, a missing bug fix on the host system has been detected during build. In this case, it is classified as non-justifiable error and may cause the CI system to deny a commit.


- > PROJECT INFO
  - Dependencies
  - Artifact Tree
  - Traceability
- > CLEANUP ✓
  - CleanCache ✓
  - CleanCodeCache ✓
- > ENVCHECK ☒
  - CheckSLECI ✓
  - CheckBugFixes ☒
  - VerifyMrailsVersions ✓

Execution Status

- ☒ once
  - ☒ validateOnce
    - ☒ BugFix 1519083_16b_2017-01-11 is not installed correctly.

Figure 5: Detailed status information on an occurred build error classified as non-justifiable.

In contrast to many common build tools, more effort has been spent on handling the return status information of jobs. The intention is to return not just a pass or fail state, but to keep the result customizable. Ideally, the output of any other tool called by the job is translated into the provided format. This way, the build tool and its GUI become a unique place to review the complete status of the project.

Therefore, the tool provides a status object, which holds a general state (Figure 6) and detailed formatted descriptions. Furthermore, the user can specify whether the state can be mitigated by justification or not, and to which target state. By nesting status objects, almost any output structure can be rebuilt. The status object also indicates, whether a job is suspected or not and aggregates status and the suspected state through the whole status hierarchy.

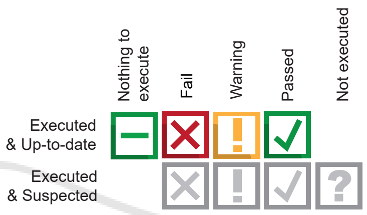
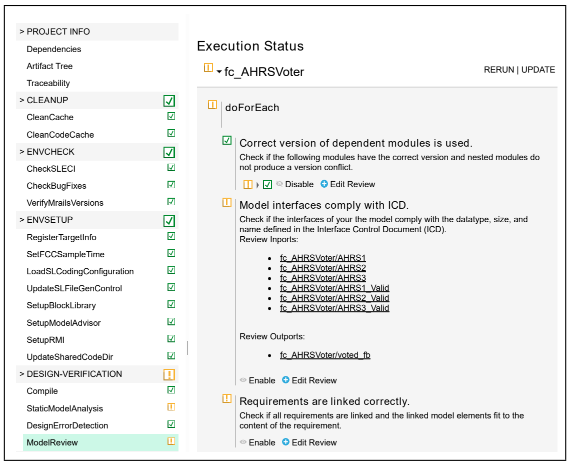


Figure 6: Basic execution status types.

The status objects can also be used to create interactive and dynamic check lists. Since the detailed description can be formatted with plain HTML, and the job execution has full access to all MATLAB, Simulink, and Stateflow APIs, tailored and dynamic review lists can be generated upon model information.

In Figure 5, a non-justifiable status information has been introduced. Alternatively, Figure 7 shows an exemplary interactive model review checklist, embedded in the build workflow. In a first run, checklist items are marked with the status WARN. By manually ticking items and/or adding a justification text, the status can be mitigated to PASS or any other status according to a specified mitigation rule. The checklist in Figure 7 is dynamic and supports incremental deprecation. The item "Model interfaces comply with ICD" directly lists all relevant Inport and Outport model elements to be reviewed with a direct link into the model elements.

The advantages of the tight integration with the build tool are obvious. At first, checklists can be created iteratively for a set of artefacts (see also Section 3.2.4). The Assessor can go through the checklist of every model step by step and check the items. Like jobs, reviews can deprecate or get under suspicion if upstream dependencies change. The user gets a seamless review interface containing both job results and dynamic checklists in the end.



> PROJECT INFO
Dependencies
Artifact Tree
Traceability
> CLEANUP ✓
CleanCache ✓
CleanCodeCache ✓
> ENVCHECK ✓
CheckSLECI ✓
CheckBugFixes ✓
VerifyMrailsVersions ✓
> ENVSETUP ✓
RegisterTargetInfo ✓
SetFCCSampleTime ✓
LoadSLCodingConfiguration ✓
UpdateSLFileGenControl ✓
SetupBlockLibrary ✓
SetupModelAdvisor ✓
SetupRMI ✓
UpdateSharedCodeDir ✓
> DESIGN-VERIFICATION !
Compile ✓
StaticModelAnalysis !
DesignErrorDetection ✓
ModelReview !

Execution Status

! fc_AHRSVoter RERUN | UPDATE

! doForEach

✓ Correct version of dependent modules is used.
Check if the following modules have the correct version and nested modules do not produce a version conflict.
! ✓ Disable Edit Review

! Model interfaces comply with ICD.
Check if the interfaces of your the model comply with the datatype, size, and name defined in the Interface Control Document (ICD).
Review Inports:

- fc_AHRSVoter/AHRS1
- fc_AHRSVoter/AHRS2
- fc_AHRSVoter/AHRS3
- fc_AHRSVoter/AHRS1_Valid
- fc_AHRSVoter/AHRS2_Valid
- fc_AHRSVoter/AHRS3_Valid

Review Outports:

- fc_AHRSVoter/voted_fb

Enable Edit Review

! Requirements are linked correctly.
Check if all requirements are linked and the linked model elements fit to the content of the requirement.
Enable Edit Review

Figure 7: Exemplary model review checklist.

4 TECHNICAL CONCEPTS

4.1 Traceability

The traceability graph mentioned in the previous paragraph must be captured and maintained.

Traceability exists with various characteristics and for different purposes. In its most general definition, a trace is a triplet of elements. A source and target artefact as well as a trace link in-between (Gotel et al., 2012).

The main requirement for the traceability graph maintained by the build tool is to keep it adaptable to any custom Traceability Information Model (TIM, (Mäder et al., 2013)). Thus the tool only provides the infrastructure to store, relate, update, and query meta data of artefacts in a directed graph. The data model itself is further discussed in Section 4.2.

With regard to this flexibility, a trace can be defined between any type of software artefact, from informal to formal (cf. (Seibel et al., 2012)), in-memory, or on the file system. Also the type of link, by (Seibel et al., 2012) and (Lochmann and Hessellund, 2009) categorized under hard (explicit) references, soft (implicit) references, and semantic connections, is unrestricted.

(Asuncion et al., 2010) distinguish between *prospective* and *retrospective* approaches. Retrospective traceability approaches derive traces from a set of static software artefacts. Prospective methods update traces during artefact generation or modification. They are tightly connected to a development environment or techniques (Seibel et al., 2012), like a model transformation into source code. The method here represents a prospective approach, since traces are modified *in situ* with the build process.

The methods chosen to create traces are rule-based techniques. They provide the most flexibility and can cope with the heterogeneity of artefacts (Zisman, 2012). Furthermore, these methods support the definition of *candidate traces*, which are traces that finally must exist but are not created yet. Rules are either defined when setting up the model scaffolding interface or with the specification of the job. In the latter case, the rules describe traceability creation, but at the same time define inputs and outputs for the build. This double function is one of the core principles, since it reduces configuration effort. Traceability and the build process benefit from each other.

Figure 8 illustrates the interaction between the build process and the traceability graphs. Every build run starts with a copy of the design tree, which is the basis for the so-called execution tree. The design tree is the output of model scaffolding. For every job, a selection of artefacts to iterate on is determined. The rules for the required inputs and expected outputs are evaluated and created artefacts pushed into the job tree. Then, the core function of the job ("do") is executed. After that, a recovery rule is optionally called to get traces for artefacts, which are not easily predictable beforehand (like the output of the translation to source code).

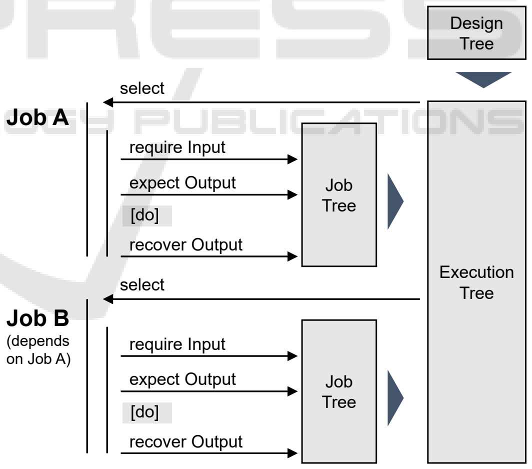


Figure 8: Interaction between build jobs and traceability trees.

The job trees are serialized, cached, and stored, which allows independent and incremental job execution. The separate treatment also increases robustness. In contrast, it comes along with larger memory requirements. A full traceability tree is only available in the end, when all job trees are merged to a single tree (cf. Section 3.2.5).

The build tool itself provides the infrastructure to capture traces. The algorithms for recovering or establishing traces are implemented when setting up the

Life Cycle Package. For the application example presented later on, various convenient APIs have been be combined to obtain traces. Embedded Coder generates traces between model and source code, Simulink provides built-in methods to identify variable dependencies in models, and Simulink Projects offers file dependency analysis. However, also every non-MATLAB based algorithm can be included.

4.2 Data Model

To efficiently handle artefact meta data and traceability graphs, an optimised data model is required. Especially the repeated calculation of revisions and the high number of artifact trees must remain manageable. The implementation consists of three core elements as shown in the UML diagram of Figure 9: An Artefact Pool, multiple Artefact Trees, and Meta-Artefacts.

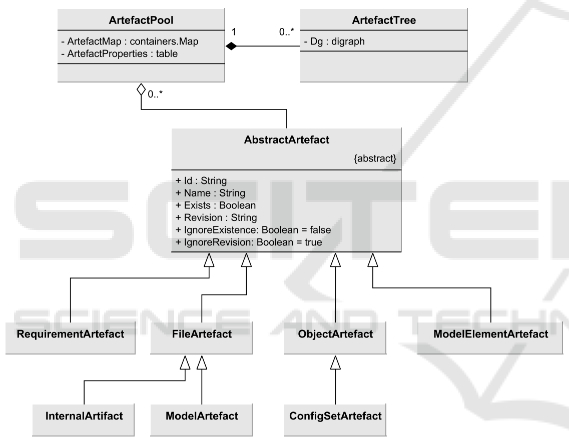


Figure 9: UML class diagram of data model (simplified).

Meta-Artefacts are the core element. They store (at minimum) a unique ID, a name, information about existence, and the current revision. A Meta-Artefact is not necessarily bound to a tangible software artefact, but can be created for every element whose existence and revision can be identified. For example, Meta-Artefacts may be created for different kinds of files, like reports, scripts, or code, but also for objects in the memory of MATLAB, like Simulink Buses, Parameters, and even for model elements. Meta-Artefacts do not need to relate to an existing artefact, they can also represent a so-called candidate artefact, which is expected to be generated after job execution. A build scan, for example, can create such candidate artefacts and show the user in advance, what is missing. Figure 9 gives an overview of the currently used types of artefacts, which have a differing implementation to evaluate existence and revision.

All artefacts are managed in Artefact Pools, which are collections of artefacts without relations between artefacts. Main function of the pool is to centralize access and checking for deprecation. Artifact Pools ensure that artefact meta data is just computed once per run (unless explicitly enforced otherwise).

Artefacts plus their trace links are organized in Artefact Trees. Each Artefact Tree consists of a set of connected or unconnected directed graphs. It provides efficient methods to query artefacts with customized selectors, traverse the graphs, as well as serialize, read, or reduce them to a Traceability Information Model. Important is that Artefact Trees only organize the relations and do not store the Meta-Artefacts. This keeps the graph algorithms slim and efficient.

5 PROCESS COVERAGE EXAMPLE

This section presents parts of an example application to show the necessity and capabilities of the presented build tool approach. It covers activities of a DO-178C/DO-331 process using a so-called Design Model (Simulink and Stateflow model) to replace DO-178C Software Low-Level Requirements and Software Architecture. The mapping of activities and tools is as proposed in the Model-Based Design Workflow for DO-178C (The MathWorks Inc., 2016a). The data bases on the DO-178C case study performed by Bill Potter for an helicopter autopilot[7].

Three build phases are hooked into the INIT stage, being executed before the developer starts the design. They can be found in Figure 11 on the left-hand side. The CLEANUP phase clears caches, the ENVCHECK phase analyses the software environment and verifies that required bug fixes are installed. The ENVSETUP updates Simulink default settings and loads custom checks, libraries, or code.

In the DESIGN stage, scaffolding methods are provided for the following model data: Simulink Buses (exported and imported), Simulink Parameters (constants and data-items), Simulink Signals (global, and model scope), Simulink Models (top-level, reusable, non-reusable), Libraries, and Enums.

After entering the BUILD stage, the phases and jobs listed in Table 1 are available to the Developer and Assessor, covering various DO-331 objectives. The DESIGN-VERIFICATION phase starts with a compile, which is also used to determine dependencies between models and model data. Subsequent

[7]https://nl.mathworks.com/matlabcentral/fileexchange/56056-do178-case-study

phases analyse the model for standard compliance, runtime errors, compliance to requirements, or coverage. In the next phases, the model is transformed to C-Code and verified with traditional software analysis afterwards. Table 1 also indicates, how the traceability graph is updated using inputs and outputs of the jobs. Execution of the build assembles traces as superficially described by the TIM in Figure 10.

Table 1: Example build workflow. Each job creates traces to required inputs (R), expected outputs (E), or recovered outputs (RC)). Numbers behind job name indicate dependent jobs, which must be executed beforehand.

Build Workflow	R	E	RC
Design Verification			
(1) Compile	x		
(2) Model Compliance Checks (1)	x	x	
(3) Design Error Detection (1)	x	x	
(4) Model Review (1)	x		
Simulation Testing			
(5) Compile Tests	x		
(6) Simulation Tests (5)	x	x	
Design Deployment			
(7) Unprotected Deployment (1)	x	x	
(8) Protected Deployment (7)	x	x	
Code Generation			
(9) Shared Code Gen. (1)	x	x	x
(10) Code Generation (9)	x	x	x
Code Verification			
(11) Auto C. Review (10)	x	x	
(12) Code Compliance Checks (10)	x	x	
(13) Code Defect Analysis (10)	x	x	

The final workflow and the composed, holistic traceability tree is plotted in Figure 11. The graph is arranged using hierarchical edge bundling (Holten, 2006), which positions artefacts depending on their type in a circle. For this relatively small model and the simplified workflow, the traceability tree consists of 237 artefacts and 592 traces. The huge amount of dependencies gives an impression of the effort to maintain consistency between artefacts in a safety critical process. It can barely be managed by hand and substantiates the need for an automated solution as presented in this research.

6 RELATED WORK AND TOOLS

This section gives an overview of existing approaches and related tools and works out the differences. For the separate disciplines combined in the presented tool, a variety of alternatives could be listed providing support in build automation, traceability, review, and scaffolding. However, they rarely integrate with MATLAB, Simulink, and Stateflow in the depth as the presented tool does, and no tool provides a similar seamless approach. In the following, some tools with the largest overlap are discussed.

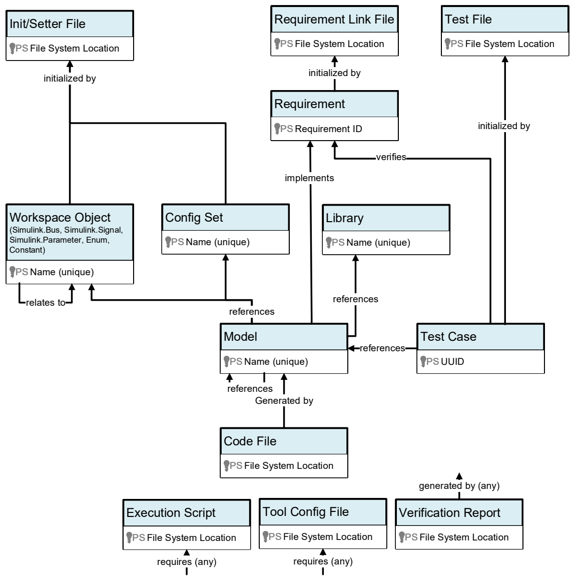


Figure 10: Example Traceability Information Model (simplified).

Traceability in the context of MBD is treated in various research publications. Tools like Reqtify[8] or Yakindu Traceability[9] provide powerful traceability features, but are not integrated with build automation and mainly focus on retrospective traceability capture.

Build tools like Apache Ant[10], Apache Maven[11], or Gradle[12] are known among developers for years. They can evaluate the up-to-dateness of jobs, incrementally rerun jobs, and manage transitive software dependencies. However, the traditional areas of application of these tools are Java and C/C++ builds. Model-based design, which comes along with new challenges like inherent dependencies (Seibel et al., 2010), is rarely supported. Furthermore, a strong binding and interaction with modeling and verification tools is necessary in the presented use case. The interaction with a multi-functional traceability graph as

[8]Traceability management tool of Dassault Systems, https://www.3ds.com/products-services/catia/products/reqtify/ [Accessed on 2017/03/08]

[9]Traceability management solution of itemis AG, https://www.itemis.com/en/yakindu/traceability/ [Accessed on 2017/03/08]

[10]Apache Ant is a Java library to setup build processes, http://ant.apache.org [Accessed on 2017/27/07]

[11]Apache Maven is a project management and comprehension tool, https://maven.apache.org, [Accessed on 2017/07/27]

[12]Gradle is a build management and automation tool, https://gradle.org [Accessed on 2017/07/27]

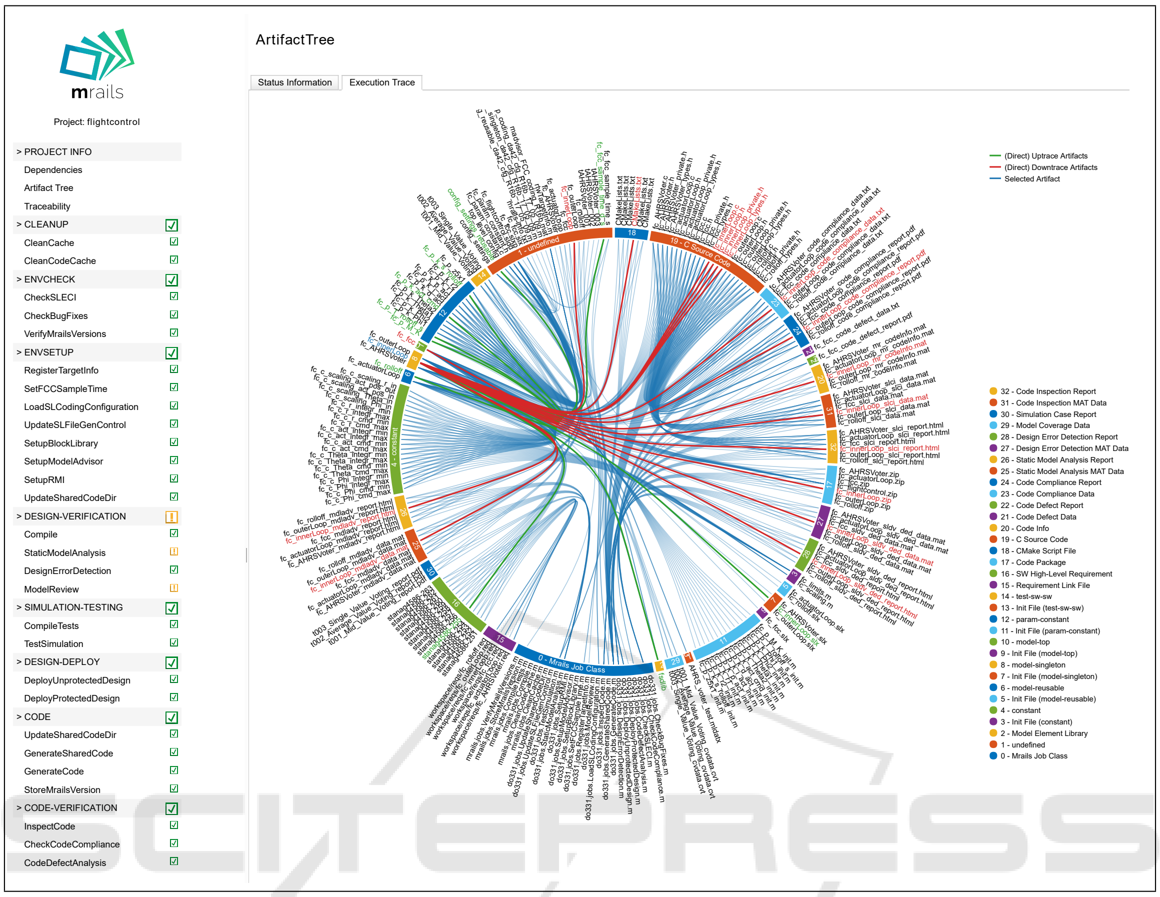


Figure 11: Full artefact traceability graph.

well as the review and justification capabilities are further features not supported by these tools. As a consequence, it has been a dedicated design decision to implement a separate build tool in MATLAB.

Beyond build tools, the web platform Models Refinery (Modelery) (Couto et al., 2014) promotes a collaborative, web-based repository for sharing, versioning, and organizing artefacts of a MBD process. It does not analyse generated traceability, dependencies or evaluate up-to-dateness artefacts.

With Simulink Projects, various process-related features have found their way directly into MATLAB and Simulink, amongst others a closer link to version control systems, file dependency analyses, or template features (Mahapatra et al., 2012). However, these features are mostly standalone, isolated from a workflow, and only provide a subset of the functionality used in the demonstrated tool. Of course, any build structure can be implemented by investing time and money, but an infrastructure as provided by the presented tool is not available out of the box.

7 LIMITATIONS AND FUTURE WORK

Different assumptions have been necessary to implement the tool in the scope of the research project. Especially the functionality of Simulink and Stateflow has to be limited to a reasonable subset due to its feature-richness. Chosen are the features supported by Simulink Code Inspector (cf. (The MathWorks Inc., 2016b)). Furthermore, the presented build tool bases on some design decisions. For example, it assumes the usage of data dictionaries instead of the base or model workspace.

Particularly in the context of safety-critical applications, tool qualification, e.g., as described in DO-330 (RTCA, 2011b), is an important corner-stone for safety. If a tool eliminates, automates, or reduces a required process, and the output itself is not verified by a qualified tool or manual inspection, the user has to perform a tool qualification (Rierson, 2013). In production, also the presented tool would require tool qualification in most cases. Since the tool is a research project, tool qualification has not been focused up to now, but a tool qualification kit may be developed on top later on.

Next step is the application of the tool on a large multi-module project to evaluate its scalability. Also other process workflows beyond DO-178C should be integrated in subsequent projects to prove general applicability. Future research may also focus on a stronger interaction with MATLAB, Simulink, and Stateflow. Instead of manually triggering a rescan of the project, callbacks might be used to automatically update the status of jobs in the background *in situ*, whenever a model or object has been changed.

8 CONCLUSIONS

The paper has introduced a build tool for MBD, which seamlessly integrates process-oriented features, like model scaffolding based on CoC, to improve standard compliance and standardisation from the beginning. Additionally, a unique interface for automatic, semi-automatic, and manual tasks, including review and justification capabilities, as well as traceability capturing in the background has been presented.

A short example has demonstrated the necessity for build automation on the one hand, and the high process coverage, which is achievable by integrating various tools into the build and parsing the individual results, on the other hand. The discussion of related work has highlighted the difference to existing solutions. Finally, a summary of limitations coming along with the build tool and future enhancement plans has been provided.

The presented tool has the ability to solve problems of different process participants. With the Life Cycle Package, the Process Manager is able to distribute workflows, standards, checklists, and procedures in a new level of detail. Model scaffolding improves the structure and clarity of software modules. Developers must read less documents and a much higher standard compliance is already expected by construction. Additionally, less time is invested in reviews, since checklists must only be re-reviewed if the tool detects deprecation (in contrast to written checklists). Traceability does not have to be created retrospectively or manually, but is implicitly captured during build. This eliminates deprecated traceability links. Various methods to evaluate completeness of artefacts, compliance to standards, consistency, and cleanliness support assessors in their daily work or are pluggable to other build automation systems, like a CI server.

With the presented tool, higher process compliance is achieved whilst investing less effort. It aligns with the trends of modern software development and has the potential to support increasing software complexity and size, but also to accelerate development cycles in high-integrity MBD.

REFERENCES

Asuncion, H. U., Asuncion, A. U., and Taylor, R. N. (2010). Software traceability with topic modeling. In Kramer, J., Bishop, J., Devanbu, P., and Uchitel, S., editors, *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, page 95, New York, New York, USA. ACM Press.

Basagiannis, S. (2016). 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)*, pages 1–6. IEEE.

Berglund, T. and McCullough, M. (2011). *Building and testing with Gradle*. O'Reilly, Beijing and Sebastopol.

Bhatt, D., Madl, G., Oglesby, D., and Schloegel, K. (2010). Towards Scalable Verification of Commercial Avionics Software. In AIAA, editor, *Proc. of AIAA Infotech@Aerospace 2010*.

Broy, M., Kirstan, S., Krcmar, H., and Schätz, B. (2014). What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry? In Management Association, I. R., editor, *Software Design and Development*, pages 310–334. IGI Global.

Conrad, M., Englehart, M., Erkkinen, T., Lin, X., Nirakh, A. R., Potter, B., Shankar, J., Szpak, P., Yan, J., and Clark, J. (2012). Automating Code Reviews with Simulink Code Inspector. In Dagstuhl, editor, *Proc. of VIII Dagstuhl-Workshop*, München. fortiss GmbH.

Conrad, M., Friedman, J., and Sandmann, G. (2009). Verification and Validation According to IEC 61508: A Workflow to Facilitate the Development of High-Integrity Applications. *SAE International Journal of Commercial Vehicles*, 2(2):272–279.

Couto, R., Ribeiro, A. N., and Campos, J. C. (2014). The Modelery: A Collaborative Web Based Repository. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Kobsa, A., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Terzopoulos, D., Tygar, D., Weikum, G., Murgante, B., Misra, S., Rocha, A. M. A. C., Torre, C., Rocha, J. G., Falcão, M. I., Taniar, D., Apduhan, B. O., and Gervasi, O., editors, *Computational Science and Its Applications – ICCSA 2014*, volume 8584 of *Lecture notes in computer science*, pages 1–16. Springer International Publishing, Cham.

Dillaber, E., Kendrick, L., Jin, W., and Reddy, V., editors (2010). *Pragmatic Strategies for Adopting Model-Based Design for Embedded Applications*. SAE International.

Erkinnen, T. and Potter, B. (2009). *Model-Based Design for DO-178B with Qualified Tools: AIAA Modeling and Simulation Technologies Conference and Exhibit*. American Institute of Aeronautics and Astronautics Inc, Hyatt Regency McCormick Place, Chicago Illinois.

Estrada, R. G., Sasaki, G., and Dillaber, E. (2013). *Best practices for developing DO-178 compliant software using Model-Based Design*. AIAA Infotech@Aerospace, Boston, MA.

Gallina, B. (2014). A Model-Driven Safety Certification Method for Process Compliance. In *Proc. of 2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 204–209. IEEE.

Gotel, O., Cleland-Huang, J., Hayes Huffman, J., Zisman, A., Egyed, A., Grünbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J., and Mäder, P. (2012). Traceability Fundamentals. In Cleland-Huang, J., Gotel, O., and Zisman, A., editors, *Software and Systems Traceability*, pages 3–22. Springer London, London.

Hochstrasser, M., Hornauer, M., and Holzapfel, F. (05 Oct. 2016). Formal Verification of Flight Control Applications along a Model-Based Development Process: A Case Study.

Hochstrasser, M., Schatz, S. P., Nürnberger, K., Hornauer, M., Myschik, S., and Holzapfel, F., editors (2017). *Aspects of a Consistent Modeling Environment for DO-331 Design Model Development of Flight Control Algorithms*.

Holten, D. (2006). Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Transactions on Visualization and Computer Graphics*, Volume 12:p. 741–748.

Hornauer, M. and Holzapfel, F. (2011). Model Based Testing for CS-23 Avionic and UAV Applications: DGLR Workshop 2011.

Hornauer, M., Schuck, F., and Holzapfel, F. (2013). Wechselwirkungen zwischen GNC Algorithmus und Software.

Humble, J. and Farley, D. (2015). *Continuous delivery: Reliable software releases through build, test, and deployment automation*. A Martin Fowler Signature Book. Addison-Wesley, Upper Saddle River, NJ, tenth printing edition.

Lochmann, H. and Hessellund, A. (2009). An integrated view on modeling with multi domain-specific languages. In *Proc. of the IASTED International Conference Software Engineering*.

Mäder, P., Jones, P. L., Zhang, Y., and Cleland-Huang, J. (2013). Strategic Traceability for Safety-Critical Projects. *IEEE Software*, 30(3):58–66.

Mahapatra, S., Ghidella, J., and Walker, G. (2012). Team-Based Collaboration in Model-Based Design. In *AIAA Modeling and Simulation Technologies Conference*, Reston, Virigina. American Institute of Aeronautics and Astronautics.

Marques, J. C., Yelisetty, S. M. H., Dias, L. A. V., and da Cunha, A. M. (2012). 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*, pages 431–436. IEEE.

OMG Object Management Group (Apr. 2008). Software & System Process Engineering Meta-Models Specification (SPEM 2.0).

Paz, A. and El Boussaidi, G. (2016). 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)*, pages 229–236. IEEE.

Potter, B. (2012). Complying with DO-178C and DO-331 using Model-Based Design.

Rahman, A., Partho, A., Meder, D., and Williams, L. (2017). Which Factors Influence Practitioners' Usage of Build Automation Tools? In *Proc. of 2017 IEEE/ACM 3rd International Workshop on Rapid Continuous Software Engineering (RCoSE)*, pages 20–26. IEEE.

Rierson, L. (2013). *Developing safety-critical software: A practical guide for aviation software and DO-178C compliance*. CRC Press LLC, Boca Raton, FL.

RTCA (2011a). DO-178C - Software Considerations in Airborne Systems and Equipment Certification.

RTCA (2011b). DO-330 Software Tool Qualification Considerations.

RTCA (2011c). DO-331 - Model-Based Development and Verification Supplement to DO-178C and DO-278A.

Ruby, S. (2016). *Agile Web Development with Rails 5*. Pragmatic Bookshelf.

Seibel, A., Hebig, R., and Giese, H. (2012). Traceability in Model-Driven Engineering: Efficient and Scalable Traceability Maintenance. In Cleland-Huang, J., Gotel, O., and Zisman, A., editors, *Software and Systems Traceability*, pages 215–240. Springer London, London.

Seibel, A., Neumann, S., and Giese, H. (2010). Dynamic hierarchical mega models: Comprehensive traceability and its efficient maintenance. *Software & Systems Modeling*, 9(4):493–528.

Sonatype Company (Oct. 2008). *Maven: The Definitive Guide*. O'Reilly, Cambridge.

The MathWorks Inc. (2016a). *DO Qualification R2016b: Model-Based Design Workflow for DO-178C*. Natick, MA, USA.

The MathWorks Inc. (Sep. 2016b). *Simulink Code Inspector Reference: R2016b*. Natick, MA, USA.

Zisman, A. (2012). Using Rules for Traceability Creation. In Cleland-Huang, J., Gotel, O., and Zisman, A., editors, *Software and Systems Traceability*, pages 147–170. Springer London, London.