A modular, model-based DO-178C software life cycle – planning, realization, and preservation

München, 10th of October, 2018

Markus Hochstrasser, Stephan Myschik and Florian Holzpfel
Agenda

• Introduction
• Modular, model-Based DO-178C development process for agile workflows
• Process-oriented build tool for model-based design for DO-178C
• Conclusion
Introduction

- Institute of Flight System Dynamics uses **model-based software development** with MATLAB, Simulink, and Stateflow to develop flight control algorithms

- Application to small and large vehicles (e.g. safety-critical software for DA42 flying testbed)

- „Certification-worthy“ software compliant with DO-178C / DO-331
Can a DO-178C process be agile?

Agile principles
1. Customer satisfaction and continuous delivery
2. Welcome changing requirements
3. Deliver working software frequently
4. Business people and developers constantly work together
5. Projects around motivated individuals
6. Face-to-face communication
7. Working software is the primary measure of success
8. Sustainable processes with constant pace
9. Technical excellence and good quality
10. Simplicity
11. Self-organizing teams
12. Regular retroperspectives

Objective

• **DO-178C/DO-331 objectives shall be achieved by applying agile software development principles**
  
  – What is the framework DO-178C gives us to introduce agile workflows and methods?
  – How does an agile DO-178C software process look like?
  – Agile workflows increase effort for complexity-, interface-, and change management – how can this be improved in the DO-178C process?
  – How can we avoid process corrosion without loosing agility?

**PLANNING – REALIZATION - PRESERVATION**
Research

1. Adoption and tailoring of a modular, model-based DO-178C/DO-331 process supporting agile workflows.

2. Realization and compliance preservation of the process using a process-oriented build tool
Planning

MODULAR MODEL-BASED DEVELOPMENT PROCESS
DO-178C about the software life cycle

This section discusses the software life cycle processes, software life cycle definition, and transition criteria between software life cycle processes. This document does not prescribe preferred software life cycles and interactions between them. The separation of the processes is not intended to imply a structure for the organization(s) that perform them. For each software product, the software life cycle(s) is constructed that includes these processes.

The processes of a software life cycle may be iterative, that is, entered and re-entered. The timing and degree of iteration varies due to the incremental development of system functions, complexity, requirements development, hardware availability, feedback to previous processes, and other attributes of the project.

- Software life cycle is the sequence of processes
- Requires planning, development, and integral processes
- The sequence of processes can be chosen by the project, can be iterative, re-entered, and have a project-specific timing
- Not every input to a process must be complete, as long as the transition criteria are satisfied
- Significant freedom
Embedded model-based software development process

R – Requirements
D – Design
C – Code
T – Testing
I – Integration
[] – Partially

- ARP4754A
- SW DO-331 (MBSwD)
- SW DO-178C (main)
- HW DO-254
Embedded model-based software development process

Model-based development process is split into a top-level process \((T)\) and multiple sub-level processes \((S)\)

R – Requirements
D – Design
C – Code
T – Testing
I – Integration
[] – Partially
Experiences with MBD and a modular workflow

- Module architectures and interfaces
- Scoping and access limitation
- Requirements and traceability
- Modular code generation
- Reachability
Module architecture and interfaces (1)

**CHALLENGE**

- The process is broken down into sub-level processes to allow concurrent, agile development
- Interface definitions are extremely important for both development and verification
- Interface definitions need to be shared amongst modules
- No tool in Simulink supporting creation of architectures / interface definitions
- Many companies have their home-grown solution – there is currently no product out-of-the box
Module architecture and interfaces (2)

**Our Solution**

- A mixture between manual drawing (Visio)/independent UML architecture tools (MagicDraw, Capella,…) and interface definitions with Simulink Buses in a global data dictionary
- But assuring consistency between architecture design / interface definition and Simulink Busses is still challenging
- Has to be solved in the future
Requirements allocation to modules and traceability (1)

**CHALLENGE**

- Allocation of requirements to modules (to refine or implement)
- Traceability from model elements to requirements
- Traceability robustness against "welcome requirements changes"
- Impact analysis from requirement changes to model elements and test cases

- Requirements are managed in **SIEMENS Polarion**
  - Inter-tool traceability (tool gap)
  - Challenge for configuration management
Requirements allocation to modules and traceability (2)

**OUR SOLUTION**

1. Identify the requirement subset
2. Formulate query and store with module
   WHERE COMPONENT =...
3. Establish traces to subset

- Simple but powerful setup
- Definition and allocation of a set of requirements by queries allows
  - Identification of allocation changes (new, deleted and reallocated requirements)
  - Coverage prediction
  - Simple config management

(POLARION® Server)
SimPol Manager View
Deep integration

Demo of a Stateflow Home Security System, including a motion detector.

The software shall return the active mode `mode_lgx` representing the current mode according to the following table:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>unset()</td>
<td>stanag586-10</td>
</tr>
<tr>
<td>standby(1)</td>
<td>stanag586-10</td>
</tr>
<tr>
<td>xhi(2)</td>
<td>stanag586-10</td>
</tr>
<tr>
<td>op_1c_airborne(10)</td>
<td>stanag586-39</td>
</tr>
</tbody>
</table>
Bridging the gap…

SIMPOL …

- Allows allocation of requirements to modules (to refine or implement)
- Ensures traceability from model elements to requirements
- Provides means for impact analysis from requirement changes to model elements and test cases
- Closes the tool gap with an simple and easy setup
- Integrates into the design environment
Scoping and access control (1)

**CHALLENGE**

- Team work requires access control of variables and functions
- Good practice in almost all programming languages, e.g., *private*, *protected* and *public* in C++
- There is no out-of-the-box solution in Simulink to do scoping on module-level
- Scoping may influence coverage
Scoping and access control (2)

**Our Solution**

- Scoping of model data for models by model workspaces
- Scoping of model data for modules by two separate SL data dictionaries types (one for private and one for public data)
- Scoping of functions with SL Projects labels
- No scoping of SL library elements
- No built-in and forced access control → retrospective checks

<table>
<thead>
<tr>
<th>Scope</th>
<th>Functionality</th>
<th>Model data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td><em>SL Projects</em></td>
<td>Model Workspace</td>
</tr>
<tr>
<td>Module</td>
<td></td>
<td><em>Data Dictionary</em></td>
</tr>
</tbody>
</table>
Modular code generation (1)

CHALLENGE

- Each module consists of software units, which are finally integrated into a single model hierarchy
- MathWorks workflows for Embedded Coder propose a single top-level build
- Full modular code generation not out-of-the-box
- Reason is mainly shared code (some source files are incrementally generated)

Consequences
- Small changes always require a full top-level build and reverification of all code artifacts
- Reuse of code from other modules not easily possible
- Long workflows and late detection of rework
Modular code generation (2)

**Our Solution**

- Correct coder configuration settings
  - Limitations on shared code options and suitable coder configuration settings (target settings, shared library settings, …)

- Modeling guidelines
  - Storage classes, explicit header file names

- Multi-step build process
  - Code relocation, separate build of shared variables, change detection

- Last but not least: DISCIPLINE!
Reachability (1)

**CHALLENGE**

- Throughout the DO-178C process, multiple reachability requirements must be satisfied
  - Model coverage
  - Code coverage
  - Formal reachability analysis (e.g., using Polyspace Code Prover, MISRA checking,…)

- Reachability can not only be evaluated for a module
- When is a model element unused?
Reachability (2)

Our Solution

- Use scoping information
- Slightly different activities for top- and sub-level process
  - E.g., different Polyspace MISRA checks
- Coverage accumulation
Realization and Preservation

PROCESS-ORIENTED BUILD TOOL
Now let's be AGILE!

Plans, Standards, Rules, Guidelines, Template,…
Problems of realizing and preserving process compliance in an agile process

- Modular process is complex and difficult to maintain
- Development standards and plans decrease agility
- Agile requires fast iterations and frequent changes → consistency
- Continuous delivery, continuous integration,… → to plug into a CI system, a certain framework must be available

- Artifact completeness, correctness, and consistency not visible to the developer
- Process compliance status not easily accessible for "buisness people"
8 Commandments

1. Automatic environment and workflow integration
2. Compliant by construction
3. Support of a modular process (dependency hell)
4. Artifact and activity status, process guidance
5. Dynamic review
6. CI Integration
7. Flexible threshold
8. Process status

How can a tool support agile workflows?
Our solution: Process-Oriented Build Tool

• Build tool in MATLAB enhanced with process-oriented features

• Traditional Build Tool
  – Determine and execute a sequence of dependent, pre-configured build jobs
  – Incremental build
  – CI-Integration

• Process-Oriented Build Tool
  – Holistic workflow covers
    • Planning and Setup
    • Implementation
    • Verification
    • Review
    • CI Integration
  – Construction of artifact tree
  – Evaluation/normalization of results
  – Evaluation of completeness, consistency, correctness of artifacts
  – Dynamic review lists and result mitigation by review

Holistic supported module workflow

1. Define / change process and formalize process setup
2. Perform change impact analysis (web-based UI)
3. Implement model with model scaffolding
4. Incrementally build code, verification artefacts...
5. Check: Fulfills minimum requirements? e.g., missing, suspected, or failure
6. Review: Perform review and justify in central UI (web-based)
General tool structure

- „Ship“ an independent life cycle package with design patterns, a build workflow and the modeling environment
- Integrate into project, construct and build
- In the background, the tool creates an artifact graph
- Perform incremental build and impact analysis on graph
- Provide access to and visualize status
Life Cycle Package (1)

- Independent source control repository, which is automatically embedded into the project (like shipping the workflow)

- Consists of
  - Design Schema
  - Build Workflow
  - Modeling Environment
Life Cycle Package (2)

- Each build job evaluates the results and returns a normalized status

<table>
<thead>
<tr>
<th>MISRA Category of Violation</th>
<th>Status</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mandatory</td>
<td>❌ (fail)</td>
<td>Violation requires rework of model and update of generated code.</td>
</tr>
<tr>
<td>Required</td>
<td>🔴 (warn)</td>
<td>To justify, review Polyspace annotation, set status in Polyspace to “Justified” and add comment. Reference to specific deviation in [Procedure supplement] is possible.</td>
</tr>
<tr>
<td>Advisory</td>
<td>✔️ (pass)</td>
<td>Advisory violations should be reviewed and are automatically documented.</td>
</tr>
</tbody>
</table>
Workflow

- Auto-include transient dependencies (GIT) and version conflict detection
- Command-line based interface
- Similar to ruby-on-rails

Sample commands:
- mrails stage design
- mrails create model-reusable <name>
- mrails create param-data-item <name>
- mrails codegen
Artifact Graph

- Centrally built
  - Implicitly defined by build jobs ( specification of expected outputs)
- Decentrally stored subgraph with each tasks
- Central graph is rebuilt from subgraphs

- Advantage:
  - Very robust
  - Partial deletion of task results

- Disadvantage:
  - Performance
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",
- distinguishes between model references and top-level models during execution,
- generates an 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

5 Suspected task

4 Inconsistent model report

3 Traceability (up- and downwards)

2 Multiple tasks (one for every model)

6 Status overview
Tight integration with various tools
Summary

- Adoption and tailoring of a modular, model-based DO-178C/DO-331 process to agile workflows has been discussed.

- As a solution to overcome process corrosion in agile development teams and to streamline development, a process-oriented build tool in MATLAB is presented.

- Supports process realization and assure compliance through pre-defined development steps.

- Integrates into CI environments.
Outlook

- Increase tool experience and maturity through application to a large scale example project

- Performance optimization and scaling (e.g. parallelization of checks)

- Automatic background update of jobs based on modifications on artifacts

- Generation of documents (e.g., Software Accomplishment Summary)