AIAA SciTech Forum
7-11 January 2019, San Diego, California
AIAA Scitech 2019 Forum

10.2514/6.2019-0500

Traceable and Model-Based Requirements Derivation, Simulation, and Validation Using MATLAB Simulink and Polarion Requirements

Kevin Schmiechen*, Markus Hochstrasser†, Julian Rhein‡, Christopher Schropp§, and Florian Holzapfel¶
Technical University of Munich, Garching, 85748, Germany

In contrast to classical manual requirement derivation and validation, this paper presents a cross-platform traceable model-based requirement development process. The goal is to improve the quality of requirements and thus reduce development costs and time to market of the product. The requirements management platform *Polarion Requirements* is used to record the requirements in natural language and in a textual, formalized form. With *MATLAB Simulink* and *Stateflow*, system, specification, and assessment models are created. Furthermore, *Simulink Design Verifier* is used for formal validation of the requirements. Full traceability between *Polarion* and *Simulink* is provided by the tool *SimPol*, which has been developed at the Institute of Flight System Dynamics at the Technical University of Munich. This process is illustrated using requirements for a signal source selection function. The process showed to be effective for the exemplary requirements.

I. Introduction

The quality of requirements is a crucial factor for a development project. It is worthwhile to invest a high amount of effort to make sure that the requirements are, amongst other properties, correct, complete, and not in contradiction to each other. According to [1], about 60% of all errors in system development are based on insufficient requirements and fixing these errors during implementation is up to 20 times more expensive than during requirements engineering. The process, which is defined in Section II of this paper, has the goal to ensure a high level of quality for the requirements by preventing or respectively revealing defects early in the requirements engineering process. In this way, the development costs and the time to market can ultimately be reduced [2].

Figure 1 shows a common V-Model that is applicable for system, hardware (HW), and software (SW) development. The left branch captures and validates the requirements top down from the aircraft level to the software and hardware level. Classically, requirements are only captured in natural language. The review is then performed manually based on these textual requirements. However, [3] recommends requirements validation by analysis, modeling, or test for certification of Development Assurance Level (DAL) A and B artifacts.

This paper addresses the left branch of Fig. 1 as well and describes an approach completely supported by simulation models. It is use-oriented and has the intention to present a process that can be applied to a wide range of problems in the engineering industry with commonly used tools. The approach includes formalization and modeling of the requirements to improve the possibilities and quality of requirement derivation and validation. The three steps of informal analysis, requirements formalization, and requirements validation are already described in [2, 4]. An aeronautical case study for requirements formalization with *Modelica* is demonstrated in [5]. "Requirement modeling and automated requirements-based test generation" is presented in [6]. In [7], "best practices for verification, validation, and test in model-based design" are described and this paper also emphasizes the importance of model-based design early in the development phase.

Another purpose of introducing formalization and modeling of the requirements is to make sure that they are verifiable. In [3], a requirement on a system level is defined as "an identifiable element of a function specification that can be validated and against which an implementation can be verified." On the software level, [8] also demands the high and low-level requirements to be verifiable. Furthermore, it defines verification as "the evaluation of the outputs of a

*PhD Candidate, Research Associate, Institute of Flight System Dynamics, kevin.schmiechen@tum.de, AIAA Member
†PhD Candidate, Institute of Flight System Dynamics, markus.hochstrasser@tum.de
‡PhD Candidate, Research Associate, Institute of Flight System Dynamics, julian.rhein@tum.de
§PhD Candidate, Institute of Flight System Dynamics, schropp@tum.de
¶Full Professor, Head of Institute, Institute of Flight System Dynamics, florian.holzapfel@tum.de, AIAA Associate Fellow

process to ensure correctness and consistency with respect to the inputs and standards provided to that process." As already stated in [9], verifiability is established as soon as the requirements are formalized and modeled. As a result, this property can already be ensured during the requirements capturing phase.

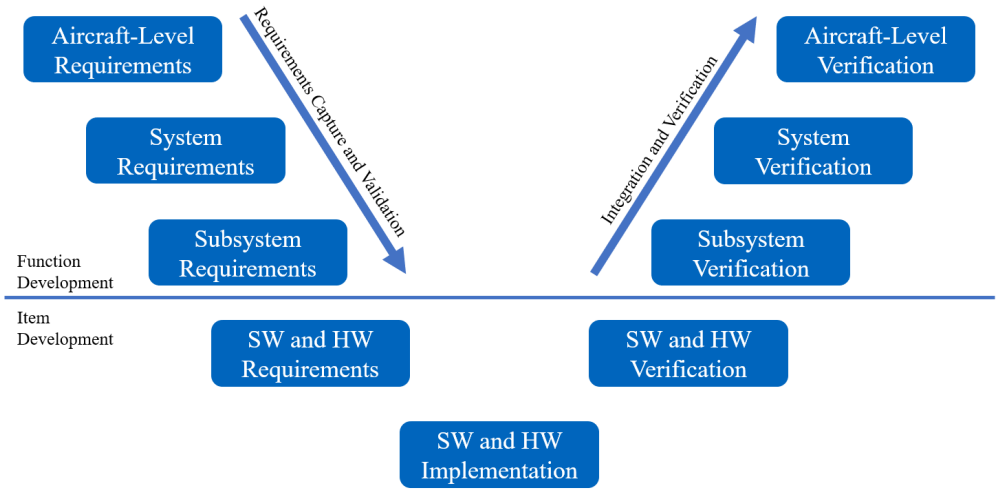

Fig. 1 V-Model According to [10]

The industry tools used in this paper are *MATLAB Simulink*, *Stateflow* and *Polarion Requirements*. *Simulink* and *Stateflow* are extensions to *MATLAB* by *MathWorks, Inc.* *Simulink* is a common graphical programming tool for modeling, simulating and analyzing dynamic systems. *Stateflow* enables *Simulink* to model reactive systems via state machines and flow charts. *Polarion Requirements* is part of the web-based Application Lifecycle Management (*ALM*) suite by *Siemens Industry Software*. The module features change/configuration management, requirements management, and branch management for engineering projects. Requirements, as well as all other artifacts, are stored as work items on the platform. Links between the work items enable bi-directional traceability and make it easier for the user to capture all the correlations between the requirements.

In addition to the traceability within the platform, the tool *SimPol*, which has been developed at the Institute of Flight System Dynamics, ensures traceability between the textual requirements in *Polarion* and the simulation models in *Simulink*. *SimPol* is described separately in Section III. The *Simulink Design Verifier* is used for formal validation of the requirements. A general description of model checking along with an introduction to the *Simulink Design Verifier* is given in Section IV.

The model-based requirement development process including traceability will be illustrated in Section V by means of exemplary requirements for a signal source selection function. This function determines which cockpit control inceptor signals (Captain or First-Officer side) to use for the computation of the aircraft control surface commands.

II. Process Definition

The continuously model-based requirement development process presented in this paper is depicted in Fig. 2. It also contains requirements capture and validation as shown in Fig. 1 but adds requirement formalization and multiple types of models that are used throughout the process. The bars above the blocks in the top row illustrate the progress regarding the model-based requirement development process. An overview of the different model types is given in Fig. 3.

A. Requirement Derivation

The requirement development process starts with requirement derivation in textual natural language or graphical form, whichever is best. It is suggested that the requirements are recorded on a management platform such as *Polarion* for configuration management and traceability. Models of the system are used as assistance for the derivation process. These can not only be models of the plant for which a function or system is to be developed (e.g. aircraft dynamics for a controller) but also simplified models of the system itself (e.g. linear second-order systems for the closed-loop moment dynamics). In addition, based on the requirements, prototype implementations of the system or function under

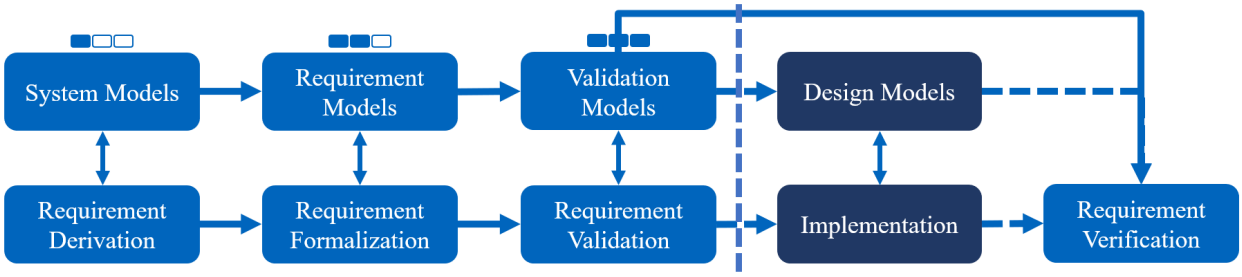

Fig. 2 Continuous Model-Based Requirement Development Process

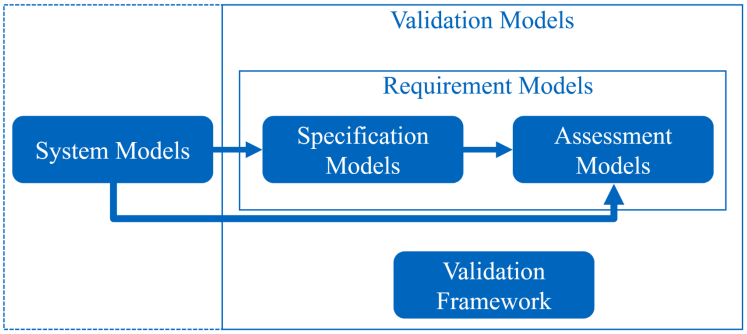

Fig. 3 Overview of Model Types

ideal conditions where all states and signals are perfectly known can help to increase the understanding of the complex system or the function. In Fig. 2, these system models are marked with 1/3 bars to indicate that they form the start for the models used in the process presented in this paper but will be reused in the following steps.

B. Requirement Formalization

In the next step, the captured requirements are formalized. This means they are precisely described in a formal specification where the notation is uniquely defined. However, formalization can of course also be already conducted in parallel to the derivation of the requirements. That way, they are available in an unambiguous form right in the beginning of the development phase.

Formalization is the link between the requirements in natural language and the *Simulink* models. As mentioned in [4], there are several formal specification languages such as Z, B, OCL and Alloy. This paper though does not suggest a certain formal specification language. The focus lies on the overall process and the integration into the commercial tool-chains. Furthermore, the example requirements in this paper describe only state transitions for which tables with the transition conditions are sufficient. The formalization is stored in a textual form in *Polarion* in the same work item as the natural language text of the requirement. Work items are the artifacts within *Polarion*.

C. Requirement Modeling

The requirement models in *Simulink* can be created based on the formalized requirements without any knowledge of the system or the project, if formalization has been performed properly. It is just another representation of the same content. Requirement models can be separated in the two categories specification models and assessment models. Together, both types of models make up 2/3 bars in Fig. 2.

For controlled dynamic variables, specification models can implement the desired closed-loop response. For logical functions, they can implement state sequences independent of specific dynamics (e.g. ground spoiler deployment after touchdown). The specification models use the output of the system models from the first step as input.

The output of the specification models or prototype implementations is used as input for the assessment models. For dynamic behavior, they utilize inequalities based on metrics boundaries for adequate behavior to determine compliance or non-compliance. A distance measure from the metrics boundary for desired behavior is used to determine the quality

of compliance. For logical and abstracted system behavior, the assessment models evaluate logical conditions on system variables. In contrast to the specification models, assessment models can also be created for requirements that define the absence of a certain behavior. An example for this is a variable that shall never exceed a certain threshold.

D. Requirement Validation

The models for requirement validation consist of the validation framework, the assessment models, specification models and optional system models. Incorporating the models from the previous steps, validation models have 3/3 bars in Fig. 2. The dashed line in Fig. 3 indicates that the system models are optional for requirement validation. In general, they are only necessary for requirements describing closed-loop behavior. Consequently, an aircraft or cockpit control inceptor system model is not necessary in order to test the requirements for the signal source selection function of this paper. However, specification models are necessary. In [11], two roles are assigned to a design model. The design model serving as a basis for automatic code generation will be addressed in Section VI. The second and interesting role, here called specification model, for this paper is the replacement for low-level software requirements by a design model. This means, the design model is located below the dividing horizontal line in Fig. 1, and thus belongs to item development, but is still on the left branch. It is a model that represents the desired behavior of the function on a software level before the actual implementation.

In this paper, the property proving capability of the *Simulink Design Verifier* is used for formal model checking of the specification model against the assessment models. Applying the tool to the assessment models and a specification model for test vector generation is already described in [6]. However, the general approach will be described in Section IV and it will be applied in Subsection V.E.

By verifying the specification models against the assessment models, not only inconsistencies between higher-level requirement models and the lower-level specification models can be detected but also inconsistencies between the specification models. In the case of requirements violation, formal verification tools such as the *Simulink Design Verifier* provide explanations of the inconsistencies in the form of counterexamples. They can be used for the iterative refinement of specification and assessment models. By verifying against all requirements at once, it is tested whether the requirements are not only correct but also if they are consistent with each other.

III. Cross-Platform Bi-Directional Traceability

Traceability is a well-known method of software development enabling assessment of thoroughness and completeness of derived artifacts. Especially bi-directional traceability between textual requirements and requirement models enables

- evaluation of the completeness of requirement models
- evaluation of whether all requirement models reflect textual requirements
- impact analysis in both directions upon changes.

However, traceability information is only a reliable source for verification if it is accurate and up-to-date. The effort to establish and maintain traceability is considerable and developers often do not directly recognize the benefit which trace data provides later in the life cycle. The implementation effort even grows if the traces span between artifacts which are maintained by different tools, as is in this case. The textual requirements reside in *Polarion* and the derived requirement models in *Simulink*.

Both tools only offer bi-directional traceability within their environment. They can also trace to external objects, however, without an automatic backtrace. To bridge the gap, *SimPol*, a tool which has been developed by Markus Hochstrasser at the Institute of Flight System Dynamics of the Technical University of Munich, is used.

On Simulink or Stateflow side, *SimPol* docks to the so-called Requirements Management Interface (RMI)*. With *Polarion*, it communicates through the Open Java API †. Traces are realized with unique HTTP hyperlinks in both directions, directly pointing to work items in *Polarion* or to model elements through the local web server provided by the RMI.

The main challenge is configuration management. Firstly, *Polarion* requirements and *Simulink* models reside in different version control systems. If older revisions must be restored at a later point in time, trace links must point to the correct artifact revisions as well.

The flexibility to handle different version control systems is given by separating the revision information from the actual artifact identification of the link. The revision pointer is saved in a separate file within the Simulink project. It

*https://www.mathworks.com/help/slrequirements/requirements-management-interface.html

†https://almdemo.polarion.com/polarion/sdk/doc/javadoc/index.html

can be connected to any previous baseline in *Polarion* without modifying the links.

Secondly, a trace link should not change the revision of the linked artifact itself due to consistency reasons. For example, requirement baselines are reviewed and verified before being handed-over for implementation. Reports often contain this revision information and would otherwise have to be updated after linking.

The RMI in *Simulink* supports external storage of links without influencing the model. *Polarion*, in contrast, does not support storing independent, supplemental data. As a solution, *SimPol* supports so-called surrogate models. For each linked model element, a work item in *Polarion* is created as illustrated in Fig. 4. It does not only contain the hyperlink to the model element, but also *Polarion*-internal links to requirements. It can be considered as a binding object. Since *Polarion* only saves links on the source side and implicitly maintains the reverse links, the revision of the requirement work item remains untouched. Keeping the surrogate work items up-to-date is a core functionality of *SimPol*.

Impact analysis is performed with *SimPol* as well. *Polarion* provides a mechanism, which allows the user to manually mark artifacts as suspected (or define rules to do so) and propagates the suspicion mark down through the requirement hierarchy. SimPol extends the propagation to model elements and thus enables complete impact analysis.

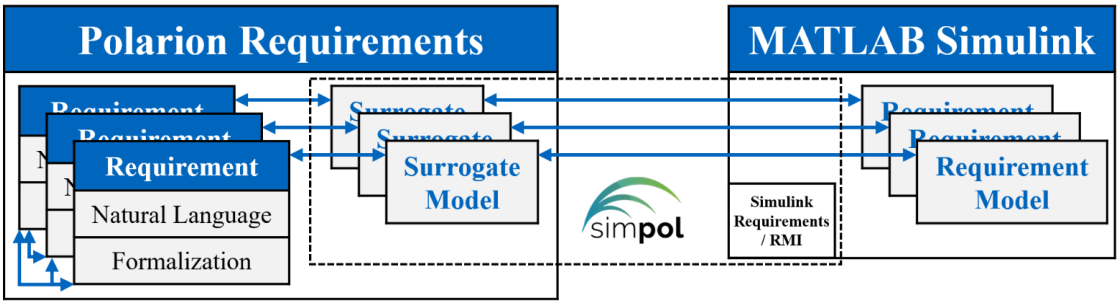

Fig. 4 Cross-Platform Bi-Directional Traceability with *SimPol*

IV. Model Checking

A. General

The objective of requirements validation and verification is to ensure that the specified system fulfills its requirements under all circumstances. Traditionally, this is mainly addressed by extensive simulation and testing. While these are useful techniques during the early stages of design, it becomes increasingly difficult to detect design flaws or bugs as the development matures, when there are fewer remaining errors that are also unlikely to be covered by the executed test cases. Essentially, simulation and testing can, within realistic time frames, evaluate only a subset of the modes and states which a system can have, thereby neglecting some behavioral aspects of the system which may still contain errors. Therefore, it cannot be guaranteed that the systems tested with such methods are error-free.

Model Checking [12, 13] is a complementary technique for the automatic verification of a formal system model against a set of requirements, also called *properties*. In order to verify that a desired property holds for the system under all possible conditions, an efficient search procedure, which evaluates all reachable system states and transitions between them, is executed. This search is usually based on a symbolic representation of the system. In most of today's model checking tools, the system to be verified is modeled as a (finite or infinite) state machine, and the formal requirements are represented in some form of temporal logic, such as Computation Tree Logic (CTL) or Linear Temporal Logic (LTL) [14]. The underlying analysis techniques include Binary Decision Diagrams (BDDs) [15, 16], Boolean Satisfiability Procedures (SAT) [17–19], and Satisfiability Module Theory (SMT) [20–22] solvers.

The output of a model checker is a mathematical proof that the formalized requirements for a system are either met or violated. When a requirement is violated, the model checker produces a counterexample that illustrates the system behavior violating the requirements. A counterexample is a sequence of system input stimuli, which drives the system along a trace of states eventually leading to a violation of a property. This is particularly helpful to understand the root cause of the requirements violation and to point out the design flaw or error of the investigated implementation. In general, model checking is well suited for the automated analysis of finite state systems, which are characterized in terms of highly complex logical relations and modes and are thereby difficult to test extensively. It should be noted that

model checking is not the right tool to assess system models that are described by a continuous state space or differential equations.

B. Model Checking with the Simulink Design Verifier

The *Simulink Design Verifier* (SLDV) by *MathWorks, Inc.* is a formal verification tool that is integrated in the *Simulink* development environment. The SLDV features a property-proving mode, in which the *Design Verifier* formally analyzes the model in order to prove or disprove the modeled properties. A property represents a formal requirement of the system under test that is modeled in *Simulink* in the form of a constraint (a set of values) that a signal must always fulfill [23]. This type of property, an expression which must always be fulfilled on every possible execution trace of the system, is known as a safety property in CTL. Safety properties are modeled by the *Proof Objective* block in *Simulink* (see Fig. 5). Certain temporal properties can be modeled using the temporal operators of the *Simulink Design Verifier* library, which allow the specification of past-time LTL expressions to some extent by constructing semantically equivalent observers (e.g., the *Detector* block represents a bounded *Historically* operator, the *Extender* block can be seen as equivalent to the *Once* operator). Additionally, the rigor of the proof can be relaxed by *Assumption* blocks, which specify preconditions for the proof objectives, i.e., a trace violating an assumption will never cause the proof to be violated. An illustrative introduction to the SLDV is given in [24].

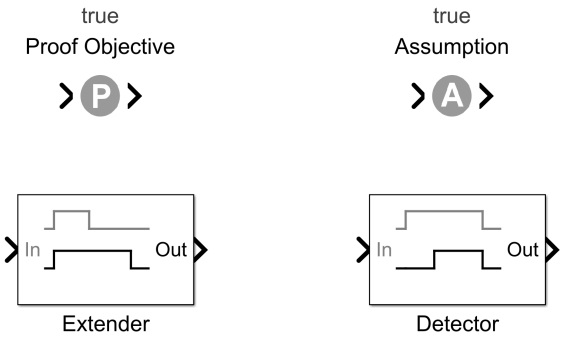

Fig. 5 Simulink Design Verifier Proof Assumption, Proof Objective and Temporal Operator Blocks

Figure 6 shows an exemplary SLDV property-proving setup. The inputs to the model are not specified explicitly but left non-deterministic. However, they can be partially constrained by a set of assumptions. After modeling the requirements to the system using the Proof Objective blocks, the property proving mode of the SLDV can be executed. As a result, the *Design Verifier* either proves the properties (i.e., Proof Objectives) valid for all possible inputs (under the given assumptions) or generates a trace of input signal values leading to a violation of the system requirements – a counterexample. Additionally, it provides the option to create a harness model, where the original system model is connected to a signal generator representing the input trace described by the counterexample.

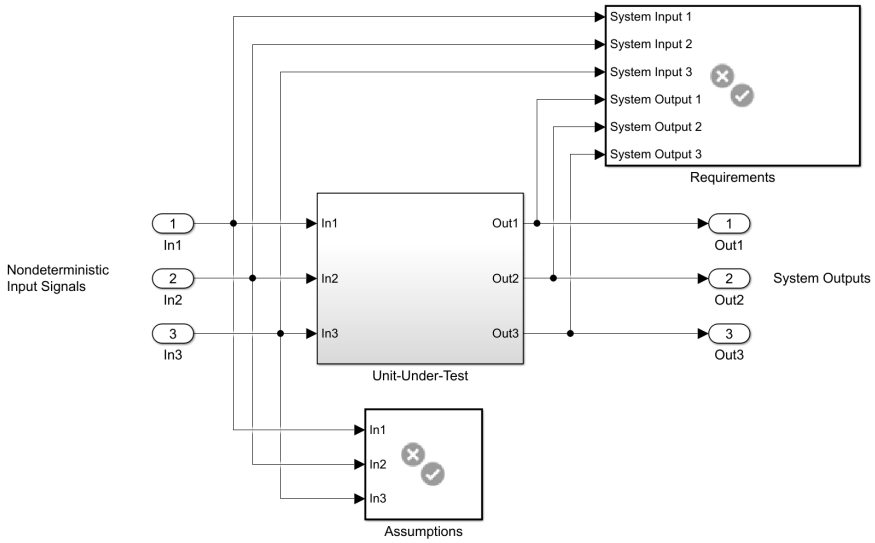

Fig. 6 Typical Simulink Design Verifier Model Checking Setup

V. Process Application

The process that was defined in Section II is illustrated in this section by means of exemplary requirements for a cockpit control inceptor signal source selection function.

This function determines which control inceptor signals (Captain (CPT) or First-Officer (F/O) side) to use for the computation of the aircraft control surface commands. In this example, the control inceptors are two yokes of a passive fly-by-wire system. They are mechanically connected in the nominal case but can be disconnected by a clutch in abnormal conditions. Disconnects can originate from different causes but this lies beyond the scope of this paper. In the nominal case, the command signal for the control surfaces is generated from the signals of both sides. Exclusive priority can either be assigned automatically by the system or manually by pressing the auto-pilot (AP) disengage button on the side that is supposed to have priority.

For better understanding, the high-level state chart of the complete system is given in Fig. 7. It consists of the three main modes, *no priority*, *CPT priority*, and *F/O priority*. Both priority modes have the submodes *Auto*, *Temporary*, and *Permanent*. *Auto* is activated automatically by the system (e.g. due to failures). The submode *Temporary* is e.g. activated by pilot input but is instantaneously left when the activation conditions are not valid anymore. In contrast, the system stays in the submode *Permanent* even when the activation conditions for this submode are not valid anymore. The labels of the transitions are the abbreviations for the transition conditions. The two exemplary requirements shown in the following subsections define the transition to the CPT or F/O temporary command priority by manual pilot input. In Fig. 7, these transition conditions are labeled as C_1 and C_2, respectively.

A. Requirement Derivation

As shown in Fig. 2, a model of the control inceptor system was used for the derivation of the requirements. However, it will not be required for validation in Section V.E. The control inceptor system is not part of the source selection function but forms the environment for the function to be developed. The model helped us to understand not only the nominal behavior but also the behavior under failure conditions, especially regarding the passive fly-by-wire system and mechanical disconnect.

The requirements were captured on the management platform *Polarion*. The fields in the following only include those of the custom requirement template used for this paper. Polarion includes basic fields (e.g. *ID*, *Title*, *Author*, *Description*), but additional fields of different types (e.g. string, rich text, enumeration, date, integer, Boolean) can be defined freely by the user. For reasons of readability, the content of the exemplary requirements was extracted from Polarion and put into Tables 1 and 2.

In Polarion, the requirements also include the fields *Rationale & Discussion*, *Linked Work Items*, and *Comments* which are, however, disregarded for this paper. The field *Rationale & Discussion* is used to give the reasoning for the overall requirement and specified values. *Linked Work Items* contains references to other artifacts in Polarion such as

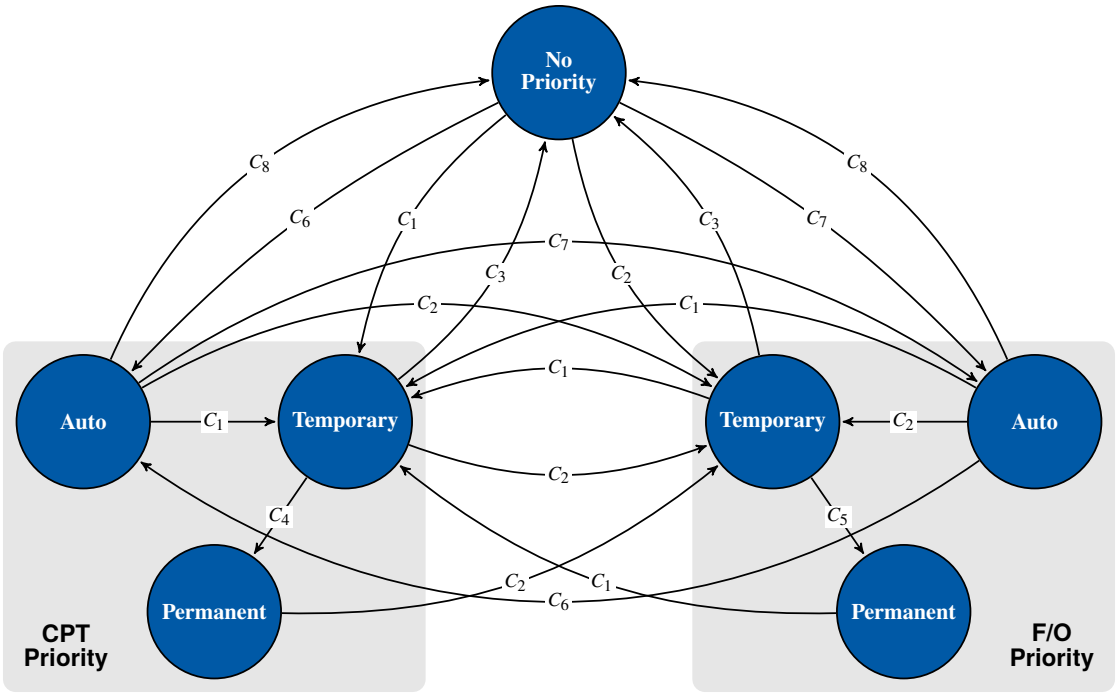

Fig. 7 Signal Priority State Chart

requirements, Simulink surrogate models, or test cases. *Comments* documents the reviews or the discussions of the engineers that led up to the requirement.

The field *ID* is used for unique identification. The prefix is specific for every project. The *Author* is the person who creates the requirement. The field is not updated when another person edits the requirement. However, the history of a work item clearly tracks all changes and their authors. The *Title* is only used for a better overview for the user. The field *Status* stores the current status of the work item within the specified custom workflow. Examples of states are *Draft*, *In Review*, *Reviewed*, and so on. The severity of a requirement (should, shall, must) is already given by the *Description*. However, the additional drop-down menu *Severity* is used for filtering. *Category* is also a drop-down menu to specify the requirement category. It is advisable to use the categories defined in [3].

Description contains the actual text of the requirement. The blue text in the tables indicates the links to the other requirement work items. These links are another benefit of the requirement management platform and are very useful when writing or reviewing the requirements.

The requirement for the F/O side in Table 2 is erroneous on purpose, to show the capabilities of model-based validation in Subsection V.E. In the current state of the two requirements, both activation conditions would evaluate as *True* at the same time when both pilots pressed their AP disengage buttons simultaneously, with the system being in the mode *no priority* and with both inceptor signals valid. Consequently, one requirement would be violated because only one of the priority modes can be activated at the same time. This is an example of two requirements which are correct when considered individually. However, they contradict each other when considered together as part of a system.

B. Requirement Formalization

Tables 3 and 4 contain the formalized form of the activation conditions for the requirements in Tables 1 and 2. They are each stored in the same work item in *Polarion* as the natural language description of the requirement.

The far-left column of each table lists the elementary variables of the activation conditions. Each of the other columns represents a conjunction of value assignments for those variables, i.e. a column evaluates to *True* if the conditions of all its elements are *True*. An asterisk (*) denotes a value that is not relevant. The collection of the columns represents a disjunction (i.e. the activation condition evaluates as *True* if one of the columns is *True*).

Submode is the submode according to the state chart illustrated in Fig. 7. *CPT_AP_flg* and *F/O_AP_flg* are the signal flags of the CPT and F/O AP disengage push buttons. *CPT_valid_flg* and *F/O_valid_flg* are flag signals that indicate

Table 1 Requirement FCS-3064

ID	FCS-3064	Title	Manual CPT Priority Activation				
Author	Kevin Schmiechen	Status	In Validation	Severity	Shall	Category	Functional
Description							
The FCS-3093 - Submode: Temporary CPT Priority of FCS-2897 - Operating Mode: CPT Control Inceptor Priority shall be activated when • the system is not in the FCS-3094 - Submode: Permanent CPT Priority - **and** - • the CPT AP disengage button is pushed down - **and** - • the CPT control inceptor deflection signal is valid - **or** - the CPT and F/O control inceptor deflection signals are invalid.							

Table 2 Requirement FCS-3066 (Erroneous)

ID	FCS-3066	Title	Manual F/O Priority Activation				
Author	Kevin Schmiechen	Status	In Validation	Severity	Shall	Category	Functional
Description							
The FCS-3090 - Submode: Temporary F/O Priority of FCS-2904 - Operating Mode: F/O Control Inceptor Priority shall be activated when • the system is not in the FCS-3091 - Submode: Permanent F/O Priority - **and** - • the F/O AP disengage button is pushed down - **and** - • the F/O control inceptor deflection signal is valid - **or** - the CPT and F/O control inceptor deflection signals are invalid.							

whether the CPT and F/O control inceptor signals are valid. The additional elementary variables *Operating_Mode* and *Mech_Disconnect_flg* do not have an influence on the examples presented and are thus disregarded here. For the AP disengage push button operation, the following wording is used:

- Rising edge (change from *False* to *True* in one time step): pushing down
- Falling edge (change from *True* to *False* in one time step): releasing
- *True*: pressed
- *False*: not pressed

Together with the other requirements that are not considered in this paper, the specification and assessment models can be created to enable not only manual but also automated assessment.

Table 3 Formalized Activation Conditions for FCS-3064

VARIABLE	CONDITION_1	CONDITION_2
Submode	NOT FCS-3094 - Submode: Permanent CPT Priority	NOT FCS-3094 - Submode: Permanent CPT Priority
CPT_AP_flg	RISING	RISING
F/O_AP_flg	*	*
CPT_valid_flg	True	FALSE
F/O_valid_flg	*	FALSE

C. Requirement Modeling

A *Simulink* model with a *Stateflow* chart was created based on the complete set of requirements. This is not a model for code generation but a specification model that simulates the behavior of the signal source selection function. The *Stateflow* chart represents the specification model of the requirements. It includes not only the architectural requirements

Table 4 Formalized Activation Conditions for FCS-3066 (Erroneous)

VARIABLE	CONDITION_1	CONDITION_2
Submode	NOT FCS-3091 - Submode: Permanent F/O Priority	NOT FCS-3091 - Submode: Permanent F/O Priority
CPT_AP_flg	*	*
F/O_AP_flg	RISING	RISING
CPT_valid_flg	*	FALSE
F/O_valid_flg	True	FALSE

but also the mode activation conditions as in Tables 1 and 2. A model for manual testing was created including push buttons, toggle switches and lamps from the category *Dashboard* of the *Simulink* library.

For automatic validation with the *Simulink Design Verifier*, which is described in Subsection V.E, the assessment models of the requirements were added to the model and the dashboard blocks were replaced by inputs that can be controlled by the *Simulink Design Verifier*. The assessment model for the manual CPT priority activation of Table 1 is shown in Fig. 8.

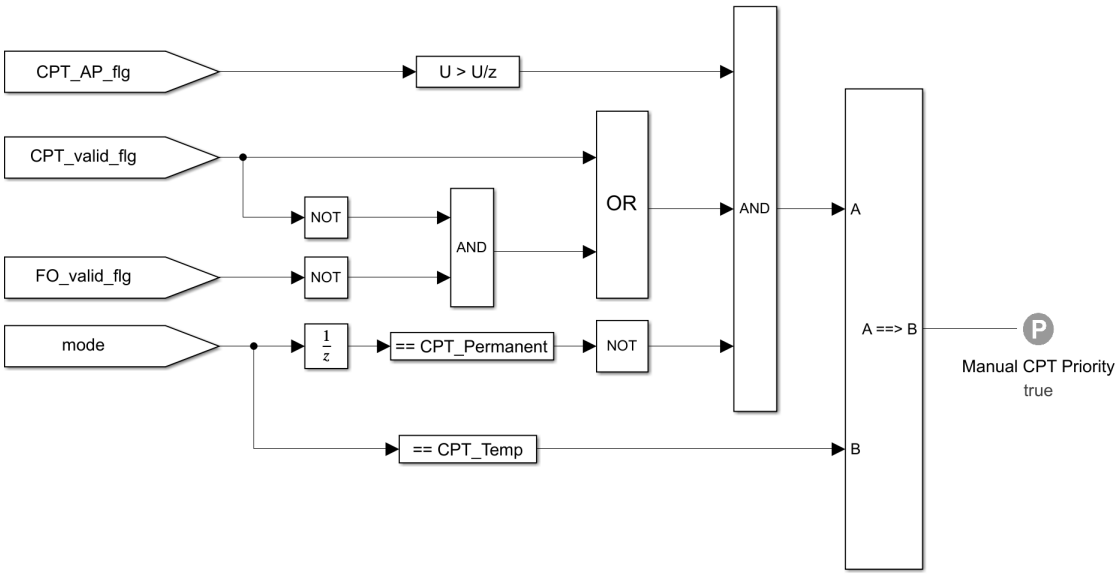

Fig. 8 Assessment Model for FCS-3064

D. Requirement Linking

SimPol was used to create traceability links between the subsystems containing the assessment models as in Fig. 8 and the requirement work items in *Polarion*. On *Simulink* side, the *Polarion* work items can be reached by right-clicking the linked subsystem and selecting the desired requirement from the *Requirements* menu. *SimPol* then opens the *Polarion* web-page of the requirement. In *Polarion*, *SimPol* creates a surrogate of the *Simulink* model which includes a screenshot of the subsystem or block. This surrogate model contains a link to the requirement work item and also a hyperlink to the *Simulink* block or subsystem. The hyperlink is a *URL* to the *localhost* providing the model name and block *ID* to the *RMI* of *MATLAB*. As long as *MATLAB* is running and the respective model is open or on the search path, the *RMI* opens the model and highlights the linked block or subsystem.

E. Requirement Validation

The example requirements were validated using the *Simulink Design Verifier*. As already indicated in Subsection V.A, the requirement for the manual F/O priority activation is erroneous in purpose. However, the *Simulink Design Verifier* did not only find a counterexample where this requirement is violated (see Fig. 9). It also found a counterexample where the requirement of the CPT priority activation is violated (see Fig. 10). In this counterexample, the function transitions to the temporary F/O priority at $0.01s$ due to the pushbutton signal from the F/O. At $0.02s$ it transitions to the automatic F/O priority because both, the CPT and F/O, control signals become invalid at the same time. In the next step, the function transitions to the temporary F/O priority again, while the conditions for the temporary CPT priority are also *True*. The reason for this is the execution order of transitions in *Stateflow*. In the example model, the transition to the temporary F/O priority is the first one to be executed from the automatic F/O priority. It would be a solution to correctly sort the execution order between the modes. That way, some information of the requirement would be included in the transition order instead of in the transition conditions. However, since the correct requirements were formulated such that the transition conditions are mutually exclusive, the execution order was not considered in the *Stateflow* model.

Fig. 9 Counterexample for FCS-3066

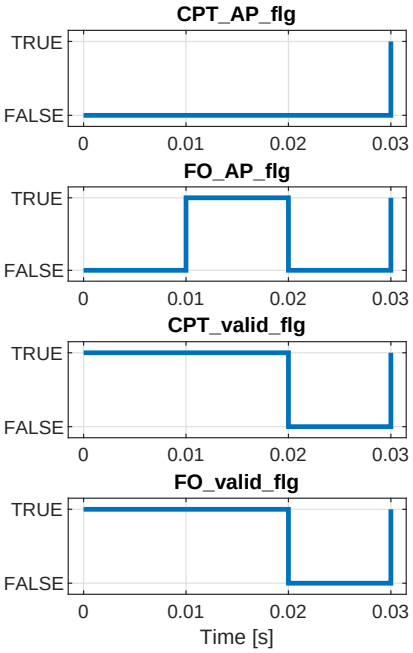

Fig. 10 Counterexample for FCS-3064

In order to make the transition conditions mutually exclusive again, requirement FCS-3066 needs to be updated. One possible correction is shown in Table 5 and 6. Having implemented this correction, the *Simulink Design Verifier* confirms that the two requirements are proven valid.

Table 5 Requirement FCS-3066 (Correct)

ID	FCS-3066	Title	Manual F/O Priority Activation				
Author	Kevin Schmiechen	Status	Change	Severity	Shall	Category	Functional
Description							

The FCS-3090 - Submode: Temporary F/O Priority of FCS-2904 - Operating Mode: F/O Control Inceptor Priority shall be activated when

- the system is not in the FCS-3091 - Submode: Permanent F/O Priority - **and** -
- the CPT AP disengage button is not pushed down - **and** -
- the F/O AP disengage button is pushed down - **and** -
- the F/O control inceptor deflection signal is valid - **or** - the CPT and F/O control inceptor deflection signals are invalid.

- **or** -

- the system is not in the FCS-3091 - Submode: Permanent F/O Priority - **and** -
- the F/O AP disengage button is pushed down - **and** -
- the CPT control inceptor deflection signal is invalid - **and** -
- the F/O control inceptor deflection signal is valid

Table 6 Formalized Activation Conditions for FCS-3066 (Correct)

VARIABLE	CONDITION_1	CONDITION_2	CONDITION_3
Submode	NOT FCS-3091 - Submode: Permanent F/O Priority	NOT FCS-3091 - Submode: Permanent F/O Priority	NOT FCS-3091 - Submode: Permanent F/O Priority
CPT_AP_flg	NOT RISING	*	NOT RISING
F/O_AP_flg	RISING	RISING	RISING
CPT_valid_flg	*	FALSE	FALSE
F/O_valid_flg	True	True	FALSE

VI. Conclusion

In this paper we described a model-based process for derivation, simulation and validation of requirements in the context of flight control system development. Having used the flexible and common industry tools *MATLAB* and *Polarion Requirements*, applicability is facilitated for a wide range of domains. Traceability between the two software environments is provided using the tool *SimPol* that has been developed. A general introduction into formal model checking was given and it was applied for the presented process using the *Simulink Design Verifier*.

The process has been implemented for two exemplary requirements regarding command signal source selection. The first version of one requirement was erroneous on purpose to show the capabilities of the model-based approach. Because of this error, the *Simulink Design Verifier* found counterexamples for both requirements. After correcting this error, both requirements were proven valid by the tool. The example showed how the model-based process helped us to improve the quality of the requirements.

The process described in this paper ends after the vertical dividing line in Fig. 2. However, as depicted in the figure, the validation models can be reused for verification of the implementation against the requirements. In Subsection II.D, it was already described that the design models can be used as a basis for automatic generation of the code that will run on the platform. For implementation, however, the design models are significantly more complex than it is necessary for requirement validation. As described in [7], the models can be applied for software- and processor-in-the-loop verification.

References

[1] Pohl, K., and Rupp, C., *Requirements engineering fundamentals: A study guide for the Certified Professional for Requirements Engineering exam : foundation level, IREB compliant*, 1st ed., Rocky Nook computing, Rocky Nook, Santa Barbara, CA, 2011. URL `https://ebookcentral.proquest.com/lib/subhh/detail.action?docID=741268`.

[2] Cecconi, M., and Tronci, E., "Requirements formalization and validation for a telecommunication equipment protection switcher," *Proceedings, Fifth IEEE International Symposium on High Assurance Systems Engineering (HASE 2000)*, IEEE Computer Society, Los Alamitos, Calif, 2000, pp. 169–176. doi:10.1109/HASE.2000.895456.

[3] SAE Aerospace, "Guidelines for development of civil aircraft and systems: SAE ARP 4754 rev. A," Tech. rep., 2010.

[4] Cimatti, A., Roveri, M., Susi, A., and Tonetta, S., "Formalization and Validation of Safety-Critical Requirements," *Electronic Proceedings in Theoretical Computer Science*, Vol. 20, 2010, pp. 68–75. doi:10.4204/EPTCS.20.7, URL `http://arxiv.org/pdf/1003.1741v2`.

[5] Schamai, W., Buffoni, L., Albarello, N., Fontes De Miranda, P., and Fritzson, P., "An Aeronautic Case Study for Requirement Formalization and Automated Model Composition in Modelica," *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, Linköping University Electronic Press, 2015, pp. 911–920. doi:10.3384/ecp15118911.

[6] Lee, C.-C., and Friedman, J., "Requirements Modeling and Automated Requirements-Based Test Generation," *SAE International Journal of Aerospace*, Vol. 6, No. 2, 2013, pp. 607–615. doi:10.4271/2013-01-2237.

[7] Murphy, B., Wakefield, A., and Friedman, J., "Best Practices for Verification, Validation, and Test in Model-Based Design," SAE International400 Commonwealth Drive, Warrendale, PA, United States, 2008. doi:10.4271/2008-01-1469.

[8] RTCA, "DO-178C: Software Considerations in Airborne Systems and Equipment Certification," Tech. rep., Washington DC, 2011.

[9] Cofer, D., and Miller, S., "DO-333 Certification Case Studies," *NASA Formal Methods*, Lecture Notes in Computer Science / Programming and Software Engineering, Vol. 8430, edited by D. Hutchison, T. Kanade, and J. Kittler, Springer International Publishing, Cham, 2014, pp. 1–15. doi:10.1007/978-3-319-06200-6_1.

[10] Rierson, L., *Developing safety-critical software - a practical guide for aviation software*, Taylor & Francis Inc, 2013.

[11] Hochstrasser, M., Schatz, S. P., Nürnberger, K., Hornauer, M., Myschik, S., and Holzapfel, F., "Aspects of a Consistent Modeling Environment for DO-331 Design Model Development of Flight Control Algorithms," *Advances in Aerospace Guidance, Navigation and Control*, edited by B. Dołęga, R. Głębocki, D. Kordos, and M. Żugaj, Springer International Publishing, Cham, 2018, pp. 69–86. doi:10.1007/978-3-319-65283-2_4.

[12] Clarke, E. M., and Emerson, E. A., "Design and synthesis of synchronization skeletons using branching time temporal logic," *Logics of Programs*, edited by D. Kozen, Springer Berlin Heidelberg, Berlin, Heidelberg, 1982, pp. 52–71.

[13] Clarke, E. M., Jr., Grumberg, O., and Peled, D. A., *Model Checking*, MIT Press, Cambridge, MA, USA, 1999.

[14] "Linear Temporal Logic Symbolic Model Checking," *Computer Science Review*, Vol. 5, No. 2, 2011, pp. 163 – 203.

[15] "Symbolic model checking: 1020 States and beyond," *Information and Computation*, Vol. 98, No. 2, 1992, pp. 142 – 170.

[16] Clarke, E. M., Grumberg, O., and Hamaguchi, K., "Another Look at LTL Model Checking," *Formal Methods in System Design*, Vol. 10, No. 1, 1997, pp. 47–71.

[17] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y., "Symbolic Model Checking without BDDs," *Tools and Algorithms for the Construction and Analysis of Systems*, edited by W. R. Cleaveland, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 193–207.

[18] Bradley, A. R., "SAT-Based Model Checking without Unrolling," *Verification, Model Checking, and Abstract Interpretation*, edited by R. Jhala and D. Schmidt, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 70–87.

[19] Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A., "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," *Proceedings of the 14th International Conference on Computer Aided Verification*, Springer-Verlag, London, UK, UK, 2002, pp. 359–364. URL `http://dl.acm.org/citation.cfm?id=647771.734431`.

[20] Tinelli, C., "SMT-Based Model Checking," *NASA Formal Methods*, edited by A. E. Goodloe and S. Person, Springer Berlin Heidelberg, 2012.

[21] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., and Tonetta, S., "The nuXmv Symbolic Model Checker," *Computer Aided Verification*, edited by A. Biere and R. Bloem, Springer International Publishing, 2014, pp. 334–342.

[22] Gacek, A., Backes, J., Whalen, M., Wagner, L. G., and Ghassabani, E., "The JKind Model Checker," *CoRR*, Vol. abs/1712.01222, 2017. URL `http://arxiv.org/abs/1712.01222`.

[23] Mathworks, T., "Simulink Design Verifier User's Guide," Tech. rep., 2016.

[24] Storm, W., "Solving Sudoku using Simulink Design Verifier-A Model Checking Example," *AIAA Infotech at Aerospace 2010*, 2010.