Specification of Embedded Systems
Summer Semester 2020

Session 4
Requirements Tracing

Jan Peleska
peleska@uni-bremen.de
Issue 2.0
2020-06-18

Chapter 1

Preface

In this document, the material for **Session 4** of the course **Specification of Embedded Systems** is provided.

This document is structured as follows. Overview

- In Section 2, the objectives of requirements tracing are explained.
- In Section 3, the SysML modelling techniques for tracing requirements are described.
- In Section 4, we slightly extend the material of Session 1 regarding requirements and associations between them. This extension is presented now, because it requires a deeper understanding of system design which is now available, since we have modelled a substantial portion of the Turn Indication Control System.

  In Section 5, we give a short introduction into model-based testing, where requirements tracing information can be used to identify test cases that are suitable for verifying the given requirement.
- As usual, these lecture notes end with questions and exercises in Section 6.

For this session, we advise you to study Chapter 13 in [1] for more elaborate background information about what is presented here in condensed form.

Contents

List of Figures

Chapter 2

Objectives of Requirements Tracing

You remember from Session 1 that requirements are *"first-class citizens"* of the SysML modelling language. This means, that they have their own SysML language elements and can be related to other model elements. This feature distinguishes SysML significantly from other modelling formalisms, where requirements come in as "an afterthought" and have to be represented by other existing language constructs that were not originally invented for this purpose.[1]

Typically, requirements are stored in a separate package of the model, and the other packages "show how these requirements are realised". We also say that the other packages show how the requirements **are implemented**, because the formalised model represented by these packages should represent an **implementable refinement** of the informal requirements specified in the first package.

From requirements capture ...

So far, we have only *captured* (i.e. identified) requirements for the Turn Indication Control System, and we have structured them, for example, according to behavioural, structural, and non-functional requirements. We recall from Session 1 that the full requirements data can be extracted from *leaves* of the requirements diagrams: there, atomic requirements without further decomposition are specified. The requirements further up in the de-

[1] For example, requirements are often represented as comments with a special format in programming languages. Other modelling formalisms require to set up tables relating requirement identifiers created with other tools to behavioural or structural model elements.

composition tree are only needed for structuring the atomic requirements into suitable sub-collections.

. . . to requirements tracing

The main objective of this session is to learn how to **trace** requirements to other parts of the model, showing that

- the requirement has not been forgotten, and
- which design is considered as appropriate for realising the requirement in an effective way.

Chapter 3

The SysML Approach to Requirements Tracing

3.1 The Satisfy Relationship

Satisfy relationship

As you may have guessed, requirements tracing is performed by means of specific associations, linking the requirement to one or more model elements representing structure, behaviour, or non-functional system properties. The most important association is the **satisfy relationship** which is depicted by a dashed-line arrow decorated with the stereotype ≪satisfy≫ and leads from a structural or behavioural model element to a requirement. The interpretation of this relationship is

> *The model element at the base of the* ≪satisfy≫*-arrow (called the* **client***) contributes to the realisation of the requirement at the arrow head (called the* **supplier***) of this association.*

The ≪satisfy≫-relationship always has a requirement at its supplier end. At the arrow base, any named element may occur as client, including even requirements. The typical usage, however, is to linke a named element which is *not* a requirement to a requirement.

The ≪satisfy≫-relationship is *many-to-many*:

- Many structural and/or behavioural model elements may be needed to realise one requirement.
- One structural or behavioural model element may partake in the realisation of many requirements.

In Fig. 3.1, which is part of a requirements diagram, it is shown how the fact that the proxy port EmerSwitchPressed contributes to the implementation of requirement REQ-023 is expressed by means of a ≪satisfy≫-relationship. Arbitrary model elements may be dragged and dropped to requirements diagrams, and then linked to requirements by means of the ≪satisfy≫-relationship.

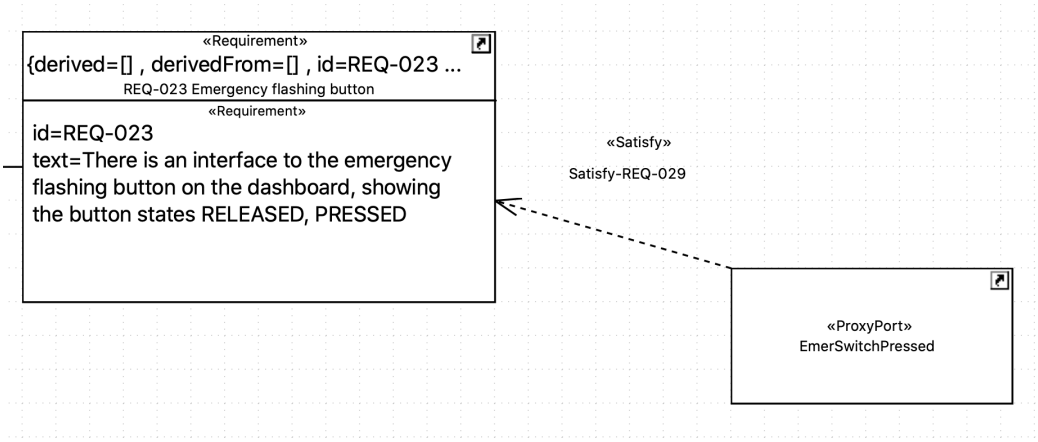

Figure 3.1: Example of a ≪satisfy≫-relationship.

Trace relationship

For showing in a very general way that one model element (including a requirement) is "somehow" related to another, SysML provides a second association called **trace relationship** which is represented by a dashed-line arrow labelled by stereotype ≪trace≫. This association should *not* be used in places where a ≪satisfy≫-relationship is appropriate. Instead, the ≪trace≫-relationship is used to point out that a requirement is related in any general way to artefact it points to. In Fig. 3.2, a typical example of an application of the ≪trace≫-relationship is shown: The requirements related to the ON/OFF flashing periods for the turn indication control system are based on German traffic law. The ≪trace≫-relationship just points from the requirements to this document.

We won't consider the trace relationship further in the remainder of this session, because its use is quite informal, so that trace relationships are never evaluated for automated code generation or automated generation of tests.

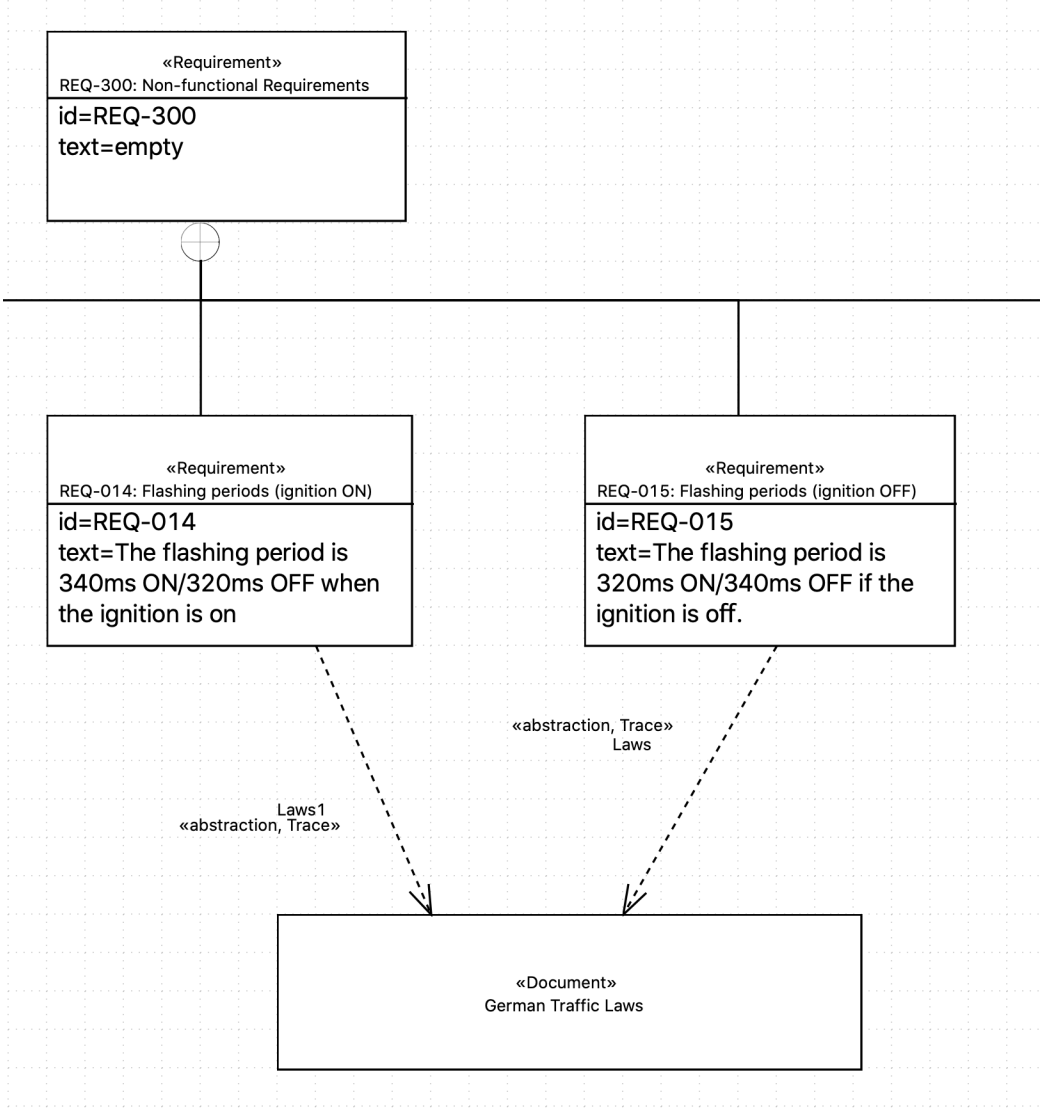

Figure 3.2: Example of a ≪trace≫-relationship.

3.2 Choice of Clients in Satisfy Relationships

Depending on the nature of the requirement, different choices of clients satisfying the requirement need to be made. Some choices are quite obvious.

1. A behavioural requirement can never be satisfied by a collection of satisfy relationships containing only structural model elements as clients.

2. A structural requirement can never be satisfied by a collection of satisfy relationships containing only behavioural model elements as clients.

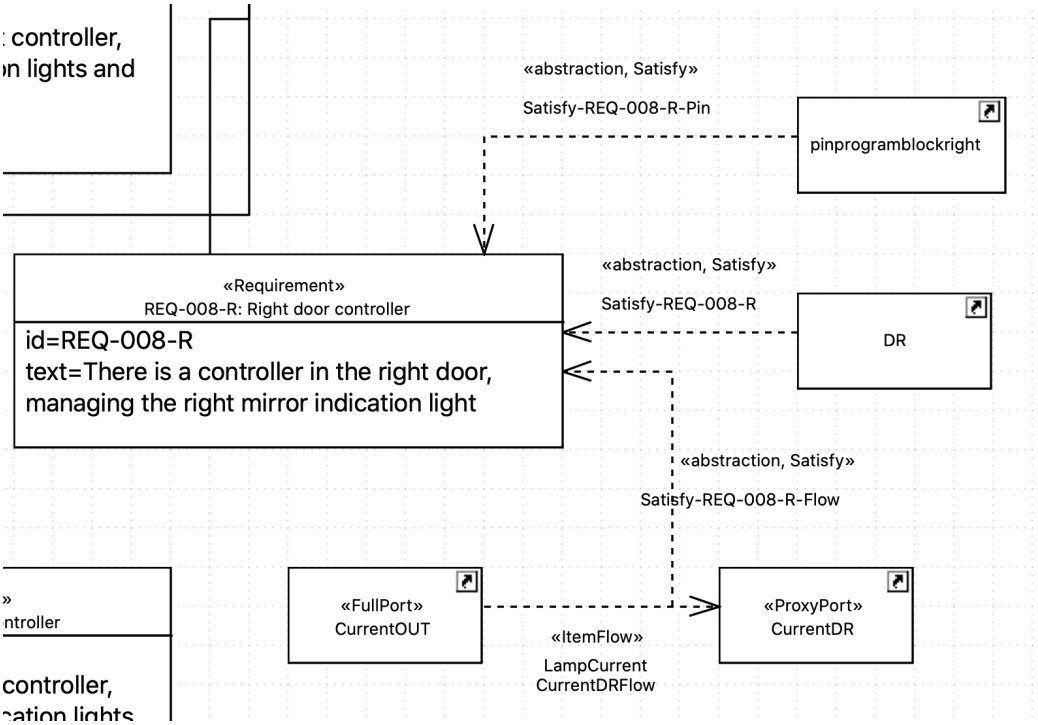

Figure 3.3: Tracing a requirement to parts and flows.

Using parts instead of blocks

For structural requirements, a distinction between blocks and parts (that is, instances of blocks) will usually become necessary. To give an example, consider the structural requirement REQ-008-R in Fig. 3.3. This requirement states that a door controller must be allocated in the right door. Therefore, it would not suffice to use the block DoorController as the client of the ≪satisfy≫-relationship. Instead, we need the part property DR : DoorController of the TurnIndicationController as client, because this confirms that there will be a door controller instance at the right-hand side. Now here, we should be a bit more elaborate and show that this door controller also "knows"

that it is integrated in the right door. This is ensured by the part property pinprogramblockright : PinProgramBlockRight of TurnIndicationController. Finally, the requirement states that this door controller should manage the right mirror indication light. This is ensured by the item flow connecting the output port CurrentOUT to the proxy port CurrentDR. This results in the collection of ≪satisfy≫-relationships shown in Fig. 3.3.

Tracing behaviour

When it comes to behaviour, there is usually at least one *part* involved as client in the collection of ≪satisfy≫-relationships: This part is an instance of the block possessing the owned classifier behaviour implementing the requirement. Tracing to this part means that "the behaviour has not only been modelled in some state machine (and then forgotten), but it is also allocated in a suitable place of the system design". The proper behaviour may be expressed by a collection of

- state machines,
- activities,
- interactions,
- operations.

We focus here on state machines. Typically, only some parts of a state machine contribute to the satisfaction of a given requirements. This means, that single transitions, states, or actions need to be related to the requirement. Consider, for example, requirement REQ-16 in Fig. 3.4 which states that OFF commands are sent by the rear controller. The part responsible for executing the state machine implementing this behaviour is candriver : CANdriver. The state machine CANdriver has two transitions reacting to OFF commands, these are linked to the requirement as well, so that one structural modelling element and two behavioural elements together implement the requirement.

Note that in the requirements diagram, the transition *names* are used in the client boxes. This show again that it is useful to associate names with transitions.

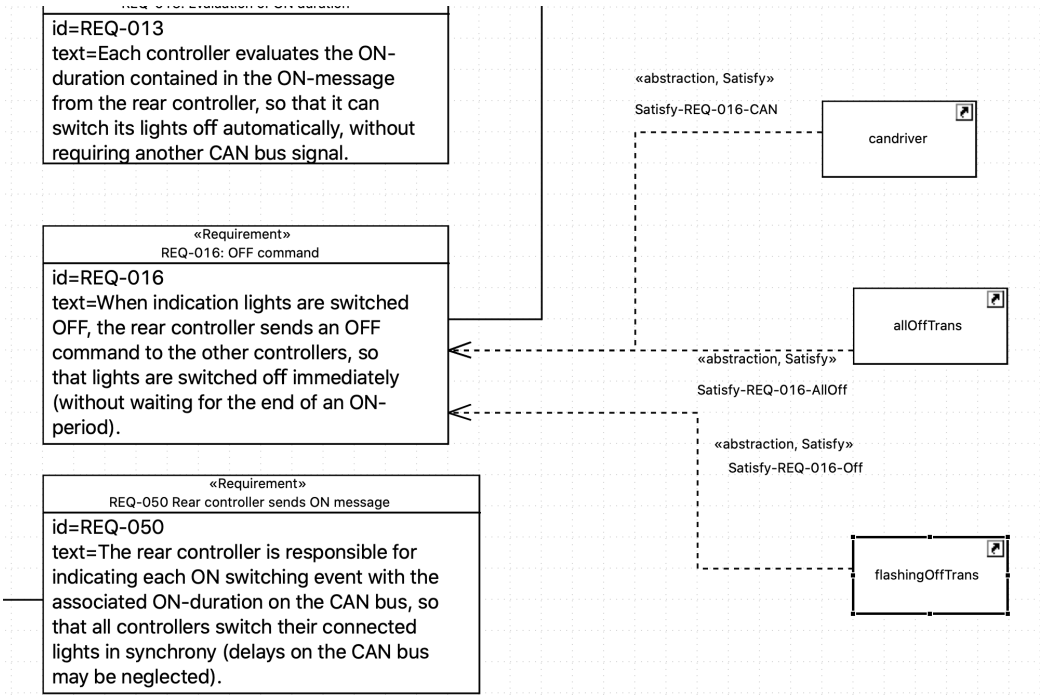

Figure 3.4: Tracing a behavioural requirement to parts and transitions.

Trace to opaque behaviour

In several situations, it is necessary to refer to opaque behaviour or opaque expressions, because the modelled behaviour can only be fully understood when analysing the actions and conditions specified there. An example is given in Fig. 3.5, where the requirement for left-hand-side flashing is specified. Apart from the part executing the ControlLogic state machine and the transitions involved, the behaviour is determined by the entry action LRFlashingAssignment, which has the opaque behaviour

```
1 FlashCmdOut.cmd = ON;
2 FlashCmdOut.applyLeft = (LeverPositionIN == LEFT);
3 FlashCmdOut.applyRight = (LeverPositionIN == RIGHT);
```

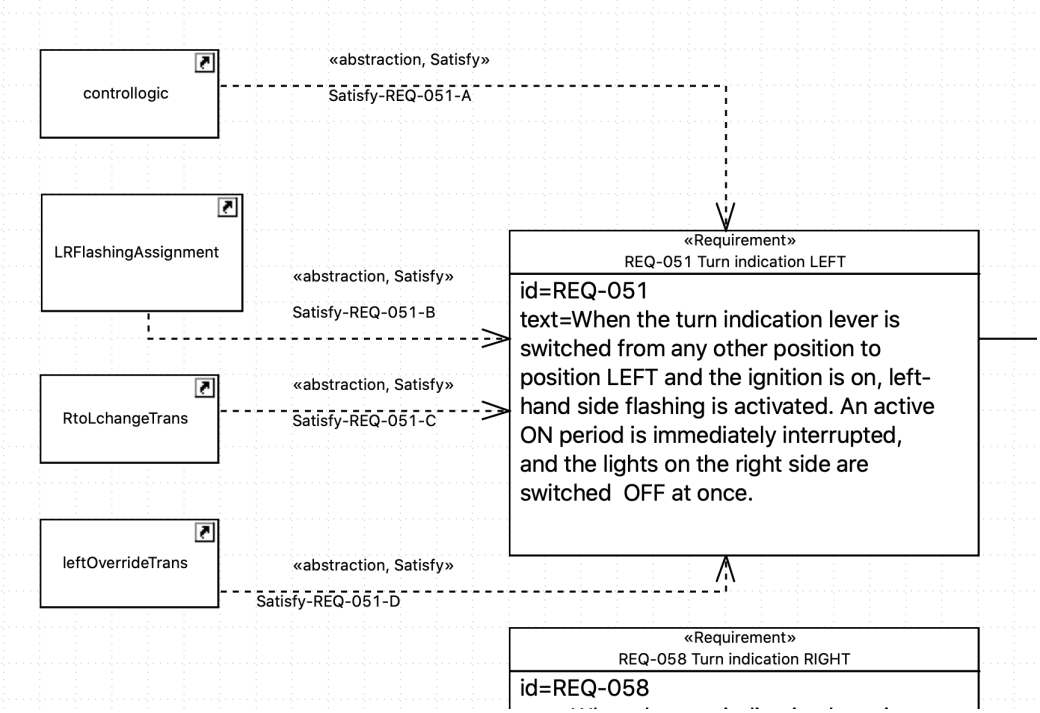

Figure 3.5: Tracing a behavioural requirement to parts, transitions, and opaque behaviours.

3.3 Different Ways to Represent Traceability Information

In the previous section, we have presented the standard way to represent satisfaction relations: the clients of ≪satisfy≫-relationships are dragged and dropped to requirements diagrams and linked there to the associated supplier requirement. While this is the preferable solution for small collections of requirements and clients, the diagrams may become quite cluttered when many requirements with numerous clients are involved.

Callout notation

Some tools allow for referencing requirements in other diagrams (so-called **callout notation**), drawing a ≪satisfy≫-relationship, for example, from a transition to a requirements callout box in a state machine diagram [3, p. 184]. This notation style, however, is not supported by the Papyrus tool.

satisfiedBy compartment

Specifying traceability information without graphical representation is possible by creating a satisfy association from the context menu SysML Relationship ⟶ Satisfy of the *client model element*. There, the *supplier requirement* can be selected, and the client is registered in the Satisfied By compartment of the requirement. This avoids cluttered graphical representations, but one misses a condensed representation of the satisfied-by information.

Tabular views

To overcome this problem, SysML introduces tabular representations of requirements and their related elements, such as the clients of the ≪satisfy≫-relationship, or other requirements related by the associations described in the next section [3, 16.3.1.5]. In the Papyrus tool, these tables can be created using the New Table ⟶ Generic Tree Table command on a higher-level requirement R. The table will then contain all the the requirements underneath R. Then it is possible to select additional columns using the Columns ⟶ Create/Destroy Columns command in the context menu of the table itself. This is explained in the video accompanying this session. An example of such a table is given in Fig. 3.6.

	A	B	C
	name : String [0..1]	id : String [1]	/satisfiedBy : NamedElement [*]
/ownedElement			
REQ-011: Controllers communicate via CAN bus	REQ-011: Controllers communicate via CAN bus	REQ-011	
REQ-100-A: Interfaces to the operational environment	REQ-100-A: Interfaces to the operational environment	REQ-100-A	
/ownedElement			
REQ-100-A1 Indication Light Interfaces	REQ-100-A1 Indication Light Interfaces	REQ-100-A1	
REQ-100-A2 Dashboard interfaces	REQ-100-A2 Dashboard interfaces	REQ-100-A2	
/ownedElement			
REQ-002-L Left dashboard indication LED	REQ-002-L Left dashboard indication LED	REQ-002-L	
REQ-002-R Right dashboard indication LED	REQ-002-R Right dashboard indication LED	REQ-002-R	
REQ-023 Emergency flashing button	REQ-023 Emergency flashing button	REQ-023	EmerSwitchPressed
REQ-020 Battery interface	REQ-020 Battery interface	REQ-020	
REQ-021 Ignition interface	REQ-021 Ignition interface	REQ-021	
REQ-022 Turn Indication Lever interface	REQ-022 Turn Indication Lever interface	REQ-022	
REQ-100-B Internal interfaces	REQ-100-B Internal interfaces	REQ-100-B	
REQ-100-C Controllers	REQ-100-C Controllers	REQ-100-C	
/ownedElement			
REQ-006: one front controller	REQ-006: one front controller	REQ-006	
REQ-008-L: Left door controller	REQ-008-L: Left door controller	REQ-008-L	
REQ-008-R: Right door controller	REQ-008-R: Right door controller	REQ-008-R	DR, <Information Flow> CurrentDRFlow, pinprogramblockright
REQ-007: one rear controller	REQ-007: one rear controller	REQ-007	

Figure 3.6: Traceability matrix created as Papyrus Generic Tree Table.

Recommendation for displaying traceability information

Summarising, we recommend the following two approaches for representing traceability onformation in the model.

1. If graphical representation is important[1], use requirements diagrams

[1]For example, if walkthroughs for checking how requirements have been implemented in the model are performed with several participants.

showing only a small number of requirements, so that they do not become cluttered, when clients and ≪satisfy≫-relationships are displayed on the same diagram.

2. If graphical representation is not important, create ≪satisfy≫-relationships by means of the SysML Relationship ⟶ Satisfy command in the context menu of the client, and collect all ≪satisfy≫-relationships in a table showing requirements and associated clients in separate columns.

Please note that these options are not an "either-or" choice: you can always choose Approach 2 and add requirements diagrams with graphical display of clients and ≪satisfy≫-relationships for some specific requirements only.

Traceability matrix

In general, and independent of the SysML formalism, tabular representations of requirements and related artefacts are called **traceability matrix**. Of course, the term "matrix" is a bit naive, since more complex structures involving several tree tables are needed to represent all related information in a comprehensive view. In this course, we focus on other model elements acting as clients to the ≪satisfy≫-relationship to be displayed in traceability matrices. In a complete system development involving HW and SW development, reviews, analyses, and tests on unit level, software integration level, HW/SW integration level, and system level, there are far more artefacts to be traced back to requirements with different relationships not used in this course, but briefly discussed in the next section.

For example, a system test case can be represented in SysML by means of an interaction. To express that a requirement R is tested by interaction I, a ≪verify≫-relationship with I as client and R as supplier can be used.

SysML model as central source of traceability information?

It is currently discussed in the engineering communities whether it is advisable to capture *all* artefacts of a system development in a SysML model, which would allow us to perform *complete* requirements tracing inside the system model. The SysML syntax and semantics is expressive enough to do this. There are, however, some serious obstacles to be overcome until such an ideal situation could be reached:

- The SysML tools currently available (both commercial and free open source) are far too weak to provide a satisfactory user experience for managing all artefacts in the development life cycle.

- There is a separate market for **requirements management tools**[2] which provide a database for managing requirements, other artefacts, and traceability relations between them.

As of today, separate tools for requirements management and tracing on the one hand, and modelling on the other hand, are therefore applied in industry.

[2]See, for example, `https://www.ibm.com/support/knowledgecenter/SSYQBZ_9.5.0/com.ibm.doors.requirements.doc/topics/c_welcome.html` for one of the commercial tools focused on requirements management.

Chapter 4

Requirements and Related Associations

4.1 Association summary

Known associations

By now, we have become acquainted with four requirements-related associations:

1. The **containment** relationship, denoted by a "crosshair" $\oplus$ is used to specify that one requirement is comprised of other, more specific (potentially atomic) ones.
2. The **satisfy** relationship has been introduced above to trace supplier requirements back to client model elements.
3. The **trace** relationship has been briefly mentioned above, and it is used to link requirements to other artefacts having only a weakly specified relationship to the requirement.[1]
4. The **verify** relationship (also briefly mentioned above) is used to link a verification artefact (typically, a test case or an analysis document) to a requirement, stating that the proper implementation of the requirement is checked by the artefact.

New associations

To this little zoo of association animals, the following relationships need to be added, and they will be explained in the following paragraphs.

[1] So the «trace»-relationship must never be used for stating that requirements are implemented by other model elements.

5. The **DeriveReqt (derive requirement)** relationship connects a client requirement to a supplier requirement and states that the client is a more concrete requirement which has been "invented" to realise the supplier requirement.

6. The **refine** relationship makes a requirement more precise by associating a model at the client end explaining the requirement at the supplier end.

7. The **copy** relationship states that the client requirement is just a verbatim copy of the supplier requirement "living" in another project context or any other context (e.g. a standard).

4.2 The Derive Requirement Relationship

The ≪DeriveReqt≫-relationship is typically introduced during the modelling process, together with a new design-related requirement as client of this relationship. The client is a requirement which typically states one of several possible alternatives for realising the supplier requirement. In the standard [7] for avionic systems development, the client requirement is therefore called the **derived requirement**. It should be emphasised that the derived requirement is still informal, just as the supplier requirement.[2]

It is good style to extend the ≪DeriveReqt≫-relationship with a **rationale** if the effectiveness of the derived requirement is not immediately clear from its description.

An important distinction between original and derived requirements is that the original requirements are typically specified by the customer, while the derived requirements are identified by the supplier team during the modelling process. Stakeholders

The utilisation of derived requirements is exemplified in Fig. 4.1.

[2]For formalising a requirement, the ≪refine≫-relationship discussed next can be used.

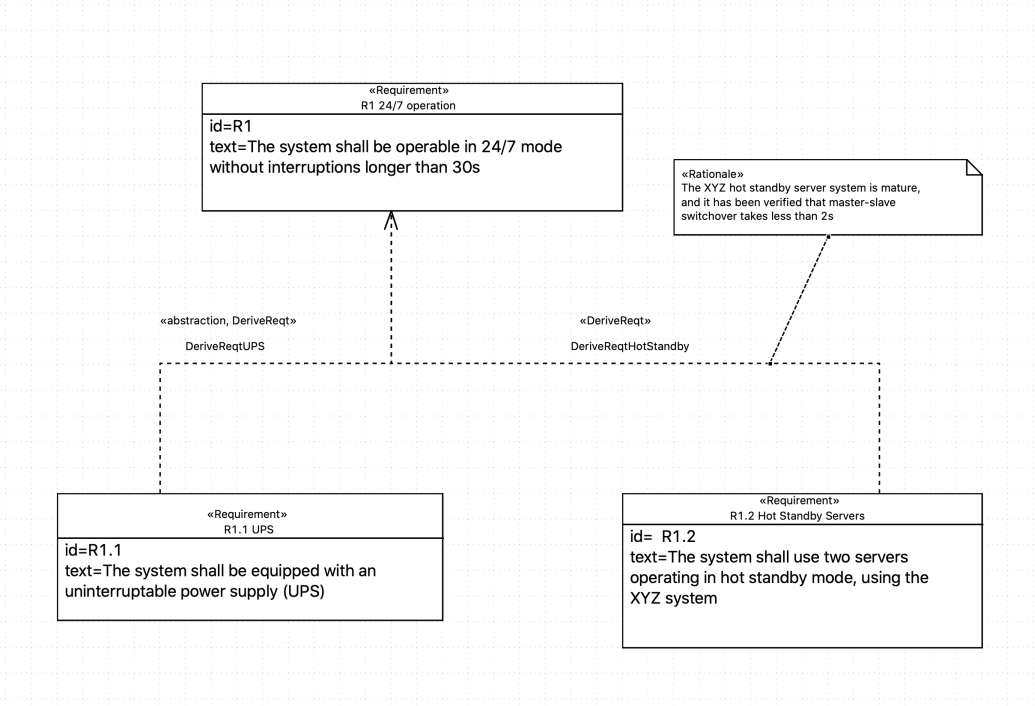

Figure 4.1: Example of supplier requirement and two derived requirements.

4.3 The Refine Relationship

The ≪refine≫-relationship complements the ≪DeriveReqt≫-relationship in the sense that it *formalises* the supplier requirement by providing a sub-model at the client end of the relationship. For example, the supplier requirement could state *"the sender uses the HDLC transmission protocol"*, and the client could be a state machine modelling the HDLC protocol behaviour.

Note that the client sub-model is usually *not* a part of the system design, the latter can be realised by quite different modelling elements. For example, the protocol implementation could be modelled by concurrent activities instead of the state machine refining the requirement. Moreover, the refining sub-model needs no allocation anywhere as a part of the target system.

4.4 The Copy Relationship

The ≪copy≫-relationship has the effect that the copied requirement becomes part of the project, but *is read only*: since it has been intentionally been copied verbatim from another project (for example, with the intention to re-use an existing solution), it is not allowed to change a copied requirement.

4.5 Obsolete Stereotypes

Please note that the stereotypes for specialised requirements listed in [1, Table 13.1] (as, for example, ≪functionRequirement≫) are no longer supported in SysML Version 1.6. With today's SysML, one would introduce project-specific specialisations by creating new stereotypes in a project-specific **profile** [3, Chapter 17]. Profile creation, however, is beyond the scope of this course.

Chapter 5

Model-based Requirements-driven Testing

5.1 Terms and Definitions

The most obvious benefit of requirements tracing is the verification that no requirements have been forgotten in the system design. But there are more advantages coming from tracing requirements to elements of the system design. The most important of these advantages is **model-based testing (MBT)**, in particular, **requirements-driven testing**. We focus here on **functional testing**, that is, on test cases suitable to check that the behaviour of the **system under test (SUT)** conforms to the behaviour in the specified reference model. Model-based structural testing or non-functional testing is also possible, but our focus here is on functional testing only.

Functional MBT

In MBT, various strategies are applied to derive "useful" test cases from design models specified in SysML or other modelling formalisms. In this context, "useful" means one of two things:

1. The resulting test suites are suitable for detecting *any* deviation of the system behaviour from the specified model expressing the *expected* SUT behaviour. This approach is relevant when testing for **model conformance** of the SUT with *every* aspect of the model. Conformance testing is applied, for example, when verifying communication systems exchanging data according to certain protocols.

   Conformance testing

2. The resulting test suites are suitable for detecting *violations of a given*

   Requirements-driven testing

requirement. This approach is called **requirements-driven testing**. It is applied in the safety-critical systems domain, where safety-related requirements need to be traced to the system design and from there to HW design, SW design, code and verification artefacts, like tests, reviews, and analyses.

For the remainder of this section, we will explain the basic approach to requirements-driven MBT. We will explain this in an informal, intuitive way. Please note, however, that MBT has a solid scientific foundation, and it is still a very active research field. Moreover, many companies world-wide currently introduce MBT into their development, verification, and validation processes. This is mainly motivated by the fact that MBT can be automated in a way that test procedures including the test data and the checks to be performed can be automatically generated from models.

- For the scientific foundations of MBT, please consult our lecture notes [6] and visit our lecture on test automation.
- For MBT automation methods, please read [4, 5].

5.2 Requirements-driven Test Case Identification

We have learned how to trace structural and behavioural model elements to requirements, using the ≪satisfy≫-relationship. Intuitively speaking, every behavioural model element contributing to the implementation of a requirement should be **covered** in at least one of the test cases.

Covering a model element

The concept of **model coverage** needs some explanation: The design model is associated with a **behavioural semantics** which we have only dicussed in an informal way, but which can be formalised as shown, for example, in [2, Chapter 11]. Intuitively speaking, the model behaviour is specified by the state machines involved and the guard conditions, entry actions, and operation calls evaluated or executed by the state machines. Placing data or signal events on system interfaces leads to state machine transitions being triggered by means of signal events or change events. Moreover, state machine transitions can be triggered by time events. As a consequence, we can **simulate** the state machine behaviour, either by hand, or by automatically

transforming the model into simulation code. The latter technique will be explained during the next sessions.

During a simulated state machine execution, every (partial) execution step corresponds to

- a guard condition being evaluated,
- a trigger condition being evaluated,
- an entry action/exit action/do action or transition action being executed, or
- a block operation called by the state machine being executed.

Therefore, we can say that these behavioural model elements have been covered during a simulation.

As a consequence, testing a requirement in a comprehensive way means covering the related behavioural model elements in a comprehensive way.

5.3 Test Cases

Following [7], a **test case** consists of input data to the SUT and checking conditions for expected results, i.e. expected SUT reactions. For reactive control systems, input data usually consists of *sequences of input tuples*, together with timing conditions stating when the next tuple should be placed on the SUT input interfaces. The input tuples are usually called **input vectors**, though there are no mathematical vector spaces involved.

For system testing, input vectors need to refer to full ports, that is, to HW interfaces of the SUT. In software unit testing and software integration testing, input tuples may refer to the software ports interfacing to a state machine or to the parameters to be passed in a block operation call.

For expected results, we can simulate the model in **back-to-back** fashion simultaneously with the SUT: every input vector placed on the SUT input interface is also placed on the model simulation interface. Then the SUT outputs at full ports are compared to the corresponding outputs of the simulation.

For system test cases, it has to be determined which input vectors placed on the SUT HW interfaces are suitable to finally cover the desired model element(s) in the appropriate order. Based on the formal semantics available

for SysML models, this can be calculated by MBT tools in an automated way.[1]

5.4 System Test Case Creation Example

5.4.1 The Requirement to be Tested

To illustrate the requirements-driven MBT, let us consider the following requirement specified for our turn indication system case study.

REQ-051 When the turn indication lever is switched from any other position to position LEFT and the ignition is on, left-hand side flashing is activated. An active ON period is immediately interrupted, and the lights on the right side are switched OFF at once.

5.4.2 The Model Elements to be Covered

Model elements to be covered

In Fig. 5.1, the model elements contributing to the satisfaction of this requirement are shown. Element contrologic is a structural element (part) shown that the behaviour has been deployed (on the rear controller). The other elements are behavioural and need to be covered by associated test cases:

- LRFlashingInitialTrans is the transition responsible for the activation of left or right flashing when the ignition is on and no flashing activity is currently active.
- leftOverrideTrans is the transition overriding emergency flashing of flashing on the left-hand side is activated.
- RtoLchangeTrans is the transition initiating the change from right-hand side flashing to left-hand side flashing.
- LRFlashingAssignment is the entry action (opaque behaviour) setting the output interface of the control logic according to the turn indication lever position.

[1] See, for example, the tool RT-Tester developed in my company `https://www.verified.de/products/model-based-testing/`

Please look these model elements up in the turn indication controller model (state machine ControlLogic).

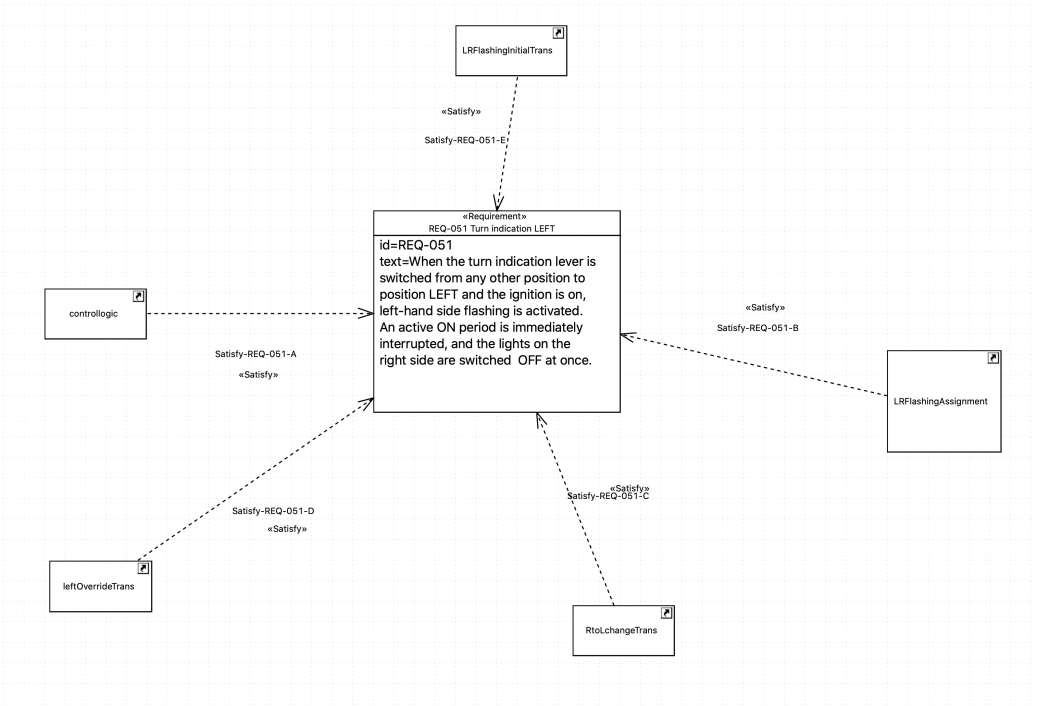

Figure 5.1: Model elements contributing to the satisfaction of requirement REQ-051.

5.4.3 Test Case Identification

Test case identification

We see that we need three test cases for covering these model elements:

1. **TC-051-001.** The first test case should cover LRFlashingInitialTrans which will always lead to the consecutive execution of the LRFlashingAssignment entry action.

2. **TC-051-002.** The second test case should cover leftOverrideTrans which will also lead to the consecutive execution of the LRFlashingAssignment entry action.

3. **TC-051-003.** The third test case should cover transition RtoLchangeTrans which will also lead to the consecutive execution of the LRFlashingAssignment entry action.

5.4.4 Test Case Elaboration: TC-051-001

To elaborate a sequence of input vectors suitable for TC-051-001[2], we proceed according to the following steps.

Step 1: Determine start state. We need to decide from which state machine state of ControlLogic the test case should be activated. A good starting point for test case TC-051-001 would be the simple state NO_FLASHING. Fig. 5.2 shows possible transitions for reaching this state: from the initial pseudo state, we can pass, for example, through transition sequence[3]

> initTrans $\longrightarrow$ emerOffInitialTrans $\longrightarrow$ ignOnInitialTrans $\longrightarrow$ noLRflashingInitialTrans

[2]Note that these sequences are not uniquely determined!
[3]We are using the transition names here.

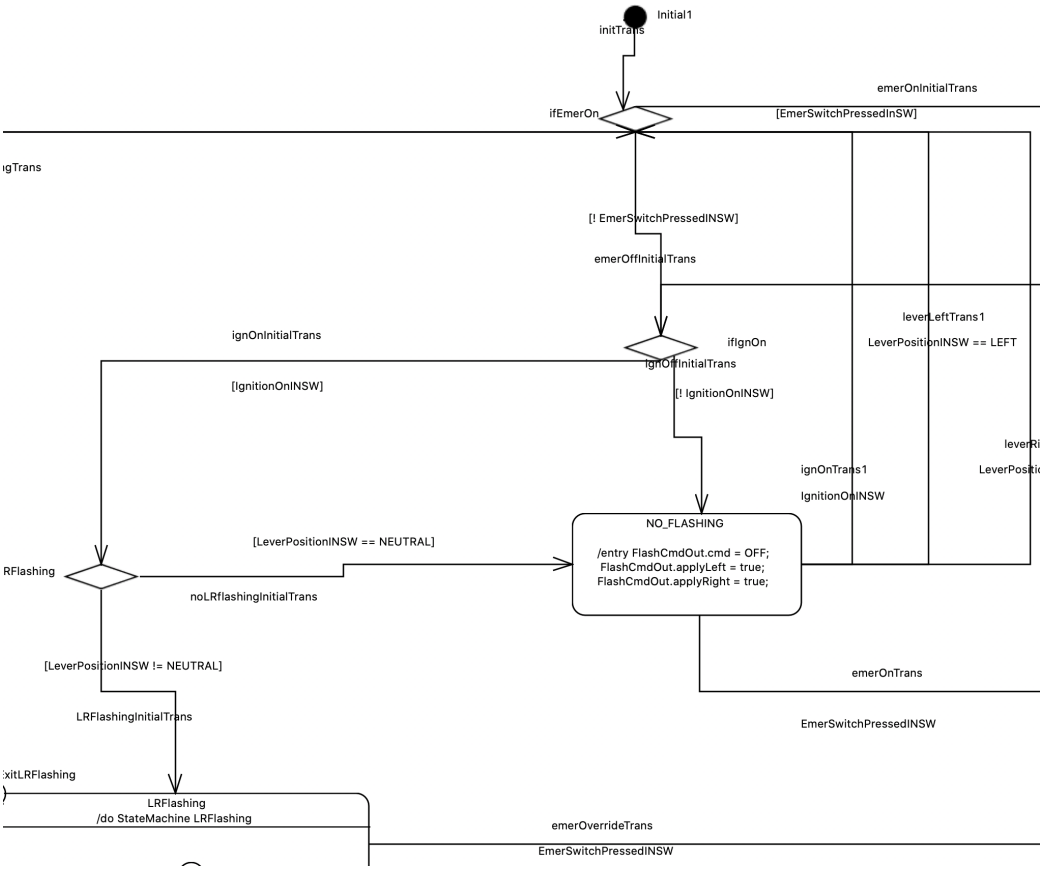

Figure 5.2: Initial transitions leading to state NO_FLASHING.

These sequences connect pseudo states and end up in the simple state NO_FLASHING. Therefore, all condition for taking these transitions need to be fulfilled at the same time.[4] This results in the following logical condition for reaching NO_FLASHING in the beginning of the test case execution:

$$\begin{aligned} \text{Condition1'} \quad \equiv \quad & \neg \text{EmerSwitchPressedINSW} \wedge \\ & \text{IgnitionOnINSW} \wedge \\ & \text{LeverPositionINSW} = \text{NEUTRAL} \end{aligned}$$

Now this condition refers to ports of the ControlLogic state machine. In

[4]Remember that no time passes when going through pseudo states.

system testing, we cannot access these software ports in the SUT directly. Instead, we need to stimulate suitable full ports at HW interfaces, in order to stimulate the desired inputs at the software ports. For identifying suitable HW interfaces, the internal block diagrams are helpful. In the RearControllerIBD, we see that the full ports EmerSwitchPressedIN, LeverPositionIN, IgnitionOnIN are connected to the software ports EmerSwitchPressedINSW, LeverPositionINSW and IgnitionOnINSW, respectively. Consequently, formula Condition1' is revised by inserting the full ports in

$$\begin{aligned} \text{Condition1} \quad \equiv \quad & \neg \mathsf{EmerSwitchPressedIN} \wedge \\ & \mathsf{IgnitionOnIN} \wedge \\ & \mathsf{LeverPositionIN} = \mathsf{NEUTRAL} \end{aligned}$$

Step 2: Determine transition sequence to target transition. From the starting state, the transitions to be covered for reaching the target transition are identified. A possible path (see Fig. 5.2) for reaching LRFlashingInitialTrans from state NO_FLASHING is

$$\mathsf{leverLeftTrans} \longrightarrow \mathsf{emerOffInitialTrans} \longrightarrow \mathsf{ignOnInitialTrans} \longrightarrow \mathsf{LRFlashingInitialTrans}$$

Again, this transition sequence is associated with a logical condition

$$\begin{aligned} \text{Condition2'} \quad \equiv \quad & \neg \mathsf{EmerSwitchPressedINSW} \wedge \\ & \mathsf{IgnitionOnINSW} \wedge \\ & \mathsf{LeverPositionINSW} = \mathsf{LEFT} \end{aligned}$$

which is in turn associated with the following condition about full ports of the rear controller:

$$\begin{aligned} \text{Condition2} \quad \equiv \quad & \neg \mathsf{EmerSwitchPressedIN} \wedge \\ & \mathsf{IgnitionOnIN} \wedge \\ & \mathsf{LeverPositionIN} = \mathsf{LEFT} \end{aligned}$$

The triggered transition sequence automatically leads to the execution of the entry action LRFlashingAssignment in state LRFlashing.SendLRFlashCmd.

Step 3: Determine the expected results to be checked. On system test level, only the lamp outputs and LIN bus outputs to the dashboard LEDs can be observed. Therefore, we need to trace the expected effect of the entry action LRFlashingAssignment on the lamps and LEDs. To this end, the internal block diagrams with their ports and flows have to be carefully analysed to come up with the result that the effect should become visible at full ports CurrentRout, CurrentLout (rear controller), CurrentFLout, CurrentFRout (front controller), DL.CurrentOut, DR.CurrentOut (door controller), and on the LIN bus. In the flows to lamp outputs, we detect that the current is only provided if the battery voltage is in range. Otherwise, the current is set to zero by the power sources, and the expected effect will not be observable. This induces an invariant condition for this test case:

$$\text{Invariant} \quad \equiv \quad \mathsf{BatVolIN} \in [10, 15]$$

Finally, also timing conditions need to be considered when checking the expected results. In our case, the ON/OFF flashing periods (340/320ms, since the ignition is on) need to be monitored, so after having made Condition2 true, we still need a checking period of at least 660ms.

Step 4: Write the test script. Finally, we can turn the sequence of conditions plus the invariants into a test script. For this example, the following script (with this or any other tool-specific syntax) would apply:

```
1  // Write initial values to input ports of the SUT
2  SET BatVolIN = 12.0;
3  SET LeverPositionIN = NEUTRAL;
4  SET EmerSwitchPressedIN = false;
5  SET IgnitionOnIN = true;
6
7  RESET; // Reset the SUT
8
9  WAIT 5s;
10
11 // Check that lamps are OFF
12 ASSERT CurrentRout == 0 and CurrentLout == 0 and DL.CurrentOUT == 0 and ...
13
14 WAIT 5s;
15
16 SET LeverPositionIN = LEFT;
17 // Check expected results and leave 5ms slack for SUT
18 ASSERT CurrentRout == 2 and CurrentLout == 2 and ... WITHIN 5ms;
19
20 WAIT 340ms;
21 ASSERT CurrentRout == 0 and CurrentLout == 0 and ... WITHIN 5ms;
22
```

```
23 // check a few more flashing periods ...
24 END OF TEST
```

Step 5: Decide about alternative test cases. The availability of a model facilitates the identification of further test case variants. These are always associated with alternative transition paths. For this test case, we could, for example, start with ignition switched off, which would lead to another path to state NO_FLASHING. Then we could *first* put the lever into LEFT position and *then* switch on the ignition. These considerations lead to the following more general observations:

- MBT helps extremely well to find meaningful test cases.
- Tool support is very desirable, since the enumeration of alternative transition paths and specification of associated logical conditions is quite tedious to perform manually.
- It is very hard to come up with these interesting test cases without using a model.

Chapter 6

Questions and Exercises

6.1 Questions

6.1.1 How Many Satisfy Links do we Need?

Consider again Fig 3.1, where a proxy port is shown to contribute to the realisation of requirement REQ-023. In principle, it would be ok to link all the other ports and also the flows associated with this interface via the ≪satisfy≫-relationship to the requirement, such as ports EmerSwitchPressedIN of block RearController, EmerSwitchPressedInSW of block ControlLogic, and flows EmerSwitchStatus, emerFlow. Why is it allowed to omit these additional associations?

6.2 Exercises

6.2.1 Derived Requirements

Extend the existing requirements package by several derived requirements, that have been set up when making certain design decisions, but have not yet been added to the requirements package. Draw the ≪DeriveReqt≫-relationship between the new requirement and its supplier in one of the existing requirements diagrams.

6.2.2 Full Requirements Tracing for the Turn Indication Model

Please trace all requirements back to elements of the realisation model, using the «satisfy»-relationship. Display the relationship between requirement suppliers and model element clients by means of (several) generic tree tables.

6.2.3 Requirements-driven Test Case Specifications

Take three behavioural requirements of the turn indication model for which you have performed complete requirements tracing and specify test cases for each of these requirements, as described in the example in Section 5.

Alternatively, work out the details for the two test cases TC-051-002 and TC-051-003 identified above.

Bibliography

[1] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML, Third Edition: The Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2014.

[2] Wen-ling Huang, Jan Peleska, and Uwe Schulze. Test automation support. Technical Report D34.1, COMPASS Comprehensive Modelling for Advanced Systems of Systems, 2013. Available under `http://www.compass-research.eu/deliverables.html`.

[3] Object Management Group. OMG Systems Modeling Language (OMG SysML), Version 1.6. Technical report, Object Management Group, 2019. http://www.omg.org/spec/SysML/1.4.

[4] Jan Peleska. Industrial-strength model-based testing - state of the art and current challenges. In Alexander K. Petrenko and Holger Schlingloff, editors, Proceedings Eighth Workshop on *Model-Based Testing,* Rome, Italy, 17th March 2013, volume 111 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–28. Open Publishing Association, 2013.

[5] Jan Peleska, Jörg Brauer, and Wen-ling Huang. Model-based testing for avionic systems proven benefits and further challenges. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, volume 11247 of *Lecture Notes in Computer Science*, pages 82–103. Springer, 2018.

[6] Jan Peleska and Wen-ling Huang. *Test Automation - Foundations and Applications of Model-based Testing*. University of Bremen, January 2017.

Lecture notes, available under http://www.informatik.uni-bremen.de/agbs/jp/papers/test-automation-huang-peleska.pdf.

[7] RTCA SC-205/EUROCAE WG-71. Software Considerations in Airborne Systems and Equipment Certification. Technical Report RTCA/DO-178C, RTCA Inc, 1140 Connecticut Avenue, N.W., Suite 1020, Washington, D.C. 20036, December 2011.