checsdm

Documentation for consistency management in heterogeneous control software design models

March 2019

Andrés Paz · Ghizlane El Boussaidi

Andrés Paz
andres.paz-loboguerrero.1@ens.etsmtl.ca
Ghizlane El Boussaidi(✉)
ghizlane.elboussaidi@etsmtl.ca

Table of Contents

List of Abbreviations

EMF **E**clipse **M**odeling **F**ramework
FCS **F**light **C**ontrol **S**oftware
HEV **H**ydraulic **E**lectro-**V**alve
MDE **M**odel-**D**riven **E**ngineering
LGCS **L**anding **G**ear **C**ontrol **S**oftware

1 Introduction

1.1 *checsdm*

checsdm[1] (Consistency of Heterogeneous Embedded Control System Design Models) is a systematic approach, based on model-driven engineering (MDE), for assisting engineering teams in ensuring consistency of heterogeneous design of safety-critical software. The approach is developed as a generic *methodology* and a *tool framework*, that can be applied to various scenarios involving different modelling languages and different design guidelines. The methodology comprises an iterative three-phased process:

1) An *elicitation phase* consisting in eliciting the *requirements* of the design scenario at hand in terms of: 1) determining the mix of modelling languages that are going to be used and how to use them depending on the system's nature and the languages' purposes, 2) identifying the mapping rules between the different modelling languages, 3) defining intra-model design guidelines, *i.e.* design guidelines specific to models in each language taken separately, and 4) defining inter-model design guidelines, *i.e.* design guidelines that concern cross-model constructs.

2) A *codification phase* consisting in using meta-level functionalities of the proposed tool framework to codify, as required: 1) metamodels of the various modelling languages used, 2) the mapping rules between the modelling languages, 3) the intra-model design guidelines, and 4) the inter-model design guidelines.

3) An *operation phase*, where the tools are applied to actual system designs. This phase can help refine the mapping rules and guidelines or identify new ones.

1.2 About this Report

This report offers documentation and examples useful in working with *checsdm* and *checsdm4uss*, an instantiation for UML, Simulink and Stateflow design models.

1.2.1 Audience

This report is intended for engineers wishing to use or extend *checsdm* or *checsdm4uss*. Engineers wishing to work with *checsdm4uss* are expected to know and understand UML, Simulink and Stateflow. Engineers wishing to extend *checsdm4uss* are expected to know and understand the Eclipse Modeling Framework (EMF), Viatra, Epsilon Object Language (EOL) and Epsilon Comparison Language (ECL).

1.2.2 What's in the Report

This report will cover all aspects of working with *checsdm* and *checsdm4uss*.

Chapter 2 presents *checsdm* in detail.

Chapter 3 presents *checsdm4uss*, the sample instantiation of *checsdm*.

Chapter 4 gives a usage example of *checsdm4uss* in the context of an avionics software.

Chapter 5 offers documentation on and usage guidance of the derived Eclipse toolchain for *checsdm4uss*.

[1] Pronounced "checks them".

2 *checsdm*

Recall the *checsdm* approach consists of an iterative three-phased process:

1) Elicitation phase
2) Codification phase
3) Operation phase

Figure 2.1 illustrates the general flow of the proposed *checsdm* approach.

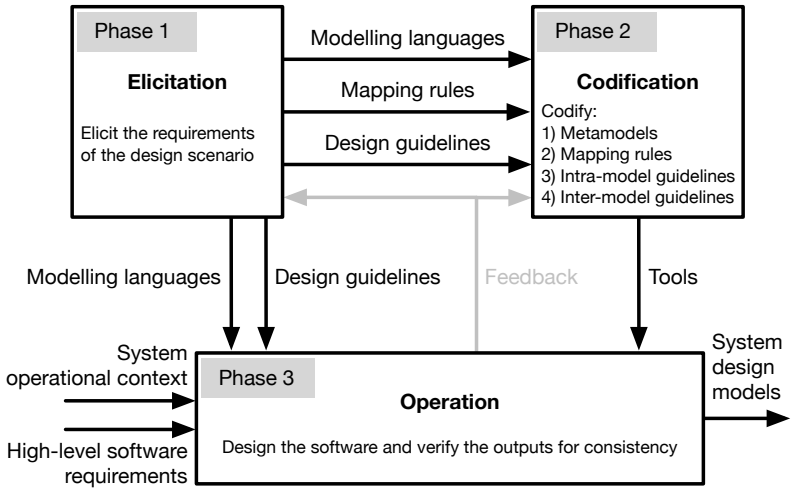


FIGURE 2.1: General flow of the *checsdm* approach.

2.1 Elicitation phase

Four steps make up the *elicitation* phase:

1. **Determination of the mix of modelling languages.** The combination of modelling languages that are going to be used is determined based on the systems' natures and the languages' purposes.
2. **Identification of mapping rules.** Correspondences between the constructs of the different modelling languages are identified and mapping rules are established.
3. **Definition of intra-model design guidelines.** Design guidelines specific to models in each language are introduced to direct the creation of the individual models.
4. **Definition of inter-model design guidelines.** Design guidelines that concern cross-model constructs are introduced to instruct on elaborated syntactical and semantical relations between the modelling languages.

Eliciting requires a careful analysis of the modelling languages' specifications to acquire a proper understanding of the system natures they can represent and how to do so.

The different mixes of modelling languages can be characterized according with 1) the degree of coverage of system elements, 2) the perspectives covered by the design models, 3) the level of abstraction at which the models represent the design, and 4) the degree of overlapping between the elements of the design models. Figure 2.2 presents this characterization in a feature diagram defining four dimensions: coverage of system elements (partial, complete),

design perspective (structural, behavioural), level of abstraction (same, different), and overlap of system elements (partial, complete). The first two dimensions are cloneable features with a minimum cardinality of 2, since at least two modelling languages are expected to be used. Models created using these modelling languages may cover different extents of the system at different perspectives.

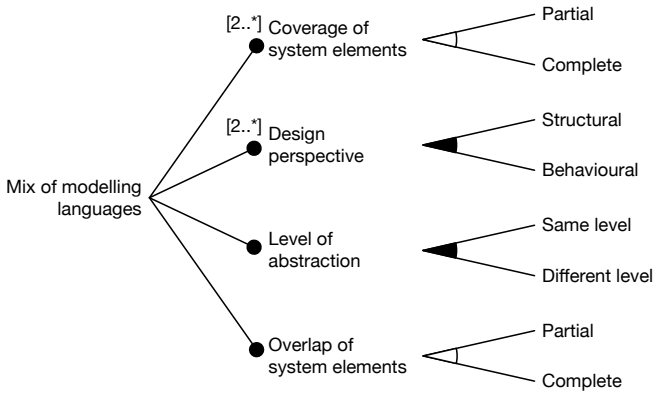


FIGURE 2.2: Feature model characterizing the mix of modelling languages.

A mapping rule relates a pair of model constructs from two modelling languages of interest (*e.g.*, UML and Simulink). The mapping rules are documented using the metamodel in Figure 2.3. Each mapping rule is given an identifier (ID) and a name for the relationship between the two model constructs. The set of *when* clauses specifies the conditions under which the current relationship holds. The set of *where* clauses specifies the relationships (*i.e.* other mapping rules) that must also hold whenever elements are participating in the current relationship.

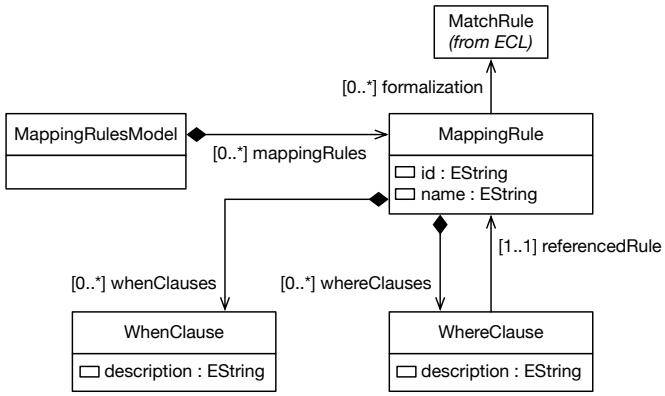


FIGURE 2.3: Mapping rules metamodel.

Guidelines are documented using the metamodel in Figure 2.4, which is derived from the textual template used for the MAAB style guidelines [1]. Each guideline is given an identifier (ID), a title and one of two possible priorities: *mandatory* or *recommended*. The description and the rationale fields provide the textual narrative for the guideline and its justification, respectively. A guideline is scoped to one or more modelling languages to which its guidance applies. This will categorize the guideline as intra-model or inter-model, respectively. A guideline may reference other guidelines as prerequisites or as referenced guidelines providing additional details where applicable. Guideline formalization/codification is explained as part of the *codification* phase.

2.2 Codification phase

The *codification* phase consists in using a tool framework to derive a toolchain that will assist engineering teams in ensuring consistency of the heterogeneous design models during the *operation* phase. Figure 2.5 presents the

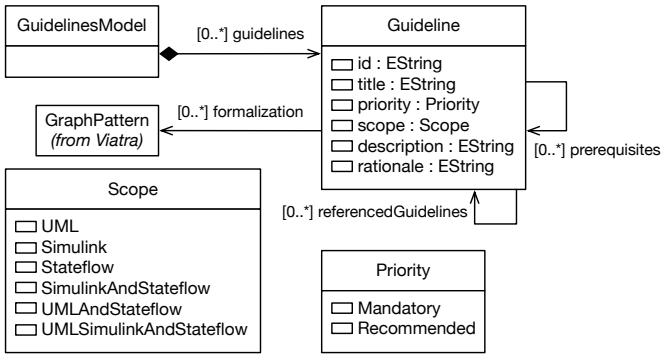


FIGURE 2.4: Guidelines metamodel.

high-level components of the tool framework. The framework is based on the Eclipse platform and its modelling technologies. Three steps make up the *codification* phase:

1. **Metamodelling.** Metamodels representing the various modelling languages are created using the Eclipse Modeling Framework (EMF). This step also involves developing custom model importers to transform original model files into their EMF representations.

2. **Codification of mapping rules.** Mapping rules are codified using the Epsilon Comparison Language (ECL) and Epsilon Object Language (EOL) [2].

3. **Codification of design guidelines.** Design guidelines (intra-model and inter-model) are codified using the Viatra Query Language (VQL) [3].

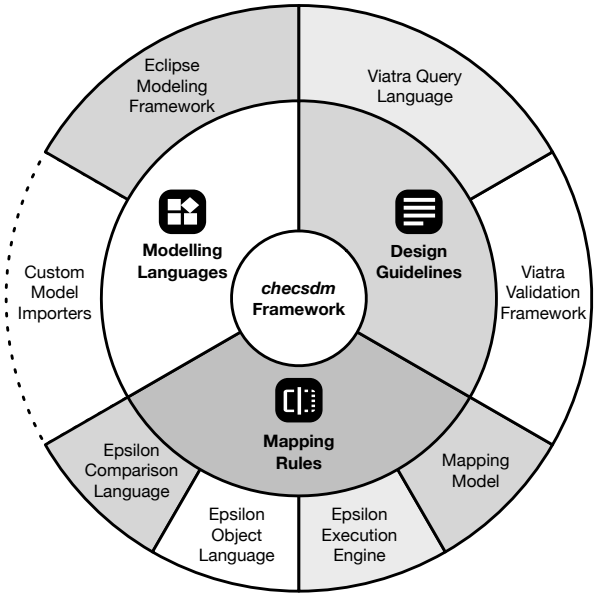


FIGURE 2.5: Tool framework.

Modelling languages are codified as Ecore metamodels using EMF. EMF provides metamodels for commonly used modelling languages, like UML and SysML. EMF-based metamodels for many other modelling languages can be downloaded and added into Eclipse. Modelling languages for which an EMF-based metamodel cannot be found, one must be created. This activity may require a careful analysis of the modelling languages' specifications in order to develop a technically valid metamodel (*i.e.* do not contravene the language specification). If a language specification is not openly available, a technically valid metamodel may have to be reverse-engineered. A custom

model importer tool, transforming from the original format into its equivalent EMF model, will need to be codified as well.

Mapping rules are codified as `MatchRules` using the Epsilon Comparison Language (ECL) [2]. Each mapping rule is, therefore, associated to a set of ECL `MatchRules`, as defined in the mapping rules metamodel (see Figure 2.3). ECL makes use of EOL (Epsilon Object Language) statements.

The tool framework includes a mapping metamodel for storing identified correspondences from the execution of mapping rules. In addition, the mapping metamodel includes facilities to flag consistency issues between the design models. Figure 2.6 presents this metamodel. It has been adapted from ECL's `MatchTrace` metamodel. The `MappingModel` comprises a collection of `Mappings`. A `Mapping` relates two model elements (`left` and `right`) belonging to two given design models. If both elements are consistent, according with the applicable `mappingRule`, then the `matching` flag is set to *true*, otherwise it is set to *false*. In this way consistency issues are flagged.

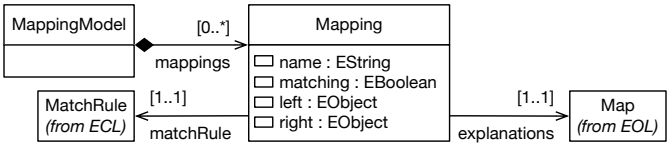


FIGURE 2.6: Mapping metamodel.

Mappings between two model elements, whether they are matching or not, are given some explanations containing the reasons for the obtained result. The `explanations` attribute is responsible for capturing explanations. Explanations contain the failed *when* or *where* clauses. In addition, when a clause fails, the model elements involved (*e.g.*, inputs, outputs, nested elements) are included to help clarify the result and locate the source of the consistency issue. Facilities for viewing and manually editing mappings are provided by a mappings editor generated by EMF from the mapping metamodel.

Design guidelines are codified as `GraphPatterns` using the Viatra Query Language (VQL) [3]. A `GraphPattern` query expresses a well-formedness constraint on a metamodel. Each design guideline is associated to a set of `GraphPattern` queries, as defined in the guideline metamodel (see Figure 2.4). Furthermore, the `GraphPatterns` must be annotated with the `@Constraint` annotation. These annotations are processed by the Viatra Validation Framework, which will automatically generate facilities to 1) link the `GraphPattern` to the given Eclipse editor ID, 2) initiate the execution of `GraphPatterns` on EMF instance models opened with the linked Eclipse editors and, 3) upon violations of the constraints, handle the creation and display of markers to the user in the Eclipse Problem View.

2.3 Operation phase

The *operation* phase covers the software design process and the part of the software verification process related to the verification of outputs from the software design process. Five steps make up the *operation* phase:

1. **Software design.** The software is designed from a given operational context and a set of high-level software requirements (HLRs) following the established mix of modelling languages and design guidelines.

2. **Verification of intra-model design guideline compliance.** The resulting design models are individually verified for intra-model guideline compliance.

3. **Mapping of design models.** Correspondences between the design models are identified and mappings are established between overlapping elements.

4. **Verification of inter-model design guideline compliance.** Using the mappings of the previous step, the design models are verified together for inter-model guideline compliance.

5. **Review and resolution of violations and consistency issues.** The flagged guideline violations and consistency issues are examined and handled accordingly.

Figure 2.7 illustrates the flow of this phase.

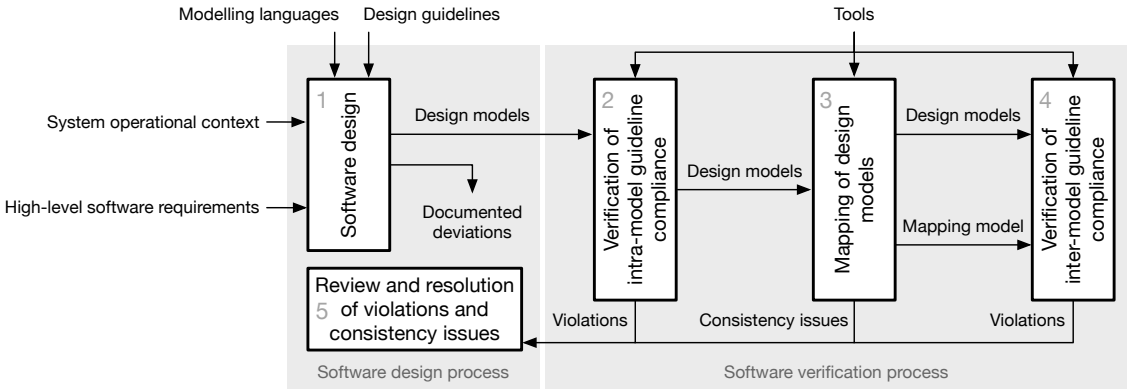


FIGURE 2.7: Flow of the *operation* phase.

The first step is the creation of the software design. This comprises both the software architecture and the detailed design. Software design is considered a generic step that must be duly adjusted with company-specific activities. However, two inputs coming from the *elicitation* phase are necessary prior to the start of the design process: the modelling languages that are to be used and the established design guidelines (both the intra- and inter-model). Two more inputs are required. These are dependent on the actual system to be designed: the system's operational context and the high-level software requirements (HLRs). Design models are developed from the set of HLRs taking into account the given operational context for the system. The design models must be developed following the established mix of modelling languages and design guidelines.

It is possible that guideline violations and consistency issues may have been introduced during the software design process since the different design models are likely developed independently following particular practices. Steps 2 through 4 verify the outputs of the design process and ensure their consistency. These steps make use of the tools codified during the *codification* phase. The verification of design guideline compliance is split into two steps. Step 2 verifies compliance of intra-model design guidelines, *i.e.* design guidelines that are scoped to a single modelling language. Step 3 involves the analysis of the design models in search for correspondences between their elements and identifying overlapping elements that must be consistent. The analysis results are stored in the mapping model. Step 4 verifies compliance of inter-model design guidelines, *i.e.* design guidelines that concern cross-model constructs, after the overlapping elements have been identified.

A feedback loop exists from each one of these steps to the design process when an analysis of the design models indicates a guideline violation or a consistency issue occurred and needs to be addressed. In step 5 the flagged guideline violations and consistency issues are examined and handled accordingly. This is an activity of the software design process and is intrinsically manual. It requires that the engineers review the outputs of the previous steps and determine the reported violations and consistency issues that will be handled and how.

The steps of the *operation* phase can be applied iteratively, until the transition criteria from the design process to the subsequent development activities (*e.g.*, coding, develop verification cases) has been met. Furthermore, the resulting mapping model from step 3 can be used along the design models to support the subsequent development activities.

3 *checsdm4uss*

This chapter describes in detail the concrete instantiation of the *checsdm* approach (see Chapter 3) in the context of its concrete implementation for ensuring consistency between UML, Simulink and Stateflow design models (*checsdm4uss*). The sets of guidelines and mapping rules for these languages, however, are not final. The inherent complexity of developing an heterogeneous design for an embedded control software as well as company-specific practices and the UML, Simulink and Stateflow modelling languages themselves, make it hard to determine all possible guidelines and mapping rules. The proposed guidelines and rules cover the most common overlaps between elements in design models created a mix of UML, Simulink and Stateflow that, when not dealt with a rigorous and systematic approach, lead to consistency issues. Additional guidelines and mapping rules can be added to expand the range of guidance and even to cover additional modelling languages.

3.1 *checsdm4uss*: Elicitation phase

The design scenario in *checsdm4uss* is characterized by employing different combinations of design models in UML, Simulink and Stateflow to describe the different aspects of the systems they develop. The mapping rules and design guidelines are not final. The inherent complexity of developing an heterogeneous design for an embedded control software as well as company-specific practices and the UML, Simulink and Stateflow modelling languages themselves, make it hard to determine all possible mapping rules and design guidelines. The proposed mapping rules and design guidelines cover the most common overlap cases between elements in design models created with a mix of UML, Simulink and Stateflow that, when not dealt with a rigorous and systematic approach, lead to consistency issues. Additional guidelines and mapping rules can be added to expand the range of mapping and guidance.

3.1.1 Mapping rules

Table 3.1 lists all the mapping rules.

Table 3.2 shows in detail mapping rule mr_0001 describing the relationship between UML classes/components and Simulink subsystem blocks/Stateflow charts. In Simulink, subsystem blocks gather blocks for the purpose of model organization and visual simplification, as well as for maximizing design reuse. Simulink subsystem blocks are semantically equivalent to UML components, which are considered modular autonomous units with hidden internals but well-defined interfaces that enable reuse. A UML component and a Simulink subsystem block are related *when* both of these elements have similar names. However, for this relationship to hold in its entirety, the inputs, outputs and nested elements of both constructs should also be related. Thus, other mapping rules (*i.e.* mr_0001, mr_0002, mr_0003, mr_0004, mr_0005) further describing such fine-grained relationships are referenced as *where* clauses.

Simulink blocks receive inputs through inports and then compute them to generate outputs through outports. In UML, the inputs for a component can come from either input parameters of operations in the component's provided interfaces, or from output parameters in the component's required interfaces. Figure 3.1 illustrates this situation. Both `Qux` and `Baz` are inputs of the UML component `Bar` (left of the figure) and the corresponding `Bar` Simulink subsystem block (right of the figure). However, in the case of the UML component, these two inputs are specified in two separate interfaces, `IFoo` and `IBar` respectively; `Qux` being an output parameter of a required interface and `Baz` being an input parameter of the component's provided interface. Mapping rules mr_0002 and mr_0003 capture such semantic equivalence. Similarly, outputs in a UML component can come from either input parameters of operations in the component's required interfaces, or output parameters of operations in the component's provided interfaces. Mapping rules mr_0004 and mr_0005 capture such semantic equivalence. Mapping inputs and outputs in this way resembles more closely how software is modelled using UML.

TABLE 3.1: Summary of the mapping rules.

ID	Name
mr_0001	UML class / component and Simulink subsystem / Stateflow chart
mr_0002	UML input parameter and Simulink block input
mr_0003	UML output parameter and Simulink block input
mr_0004	UML input parameter and Simulink block output
mr_0005	UML output parameter and Simulink block output
mr_0006	UML state machine / composite state and Stateflow chart / composite state
mr_0007	UML region and Stateflow parallel state
mr_0008	UML state and Stateflow state
mr_0009	UML choice pseudostate and Stateflow junction
mr_0010	UML fork pseudostate and Stateflow composite state
mr_0011	UML join pseudostate and Stateflow composite state
mr_0012	UML default transition and Stateflow default transition
mr_0013	UML transition and Stateflow transition
mr_0014	UML transition trigger and Stateflow transition trigger
mr_0015	UML transition guard and Stateflow transition guard
mr_0016	UML transition actions and Stateflow transition actions

TABLE 3.2: Mapping rule mr_0001 for UML class/component and Simulink subsystem/Stateflow chart.

Mapping Rule (ID: Name)	mr_0001: UML class/component and Simulink subsystem/Stateflow chart
When	The UML class/component and the Simulink subsystem/Stateflow chart have similar names.
Where	(1) Input parameters of operations in the UML classifier's provided interface and inputs of the Simulink subsystem block/Stateflow chart are matched (referenced rule: mr_0002, see Table 3.3). (2) Output parameters of operations in the UML classifier's required interface and inputs of the Simulink subsystem block/Stateflow chart are matched (referenced rule: mr_0003, see Table 3.4). (3) Input parameters of operations in the UML classifier's required interface and outputs of the Simulink subsystem block/Stateflow chart are matched (referenced rule: mr_0004, see Table 3.5). (4) Output parameters of operations in the UML classifier's provided interface and outputs of the Simulink subsystem block/Stateflow chart are matched (referenced rule: mr_0005, see Table 3.6). (5) Nested classifiers and nested subsystem blocks are matched (referenced rule: mr_0001, see Table 3.2).

TABLE 3.3: Mapping rule mr_0002 for UML input parameter and Simulink block input.

Mapping Rule (ID: Name)	mr_0002: UML input parameter and Simulink block input
When	(1) The input parameter's name is similar to the block input's name. (2) The input parameter's data type is similar to the block input's data type.

Note: Empty compartments are omitted to keep the table uncluttered.

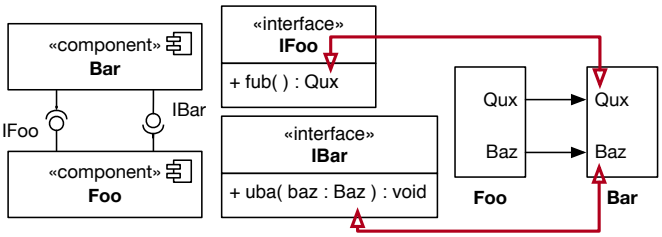


FIGURE 3.1: Illustrative example for *when* clauses 1 and 2 in mapping rule mr_0001.

TABLE 3.4: Mapping rule mr_0003 for UML output parameter and Simulink block input.

Mapping Rule (ID: Name)	mr_0003: UML output parameter and Simulink block input
When	
(1) The output parameter's name is similar to the block input's name.	
(2) The output parameter's data type is similar to the block input's data type.	

Note: Empty compartments are omitted to keep the table uncluttered.

TABLE 3.5: Mapping rule mr_0004 for UML input parameter and Simulink block output.

Mapping Rule (ID: Name)	mr_0004: UML input parameter and Simulink block output
When	
(1) The input parameter's name is similar to the block output's name.	
(2) The input parameter's data type is similar to the block output's data type.	

Note: Empty compartments are omitted to keep the table uncluttered.

TABLE 3.6: Mapping rule mr_0005 for UML output parameter and Simulink block output.

Mapping Rule (ID: Name)	mr_0005: UML output parameter and Simulink block output
When	
(1) The output parameter's name is similar to the block output's name.	
(2) The output parameter's data type is similar to the block output's data type.	

Note: Empty compartments are omitted to keep the table uncluttered.

TABLE 3.7: Mapping rule mr_0006 for UML state machine / composite state and Stateflow chart / composite state.

Mapping Rule (ID: Name)	mr_0006: UML state machine / composite state and Stateflow chart / composite state
When	The UML state machine/composite state and Stateflow chart/composite state have similar names.
Where	(1) Regions in the UML state machine/composite state and parallel states at the topmost level of the Stateflow chart/composite state are matched (referenced rule: mr_0007, see Table 3.8). This clause is dropped when the UML state machine has only one region. (2) States in the UML state machine/composite state and states in the Stateflow chart/composite state are matched (referenced rule: mr_0008, see Table 3.9). (3) Choice pseudostates in the UML state machine/composite state and connective junctions in the Stateflow chart/composite state are matched (referenced rule: mr_0009, see Table 3.10). (4) Fork pseudostates in the UML state machine/composite state and Stateflow composite states in the Stateflow chart/composite state are matched (referenced rule: mr_0010, see Table 3.11). (5) Join pseudostates in the UML state machine/composite state and Stateflow composite states in the Stateflow chart/composite state are matched (referenced rule: mr_0011, see Table 3.12). (6) Initial states in the UML state machine/composite state and default transitions in the Stateflow chart/composite state are matched (referenced rule: mr_0012, see Table 3.13). (7) Transitions in the UML state machine/composite state and transitions in the Stateflow chart/composite state are matched (referenced rule: mr_0013, see Table 3.14).

TABLE 3.8: Mapping rule mr_0007 for UML region and Stateflow parallel state.

Mapping Rule (ID: Name)	mr_0007: UML region and Stateflow parallel state
When	(1) The containing Stateflow chart or composite state has parallel (AND) decomposition. (2) The Stateflow parallel state is at the topmost level of its containing chart/composite state. (3) the UML region and Stateflow parallel state have similar names.
Where	(1) Nested elements (*i.e.* states, transitions, pseudostates) are matched (referenced rules: mr_0006 through mr_0013, see Tables 3.7 through 3.14).

TABLE 3.9: Mapping rule mr_0008 for UML state and Stateflow state.

Mapping Rule (ID: Name)	mr_0008: UML state and Stateflow state
When	The UML state and Stateflow state have similar names.
Where	(1) Incoming and outgoing transitions are matched (referenced rule: mr_0013, see Table 3.14). (2) Entry, do and exit actions are matched.

TABLE 3.10: Mapping rule mr_0009 for UML choice pseudostate and Stateflow junction.

Mapping Rule (ID: Name)	mr_0009: UML choice pseudostate and Stateflow junction
Where	
(1) Incoming and outgoing transitions are matched (referenced rule: mr_0013, see Table 3.14).	

Note: Empty compartments are omitted to keep the table uncluttered.

TABLE 3.11: Mapping rule mr_0010 for UML fork pseudostate and Stateflow composite state.

Mapping Rule (ID: Name)	mr_0010: UML fork pseudostate and Stateflow composite state
When	
(1) The Stateflow composite state has parallel (AND) decomposition.	
(2) The UML fork pseudostates and Stateflow composite state have similar names (see guideline av_0017, Table 3.35).	
Where	
(1) Incoming transitions are matched (referenced rule: mr_0013, see Table 3.14).	

TABLE 3.12: Mapping rule mr_0011 for UML join pseudostate and Stateflow composite state.

Mapping Rule (ID: Name)	mr_0011: UML join pseudostate and Stateflow composite state
When	
(1) The Stateflow composite state has parallel (AND) decomposition.	
(2) The UML join pseudostates and Stateflow composite state have similar names (see guideline av_0017, Table 3.35).	
Where	
(1) Outgoing transitions are matched (referenced rule: mr_0013, see Table 3.14).	

TABLE 3.13: Mapping rule mr_0012 for UML default transition and Stateflow default transition.

Mapping Rule (ID: Name)	mr_0012: UML default transition and Stateflow default transition
When	The target state of the outgoing transition from the UML initial state and the target state of the Stateflow default transition are matched.

Note: Empty compartments are omitted to keep the table uncluttered.

TABLE 3.14: Mapping rule mr_0013 for UML transition and Stateflow transition.

Mapping Rule (ID: Name)	mr_0013: UML transition and Stateflow transition
When	
(1) The target vertexes have similar names.	
(2) the source vertexes have similar names.	
Where	
(1) Triggers, guards and actions are matched (referenced rules: mr_0014 through mr_0016, see Tables 3.15 through 3.17).	

TABLE 3.15: Mapping rule mr_0014 for UML transition trigger and Stateflow transition trigger.

Mapping Rule (ID: Name)	mr_0014: UML transition trigger and Stateflow transition trigger
When	The trigger expressions are matched (see guideline av_0010, Table 3.28).

Note: Empty compartments are omitted to keep the table uncluttered.

TABLE 3.16: Mapping rule mr_0015 for UML transition guard and Stateflow transition guard.

Mapping Rule (ID: Name)	mr_0015: UML transition guard and Stateflow transition guard
When	The guard expressions are matched (see guidelines av_0011 and av_0012, Tables 3.29 and 3.30).

Note: Empty compartments are omitted to keep the table uncluttered.

TABLE 3.17: Mapping rule mr_0016 for UML transition actions and Stateflow transition actions.

Mapping Rule (ID: Name)	mr_0016: UML transition actions and Stateflow transition actions
When	The actions' expressions are matched (see guideline av_0015, Table 3.33).

Note: Empty compartments are omitted to keep the table uncluttered.

3.1.2 Design guidelines

Twenty-one design guidelines (intra- and inter-model) were developed, fifteen mandatory and six recommended. Eighteen of them are intra-model design guidelines and three are inter-model design guidelines. Table 3.18 lists all the design guidelines. The design guidelines are divided into four categories: 1) suggest an appropriate mix of modelling languages driven by the nature of the system being modelled and the purposes of the models, 2) instruct on the naming of elements in the design models, 3) constrain the use of modelling language constructs, and 4) provide semantic equivalences between the constructs of different modelling languages.

TABLE 3.18: Summary of the guidelines.

ID	Title	Priority
Design using mixed modelling languages		
av_0001	Mixed use of UML, Simulink and Stateflow	Recommended
Naming		
av_0002	Definition of a naming convention	Mandatory
av_0003	Naming of elements in UML models	Mandatory
av_0004	Naming of UML fork and join pseudostates	Mandatory
av_0005	Naming of elements in Simulink / Stateflow models	Mandatory
av_0006	Naming of Simulink Inport and Outport blocks	Recommended
Constraining the use of modelling language constructs		
av_0007	Decomposition type for Stateflow chart and composite state	Recommended
av_0008	Expression of triggers in UML transitions	Mandatory
av_0009	Expression of triggers in Stateflow transitions	Mandatory
av_0010 †	Expression of triggers appearing in both UML and Stateflow transitions	Mandatory
av_0011	Expression of UML guards in transitions	Mandatory
av_0012	Expression of Stateflow conditions in transitions	Mandatory
av_0013	Expression of UML actions	Mandatory
av_0014	Expression of Stateflow actions	Mandatory
av_0015 †	Expression of actions appearing in both UML and Stateflow models	Mandatory
av_0016	Use of signal receipt and send symbols in UML state machines	Recommended
av_0017	Use of UML fork and join pseudostates	Mandatory
av_0018	Data type of Simulink Inports and Outports	Mandatory
av_0019	Conjugation of a UML Port	Mandatory
Semantic equivalences between the constructs of different modelling languages		
av_0020	Entry and exit points in Stateflow chart	Recommended
av_0021	Fork and join behaviour in Stateflow	Recommended

TABLE 3.19: Guideline av_0001: Mixed use of UML, Simulink and Stateflow.

Guideline (ID: Title)	av_0001: Mixed use of UML, Simulink and Stateflow
Priority	Recommended
Scope	UML, Simulink and Stateflow
Prerequisites	None

Description

The choice of using UML, Simulink or Stateflow, or a mix of them to model all the system or given portions of it should be driven by the nature of the system being modeled and the purposes of the models.

- If the system (or the portions of it) primarily involves software, use UML components and classes to characterize the structure and behaviour of the software.
- If the system (or the portions of it) involves both software and hardware elements, use UML components and classes to characterize the structure and behaviour of the software, and use Simulink and Stateflow to characterize the structure and behaviour of the hardware elements.
- If the behaviour of the system (or the portions of it) primarily involves modal logic with a combination of past and present logical conditions, use UML state machines, Stateflow charts or a mix of them.
- If a behaviour segment primarily involves behaviour that may execute concurrently, use UML state machines.
- If the behaviour of the system (or the portions of it) primarily involves if-then-else statements, use Stateflow truth table charts.

Rationale

Embedded control software are highly heterogeneous, possessing very different characteristics. In order to cope with their complexities, these characteristics should be described using diverse modelling mechanisms, like UML, Simulink and Stateflow. This guideline is intended to manage the resulting design heterogeneity in order to facilitate understanding and communication without hindering verification and certification.

See Also

- Guideline av_0002: Definition of a naming convention (see Table 3.20)
- MAAB Guideline na_0006: Guidelines for mixed use of Simulink and Stateflow
- MAAB Guideline na_0007: Guidelines for use of Flow Charts, Truth Tables and State Machines

TABLE 3.20: Guideline av_0002: Definition of a naming convention.

Guideline (ID: Title)	av_0002: Definition of a naming convention
Priority	Mandatory
Scope	UML, Simulink and Stateflow
Prerequisites	• MAAB Guideline ar_0001: Filenames • MAAB Guideline ar_0002: Directory names • MAAB Guideline na_0035: Adoption of naming conventions • MAAB Guideline jc_0201: Usable characters for Subsystem names • MAAB Guideline jc_0211: Usable characters for Inport block and Outport block • MAAB Guideline jc_0231: Usable characters for block names • MAAB Guideline na_0014: Use of local language in Simulink and Stateflow • UML 2.5.1 subclause 9.2.4 Notation (Classifiers) • UML 2.5.1 subclause 9.5.4 Notation (Properties) • UML 2.5.1 subclause 9.6.4 Notation (Operations) • UML 2.5.1 subclause 10.4.4 Notation (Interfaces) • UML 2.5.1 subclause 11.4.4 Notation (Classes) • UML 2.5.1 subclause 11.6.4 Notation (Components) • UML 2.5.1 subclause 14.2.4 Notation (State Machines)
Description It is not possible to define a single naming convention applicable to the internal processes of all organizations. However, within an organization it is mandatory to follow a single, consistent naming convention that is compatible across UML, Simulink and Stateflow models. Such naming convention must provide guidance for naming Simulink subsystem blocks, Simulink basic blocks, Simulink input and output ports, Stateflow charts, Stateflow states, Stateflow data (input, output and local), and UML`NamedElement`s.	
Rationale This guideline is intended to make mandatory a naming convention across UML, Simulink and Stateflow models in order to reduce ambiguities when mapping UML constructs and Simulink or Stateflow constructs.	

TABLE 3.21: Guideline av_0003: Naming of elements in UML models.

Guideline (ID: Title)	av_0003: Naming of elements in UML models
Priority	Mandatory
Scope	UML
Prerequisites	None
Description All `NamedElement`s in a UML model must have a name.	
Rationale Finding matches between UML, Simulink and Stateflow models relies on naming.	

TABLE 3.22: Guideline av_0004: Naming of UML fork and join pseudostates.

Guideline (ID: Title)	av_0004: Naming of UML fork and join pseudostates
Priority	Mandatory
Scope	UML
Prerequisites	None
Description The names of UML fork and join pseudostates must be the same.	
Rationale Sharing the same name root facilitates their matching to an equivalent Stateflow construct.	
See Also Guideline av_0014: Fork and join behaviour in Stateflow (see Table 3.39)	

TABLE 3.23: Guideline av_0005: Naming of elements in Simulink / Stateflow models.

Guideline (ID: Title)	av_0005: Naming of elements in Simulink / Stateflow models
Priority	Mandatory
Scope	Simulink / Stateflow
Prerequisites	None
Description All elements in Simulink / Stateflow models that can be named must have a name.	
Rationale Finding matches between UML, Simulink and Stateflow models relies on naming.	

TABLE 3.24: Guideline av_0006: Naming of Simulink Inport and Outport blocks.

Guideline (ID: Title)	av_0006: Naming of Simulink Inport and Outport blocks
Priority	Recommended
Scope	Simulink
Prerequisites	• MAAB Style Guideline jc_0211: Usable characters for Inport block and Outport block • MAAB Style Guideline jm_0010: Port block names in Simulink models
Description The names of Simulink Inport and Outport blocks must end with the suffixes *_in* and *_out*, respectively.	
Rationale Finding matches between UML, Simulink and Stateflow models relies on naming.	

TABLE 3.25: Guideline av_0007: Decomposition type for Stateflow chart and composite state.

Guideline (ID: Title)	av_0007: Decomposition type for Stateflow chart and composite state
Priority	Recommended
Scope	Stateflow
Prerequisites	None
Description A Stateflow chart or composite state should have a parallel (AND) decomposition and at least one parallel state owning a set of vertices (states and junctions) and transitions that define a behaviour fragment.	
Rationale A UML state machine and composite state denote a set of orthogonal behaviour fragments with the use of regions. Thus, this guideline is intended to reduce ambiguities when mapping a UML state machine or composite state to a Stateflow chart or composite state, respectively. The use of a parallel decomposition and parallel states is semantically equivalent to that of UML orthogonal regions. When the state machine or composite state define only one region this guideline can be elided.	

TABLE 3.26: Guideline av_0008: Expression of triggers in UML transitions.

Guideline (ID: Title)	av_0008: Expression of triggers in UML transitions
Priority	Mandatory
Scope	UML
Prerequisites	None
Description UML triggers must be defined as event names denoted textually as described in subclause 13.3.4 of the UML 2.5.1 specification.	
Rationale This guideline is intended to allow a mapping to be established between a UML state machine and a Stateflow chart.	
See Also UML 2.5.1 subclause 13.3.4 Notation	

TABLE 3.27: Guideline av_0009: Expression of triggers in Stateflow transitions.

Guideline (ID: Title)	av_0009: Expression of triggers in Stateflow transitions
Priority	Mandatory
Scope	Stateflow
Prerequisites	None
Description	Stateflow events in transitions should not be used for the reasons pointed out in [4]. An exception to the previous restriction applies to relative time event triggers, which must be denoted with "`after`" followed by the argument values corresponding to the number and unit of time.
Rationale	This guideline is intended to allow a mapping to be established between a UML state machine and a Stateflow chart.
See Also	MAAB Guideline db_0126: Scope of events

TABLE 3.28: Guideline av_0010: Expression of triggers appearing in both UML and Stateflow transitions.

Guideline (ID: Title)	av_0010: Expression of triggers appearing in both UML and Stateflow transitions
Priority	Mandatory
Scope	UML and Stateflow
Prerequisites	• Guideline av_0008: Expression of triggers in UML transitions • Guideline av_0009: Expression of triggers in Stateflow transitions
Description	If a transition appears in a UML state machine as well as a Stateflow chart, then the name of the Stateflow condition variable must be a subset of the UML trigger event name. An exception to the previous restriction applies to relative time event triggers, which must be denoted with "`after`" followed by the argument values corresponding to the number and unit of time.
Rationale	This guideline is intended to allow a mapping to be established between a UML state machine and a Stateflow chart.

Figure 3.2 illustrates guideline av_0010. This guideline states that if a transition appears in a UML state machine (left of the figure) as well as a Stateflow chart (right of the figure), then the Stateflow transition must be guarded by a condition variable whose name must be a subset of the UML transition's trigger event name. An exception applies with regards to relative time event triggers on both languages, which are denoted with "`after`" followed by a number and unit of time.

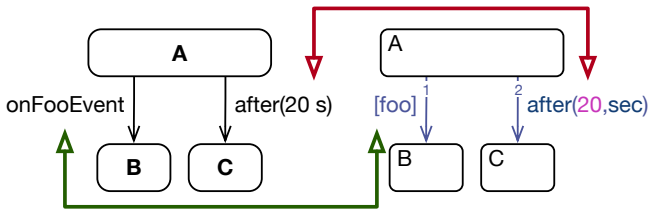


FIGURE 3.2: Illustrating example for guideline av_0010.

TABLE 3.29: Guideline av_0011: Expression of UML guards in transitions.

Guideline (ID: Title)	av_0011: Expression of UML guards in transitions
Priority	Mandatory
Scope	UML
Prerequisites	None
Description UML guards in transitions must be defined as logical expressions made up of a primary expression, or a conjunction or disjunction of two or more primary expressions. A primary expression is defined to be a relational expression containing one relational operator (<, <=, >, >=, ~=, ==, ~).	
Rationale The UML specification defines a guard as an `OpaqueExpression`. Only certain constraints are predefined in UML. Thus, this guideline borrows the Stateflow notation for the expression of guards in UML to facilitate their mapping.	
See Also • UML 2.5.1 subclause 13.3.4 Notation (Triggers) • UML 2.5.1 subclause 7.6.4 Notation (`OpaqueExpression`)	

TABLE 3.30: Guideline av_0012: Expression of Stateflow conditions in transitions.

Guideline (ID: Title)	av_0012: Expression of Stateflow conditions in transitions
Priority	Mandatory
Scope	Stateflow
Prerequisites	MAAB Style Guideline db_0150: State machine patterns for conditions
Description Stateflow conditions in transitions must be defined as logical expressions made up of a primary expression, or a conjunction or disjunction of two or more primary expressions. A primary expression is defined to be one of the following: • a Boolean input or local variable, or • a relational expression containing one relational operator (<, <=, >, >=, ~=, ==, ~).	
Rationale This guideline is intended to allow a mapping to be established between a UML state machine and a Stateflow chart.	

TABLE 3.31: Guideline av_0013: Expression of UML actions.

Guideline (ID: Title)	av_0013: Expression of UML actions
Priority	Mandatory
Scope	UML
Prerequisites	• MAAB Guideline jc_0501: Format of entries in a State block • MAAB Guideline db_0151: State machine patterns for transition actions
Description UML actions on entry, do and exit behaviours for states and on transitions must be defined as sequences of UML Operation calls.	
Rationale The UML specification provides relatively minimal concrete syntax for actions in behaviours attached to states and transitions in a state machine. Thus, this guideline clarifies the type of behaviour expressions allowed. The guideline is also intended to facilitate the mapping of actions between UML state machines and Stateflow charts.	
See Also • UML 2.5.1 subclause 16.1.1.1 Concrete Syntax • UML 2.5.1 subclause 7.6.4 Notation (`OpaqueExpression`)	

TABLE 3.32: Guideline av_0014: Expression of Stateflow actions.

Guideline (ID: Title)	av_0014: Expression of Stateflow actions
Priority	Mandatory
Scope	Stateflow
Prerequisites	• MAAB Guideline jc_0501: Format of entries in a State block • MAAB Guideline db_0151: State machine patterns for transition actions
Description Stateflow actions on entry, do and exit behaviours for states and on transitions must be defined as sequences of variable assignment statements. If the actions appear in a UML state machine as well, then the variable names must be subsets of the UML operation names used.	
Rationale The guideline is intended to facilitate the mapping of actions between UML state machines and Stateflow charts.	
See Also • UML 2.5.1 subclause 16.1.1.1 Concrete Syntax • UML 2.5.1 subclause 7.6.4 Notation (`OpaqueExpression`)	

TABLE 3.33: Guideline av_0015: Expression of actions appearing in both UML and Stateflow models.

Guideline (ID: Title)	av_0015: Expression of actions appearing in both UML and Stateflow models
Priority	Mandatory
Scope	UML and Stateflow
Prerequisites	• Guideline av_0007: Expression of UML actions (see Table 3.25) • Guideline av_0008: Expression of Stateflow actions (see Table 3.26)
Description If an action appears in a UML state machine as well as a Stateflow chart, then the Stateflow variable names must be subsets of the UML operation names used.	
Rationale The guideline is intended to facilitate the mapping of actions between UML state machines and Stateflow charts.	

TABLE 3.34: Guideline av_0016: Use of signal receipt and send symbols in UML state machines.

Guideline (ID: Title)	av_0016: Use of signal receipt and send symbols in UML state machines
Priority	Recommended
Scope	UML
Prerequisites	None
Description The UML signal receipt and send symbols should not be used. Instead, the UML default textual notation for a transition should be used.	
Rationale The signal receipt and send symbols break up the transition (into at least two segments), which makes more difficult a consistency analysis with other models (*e.g.*, Stateflow). Furthermore, these symbols are not always supported in tools.	
See Also • UML 2.5.1 subclause 14.2.4.8 Transition • Guideline av_0008: Expression of triggers in UML transitions (see Table 3.26) • Guideline av_0013: Expression of UML actions (see Table 3.31)	

TABLE 3.35: Guideline av_0017: Use of UML fork and join pseudostates.

Guideline (ID: Title)	av_0017: Use of UML fork and join pseudostates
Priority	Mandatory
Scope	UML
Prerequisites	Guideline av_0004: Naming of UML fork and join pseudostates (Table 3.22)
Description When using fork and join pseudostates, every parallel path from the fork pseudostate to the join pseudostate has only one composite state.	
Rationale This guideline is intended to reduce ambiguities when mapping a UML state machine and a Stateflow chart.	

TABLE 3.36: Guideline av_0018: Data type of Simulink Inports and Outports.

Guideline (ID: Title)	av_0018: Data type of Simulink Inports and Outports
Priority	Mandatory
Scope	Simulink
Prerequisites	None
Description The data type of Simulink Inports and Outports must be set. The data type cannot be inherited or auto.	
Rationale Finding matches between UML, Simulink and Stateflow models relies on data types.	

TABLE 3.37: Guideline av_0019: Conjugation of a UML Port.

Guideline (ID: Title)	av_0019: Conjugation of a UML Port
Priority	Mandatory
Scope	UML
Prerequisites	None
Description The value of the `isConjugated` property of a UML port must be set accordingly depending on the port's purpose. • If the port is used to provide an interface, then the value of the `isConjugated` property must be set to `false`. • If the port is used to require an interface, then the value of the `isConjugated` property must be set to `true`.	
Rationale The value of the `isConjugated` property specifies the way that the provided and required interfaces are derived from the port's type.	
See Also UML 2.5.1 subclause 11.8.14 Port	

TABLE 3.38: Guideline av_0020: Entry and exit points in Stateflow chart.

Guideline (ID: Title)	av_0020: Entry and exit points in Stateflow chart
Priority	Recommended
Scope	Stateflow
Prerequisites	None

Description
In some modelling situations when encapsulating elements in a composite state, it is useful to bind the internal elements of such composite state with incoming and outgoing transitions. In UML this is realized by means of entry and exit point pseudostates. Stateflow does not provide equivalent constructs for such purpose. However, incoming transitions can penetrate a composite state and terminate directly on one of its internal vertices, and outgoing transitions can start in an internal state and terminate in another state outside the composite state.
When these situations need to be modelled in a Stateflow chart an additional state should be added within the composite state representing an entry or exit point of the composite state. For the case of exit points, the entry action of the additional state must set a local data in the Stateflow chart that is evaluated in the conditions of the state's corresponding outgoing transitions.

Rationale
This guideline is intended to reduce ambiguities when mapping a UML state machine and a Stateflow chart. The introduction of additional states into the Stateflow chart to represent entry and exit points of composite states can be regarded as semantically equivalent to that of UML entry and exit point pseudostates.

TABLE 3.39: Guideline av_0021: Fork and join behaviour in Stateflow.

Guideline (ID: Title)	av_0021: Fork and join behaviour in Stateflow
Priority	Recommended
Scope	Stateflow
Prerequisites	None

Description
In some modelling situations where all parallel states need to terminate their execution before the execution can continue, the parallel states must perform a synchronization. To perform a synchronization function in Stateflow parallel states, additional states are required as follows:

- An additional 'Synced' state within each parallel state must be added as the final state.
- The previous 'Synced' state must be reached through a transition that evaluates to 'true' a Boolean 'syncing' local data. When the state is reached, the Boolean 'synced' local data is changed to 'true'.
- One of the parallel states needs to be chosen as the synchronizing state. Such a state will change the 'syncing' local data to 'true' in the transition to an additional 'Syncing' state placed before the 'Synced' state.

Rationale
Stateflow parallel states do not have a synchronization mechanism. Thus, this guideline is intended to define a synchronization mechanism. A word of caution, synchronizing parallel states in this manner is not scalable and is recommended only for synchronizing two parallel states as it could cause state explosion.

See Also
Stateflow documentation: Broadcast Events to Synchronize States

3.2 *checsdm4uss*: Codification phase

3.2.1 Metamodelling

In particular for *checsdm4uss*, creation of UML models is supported by the Eclipse IDE and its EMF-based UML plug-in. The Eclipse Papyrus plug-in was integrated as well to provide graphical facilities to visualize and edit the UML models. On the contrary, development of Simulink and Stateflow models is restricted to MATLAB. Simulink and Stateflow are proprietary modelling languages of MathWorks limited to the MATLAB environment. A specification of these languages is also not openly available. An existing, reverse-engineered Simulink metamodel from Massif, an open-source tool [5] providing Simulink-to-EMF transformation services, was taken as the basis for the Simulink and Stateflow metamodel in *checsdm4uss*. Two reasons prevented the use of Massif as it is. First, Massif does not support Stateflow. Second, compatibility issues were experienced between Massif and ECL, which is employed to codify the mapping rules. As a result, the Simulink and Stateflow metamodel was built from scratch, using EMF, based on Massif's metamodel for Simulink constructs with the added constructs corresponding to Stateflow.

Processing the contents from Simulink and Stateflow models requires their transformation from the MathWorks format into the developed EMF-based metamodel. Therefore, a MATLAB connector and importer tools were developed as well to 1) extract the information from the Simulink and Stateflow models opened in MATLAB, and 2) create new models conforming to the developed Simulink and Stateflow metamodel.

Codification of mapping rules

Listing 3.1 shows in detail one of the codifications for mapping rule mr_0001 (see Table 3.2) describing the relationship between UML components and Simulink subsystem blocks (see lines 2 and 3 of the listing). Mapping rules in the *where* clauses are indirectly referenced through a call to ECL's built-in `matches` operation (see lines 8, 11 and 14). Codification of the other mapping rules is analogous.

LISTING 3.1: One of the codifications of mapping rule mr_0001.

```
1  rule MatchUMLComponentAndSimulinkSubsystemBlock
2  match component : UML!Component
3  with subsystem : Simulink!SubSystem {
4   compare {
5    var names : Boolean = component.name.matches(subsystem.name);
6    var componentInputs = component.obtainInputs();
7    var subsystemInputs = subsystem.obtainInputs();
8    var inputs : Boolean = componentInputs.matches(subsystemInputs);
9    var componentOutputs = component.obtainOutputs();
10   var subsystemOutputs = subsystem.obtainOutputs();
11   var outputs : Boolean = componentOutputs.matches(subsystemOutputs);
12   var nestedComponents = component.obtainNestedElements();
13   var nestedSubsystems = subsystem.obtainNestedElements();
14   var nestedElements : Boolean = nestedComponents.matches(nestedSubsystems);
15   return names and inputs and outputs and nestedElements;
16  }
17 }
```

Codification of design guidelines

Listing 3.2 presents the codification of intra-model design guideline av_0003 (see Table 3.21) as two model queries. The query `umlNamedElementEmptyName` (see lines 4 through 6 of the listing) captures the erroneous situation where an element in a UML model does not have a name. This is accomplished by the negative composition of

LISTING 3.2: Codification of intra-model design guideline av_0003.

```
1  @Constraint(severity = "error", key = {namedElement},
2   message = "The element must have a name.",
3   targetEditorId = "org.eclipse.uml2.uml.editor.presentation.UMLEditorID")
4  pattern umlNamedElementEmptyName(namedElement : NamedElement) {
5   neg find umlNamedElementName(namedElement);
6  }
7  pattern umlNamedElementName(namedElement : NamedElement) {
8   NamedElement.name(namedElement, name);
9   check (name.matches(".+"));
10 }
```

LISTING 3.3: Codificaiton of inter-model design guideline av_0010.

```
1  @Constraint(severity = "error", key = {mapping},
2   message = "The Stateflow trigger must be a subset of the event name of $umlTrigger$.",
3   targetEditorId = "ca.ets.avio604.mappings.presentation.MappingsEditorID")
4  pattern matchingTriggerInvalid(mapping : Mapping, umlTrigger : Trigger, sfwTrigger :
       ↪ SFWTrigger) {
5   Mapping.left(mapping, umlTrigger);
6   Mapping.right(mapping, sfwTrigger);
7   Mapping.matching(mapping, _);
8   Trigger.event(umlTrigger, event);
9   Event.name(event, umlEventName);
10  SFWTrigger.statement(sfwTrigger, sfwTriggerStatement);
11  check(!umlEventName.contains(sfwTriggerStatement.substring(1, sfwTriggerStatement.length
       ↪ () - 1)) && !(umlEventName.startsWith("after") && sfwTriggerStatement.startsWith("
       ↪ after")));
12 }
```

another graph pattern `umlNamedElementName` (see lines 7 through 10), which captures elements in the UML model that have a name. `@Constraint` annotations, like the one in lines 1 through 3 of Listing 3.2, specify to the Viatra framework that a `GraphPattern` represents a well-formedness constraint linked to the UML Eclipse editor (see line 3 in Listing 3.2).

Listing 3.3 shows the codification of inter-model design guideline av_0010 (see Table 3.28) providing guidance on the expression of triggers of a transition appearing in both UML state machines and Stateflow charts. The query `matchingTriggerInvalid` states that if a transition appears in a UML state machine as well as a Stateflow chart (see lines 4 through 6 of the listing), then the Stateflow transition must be guarded by a condition variable whose name must be a subset of the UML transition's trigger event name (see line 10). An exception to the previous restriction applies with regards to relative time event triggers on both languages, which must be denoted with "`after`" followed by the argument values corresponding to the number and unit of time (see line 10). The defined `GraphPatterns` for these design guidelines are also given the `@Constraint` annotation (see lines 1 through 3 of Listing 3.3). Facilities to initiate the validation and display markers on the instance models are automatically generated when the Viatra Validation Framework processes the annotations.

Codification of the other design guidelines is analogous. Nevertheless, while codifying the design guidelines and experimenting with them on the different REVIEW ►*case studies*◄ ►◄ , it was found that mandatory design guideline av_0002 and recommended design guidelines av_0001, av_0020 and av_0021 are dependent on companies and may even vary from one project to another. These design guidelines are related to company practices as well as to choices during the design process. This means they need to be codified independently for each project or organization. Design guideline av_0002, for instance, can be formalized once the naming convention is defined. Design guideline av_0001, for instance, is difficult to codify. For this design guideline, the engineer is more likely required to carry out a manual verification of compliance.

Derived toolchain

An Eclipse toolchain can be derived with the previous codifications to support the *operation* phase. Figure 3.3 depicts an overview of the derived toolchain for *checsdm4uss*. An Eclipse plug-in contributing a contextual menu option to map two selected UML and Simulink model files from the IDE's Explorer View has been developed. This menu option provides users with a simplified way of initiating the mapping of the design models.

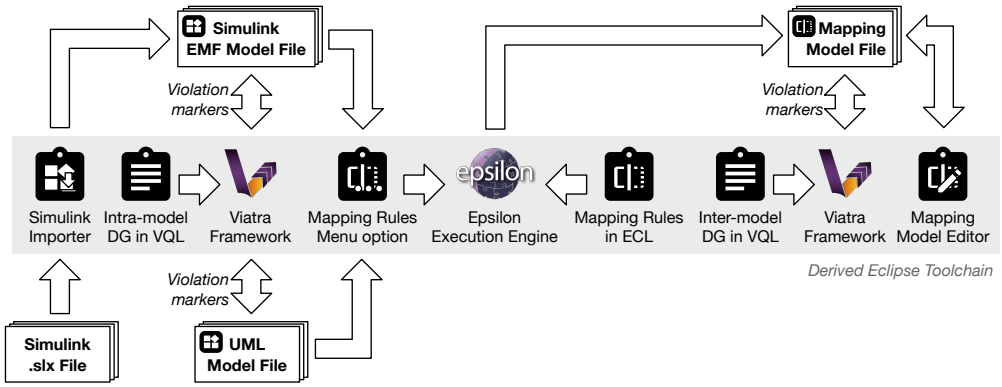


FIGURE 3.3: Overview of the derived toolchain.

3.3 *checsdm4uss*: Operation phase

This phase is presented in Chapter 4, were it is enacted in the context of an avionics software.

4 Usage Examples

In order to illustrate the *operation* phase of *checsdm4uss*, an open source avionics system specification has been used.

4.1 Landing Gear Control Software

The usage example regards an aircraft's landing gear control software (LGCS). The landing gear system specification was introduced in [6] and developed into a software specification in [7]. Following subsections present a brief system description and focus on the enactment of the steps in the *operation* phase.

4.1.1 System description

The landing gear system of an aircraft is the undercarriage that supports it during ground operations. The system is composed of three retractable gears arranged in a tricycle configuration. The gears are retracted after take off and extended before landing. Doors cover the gear compartments and are required to be open prior to and closed after any retraction or extension is performed. The operation of the landing gear system is managed by the Landing Gear Control Software (LGCS). The pilots provide the desired gear position to the LGCS. An indicator provides them with information about the state of the gears and the system itself. All the gears are moved through the sequential actuation (*i.e.* opening and closing) of five hydraulic electro-valves (HEVs). One general HEV pressurizes the hydraulic circuit, while the remaining four specific HEVs control the oil pressures in the segments of the hydraulic circuit directly attached to the gears and doors. Seventeen sensors, each with triple-mode redundancy, monitor the pressure of the hydraulic circuit, and the state of the gears and doors. The LGCS must sequence the commands that actuate the HEVs for moving the gears and doors to the given desired gear position taking into account the inputs from the different sensors.

4.1.2 Software design

Design scenario

In order to maximize coverage of *checsdm4uss*, a design scenario is chosen according with the features in the feature diagram of Figure 2.2 where the design models in UML and Simulink and Stateflow would 1) have complete overlap of elements, 2) describe elements at the same levels of abstraction, 3) have complete coverage of the system elements, and 4) cover both structural and behavioural perspectives.

Design procedure

Figure 4.1 gives a simplified, high-level view of the software design procedure that has been defined to comply with step 1 of the *checsdm* approach. The software design process has been divided as recommended by [8] to maximize adherence to DO-178C, DO-331 and DO-332, into two different activities: development of the software architecture and development of the detailed design. These activities are expanded in Figures 4.2 and 4.3, respectively.

Design models

The LGCS is designed in its entirety using UML, and Simulink and Stateflow following the previous procedure and taking into account the proposed guidelines for heterogeneous design. Most of the guidelines were followed. This was deliberate in order to insert guideline violations. Following subsections explaining the application of the

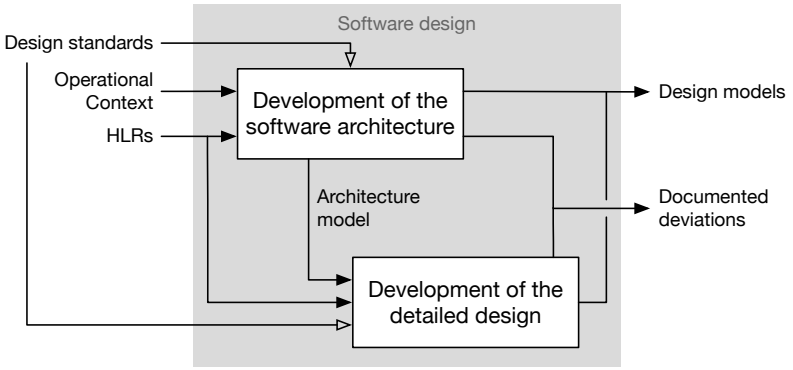


FIGURE 4.1: General flow for the software design process.

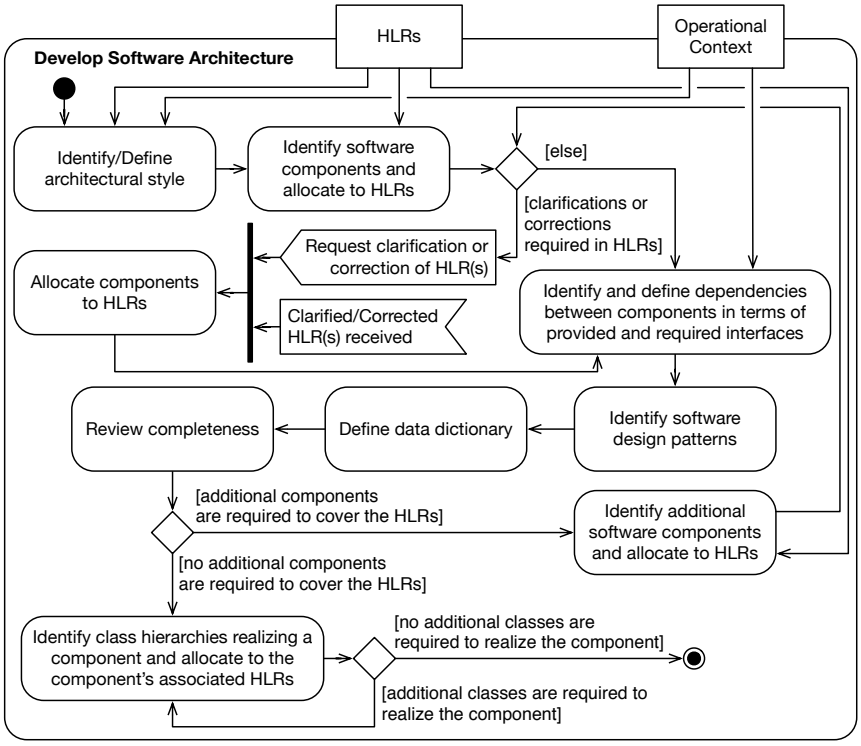


FIGURE 4.2: Expanded view of the *development of the software architecture* activity.

approach's next steps give an account of guideline compliance. Figure 4.4 presents the resulting software architecture as a UML component diagram. In it, the LGCS is described as a modular, reusable component providing its specified functionality through a set of provided interfaces. The LGCS component is further decomposed into smaller, modular, reusable components providing simpler functionality following the *Process Control* architectural style [9]. Class diagrams and state machines provide the detailed design of such components. Examples are given in Figures 4.5 and 4.6. The design with Simulink and Stateflow is analogous. The LGCS is described as a single subsystem block performing its functions over a set of inputs from other subsystem blocks, corresponding to the other system entities, to produce the set of expected outputs. The LGCS block is decomposed into smaller, interconnected subsystem blocks. The detailed design is provided through Stateflow charts, like the one presented in Figure 4.8.

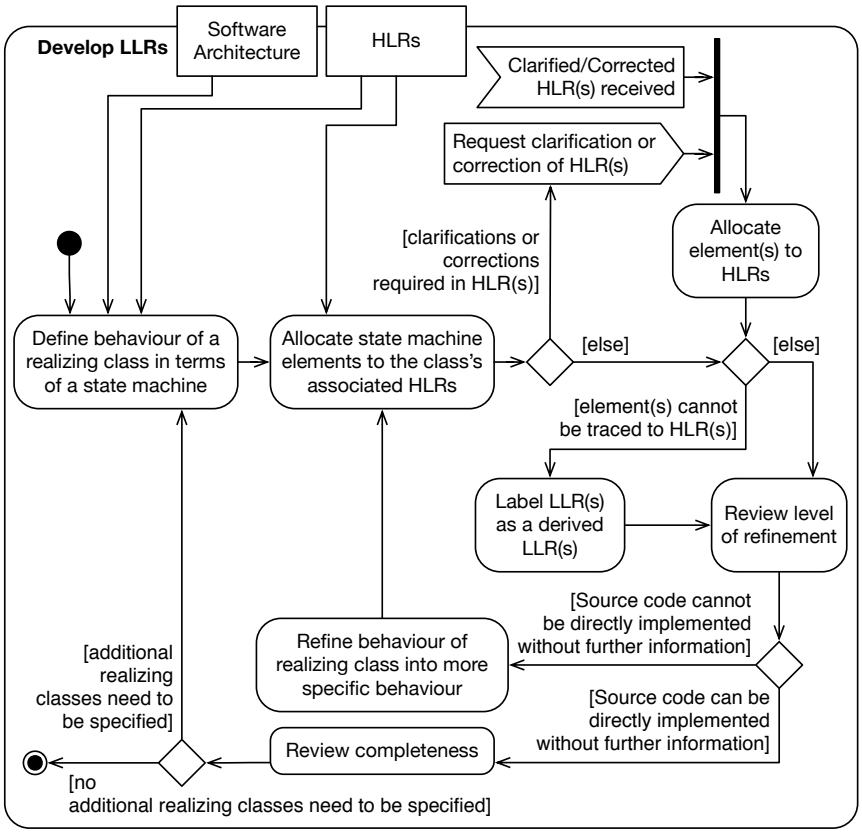


FIGURE 4.3: Expanded view of the *development of the detailed design* activity.

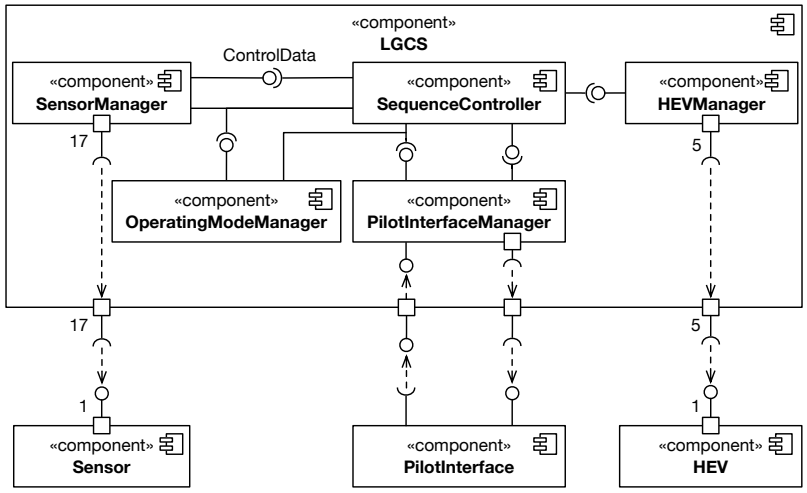


FIGURE 4.4: Architecture for the LGCS as UML components.

4.1.3 Intra-model design guideline compliance

The design models developed for the LGCS were verified for guideline compliance. This was carried out using the guidelines' formalizations as Viatra queries. The design activity resulted in no deviations from the guidelines that were verified during this step. An example of a guideline violation is shown later in Figure 4.11.

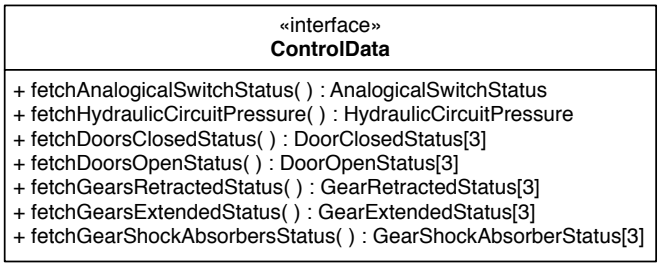


FIGURE 4.5: ControlData interface.

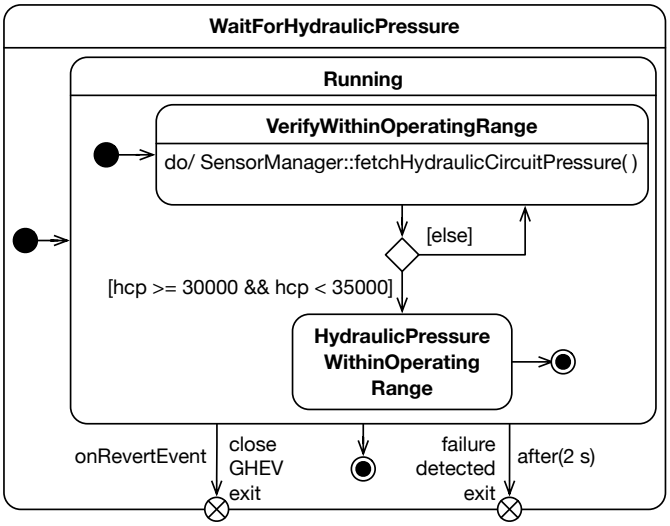


FIGURE 4.6: Excerpt from the UML state machine associated to the SequenceController component.

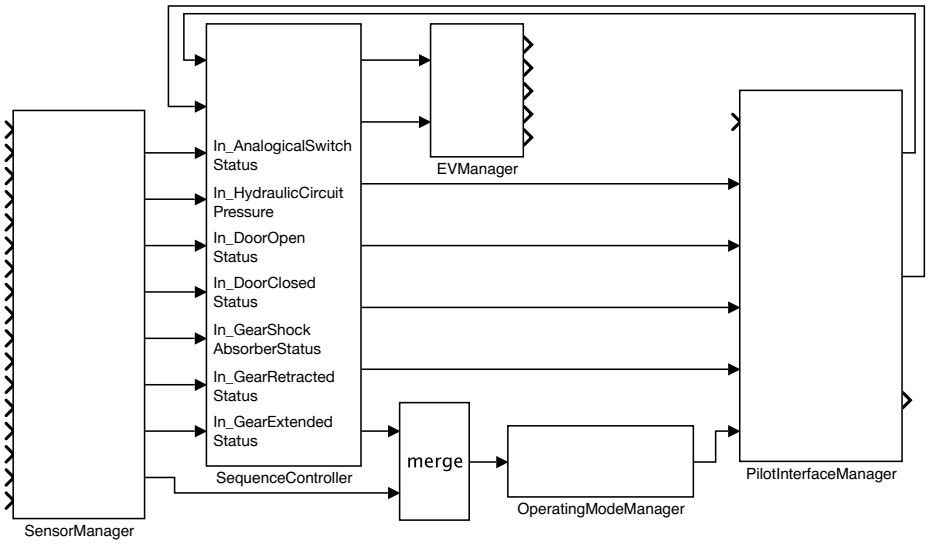


FIGURE 4.7: Architecture for the LGCS as Simulink blocks.

4.1.4 Mapping of design models

After the first pass of the verification of guideline compliance, the design models for the LGCS were analyzed in search for correspondences and verified for consistency. This was carried out using the mapping rules formalized

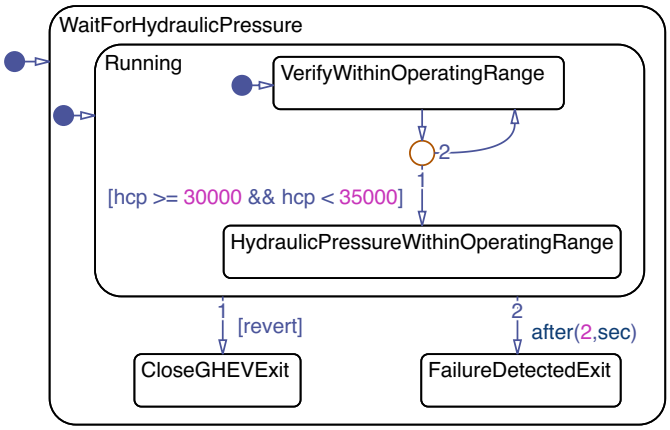


FIGURE 4.8: Excerpt from the Stateflow chart realizing the SequenceController subsystem block.

with ECL and the support tools described. Mapping rule mr_0001 presented in Table 3.2 applies to the UML component diagram in Figure 4.4 and the Simulink block diagram in Figure 4.7. Figure 4.9 presents an excerpt from the resulting mapping model from applying the proposed mapping rules. Each element in the model represents a Mapping. Figure 4.10 shows an example of the properties in every Mapping. The left property refers to an element in the UML model and the right property refers to an element in the Simulink model.

Mapping rule mr_0001 matches the SequenceController component in UML to the SequenceController subsystem block in Simulink (see line 3 of Figure 4.9). The mappingRule attribute informs this as can be seen in Figure 4.10. Both the UML component and the Simulink subsystem block have identical names (see line 6 of Figure 4.9). The other conditions defined in mapping rule mr_0001 are also met, *i.e.* inputs, outputs and nested elements were matched as well. Lines 4, 5 and 7 of Figure 4.9 show these results. For example, lines 13 and 14 of Figure 4.9 show that the analogicalSwitchStatus and hydraulicCircuitPressure return parameters expected from requiring the ControlData interface are matched to the inports In_AnalogicalSwitchStatus and In_HydraulicCircuitPressure, respectively.

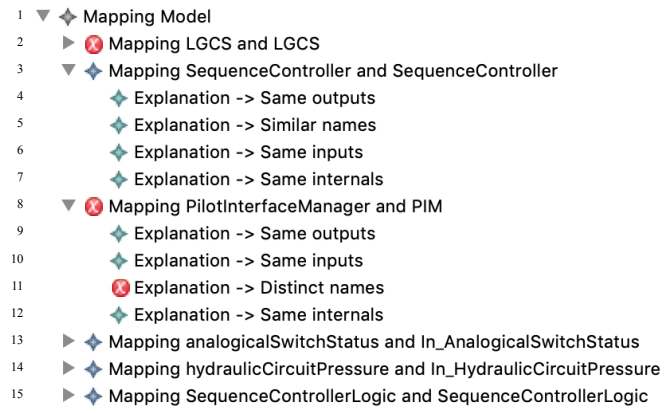


FIGURE 4.9: Excerpt from the mapping model for the LGCS.

Property	Value
Left	<Component> SequenceController
Mapping Rule	MatchUMLComponentAndSimulinkSubsystemBlock
Matching	true
Name	SequenceController and SequenceController
Right	Sub System SequenceController

FIGURE 4.10: Properties of the mapping in line 3 of Figure 4.9.

The Simulink subsystem block `PilotInterfaceManager` in Figure 4.7 was renamed to `PIM` (see line 8 of Figure 4.9) to deliberately insert a consistency issue. After applying all the mapping rules the `PilotInterfaceManager` component and the `PIM` subsystem block were not matched. The elements have the same inputs, outputs and nested elements as reported in the explanations in lines 9, 10 and 12. However, they possess distinct names, reason that is disclosed in the explanation in line 11. Since the design scenario involves a complete coverage of system elements and a complete overlapping between the UML and Simulink models, then it is expected that all elements of the UML model are related to a corresponding element of the Simulink/Stateflow model. Thus, the `PilotInterfaceManager` component and the `PIM` subsystem block are flagged as inconsistent. This caused the mapping between the LGCS UML component and the LGCS Simulink subsystem block to be flagged as inconsistent as well (see line 2 of Figure 4.9).

4.1.5 Inter-model design guideline compliance

The resulting mapping model for the LGCS heterogeneous design was verified for guideline compliance. Guideline av_0010 (see Table 3.28) was not followed in order to deliberately insert a guideline violation. In Figure 4.6, the trigger on the transition from the `Running` state to the `close GHEV exit` point is expressed as an event name (*i.e.* `onRevertEvent`). The equivalent transition in the Stateflow chart of Figure 4.8 is expressed as a condition on an input Boolean variable `cancel` instead of *revert*. Figure 4.11 displays the error obtained after guideline compliance was verified over the mapping model. Note that in Figure 4.8, the relative time event trigger `after(2,sec)` on the transition from the `Running` state to the `FailureDetectedExit` state is an example of the exception to the restriction of event names for triggers in Stateflow transitions allowed by guideline av_0010.

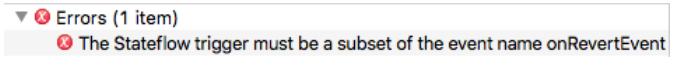


FIGURE 4.11: Example of a guideline violation.

5 Derived Toolchain for *checsdm4uss*

5.1 Overview

An Eclipse toolchain supports *checsdm4uss*. Figure 5.1 depicts an overview of the tools. A `Simulink Importer` transforms Simulink and Stateflow models into Eclipse Modelling Framework (EMF) representations that can be processed along with UML models. This is because Simulink and Stateflow models are stored in a proprietary format that requires MATLAB. The `Simulink Importer` is based on Massif[1], an open-source tool providing Simulink-to-EMF transformation services. However, Massif does not support Stateflow. Furthermore, some issues with ECL processing the resulting Massif EMF Simulink model were experienced. Thus, the `Simulink Importer` was built from scratch based on Massif's Simulink metamodel with extensions to provide Stateflow-to-EMF transformation. `Viatra Queries` check compliance of the design models with the proposed guidelines. Some of the `Viatra Queries` are used to check the compliance with the proposed guidelines once the mapping model has been created. Violations to guidelines are highlighted and reported within the design models and mapping model opened in Eclipse. The `ECL rules` analyze the system's design models in search for correspondences according with the proposed mapping rules and produce a mapping model, which flags possible consistency issues. Facilities for viewing and manually editing mappings between the models' elements are provided by a mapping model editor.

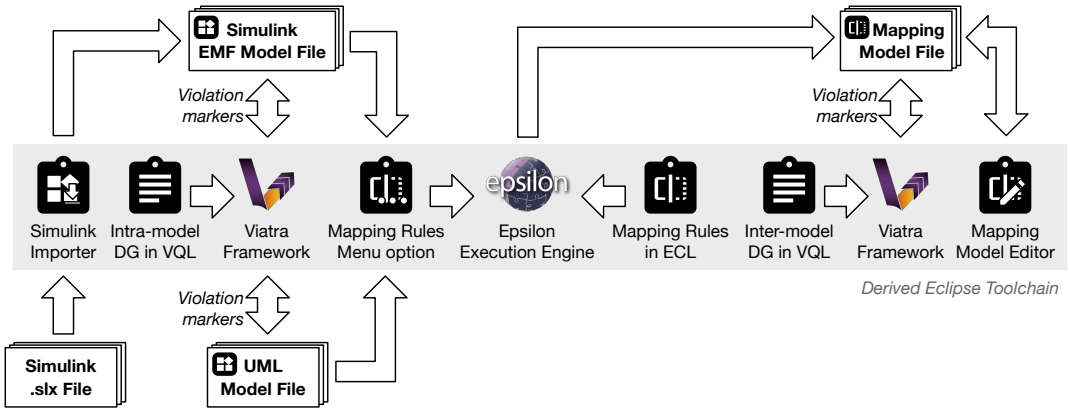


FIGURE 5.1: Overview of the support toolchain.

5.2 Developer's Guide

5.2.1 Getting started

System requirements

- Java JDK 8 or higher
- Eclipse Oxygen or newer with the Modeling package (*a.k.a.* Eclipse Modeling Tools)
- MATLAB R2016b or newer

Install required dependencies

The required dependencies are Epsilon, Viatra and Xtext.

[1] https://incquerylabs.com/en/page/show/massif

1. Go to **Help** » **Install New Software...**.
2. Work with `http://download.eclipse.org/epsilon/1.4/updates/` to install Epsilon.
   Select the following features:
   - Epsilon Core v1.4.0
   - Epsilon Core Development Tools v1.4.0
   - Epsilon EMF Integration v1.4.0
   - Epsilon UML Integration v1.4.0
3. Work with `http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/` to install Xtext.
   Select the following features:
   - Xtext » Xtext Complete SDK v2.12.0
4. Work with `http://download.eclipse.org/viatra/updates/release` to install Viatra.
   Select the following features:
   - VIATRA Core » VIATRA Query and Transformation SDK v1.6.1

Import the *checsdm4uss* projects

1. Import the source code projects of *checsdm4uss* provided with this report into the Eclipse workspace.

Create new run configuration

Create a new Run Configuration in Eclipse:

1. Go to **Run** » **Run Configurations...**.
2. From the list in the left, double click on **Eclipse Application**.
3. Give the configuration a name (suggestion: **checsdm4uss**).
4. Select the tab **Environment**.
5. Create a new variable with the following information:
   In Windows:

   Name: PATH
   Value: `«path to MATLAB»\bin\win64`

   In macOS:

   Name: DYLD_LIBRARY_PATH
   Value: `«path to MATLAB»/bin/maci64`

   In Linux:

   Name: LD_LIBRARY_PATH
   Value: `«path to MATLAB»/bin/glnxa64:«path to MATLAB»/sys/os/glnxa64`

5.2.2 Project structure

- **src-gen**

  This folder contains generated source code from models. These files should not be edited directly.

- **src**

  This folder contains manually created source code. These files can be edited directly if necessary.

- **icons**

  This folder contains image files used for icons in the plug-in.

- **META-INF** and **MANIFEST.MF**

  This folder and file contain descriptions of the extensions made to Eclipse IDE by the plug-in.

- **models**

  This folder contains EMF models.

- **build.properties**

  This file contains the build properties of the plug-in.

- **plugin.properties**

  This file contains additional properties of the plug-in.

- **plugin.xml**

  This file contains descriptions of the extensions made to the Eclipse IDE by the plug-in.

5.2.3 *checsdm4uss* projects and noteworthy files

- **ca.ets.avio604.checsdm.guidelinecompliance**

  This project contains the guideline formalizations in VQL. Each .vql file contains one guideline formalization. The files are named with the guideline IDs.

- **ca.ets.avio604.checsdm.guidelinecompliance.validation**

  This project is automatically generated by Viatra. It contains all the marker-based validation code generated from the constraints specified with the `@Constraint` annotation in the guideline formalizations.

- **ca.ets.avio604.checsdm.mappingrules**

  This project contains the mapping rules formalizations in ECL.

  - **Main.ecl**

    Entry point for mapping rule execution. This file contains initialize (`pre`) and terminate (`post`) code blocks that must be executed before and after (respectively) evaluating the mapping rules.

  - **Common.ecl**

    This file defines common operations used throughout the mapping rules formalizations.

  - **ClassesAndSubsystems.ecl**

    This file defines the formalizations for mapping rules regarding UML components and classes, and Simulink subsystem blocks, *i.e.* mapping rules mr_0001 through mr_0005 (see Tables 3.2 through 3.6).

  - **StateMachinesAndCharts.ecl**

    This file defines the formalizations for mapping rules regarding UML state machines and Stateflow charts, *i.e.* mapping rules mr_0006 through mr_0016 (see Tables 3.7 through 3.17).

- **ca.ets.avio604.checsdm.mappingrules.ui**

  This project provides a contextual menu UI contribution to the Eclipse IDE for executing the mapping rules when both a UML and a Simulink model are selected in the *Model Explorer* view.

- **ca.ets.avio604.checsdm.mappings**

  This project defines the Mapping metamodel.

  - **Mappings.ecore**

    EMF model defining the Mapping metamodel (see Figure 2.6).

  - **Mappings.genmodel**

    EMF generator model containing the control parameters on how code and other derived outputs should be generated.

- **ca.ets.avio604.checsdm.mappings.edit**

  This project provides editing facilities for mapping models. This project is generated through the *Mappings.genmodel* generator model in the *ca.ets.avio604.checsdm.mappings* project.

- **ca.ets.avio604.checsdm.mappings.editor**

  This project provides a graphical editor for mapping models. This project is generated through the *Mappings.genmodel* generator model in the *ca.ets.avio604.checsdm.mappings* project.

- **ca.ets.avio604.checsdm.matlabengine**

  This project provides a high-level, reusable, extendable framework for working programmatically with Simulink and Stateflow through MATLAB's Java API.

- **ca.ets.avio604.checsdm.simulink**

  This project defines the Simulink EMF metamodel.

  - **Simulink.ecore**

    EMF model defining the Simulink EMF metamodel.

  - **Simulink.genmodel**

    EMF generator model containing the control parameters on how code and other derived outputs should be generated.

- **ca.ets.avio604.checsdm.simulink.edit**

  This project provides editing facilities for Simulink EMF models. This project is generated through the *Simulink.genmodel* generator model in the *ca.ets.avio604.checsdm.simulink* project.

- **ca.ets.avio604.checsdm.simulink.editor**

  This project provides a graphical editor for Simulink EMF models. This project is generated through the *Simulink.genmodel* generator model in the *ca.ets.avio604.checsdm.simulink* project.

- **ca.ets.avio604.checsdm.simulink.importer.api**

  This project provides facilities for importing Simulink.slx files into the Eclipse IDE.

- **ca.ets.avio604.checsdm.simulink.importer.ui**

  This project provides a contextual menu UI contribution to the Eclipse IDE for importing a Simulink.slx file selected in the *Model Explorer* view.

- **matlabengine.bundle**

  This project bundles the MATLAB Java API as an Eclipse plug-in.

  - **engine.jar**
    The MATLAB Java API file. The current file corresponds to MATLAB R2018b. If the installed MATLAB release differs, replace the file by importing the one found in `«path to MATLAB»/extern/engines/java/jar/engine.jar`.

- **org.apache.commons.text**

  This project bundles the Apache Commons Text library as an Eclipse plug-in. This library provides algorithms working on strings. Some of these algorithms are used when comparing names in the mapping rules formalizations.

  - **commons-text-1.1.jar**
    The Apache Commons Text library. The library can be updated by replacing the file with a more recent version.

5.2.4 More information

Viatra

More information about Viatra is available at https://www.eclipse.org/viatra/documentation/index.html.

ECL

More information about ECL is available in [2].

MATLAB

More information about MATLAB, Simulink and Stateflow is available at https://www.mathworks.com/help/simulink/.

5.3 User's Guide

5.3.1 Getting started

System requirements

- Java JDK 8 or higher
- Eclipse Oxygen or newer with the Modeling package (*a.k.a.* Eclipse Modeling Tools)
- MATLAB R2016b or newer

Before running

Create a new environment variable in your operating system with the following information:

In Windows:

> Name: PATH
>
> Value: «path to MATLAB»\bin\win64 **Note:** If a PATH variable already exists, append the value to it. Make it the first value of the variable. You must restart the computer afterward before continuing, otherwise the Simulink Importer will not work.

In macOS:

> Name: DYLD_LIBRARY_PATH
>
> Value: «path to MATLAB»/bin/maci64

In Linux:

> Name: LD_LIBRARY_PATH
>
> Value: «path to MATLAB»/bin/glnxa64:«path to MATLAB»/sys/os/glnxa64

Install required dependencies

The required dependencies are Epsilon, Viatra and Xtext.

1. Go to **Help** » **Install New Software...**.
2. Work with http://download.eclipse.org/epsilon/1.4/updates/ to install Epsilon.
   Select the following features:
   - Epsilon Core v1.4.0
   - Epsilon Core Development Tools v1.4.0
   - Epsilon EMF Integration v1.4.0
   - Epsilon UML Integration v1.4.0
3. Work with http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/ to install Xtext.
   Select the following features:
   - Xtext » Xtext Complete SDK v2.12.0
4. Work with http://download.eclipse.org/viatra/updates/release to install Viatra.
   Select the following features:
   - VIATRA Core » VIATRA Query and Transformation SDK v1.6.1

Install the *checsdm4uss* plug-ins

1. Place the *checsdm4uss* plug-ins provided with this report under «path to Eclipse »/plugins.
2. (Re)Start the Eclipse IDE.

checsdm4uss plug-ins

- ca.ets.avio604.checsdm.guidelinecompliance_1.0.0.jar
- ca.ets.avio604.checsdm.guidelinecompliance.validation_1.0.0.jar
- ca.ets.avio604.checsdm.mappingrules.ui_1.0.0.jar
- ca.ets.avio604.checsdm.mappings_1.0.0.jar
- ca.ets.avio604.checsdm.mappings.edit_1.0.0.jar
- ca.ets.avio604.checsdm.mappings.editor_1.0.0.jar
- ca.ets.avio604.checsdm.matlabengine_1.0.0.jar
- ca.ets.avio604.checsdm.simulink_1.0.0.jar
- ca.ets.avio604.checsdm.simulink.edit_1.0.0.jar
- ca.ets.avio604.checsdm.simulink.editor_1.0.0.jar
- ca.ets.avio604.checsdm.simulink.importer.api_1.0.0.jar
- ca.ets.avio604.checsdm.simulink.importer.ui_1.0.0.jar
- matlabengine.bundle_1.0.0.jar
- org.apache.commons.text_1.0.0.jar

Import the *checsdm4uss* mapping rules project

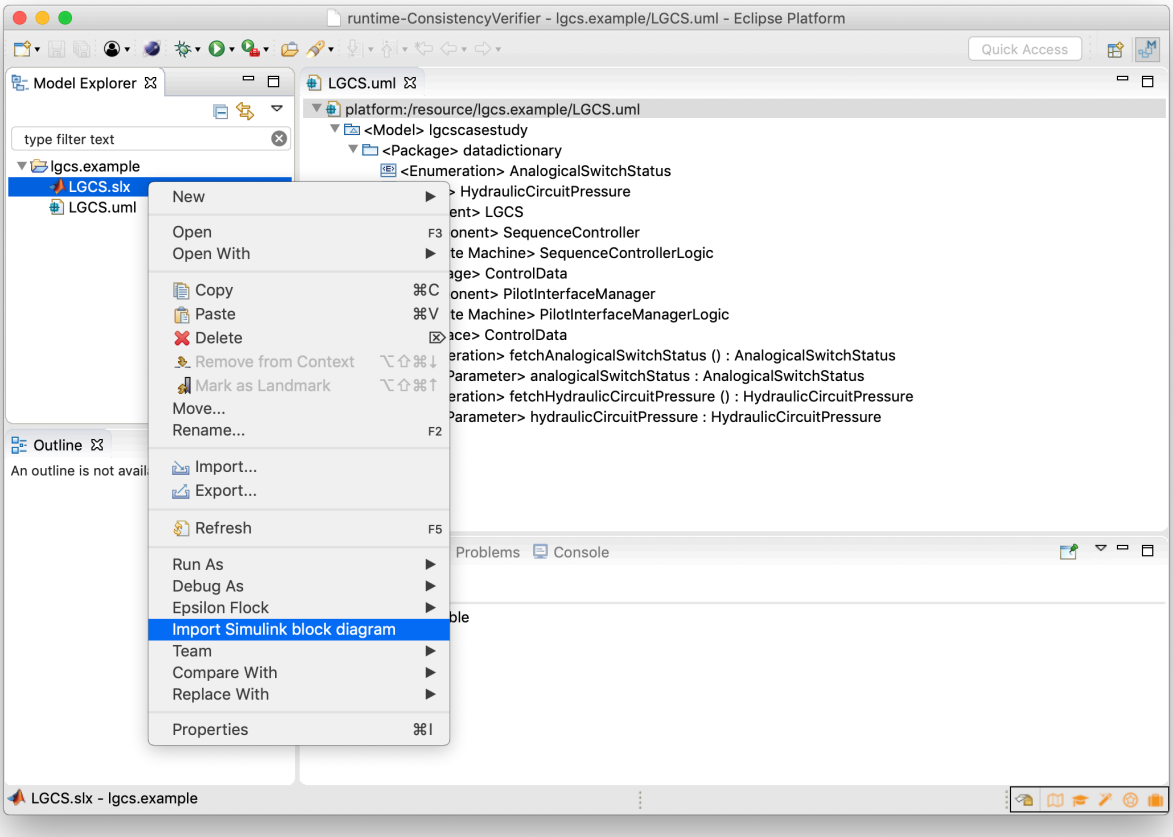
1. Import the **ca.ets.avio604.checsdm.mappingrules** project provided with this report into the Eclipse workspace.

5.3.2 Importing a Simulink model into Eclipse

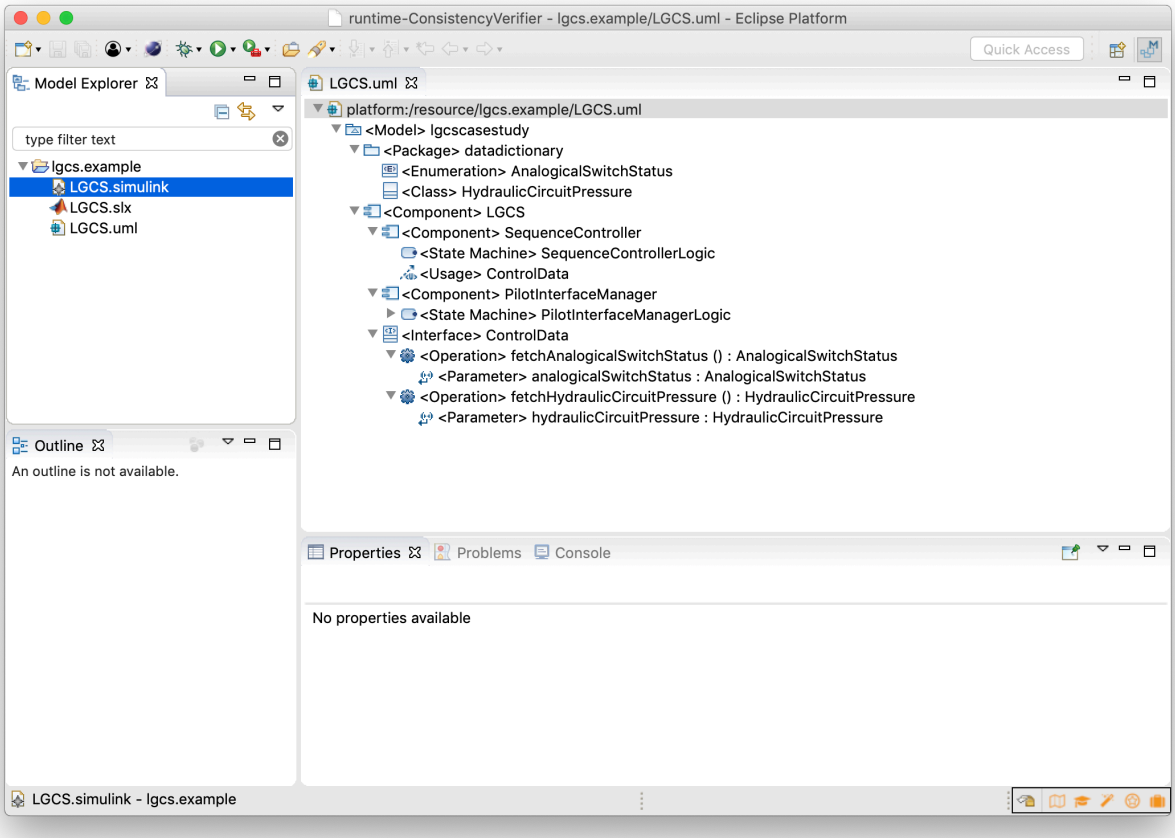
The following steps guide the import of a Simulink model into the Eclipse workspace.

1. Open the Eclipse IDE.
2. Create a project.
3. Move or copy and paste the .slx Simulink model file into the project.
4. Select the .slx file.

5. Right click on the .slx file and select **Import Simulink block diagram**.



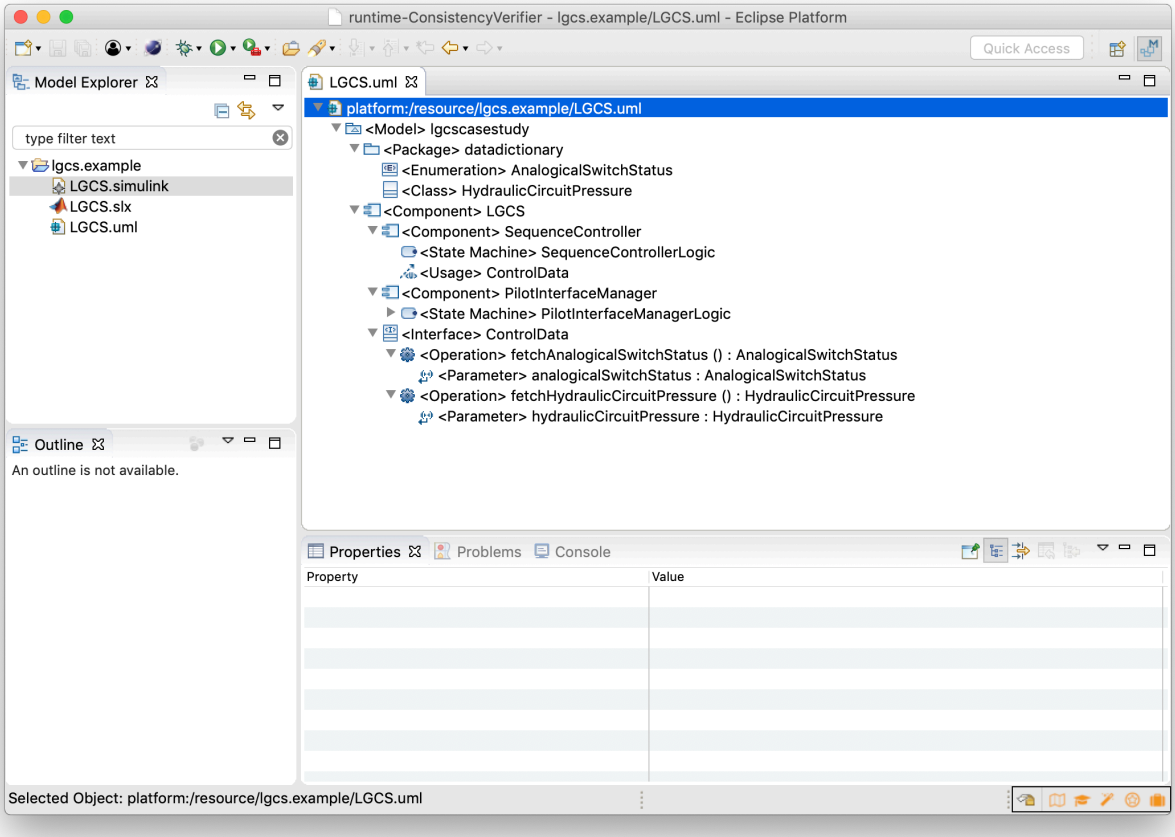
6. The import process starts. The import process may take a few minutes. MATLAB may pop up. This is the normal behaviour. Once the import process ends the MATLAB window will close and the EMF Simulink model should appear inside the project. If the block diagram makes use of library files, a dialog may pop up asking if these should be imported. A dialog may pop up if library models are used but could not be imported.



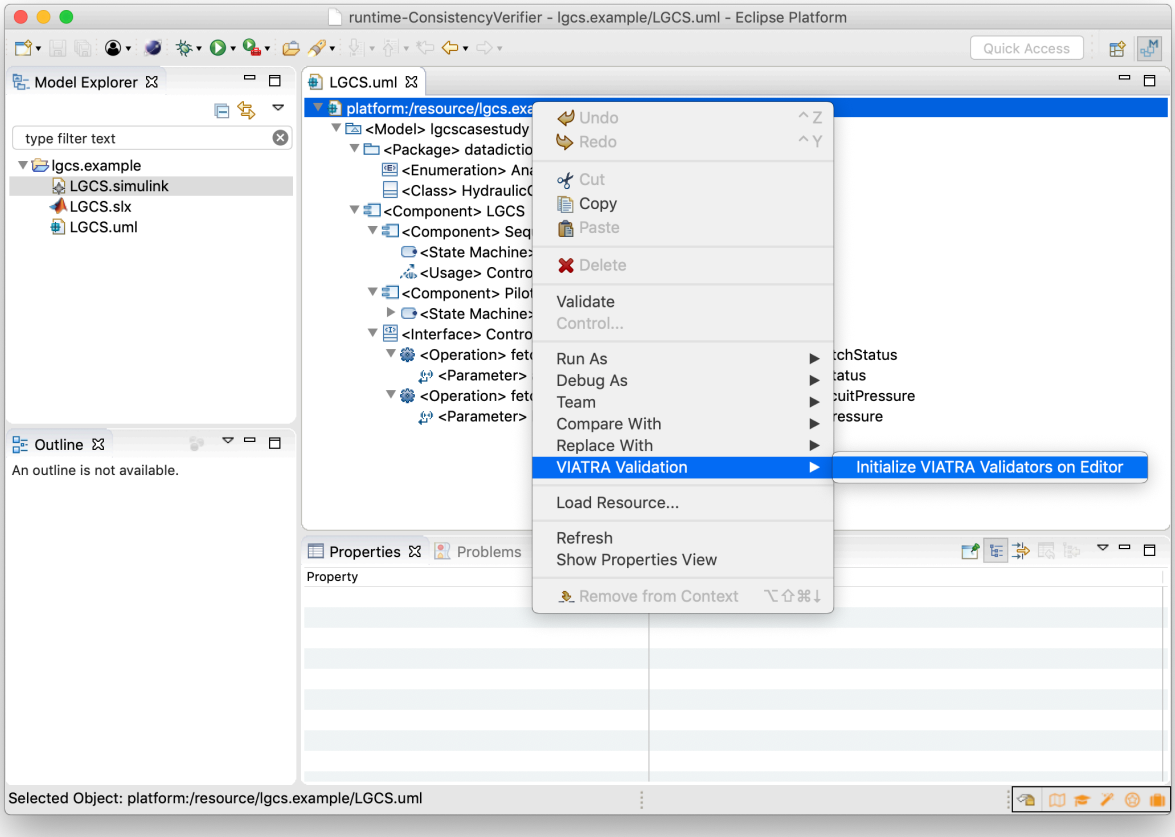
5.3.3 Verify guideline compliance

The following steps guide the verification of guideline compliance on any of the supported models (*i.e.* UML, Simulink/Stateflow, mapping model).

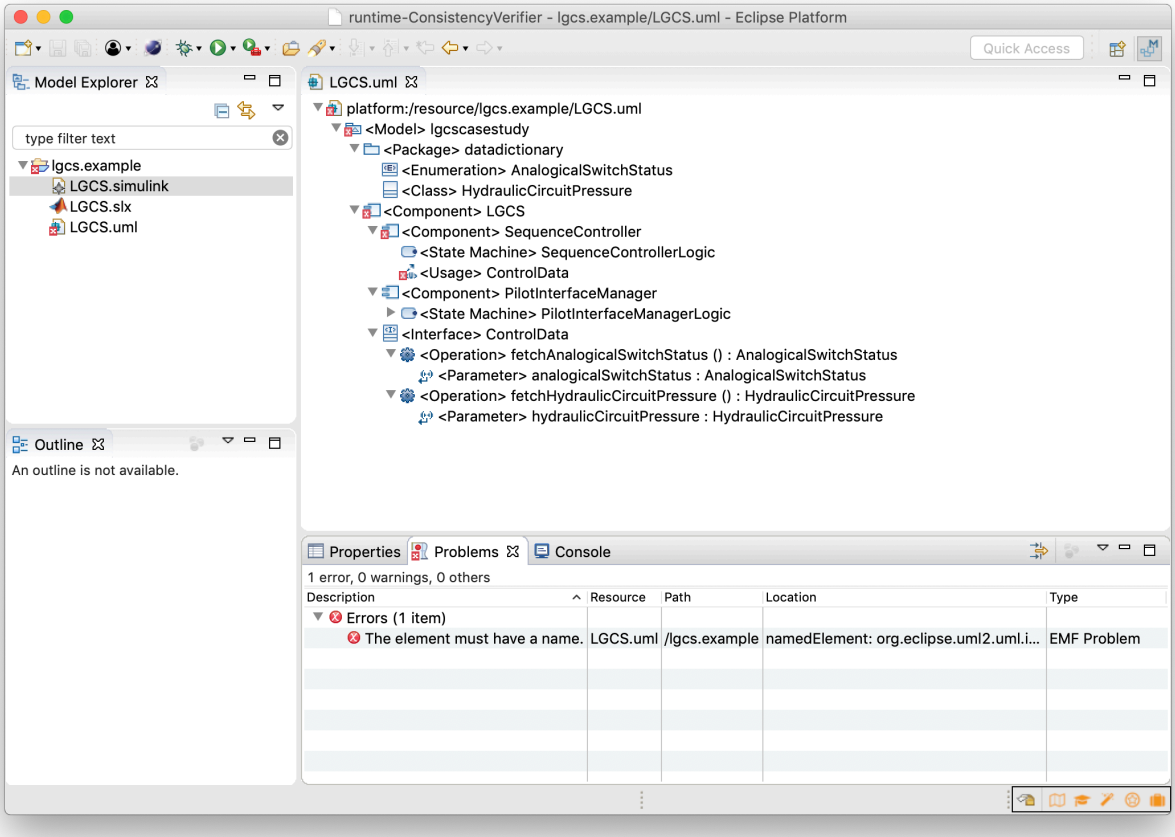
1. Open the model to be verified for guideline compliance. In this case, a UML model.



2. The verification can be carried out at anytime during modelling. Right click anywhere in the model and select **VIATRA Validation** » **Initialize VIATRA Validators on Editor**.



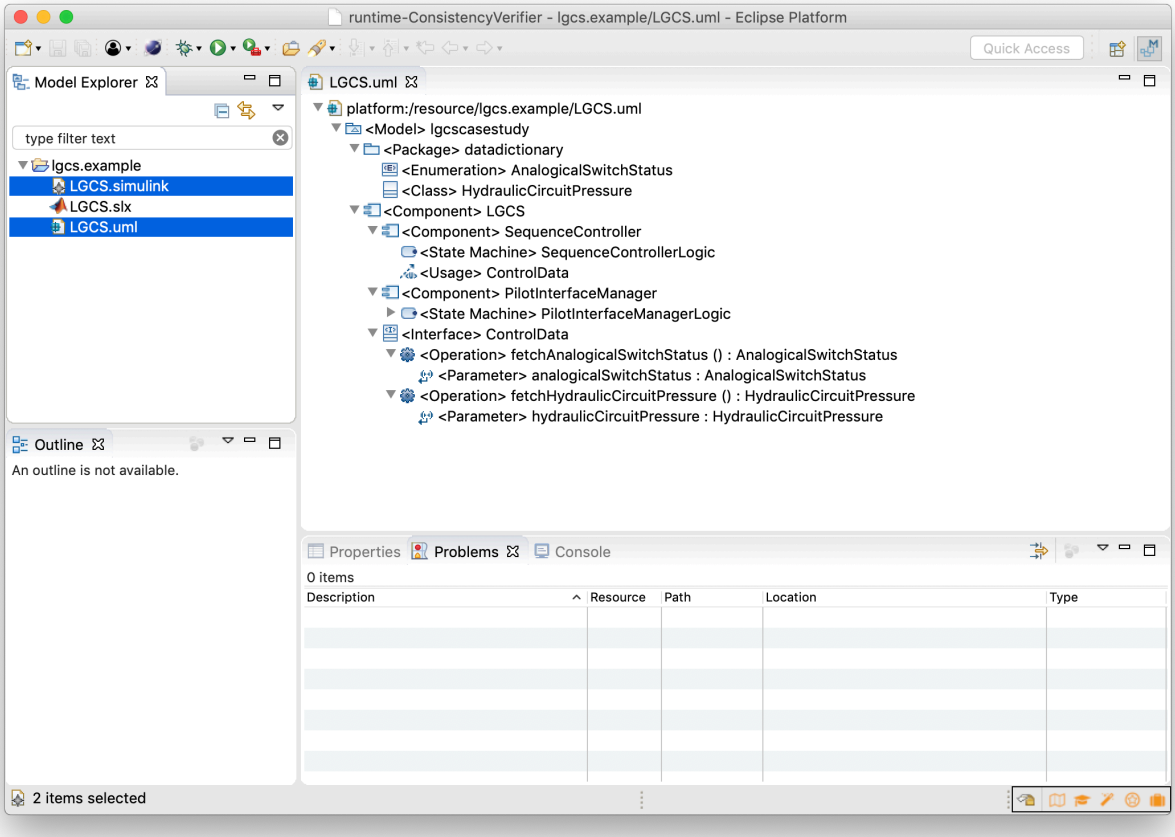
3. Violations will be presented in the **Problems** view (bottom of the screen). Any violating element will be marked in the editor view as well. To fix a violation, follow the indications given in the violation message.



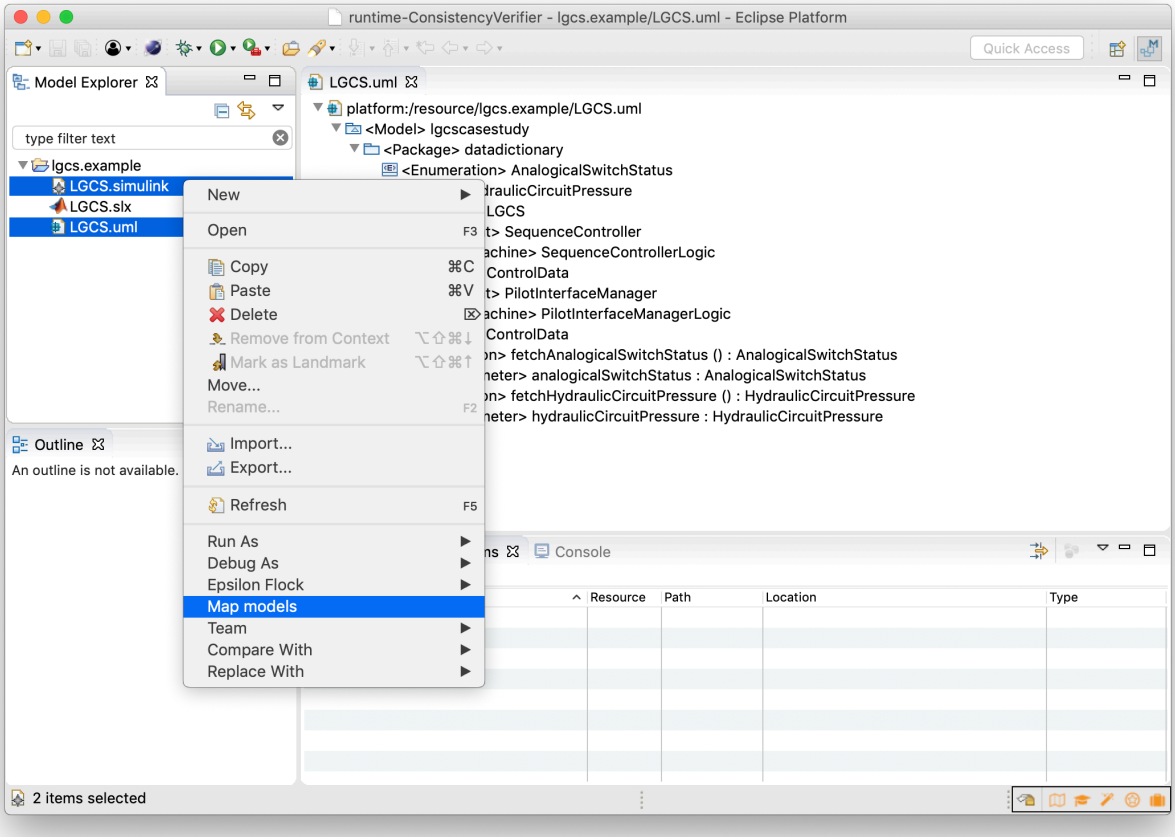
5.3.4 Map models

The following steps guide the mapping of design models.

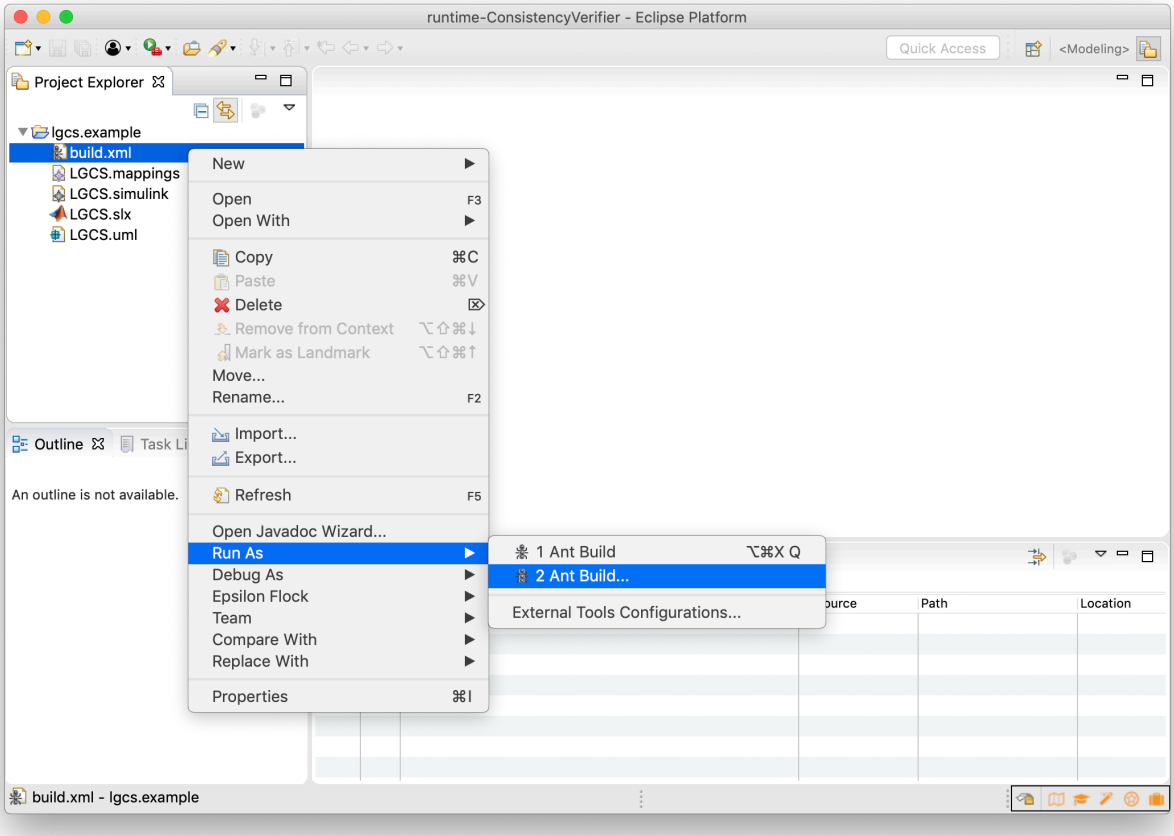
1. Select two models to map. In this case a Simulink EMF and UML models.



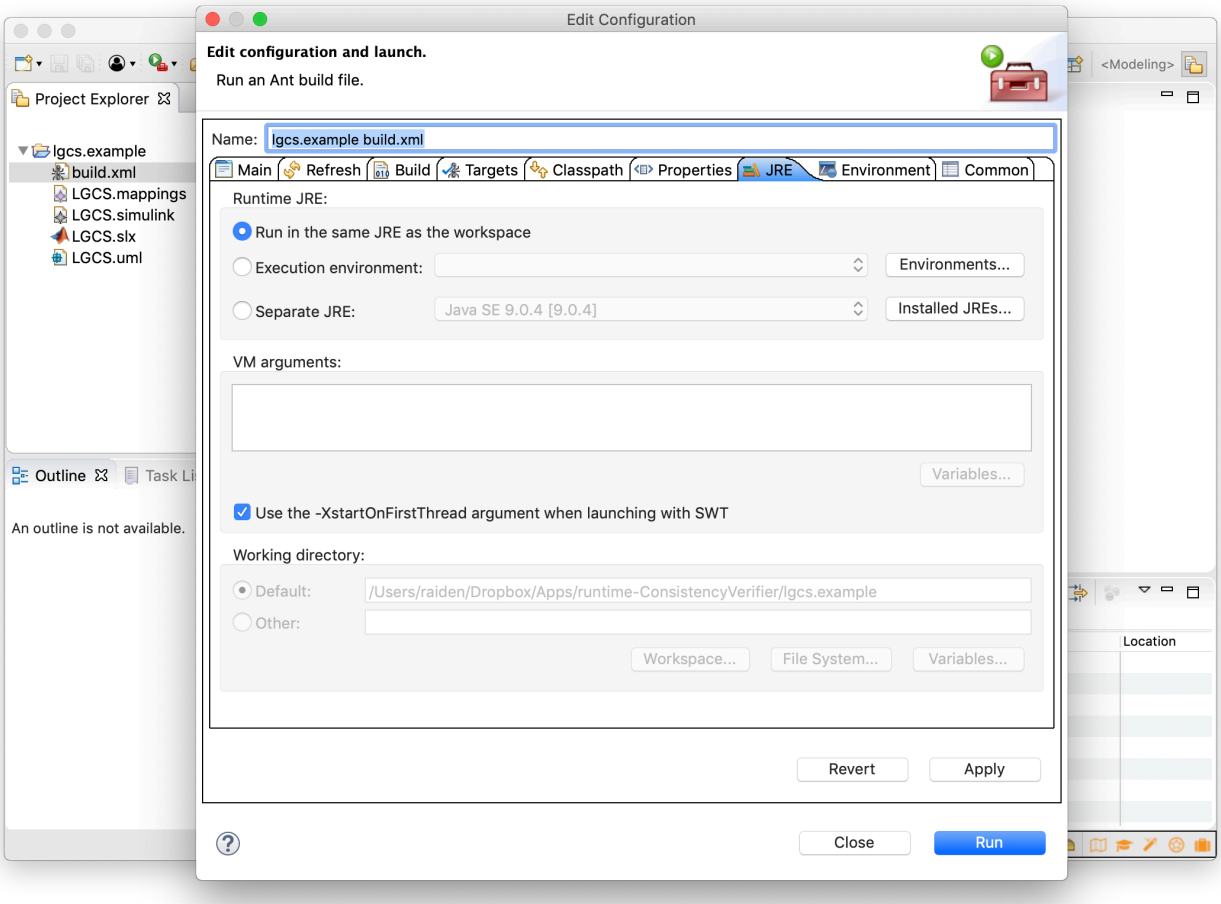
2. Right click over one of the selected models and select **Map models**.



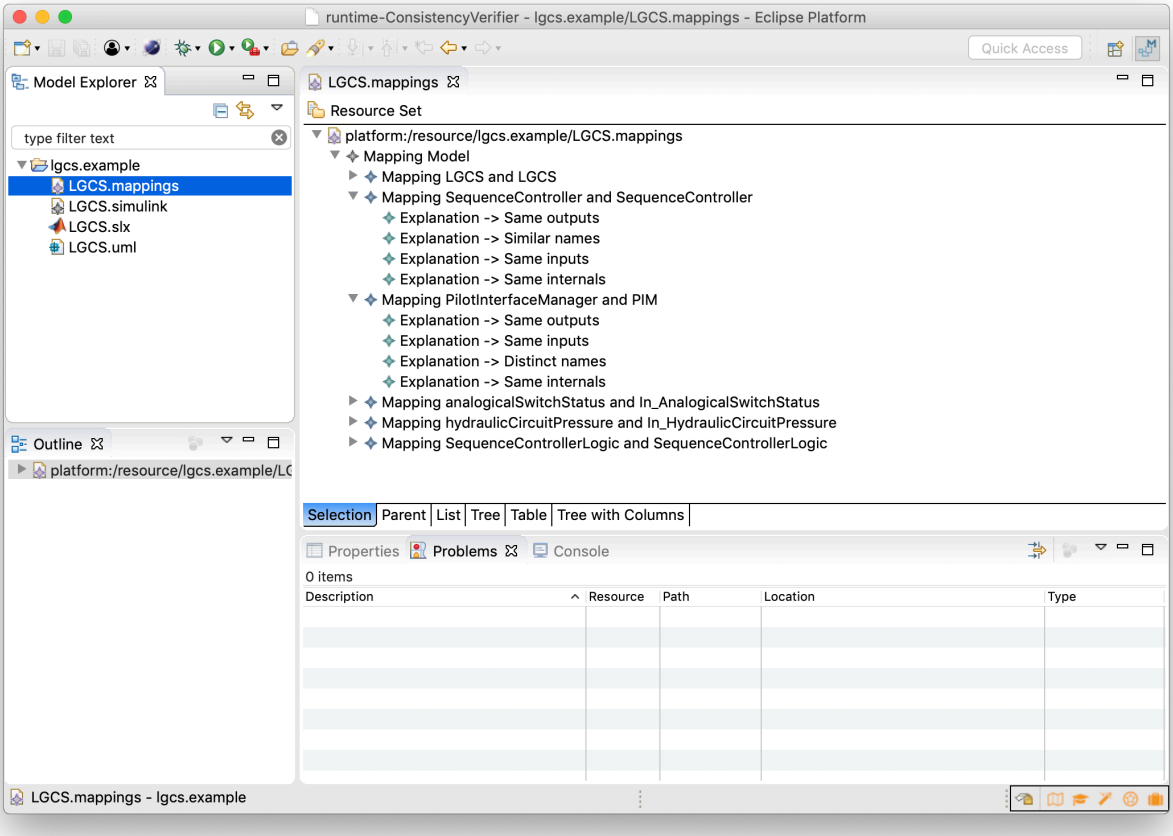
3. If the mapping fails, refresh the project and right click over the **build.xml** file that was generated inside the project. Go to **Run As** » **Ant Build...**.



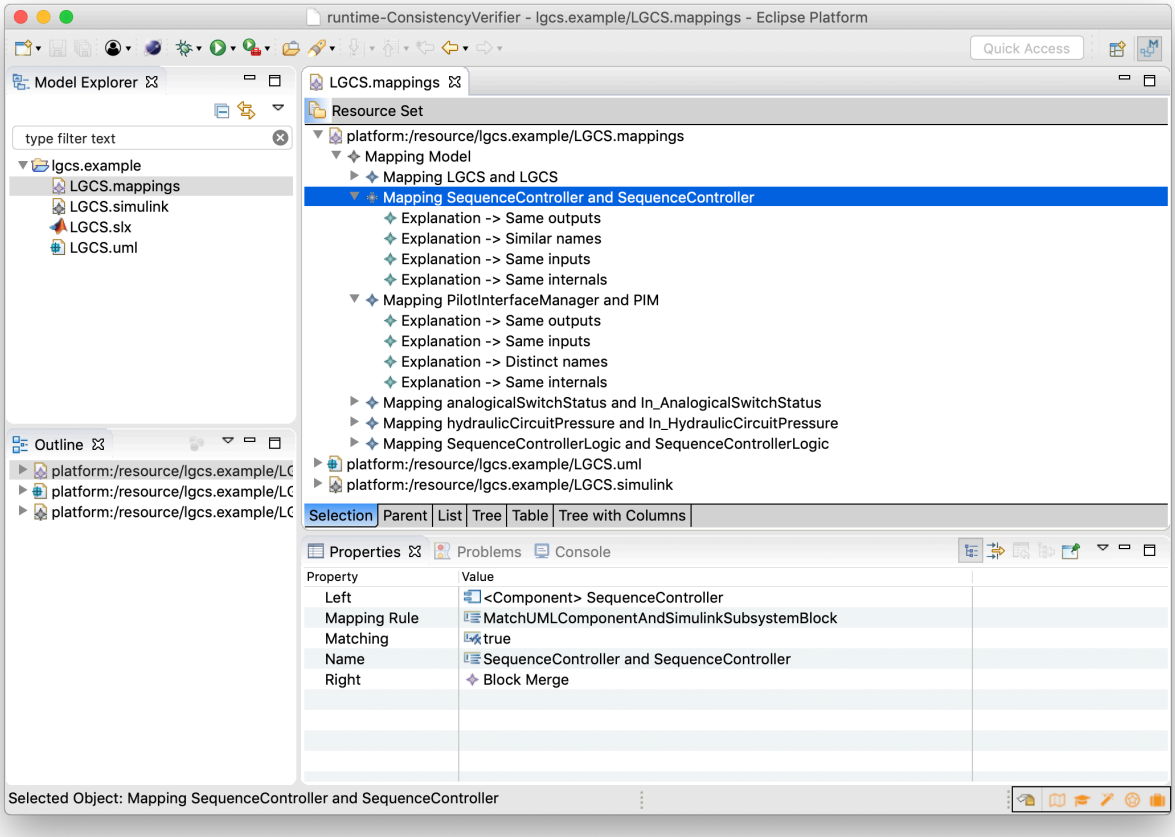
4. Select the **JRE** tab. Make sure the **Runtime JRE** is set to **Run in the same JRE as the workspace**. Click on **Apply** and then **Run**. This will rerun the mapping models operation.



5. Refresh the project. The mapping model should appear inside the project.



6. Select a mapping to review its details in the **Properties** view (bottom of the screen).



References

[1] "Control algorithm modeling guidelines using MATLAB, Simulink, and Stateflow", MathWorks Automotive Advisory Board (MAAB), Tech. Rep., 2012.

[2] D. Kolovos, L. Rose, A. García-Domínguez, and R. Paige, "The Epsilon Comparison Language (ECL)", in *The Epsilon Book*. Eclipse Epsilon Project, 2018.

[3] The Eclipse Foundation. (). Viatra, [Online]. Available: https://www.eclipse.org/viatra/ (visited on 10/19/2018).

[4] A. Ferrari, A. Fantechi, G. Magnani, D. Grasso, and M. Tempestini, "The Metrô Rio case study", *Science of Computer Programming*, vol. 78, no. 7, pp. 828–842, 2013.

[5] IncQuery Labs. (). Massif: MATLAB Simulink Integration Framework for Eclipse, [Online]. Available: https://incquerylabs.com/en/page/show/massif (visited on 03/29/2018).

[6] F. Boniol and V. Wiels, "The landing gear system case study", in *ABZ 2014: The Landing Gear Case Study*. Springer, 2014.

[7] A. Paz and G. El Boussaidi, "Building a software requirements specification and design for an avionics system: An experience report", in *Proceedings of SAC 2018: Symposium on Applied Computing*, ser. SAC 2018, Pau, France: ACM, Apr. 2018.

[8] A. Sarkis and L. A. V. Dias, "A set of rules for production of design models compliant with standards DO-178C and DO-331", in *International Conference on Information Technology: New Generations*, 2014.

[9] W. S. Levine, *The control handbook*, ser. The electrical engineering handbook series. CRC Press New York, 1996.