SpecML

Documentation for the modelling language and its reference implementation

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

CFC	**C**ontribution to **F**ailure **C**ondition
FCS	**F**light **C**ontrol **S**oftware
HEV	**H**ydraulic **E**lectro-**V**alve
HLR	**H**igh-**L**evel **R**equirement
LGCS	**L**anding **G**ear **C**ontrol **S**oftware
LLR	**L**ow-**L**evel **R**equirement
PBR	**P**roperty-**B**ased **R**equirement
PID	**P**roportional/**I**ntegral/**D**erivative
SRATS	**S**ystem **R**equirement **A**llocated to **S**oftware

1 Introduction

1.1 SpecML

SpecML is a modelling language that provides a requirements specification infrastructure for avionics software in the context of DO-178C [1]. SpecML is designed as a UML [2] profile augmenting SysML [3] Requirements. The goal of SpecML is threefold. First, enforce certification information mandated by DO-178C like the interrelationships and decomposition of requirements. Second, capture requirements in natural language to smooth the way for its adoption in industry. Third, provide facilities to capture requirements in a structured, semantically-rich formalism to enable requirements-based analyses and testing. The language is open and may be tailored to other industries and regulatory guidelines and standards.

1.2 Foundations for structured, semantically-rich requirements

One of the notable features of SpecML is its ability to capture requirements in a structured, semantically-rich formalism in order to enable requirements-based analyses and testing. This feature is founded on the property-based requirement (PBR) theory as defined in [4, 5]. PBR theory is a method for solving the problems of ambiguity, inconsistency and incompleteness in natural language-based requirements specifications. A PBR for a system Σ is defined as a constraint over a property P of an object O in Σ. The constraint enforces that the value of P is located within a domain D, which is a subset of $im(P)$ (*i.e.* the domain of possible values of P), when a condition C is met. The following expression formalizes a PBR: Req: $[\,\text{when } C\,] \rightarrow val(O.P) \in D \subset im(P)$. The term `Req` is a mandatory, unique requirement identifier. The rest of the expression is intended to read as follows: *"when condition C is met, the value(s) of property P of object O shall be in the subset D of $im(P)$"*. The presence of a condition C is optional as indicated by the presence of square brackets. The theory states that the conjunction of a finite set of PBRs $\{\text{Req}_n\}$ denotes the system Σ.

PBR theory lacks constructs for expressing timing constraints and clocks. In avionics systems, for instance, the values of their properties commonly need to be evaluated repetitively at a set frequency. This information is part of the PBR but cannot be captured. The MARTE profile [6] (particularly the *CoreElements*, *Time* and *NFPs* sub-profiles) contains constructs for the expression of such time-dependent behaviour and constraints. Thus, SpecML borrows these constructs from the MARTE profile to annotate PBRs that are time-sensitive.

1.3 About this Report

This report offers documentation and examples useful in working with SpecML and its reference implementation.

1.3.1 Audience

This report is intended for engineers wishing to use or extend SpecML. Engineers wishing to work with SpecML are expected to know and understand UML, SysML and MARTE. Engineers wishing to work with SpecML's reference implementation are expected to know and understand the Eclipse Papyrus modelling environment [7] and the Eclipse Modeling Framework (EMF).

1.3.2 What's in the Report

This report will cover all aspects of working with SpecML and its reference implementation.

Chapter 2 presents the specification of the stereotypes and constraints in SpecML.

Chapter 3 gives two usage examples in the context of avionics software.

Chapter 4 offers documentation on and example of usage of SpecML's reference implementation.

2 Profile Constructs

SpecML has the common structure of a UML profile: data types, stereotypes and constraints. No custom data types are defined, hence, UML primitive types are used to define the attributes owned by the stereotypes. Specialized stereotypes are defined to capture key concepts of requirements specification in accordance with DO-178C. Constraints are defined to enforce additional necessary checks to achieve DO-178C objectives.

The stereotypes and constraints in SpecML are categorized into four groups:

1) Requirement hierarchy
2) Requirement interrelationship
3) Requirement formalization
4) Data dictionary

Following sections describe in detail the stereotypes and constraints that belong to each of these groups.

2.1 Requirement Hierarchy

Figure 2.1 shows the stereotypes to represent DO-178C's requirement hierarchy.

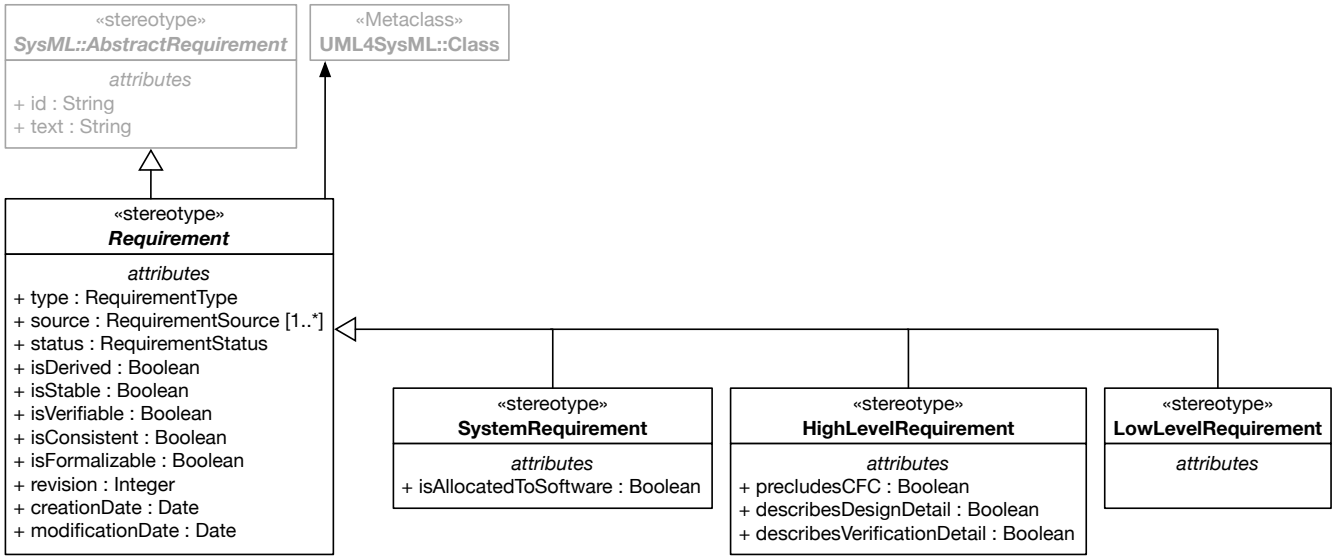


FIGURE 2.1: Requirement hierarchy stereotypes.

`Requirement`

Description:

A `Requirement` defines the general attributes and relationships essential for requirements specification under DO-178C. Specifically the stereotype generalizes the different kinds of requirements described in DO-178C, namely system requirements, high-level software requirements (HLRs) and low-level software requirements (LLRs). A `Requirement` conveys an understanding of what needs to be performed by the system of interest.

Attributes:

- **id** : *String [1]*

  The unique ID of the requirement. Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the SysML 1.5 Specification [3]).

- **text** : *String [1]*

  The requirement statement as a natural language statement. Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the SysML 1.5 Specification [3]).

- **type** : *RequirementType [1]*

  The type of the requirement. Possible values are: structural, behavioural or mixed. Structural requirements concern to a structural property (*i.e.* composition and structure) of the system being specified. Behavioural or functional requirements concern to a behavioural/functional property (*i.e.* observable behaviour) of the system being specified. Mixed requirements concern to both structural and behavioural properties of the system being specified.

- **source** : *Source [1..*]*

  The source who expresses the requirement. Possible values are: acquirer, operator, certification authority, specialty engineer, other stakeholder, or another source like certification standard, safety, costs, environmental conditions, design, production, tests, or maintenance. A requirement may have multiple sources.

- **status** : *RequirementStatus [1]*

  The status of the reviewing and acceptance of the requirement. Possible values are: pending review, reviewed and accepted, and reviewed and incorrect.

- **isDerived** : *Boolean [1]*

  Indicates if a requirement is a derived requirement under DO-178C, *i.e.* a requirement that (a) is not directly traceable to a higher level requirement, and/or (b) specifies behaviour beyond that which is specified by the system requirements or the high-level requirements.

- **isStable** : *Boolean [1]*

  Indicates if a requirement is unlikely to be changed.

- **isVerifiable** : *Boolean [1]*

  Indicates if requirement-based verification activities can be carried out.

- **isConsistent** : *Boolean [1]*

  Indicates if the requirement is consistent according with the review of the requirement.

- **isFormalizable** : *Boolean [1]*

  Indicates if the requirement statement can be expressed using a formalism that can facilitate its analysis and verification.

- **revision** : *Integer [1]*

  Indicates the number of times the requirement has been modified.

- **creationDate** : *Date [1]*

  The date on which the requirement was created.

- **modificationDate** : *Date [1]*

  The date on which the requirement was last modified.

- **/derived** : *AbstractRequirement [*]*

  Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the SysML 1.5 Specification [3]).

- **/derivedFrom** : *AbstractRequirement [*]*

  Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the SysML 1.5 Specification [3]).

- **/satisfiedBy** : *NamedElement [*]*

  Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the SysML 1.5 Specification [3]).

- **/tracedTo** : *NamedElement [*]*

  Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the SysML 1.5 Specification [3]).

- **/verifiedBy** : *NamedElement [*]*

  Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the SysML 1.5 Specification [3]).

- **/master** : *AbstractRequirement*

  Inherited from `AbstractRequirement` (see clause 16.3.2.1 AbstractRequirement from the SysML 1.5 Specification [3]).

- **/formalization** : *PropertyBasedStatement [*]*

  Indicates the requirement's formalization as a collection of property-based statements, following the definition given in Chapter 1.2. One property-based statement may not be sufficient to describe the entire domain captured in the requirement's text attribute. Thus, a collection of property-based statements may be introduced as part of the requirement's formalization. The requirement is, therefore, interpreted as the conjunction of the specified property-based statements.

Constraints:

1. The `id` must be specified and be unique.

   This constraint is applicable to adhere to DO-178C compliance needs for software level D and above.

2. A `Requirement` with the `isDerived` flag must be justified by a `Rationale` (see clause 7.3.2.5 Rationale from the SysML 1.5 Specification [3]).

   This constraint is applicable to adhere to DO-178C compliance needs for software level C and above.

3. A `Requirement`, or any of its subclasses, shall not participate as the client in dependencies stereotyped by `Satisfy`.

   The `Satisfy` relationship is a dependency between a requirement and a model element that fulfills the requirement. However, when the model element is stereotyped as a `LowLevelRequirement` the `Satisfy` relationship takes on the same meaning as the `RefineReqt` relationship. In such case the `RefineReqt` relationship shall

be used. Thus, this constraint is intended to enforce the use of the RefineReqt relationship over the Satisfy relationship.

4. A Requirement, or any of its subclasses, shall not own any nested classifiers stereotyped by Requirement. This constraint is to avoid the creation of compound requirements and subrequirements.

SystemRequirement

Description:

A SystemRequirement specifies a capability or condition that the system (or a part of it) must satisfy.

Generalizations:

- Requirement.

Attributes:

- **isAllocatedToSoftware** : *Boolean [1]*
  Indicates if the system requirement has been allocated to software.

Constraints:

1. A SystemRequirement stereotype must not be applied alongside other stereotypes that specialize the Requirement stereotype.

HighLevelRequirement

Description:

A HighLevelRequirement (HLR) specifies a capability or condition that the software must (or should) satisfy. An HLR is developed from the analysis of system requirements allocated to software (SRATS), safety-related requirements and system architecture. HLRs must be very detailed so as to guide the software's design. It could occur that SRATS are, in fact, very detailed so as to guide the software's design without any further refinement into HLRs. In this case HLRs must be defined and related to their corresponding SRATS using the Copy relationship. The Copy relationship goes from the HLR to the SRATS. HLRs must not include design details nor include verification details in accordance with DO-178C compliance needs.

Generalizations:

- Requirement.

Attributes:

- **precludesCFC** : *Boolean [1]*
  Indicates if the requirement intends to prevent one or more of the identified software contributions to failure conditions (CFCs).

- **describesDesignDetail** : *Boolean [1]*

  Indicates if the requirement statement describes design detail. HLRs should not describe design details except when there is a justified design constraint.

- **describesVerificationDetail** : *Boolean [1]*

  Indicates if the requirement statement describes verification detail. HLRs should not describe verification details except when there is a justified constraint.

Constraints:

1. A `HighLevelRequirement` stereotype must not be applied alongside other stereotypes that specialize the `Requirement` stereotype.

2. A `HighLevelRequirement` without the `isDerived` flag must have a `RefineReqt` or `Copy` dependency to a `SystemRequirementAllocatedToSoftware`.

   This constraint is applicable to adhere to DO-178C compliance needs for software level D and above.

3. A `HighLevelRequirement` with the `describesDesignDetail` flag must justify the existence of the design detail with a `Rationale` (see clause 7.3.2.5 Rationale from the SysML 1.5 Specification [3]).

   This constraint is applicable to adhere to DO-178C compliance needs for software level D and above.

4. A `HighLevelRequirement` with the `describesVerificationDetail` flag must justify the existence of the verification detail with a `Rationale` (see clause 7.3.2.5 Rationale from the SysML 1.5 Specification [3]).

   This constraint is applicable to adhere to DO-178C compliance needs for software level D and above.

`LowLevelRequirement`

Description:

A `LowLevelRequirement` (LLR) specifies a capability or condition that the software must (or should) satisfy. An LLR is developed from the HLRs and must be directly implementable in source code without further information. The stereotype can be used: 1) as a standalone element to capture natural language or formal requirement statements, or 2) to stereotype a design model element as an LLR.

Generalizations:

- Requirement.

Constraints:

1. A `LowLevelRequirement` stereotype must not be applied alongside other stereotypes that specialize the `Requirement` stereotype.

2. A `LowLevelRequirement` without the `isDerived` flag must have a `RefineReqt` dependency to a `HighLevelRequirement`.

   This constraint is applicable to adhere to DO-178C compliance needs for software level C and above.

2.2 Requirement Interrelationship

Figure 2.2 presents the stereotypes for defining the types of relationships that can occur between requirements.

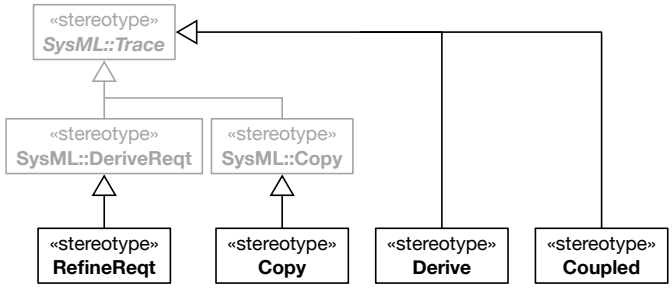


FIGURE 2.2: Requirement interrelationship stereotypes.

Copy

Description:

The Copy relationship is as defined in the SysML specification but with a more constrained usage. A Copy relationship is a dependency between two requirements at different levels of abstraction, *e.g.*, a SystemRequirementAllocatedToSoftware and a HighLevelRequirement. The dependency specifies that the client requirement (*e.g.*, HLR) is a read-only copy of the supplier requirement (*e.g.*, SRATS). The Copy relationship goes from the client requirement to the supplier requirement. For instance, this stereotype is included to make it possible to define SRATS as the HLRs when they are determined to be detailed enough to guide the design activities.

Attributes:

(from UML Dependency class)

- **client** : *NamedElement [1..*]* (from)
- **supplier** : *NamedElement [1..*]* (to)

Constraints:

1. Constraint 1 from the SysMLCopy stereotype is strengthened. A Copy dependency can only be created between subtypes of Requirement as follows:
   - SystemRequirementAllocatedToSoftware (supplier) – HighLevelRequirement (client)
   - HighLevelRequirement (supplier) – LowLevelRequirement (client)
2. Constraint 2 from the SysMLCopy stereotype. The text property of the client requirement is constrained to be a read-only copy of the supplier requirement.
3. Constraint 1 from the Trace stereotype. The Copy stereotype shall only be applied to dependencies.

`Coupled`

Description:

Requirements at the same level of the requirements hierarchy may experience some interdependence. The `Coupled` stereotype makes it possible to represent such relationship between two requirements. For instance, this stereotype makes it possible to define the interdependence of a requirement of type `StructuralRequirement` with one of type `BehaviouralRequirement`. This stereotype is included as a suggestion from experienced practitioners in the avionics industry.

Attributes:

(from UML Dependency class)

- **client** : *NamedElement [1..*]* (from)
- **supplier** : *NamedElement [1..*]* (to)

Constraints:

1. Constraint 1 from the `Trace` stereotype. The `Coupled` stereotype shall only be applied to dependencies.
2. Constraint 2 from the `Trace` stereotype is strengthened. Dependencies stereotyped by `Coupled` shall have exactly one client and one supplier. Furthermore, dependencies with a `Coupled` stereotype applied must only relate two elements stereotyped by the same subtype of `Requirement`, one of which has type `StructuralRequirement` and the other one has type `BehaviouralRequirement`.

`Derive`

Description:

A `Derive` relationship is a dependency between two requirements, a subtype of `Requirement` with the `isDerived` flag and another subtype of `Requirement` without the `isDerived` flag. The dependency specifies that the client requirement (*i.e.* the requirement with the `isDerived` flag) depends on the supplier requirement (*i.e.* the requirement without the `isDerived` flag), which is the requirement from which the former was derived. This stereotype is included to make it possible to trace the derived requirement indirectly to a higher level requirement through the mediation of the requirement from which it was derived.

Attributes:

(from UML Dependency class)

- **client** : *NamedElement [1..*]* (from)
- **supplier** : *NamedElement [1..*]* (to)

Constraints:

– Constraint 1 from the `Trace` stereotype. The `Derive` stereotype shall only be applied to dependencies.

– Constraint 2 from the Trace stereotype is strengthened. Dependencies stereotyped by Derive shall have exactly one client and one supplier. Furthermore, a Derive dependency can only be created between a subclass of Requirement with the isDerived attribute set to true (client) and a subclass of Requirement with the isDerived attribute set to false (client).

RefineReqt

Description:

A RefineReqt relationship is a bi-directional trace in which a requirement can be developed into a lower-level requirement. This stereotype is included to adhere to DO-178C compliance needs stating that SRATS are developed into HLRs and, in turn, HLRs are developed into LLRs and that there must be bi-directional traceability associations between them. The RefineReqt relationship goes from the refining requirement (*e.g.*, the LLR) to the refined requirement (*e.g.*, the HLR).

Attributes:

(from UML Dependency class)

- **client** : *NamedElement [1..*]* (from)
- **supplier** : *NamedElement [1..*]* (to)

Constraints:

1. Constraint 1 from the Trace stereotype. The RefineReqt stereotype shall only be applied to dependencies.
2. The refined requirement (supplier) shall be an element stereotyped by a subtype of Requirement.
3. The refining requirement (client) shall be an element stereotyped by a subtype of Requirement.
4. Constraint 2 from the Trace stereotype is strengthened. Dependencies with a RefineReqt stereotype applied shall have exactly one refined requirement and one refining requirement as follows:
   - SystemRequirementAllocatedToSoftware (refined) – HighLevelRequirement (refining)
   - HighLevelRequirement (refined) – LowLevelRequirement (refining)

2.3 Requirement Formalization

Figure 2.3 shows the stereotypes to capture requirements in a structured, semantically-rich formalism.

PropertyBasedStatement

Description:

A PropertyBasedStatement establishes a requirement statement formalization of the form: $[\,\text{when } C\,] \rightarrow val(O.P) \in D \subset im(P)$ [5]. The expression $[\,\text{when } C\,]$ is an optional (indicated by the presence of square brackets) *condition* of actualization in the context of the requirement. The expression $val(O.P) \in D \subset im(P)$ is a mandatory *predicate* representing the constraint over the value of a system property. O is an object or a type of object, which may

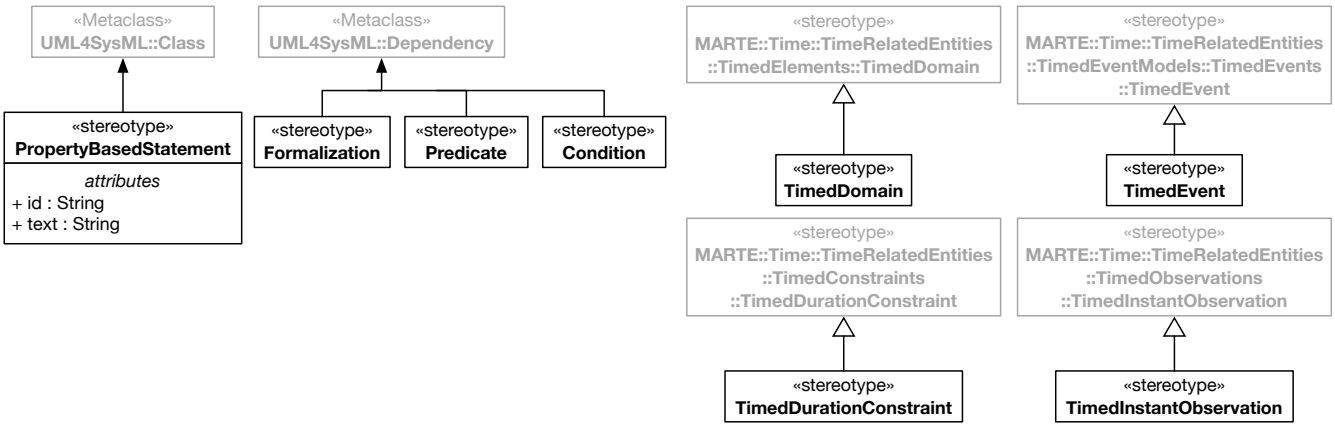


FIGURE 2.3: Requirement formalization stereotypes.

be the system being specified, or an element of its structure or its operating environment. P is a property of O. D is a domain in which the value of $O.P$ must be located when C occurs or is achieved. $im(P)$ denotes the image of P, *i.e.* the domain of possible values of P. D, thus, must be included in $im(P)$. The property-based statement reads as follows: *"when condition C is met, the value(s) of property P of object O shall be in the subset D of $im(P)$".*

Attributes:

- **id** : *String [1]*

  The unique ID of the statement.

- **text** : *String [1]*

  A textual description.

- **condition** : *ConstraintBlock [0..1]*

  The condition of actualization C. The condition is represented as a `ConstraintBlock`. The condition is optional.

- **predicate** : *ConstraintBlock [1]*

  The domain D subset of $im(P)$ in which the value of property P of object O must be located when the condition C occurs or is achieved, *i.e.* $val(O.P) \in D \subset im(P)$. The predicate is represented as a `ConstraintBlock`.

Constraints:

1. A `PropertyBasedStatement` shall not own any structural or behavioural elements beyond the properties that define its condition and predicate expressions.
2. A `PropertyBasedStatement` shall not participate in associations other than the one that binds it to the `Requirement` that it formalizes.
3. A `PropertyBasedStatement` shall not participate in generalizations.
4. A `PropertyBasedStatement` shall not have nested classifiers.
5. A `PropertyBasedStatement` stereotype must not be applied alongside other stereotypes.

`TimedDomain`

Description:

The `TimedDomain` stereotype is as defined by the MARTE specification. A package stereotyped by `TimedDomain` is a container of clocks. Elements in a `TimedDomain` package may use the clocks contained in it to express behaviour that is time dependent. A `TimedDomain` package may own nested `TimedDomain` packages.

Attributes:

- None.

Constraints:

– None.

`TimedDurationConstraint`

Description:

The `TimedDurationConstraint` stereotype is as defined by the MARTE specification. The stereotype imposes a constraint on the temporal distance between two events. This stereotype is included to allow the expression of timing constraints between specified behaviour.

Attributes:

- **kind** : *MARTE::NFP::NFP_Annotation::ConstraintKind [0..1]*

  Specifies the kind of the constraint. The constraint can be qualified by either a *required*, *offered* or *contract* kind. A constraint that is *required* indicates the minimum level that is demanded for the model element. A constraint that is *offered* establishes the space of values that can support the model element. A constraint that is *contract* defines a conditional expression between offered and required values.

- **interpretation** : *TimeInterpretationKind [1]*

  The value of this attribute is fixed to specify that the constraint applies to a duration value.

- **on** : *MARTE::Time::TimeAccesses::Clocks::Clock [1..*]*

  Specifies the associated clock(s).

- **specification** : *DurationPredicate [1]*

  Specifies the constrained duration value.

Constraints:

1. The owner of an element stereotyped by `TimedDurationConstraint` must be a package stereotyped by `TimedDomain`.

`TimedEvent`

Description:

The `TimedEvent` stereotype is as defined by the MARTE specification but with a more constrained usage. The stereotype establishes a non-functional annotation on a requirement indicating that the specified behaviour needs to be performed with a predetermined frequency (*i.e.* it is explicitly bound to a clock).

Attributes:

- **when** : *CVS::ClockedValueSpecification [1]*
  Specifies when the first occurrence occurs.
- **every** : *CVS::DurationValueSpecification [0..1]*
  Specifies the duration that separates successive occurrences of the timed event.
- **repetition** : *Integer [0..1]*
  Specifies the number of repetitive occurrences.
- **isRelative** : *ConstraintBlock [1]*
  Specifies whether the time value is relative (*i.e.* the when property is a time duration value) or absolute (*i.e.* the when property is a time instant value).

Constraints:

1. A `TimedEvent` stereotype must only be applied to an element stereotyped by a subtype of `Requirement`.
2. The owner of an element stereotyped by `TimedEvent` must be a package stereotyped by `TimedDomain`.

`TimedInstantObservation`

Description:

The `TimedInstantObservation` stereotype is as defined by the MARTE specification but with a more constrained usage. The stereotype denotes an instant in time that is associated with an event occurrence and observed on a given clock. This stereotype is included to allow the observation of event occurrences and allowing their use in the expression of timing constraints on the specified behaviour. The stereotype must only be applied to a property-based statement.

Attributes:

- **obsKind** : *EventKind [0..1]*
  Specifies the kind of the observed event. Possible values are: start, finish, send, receive, consume.
- **eocc** : *MARTE::CoreElements::Causality::RunTimeContext::EventOccurrence [1]*
  The associated observed event.
- **on** : *MARTE::Time::TimeAccesses::Clocks::Clock [1..*]*
  Specifies the associated clock(s).

Constraints:

1. A `TimedInstantObservation` stereotype must only be applied to a `PropertyBasedStatement`.
2. The owner of an element stereotyped by `TimedInstantObservation` must be a package stereotyped by `TimedDomain`.

2.4 Data Dictionary

Figure 2.4 shows the stereotypes to represent data dictionaries.

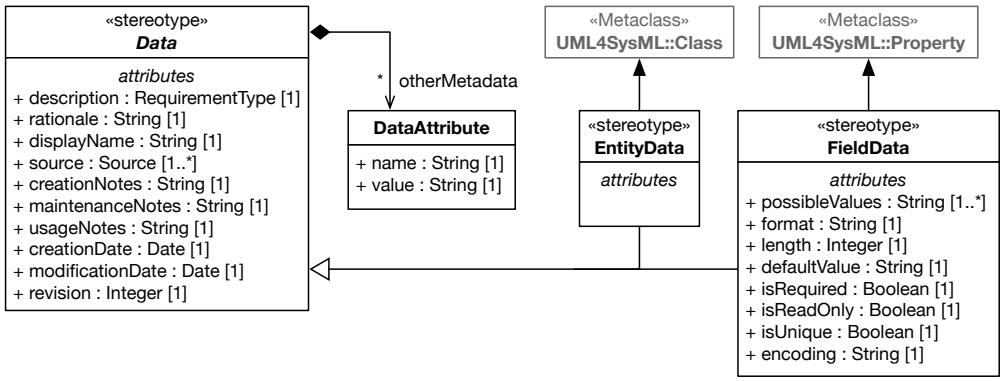


FIGURE 2.4: Data dictionary stereotypes.

`Data`

Description:

The `Data` stereotype represents an entry in the data dictionary. The stereotype defines the essential general metadata. Specifically the stereotype generalizes two kinds of data: composite data (`EntityData`) and atomic data (`FieldData`).

Attributes:

- **description** : *String [1]*

  A description of the contents of the data.

- **rationale** : *String [1]*

  A description of why the data is needed.

- **displayName** : *String [1]*

  The full readable name of the data.

- **source** : *Source [1..*]*

  The source who expresses the data. Possible values are: acquirer, operator, certification authority, specialty engineer, other stakeholder, or another source like certification standard, safety, costs, environmental conditions, design, production, tests, or maintenance. A data may have multiple sources.

- **creationNotes** : *String [*]*

  Notes about how the values for the data may be created.

- **maintenanceNotes** : *String [*]*

  Notes about how the values for the data may be obtained and/or updated.

- **usageNotes** : *String [*]*

  Information about the intended use of the data. May include one or more examples of values the data can take. Examples are intended to be illustrative.

- **creationDate** : *Date [1]*

  The date on which the data was created.

- **modificationDate** : *Date [1]*

  The date on which the data was last modified.

- **revision** : *Integer [1]*

  Indicates the number of times the data has been modified.

- **otherMetadata** : *DataAttribute [*]*

  Extension point allowing the definition of additional information for the data.

Constraints:

– None.

`EntityData`

Description:

The `EntityData` stereotype represents composite data. Composite data does not have a value of its own. Composite data are made up of other composite data or atomic data, which do have a value of their own.

Generalizations:

- Data.

Constraints:

1. None.

`FieldData`

Description:

The `FieldData` stereotype represents atomic data. Atomic data does have a value of its own.

Generalizations:

- Data.

Attributes:

- **possibleValues** : *String [1..*]*

  Indicates values for the data when the value is restricted, for instance to a list or a range.

- **format** : *String [1]*

  Indicates the proper format for entering values to the data.

- **length** : *Integer [1]*

  Indicates the maximum allowed length for the value of the data.

- **defaultValue** : *String [1]*

  Indicates a default value for the data to be used on creation.

- **isRequired** : *Boolean [1]*

  Indicates if the data is mandatory or optional.

- **isReadOnly** : *Boolean [1]*

  Indicates if the data is not meant to be changed.

- **isUnique** : *Boolean [1]*

  Indicates if the data i

- **encoding** : *String [1]*

  Indicates how the data is going to be stored for a later recall.

Constraints:

– None.

3 Usage Examples

In order to illustrate the use of SpecML open source avionics system specifications have been used. These system specifications were reviewed by industrial practitioners who have considered them to be complex and representative of their industrial needs.

3.1 Landing Gear Control Software

The first usage example regards an aircraft's landing gear control software (LGCS). The landing gear system specification was introduced in [8] and developed into a software specification in [9]. Following subsections present a brief system description and focus on fragments of the LGCS requirement specification modelled with SpecML.

3.1.1 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.

3.1.2 Specification model

There are eighteen system requirements allocated to software (identified by the prefix SRATS- and a unique number), which have been refined into eighteen high-level software requirements (identified by the prefix HLR- and a unique number). For the sake of brevity only one SRATS and its refining HLR are discussed.

SRATS-2 Retraction Sequence. When the desired gear position is up, the LGCS shall carry out the retraction sequence in under 28 seconds.

HLR-2 Retraction Sequence Control. When an `Up` value is received for the `Desired Gear Position` monitorable variable, the LGCS shall carry out the retraction sequence in under 28 seconds.

Figure 3.1 presents a fragment of the LGCS specification model in SpecML. HLR-2 describes the behaviour of the LGCS when a desired gear position is given by the pilots. There are a series of actions that the LGCS must perform as part of this requirement and a single formalization statement is not sufficient to cover the whole behaviour. Thus, thirteen property-based statements are introduced to formalize the HLR. The figure only shows formalization statements 2.1 and 2.13. Formalization 2.1 reads as follows: *when the LGCS is functioning normally and is idle, and there is a change in the desired gear position after it has been validated, then the LGCS shall become active*. Formalization 2.13 reads as follows: *when the LGCS is functioning normally and is active, the general hydraulic electro-valve has been set to close, the hydraulic circuit pressure is validated to be under 30,000 kPa, and the analogical switch status is validated to be open, then the LGCS shall become idle*. These two statements

are annotated as timed instant observations. A timed duration constraint is placed on these two timed instant observations to restrict the temporal distance between the two events to be less than 28 seconds.

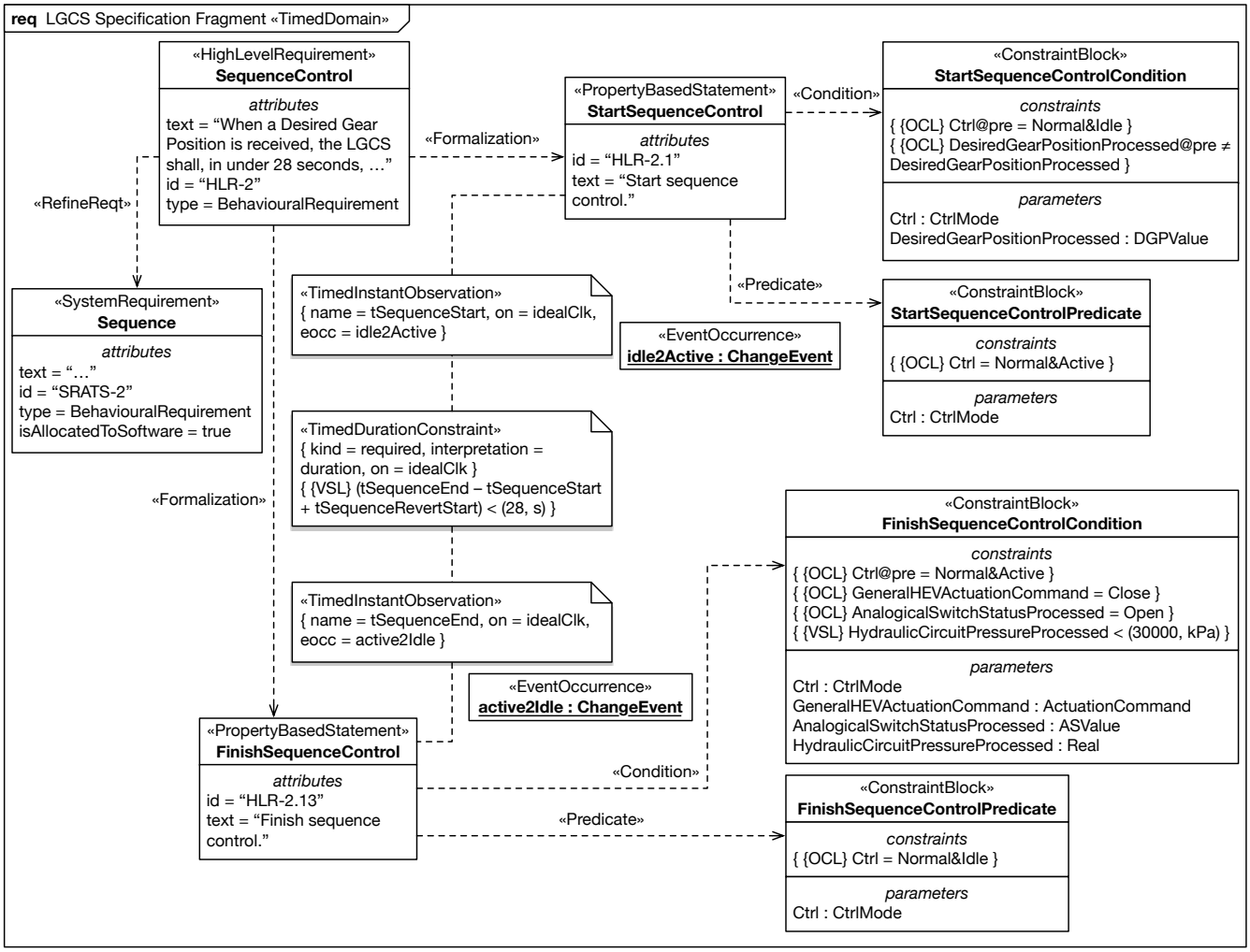


FIGURE 3.1: Fragment of the LGCS specification model.

3.2 Flight Control Software

The second usage example regards an aircraft's flight control software (FCS). An open-source system description and software requirements written in natural language of an FCS is presented in [10]. Following subsections present a brief system description and focus on fragments of the FCS requirement specification modelled with SpecML.

3.2.1 System description

The FCS is the system responsible for providing attitude (*i.e.* the aircraft's orientation about its center of mass) and attitude rate control based on pilot input commands to keep them within the flight envelope of the aircraft. The FCS controls three hydraulic actuators that allow the aircraft to pitch up or down, roll right or left, and yaw right or left.

3.2.2 Specification model

There are eleven system requirements allocated to software (identified by the prefix SR_ and a unique number), which have been refined into thirteen high-level software requirements (identified by the prefix HLR_ and a unique number). For the sake of brevity only one SRATS and its refining HLR are discussed.

SR_4 Hydraulic Actuator Control Loop Performance. The FCS shall control the hydraulic actuator position with a minimum bandwidth of 10 Hz and a minimum damping of 0.4.

HLR_4 Hydraulic Actuator Loop Control. Each hydraulic actuator loop shall be implemented as a PID (proportional/integral/derivative) control loop operating at a 1 ms frame rate. The proportional gain shall be 0.3. The integral gain shall be 0.12. The derivative gain shall be 0.02.

Figure 3.2 presents a fragment of the FCS specification model in SpecML. The system requirement SR_4 is a system requirement allocated to software, which is refined by HLR_4. HLR_4 describes the behaviour of the hydraulic actuator loop control. There are three hydraulic actuator loops, however, in this case a single formalization statement is sufficient to cover the whole behaviour. Thus, only one property-based statements is introduced to formalize the HLR. The formalization statement reads as follows: *the FCS shall implement the actuator command loop as a proportional/integral/derivative (PID) control loop*. The predicate of this statement has nested SysML constraint blocks that defined separately each term of the PID control loop. Furthermore, the formalization statement is annotated as a timed event that occurs every 1 millisecond for an indefinite number of repetitions.

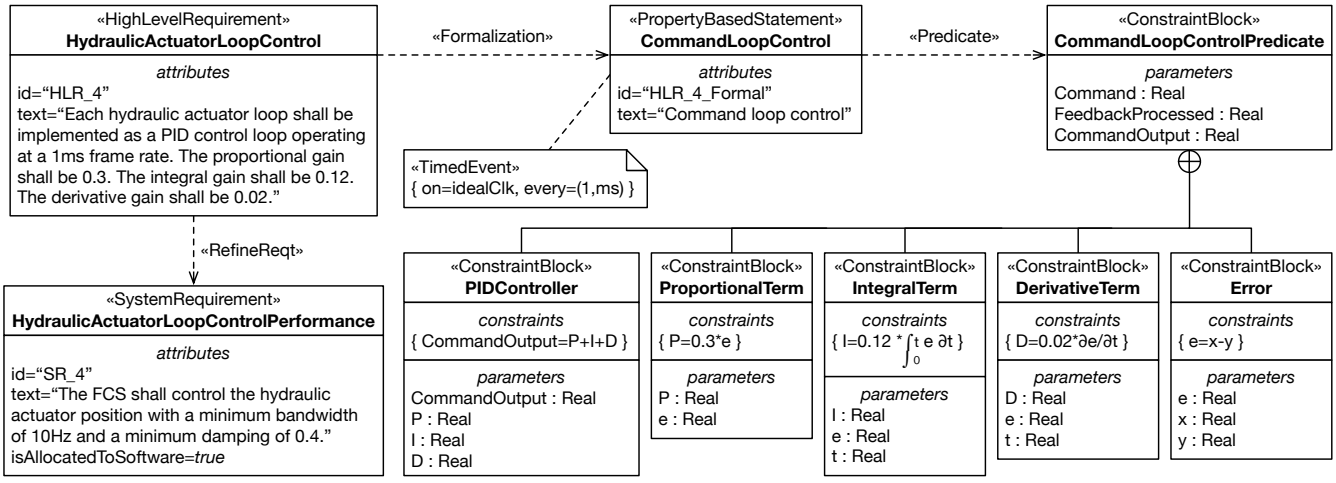


FIGURE 3.2: Fragment of the FCS specification model.

Producing the requirement specification model of the FCS with SpecML is only an intermediate goal. The requirements are allocated to entities of the design models intended to satisfy them. SysML parametric diagrams can include usages of the constraint blocks formalizing the requirements to constrain the properties of the entities in the design models. For instance, binding the parameters of the `CommandLoopControlPredicate` constraint block to the specific properties of the `PitchActuatorLoop`, `RollActuatorLoop` and `YawActuatorLoop` design entities (not shown) intended to satisfy HLR_4, will provide the values for the parameters. This modelling establishes the method of evaluating design compliance with the specified requirements. Figure 3.3 shows a SysML parametric diagram detailing the usage of the nested constraints for the PID control loop in HLR_4. This parametric diagram can be used to bind its parameters to the specific properties in a design model.

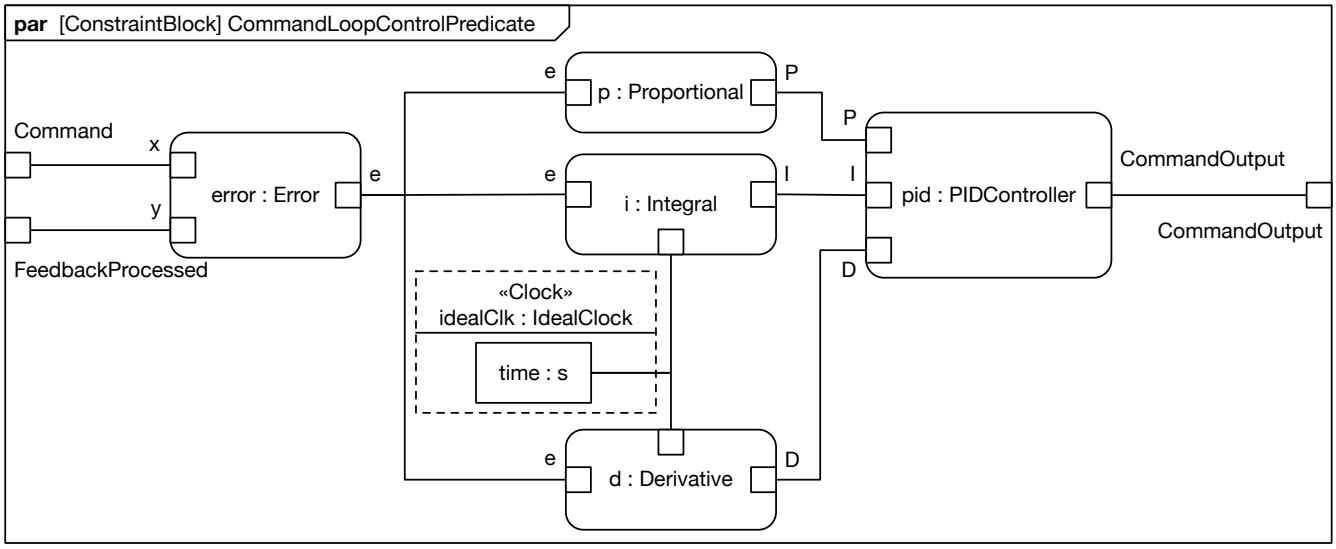


FIGURE 3.3: SysML parametric diagram of the `CommandLoopControlPredicate` constraint block.

4 Reference Implementation

4.1 Overview

SpecML is tool-independent. A reference implementation was developed with the Eclipse Papyrus modelling environment [7], but any UML modelling tool supporting UML profiles could have been used.

The reference implementation comprises three components:

1. The *profile* component defines the stereotypes for the language.
2. The *validation rules* component defines the OCL constraints for the language and utilizes Papyrus' model validation framework for their execution by the user while creating a model.
3. The *modelling tooling* component provides the user with facilities to create a model, *i.e.* editor with palette and context menus, and properties view.

Figure 4.1 displays a screenshot of the SpecML reference implementation. The middle of the screen shows the model editor with a model being created. On the right of the screen is the palette with the available language constructs. On the left center of the screen is the model explorer presenting all the elements currently in the model. The bottom of the screen displays a model validation error message indicating the violation of an OCL constraint by one of the model elements. The element causing the violation is marked in both the model editor and model explorer. The error message suggests options to the user for fixing the violation.

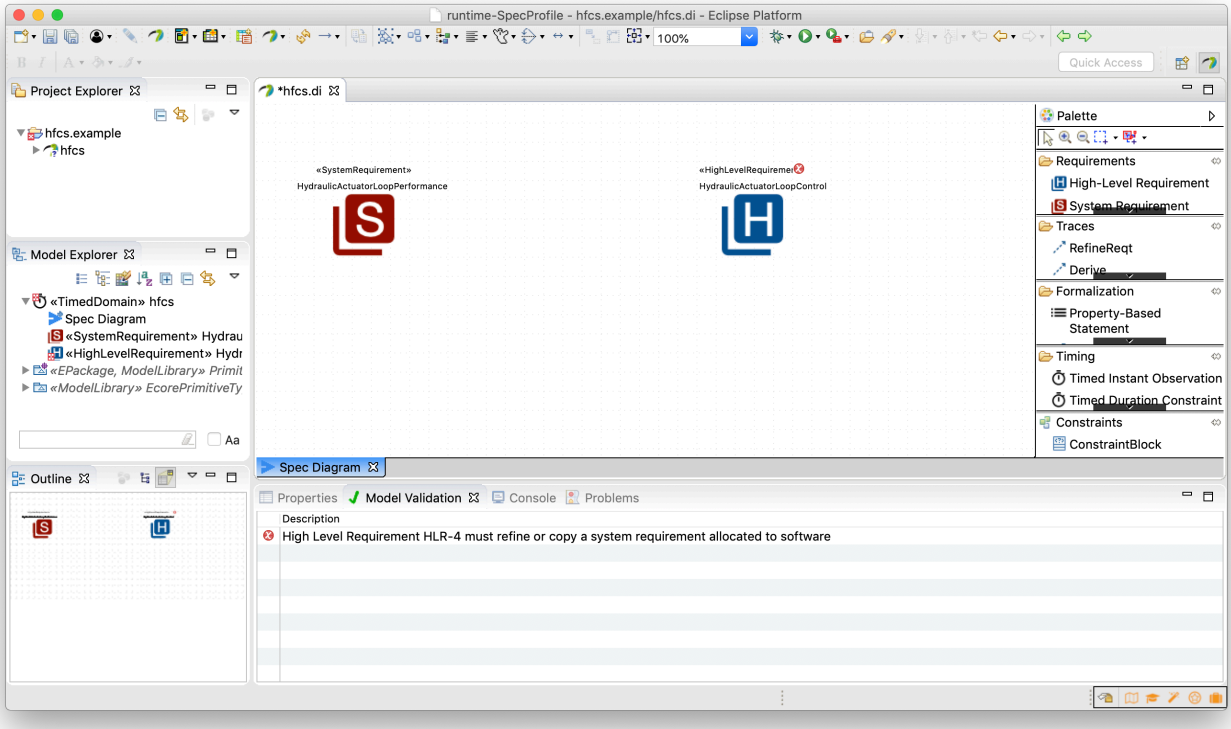


FIGURE 4.1: Screenshot of the SpecML reference implementation.

4.2 Developer's Guide

4.2.1 Getting started

System requirements

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

Install required dependencies

The required dependency is Papyrus modelling environment.

1. Go to **Help** » **Install New Software...**.
2. Work with `http://download.eclipse.org/modeling/mdt/papyrus/updates/releases/oxygen` to install the Papyrus modelling environment.

   Select the following features:

   - Papyrus
   - Papyrus Toolsmiths

Install Papyrus additional components

1. Go to **Help** » **Install Papyrus Additional Components**.
2. Select:

   - Papyrus DSML Validation
   - Papyrus SysML 1.4

Install SpecML

1. Due to a bug in Papyrus, you must place the SpecML plug-ins provided with this report under `«path to Eclipse »/dropins`. **Note:** These plug-ins must be updated with every change to the source code.
2. (Re)Start the Eclipse Papyrus modelling environment.

Import SpecML projects

1. Import the SpecML source code projects provided with this report into the Eclipse workspace.

4.2.2 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 the icons in SpecML's reference implementation.

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

  This folder and file contain descriptions of the extensions to the Eclipse Papyrus modelling environment provided by the plug-in.

- **models**

  This folder contains EMF models with extensions to the Eclipse Papyrus modelling environment.

- **resources**

  This folder contains other resources used in the extensions to the Eclipse Papyrus modelling environment.

- **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 to the Eclipse Papyrus modelling environment provided by the plug-in.

4.2.3 SpecML projects and noteworthy files

- **ca.ets.avio604.specml.architecture**

  This project specifies a new *Avionics* architecture context and the *SpecML* viewpoint in the Eclipse Papyrus modelling environment.

  - **SpecML.architecture**
    EMF model specifying the *Avionics* architecture context and *SpecML* viewpoint. Basic concepts on architecture models and a walkthrough for the definition of new architecture models can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.infra.architecture.doc/target/generated-eclipse-help/architecture.html?cp=79_1_3.
  - **SpecModelCreationCommand.java**
    Initializes a new SpecML model by applying the SpecML profile.

- **ca.ets.avio604.specml.newchild**

  This project specifies the necessary facilities for the creation and manipulation of SpecML constructs within UML models.

  - **NewChild.creationmenumodel**
    EMF model specifying a context menu for the creation of SpecML constructs within a UML model. Basic concepts and how to create or modify creation menu models can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.infra.newchild.doc/target/generated-eclipse-help/newChild.html?cp=79_1_6.

  - **SpecML.elementtypesconfigurations**

    EMF model specifying the high-level model editing facilities for UML models necessary to support the creation and manipulation of SpecML constructs through the context menu. Basic concepts on the ElementTypeConfigurations Framework can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.infra.types.doc/target/generated-eclipse-help/types.html?cp=79_1_7.

- **ca.ets.avio604.specml.palette**

  This project provides a palette in the main SpecML model editor for quick access to SpecML constructs. The palette is organized into categories for a better reference.

  - **SpecML.paletteconfiguration**

    EMF model specifying the elements in the palette and their categories. The Eclipse Papyrus modelling environment provides creation forms for the contents of this model. A demonstration can be found at https://www.youtube.com/watch?v=XnhxHPksbjc.

  - **SpecMLdi.elementtypesconfigurations**

    EMF model specifying the high-level model editing facilities for UML models necessary to support the creation and manipulation of SpecML constructs through the palette. Basic concepts on the ElementTypeConfigurations Framework can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.infra.types.doc/target/generated-eclipse-help/types.html?cp=79_1_7.

- **ca.ets.avio604.specml.profile**

  This project specifies the SpecML profile.

  - **SpecML.profile**

    Papyrus Profile resource model file defining the SpecML profile, its stereotypes and OCL constraints. Java code must be generated from this model through an EMF generator model (genmodel). Basic concepts and a walkthrough on defining profiles and stereotypes in the Eclipse Papyrus modelling environment can be found at https://help.eclipse.org/oxygen/nav/79_0_1_6.

- **ca.ets.avio604.specml.properties**

  This project provides an extension to the *Properties* view in the Eclipse Papyrus modelling environment for editing properties of the SpecML profile that are too hard to edit directly from the main graphical editor or that do not have a graphical representation.

  - **SpecML.ctx**

    EMF model specifying the properties from the SpecML stereotypes that can be edited, the types of widgets to enable editing, and their organization in a form-like view. Basic concepts and a walkthrough on the Properties View framework can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.views.properties.doc/target/generated-eclipse-help/properties-view.html?cp=79_1_0.

- **ca.ets.avio604.specml.style**

  This project specifies custom graphical representations for certain SpecML stereotypes. This is done through the use of Cascading StyleSheets (CSS). Basic concepts and a walkthrough on graphical customizations with CSS can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.infra.gmfdiag.css.doc/target/generated-eclipse-help/css.html?cp=79_1_5.

- **ca.ets.avio604.specml.tables**

  This project provides tabular representations for certain SpecML constructs. Tabular representations are available for SRATS, HLRs and property-based statements. These tabular representations can be modified or new ones can be added. Basic concepts on tables and a walkthrough for table creation can be found at https://help.eclipse.org/oxygen/nav/79_2_4.

- **ca.ets.avio604.specml.validation**

  This project provides constraint validation facilities. This project is generated from the *SpecML.profile* file in the *ca.ets.avio604.specml.profile* project. Basic concepts on constraint validation and a walkthrough on generating the validation plug-in project can be found at https://help.eclipse.org/oxygen/topic/org.eclipse.papyrus.dsml.validation.doc/target/generated-eclipse-help/dsml-validation.html?cp=79_1_2.

4.2.4 More information

More information about the Eclipse Papyrus modelling environment and how to customize it for a particular domain is available at https://help.eclipse.org/oxygen/nav/79_1 and http://wiki.eclipse.org/Papyrus_Developer_Guide?cp=79_2. A YouTube video with more explanations and live demonstrations of the Eclipse Papyrus modelling environment is available at https://www.youtube.com/watch?v=U62b2EQObRg.

4.2.5 Limitations

SpecML's reference implementation is limited by the Eclipse Papyrus modelling environment. Some of the features of SpecML could not be properly implemented because of limitations in Papyrus' UML, SysML and MARTE implementations. Papyrus does not implement SysML 1.5, only SysML 1.4, which limits extensions that can be done to the SysMLRequirement stereotype. Hence, the AbstractRequirement stereotype of SysML 1.5 was manually defined for SpecML's reference implementation. Papyrus' implementation of the MARTE profile was archived and could not be used successfully to integrate it with Papyrus' current implementation of UML. Therefore, stereotypes that were borrowed from MARTE had to be manually defined for SpecML's reference implementation. The previous implementation choices by no means limit SpecML as its reference implementation can be developed differently.

4.3 User's Guide

4.3.1 Getting started

System requirements

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

Install required dependencies

The required dependency is Papyrus modelling environment.

1. Go to **Help** » **Install New Software...**.

2. Work with `http://download.eclipse.org/modeling/mdt/papyrus/updates/releases/oxygen` to install the Papyrus modelling environment.

   Select the following features:

   - Papyrus

Install Papyrus additional components

1. Go to **Help** » **Install Papyrus Additional Components**.
2. Select:
   - Papyrus DSML Validation
   - Papyrus SysML 1.4

Install SpecML

1. Place the SpecML plug-ins provided with this report under `«path to Eclipse »/plugins`.
2. (Re)Start the Eclipse Papyrus modelling environment.

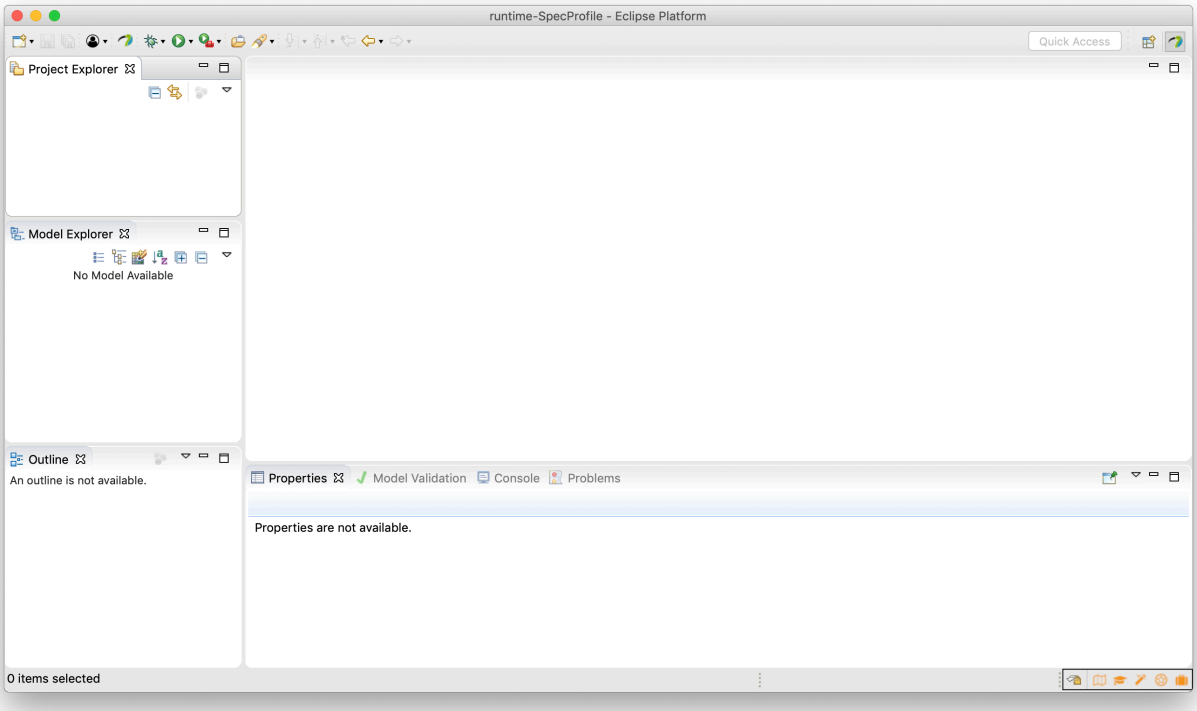
SpecML plug-ins

- ca.ets.avio604.specml.architecture_1.0.0.jar
- ca.ets.avio604.specml.newchild_1.0.0.jar
- ca.ets.avio604.specml.palette_1.0.0.jar
- ca.ets.avio604.specml.profile_1.0.0.jar
- ca.ets.avio604.specml.properties_1.0.0.jar
- ca.ets.avio604.specml.style_1.0.0.jar
- ca.ets.avio604.specml.tables_1.0.0.jar
- ca.ets.avio604.specml.validation_1.0.0.jar

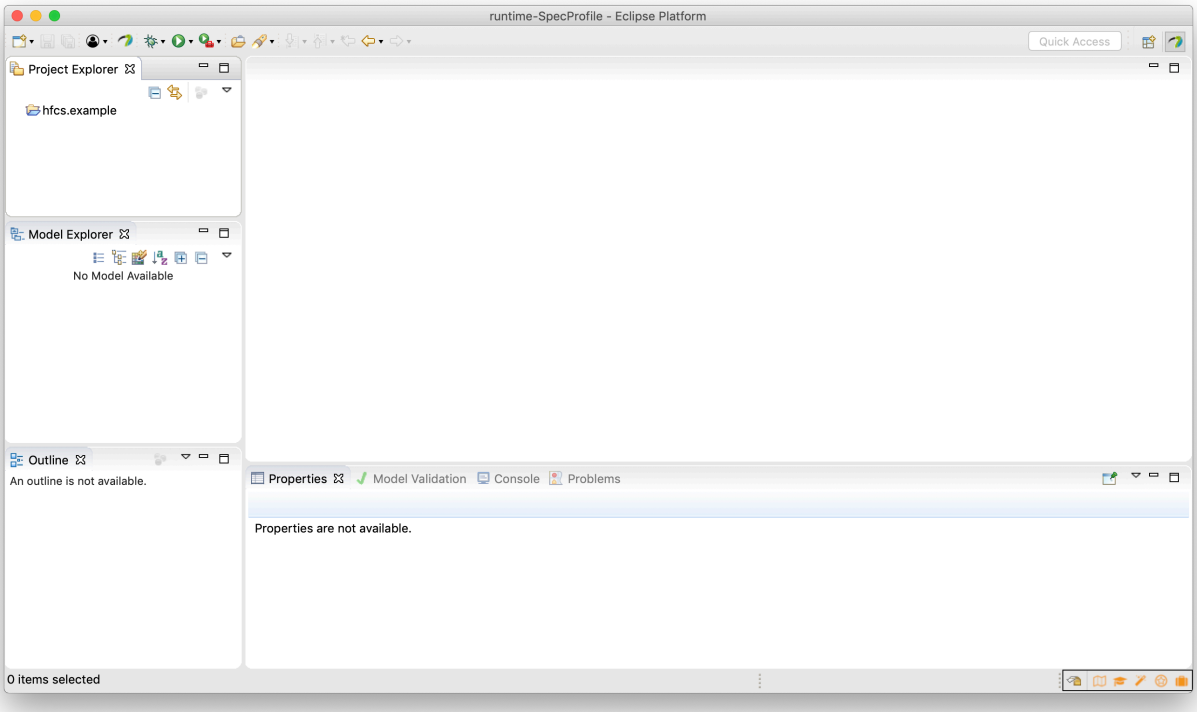
4.3.2 Creating a specification model

The following steps guide the creation of a new SpecML specification model, its verification of well-formedness, and the creation of tabular views of the requirements specification.

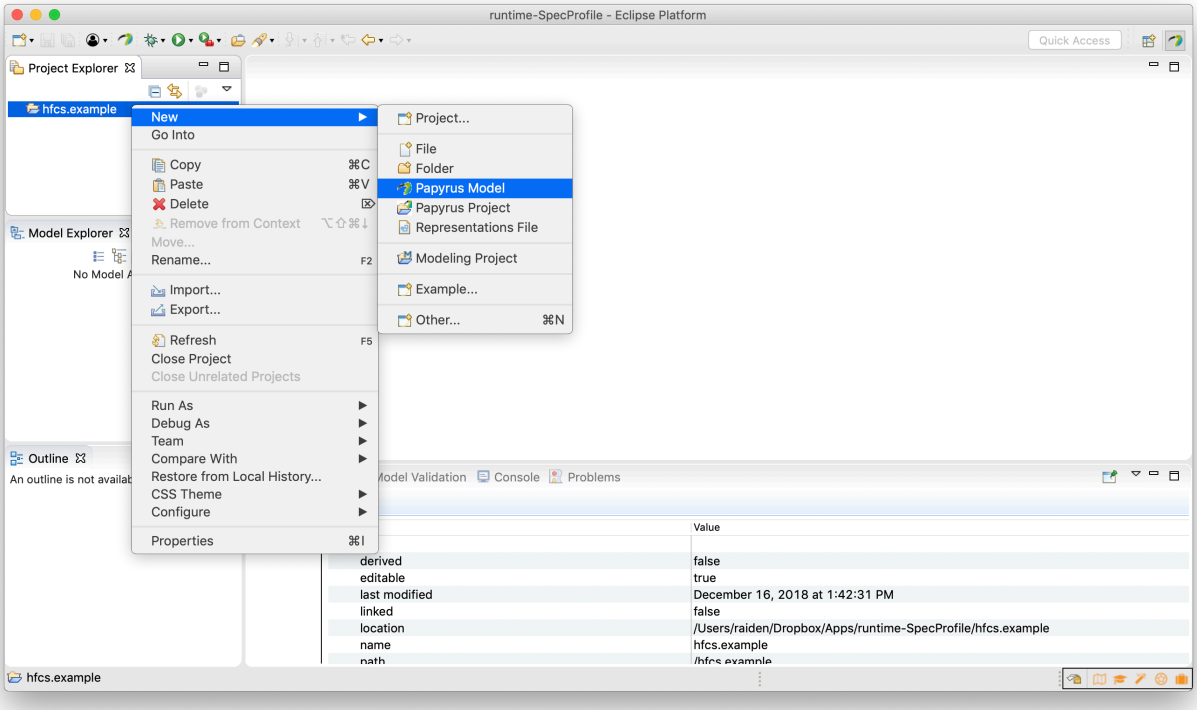
1. Open the Eclipse Papyrus modelling environment.



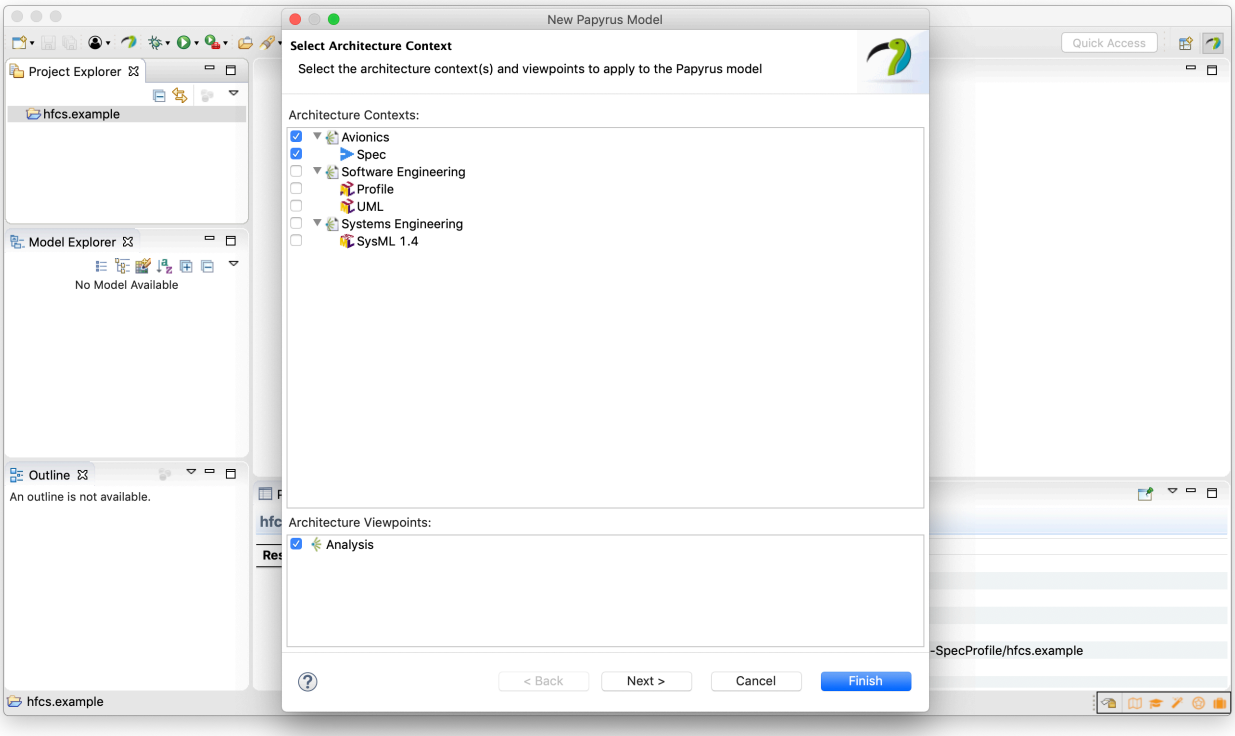
2. Create a project.



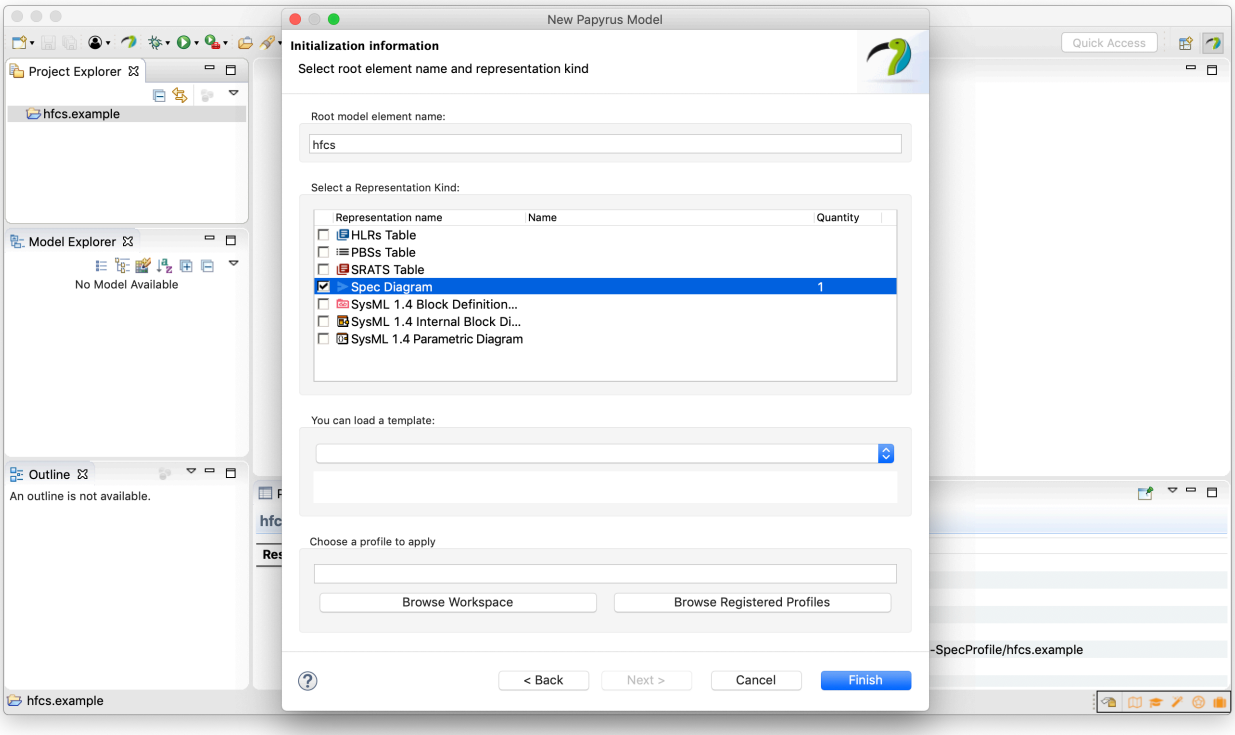
3. Create a new Papyrus model.



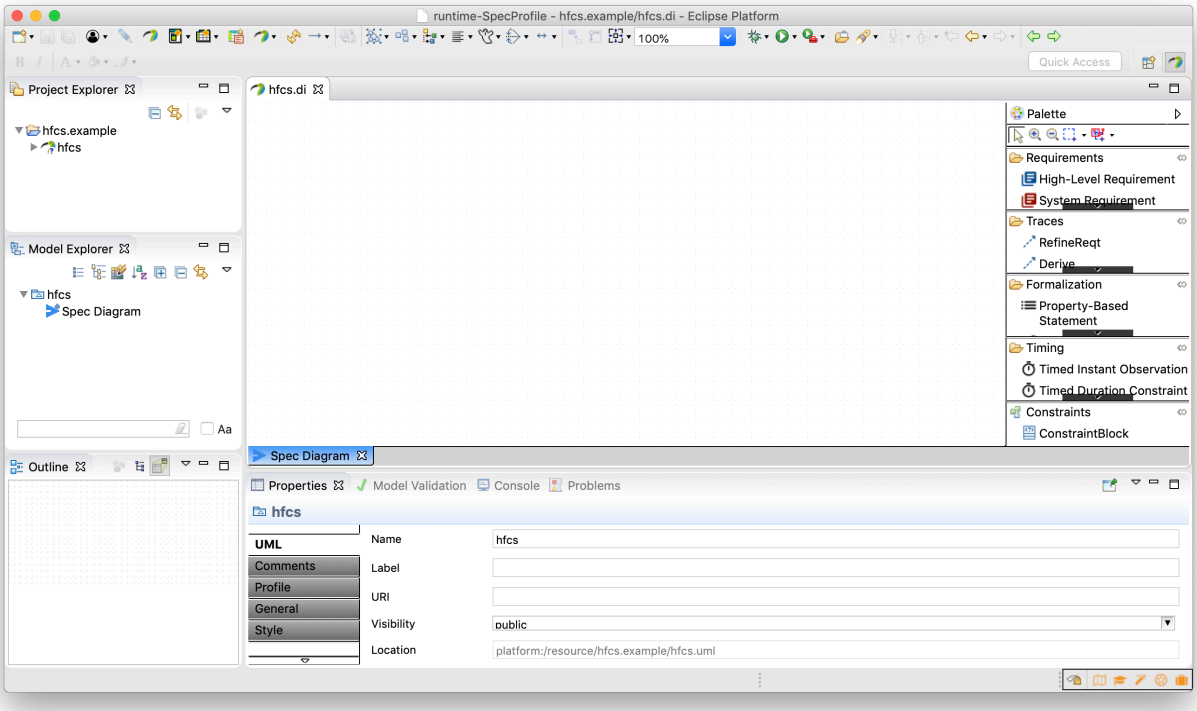
4. Select the SpecML architecture context.



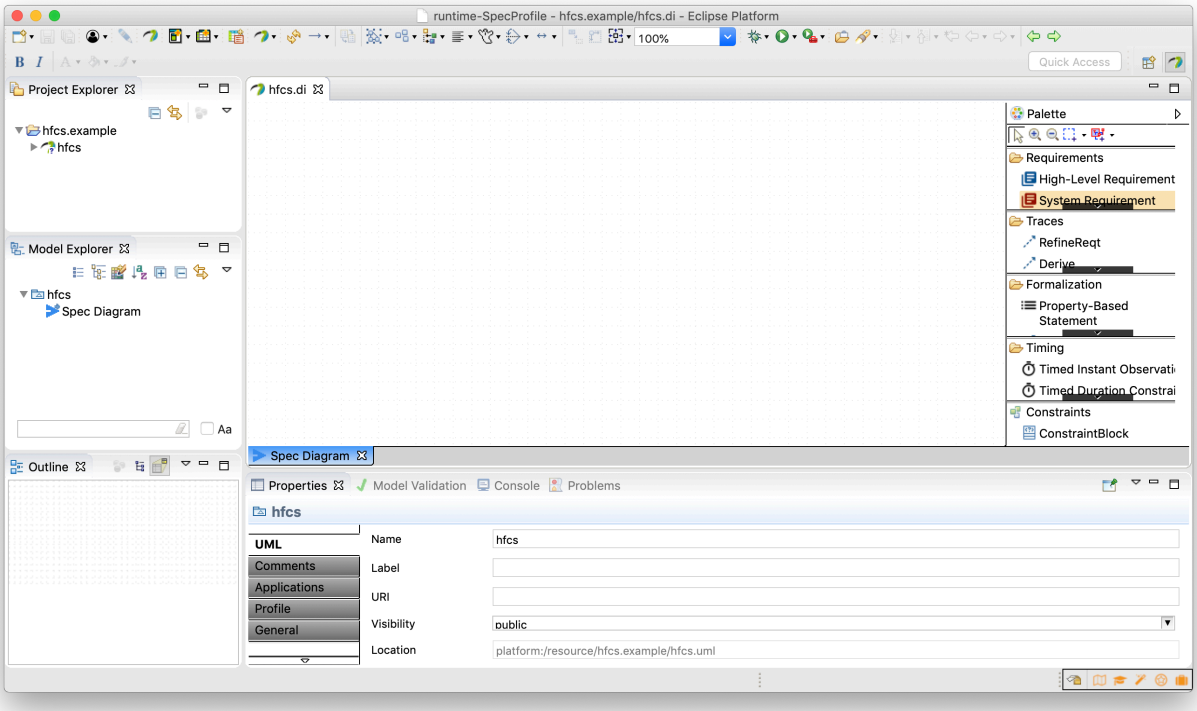
5. Select the SpecML diagram representation kind.



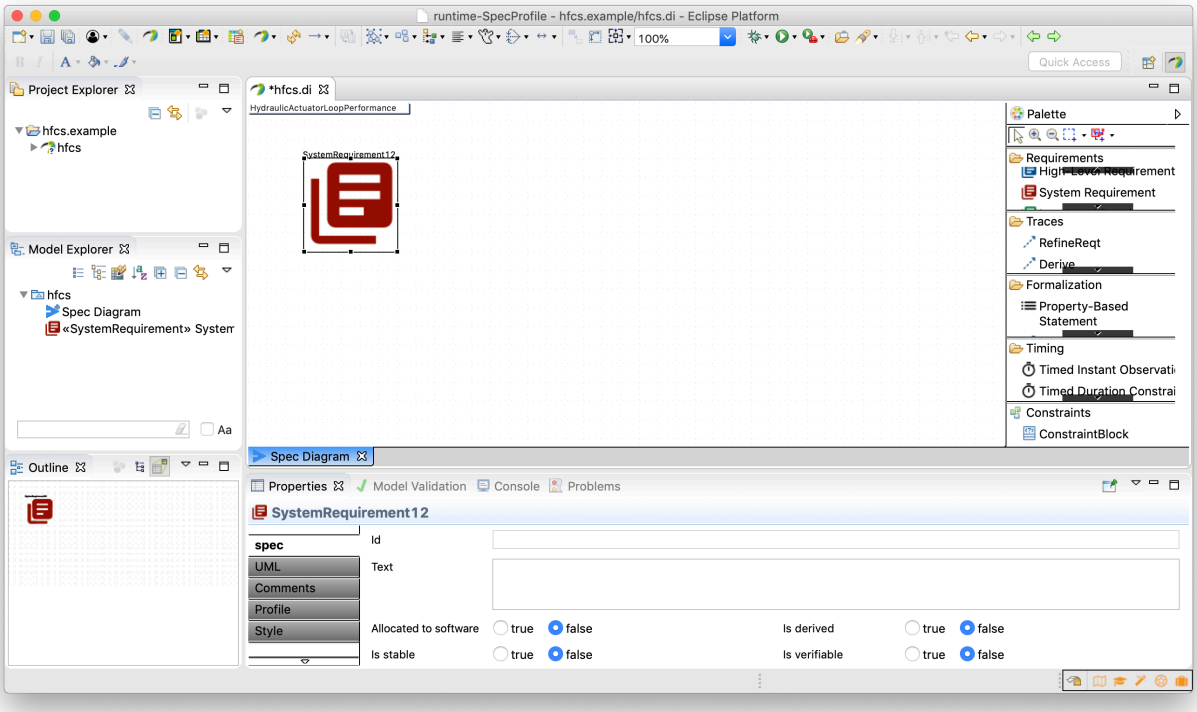
6. Click on **Finish**. The model should open.



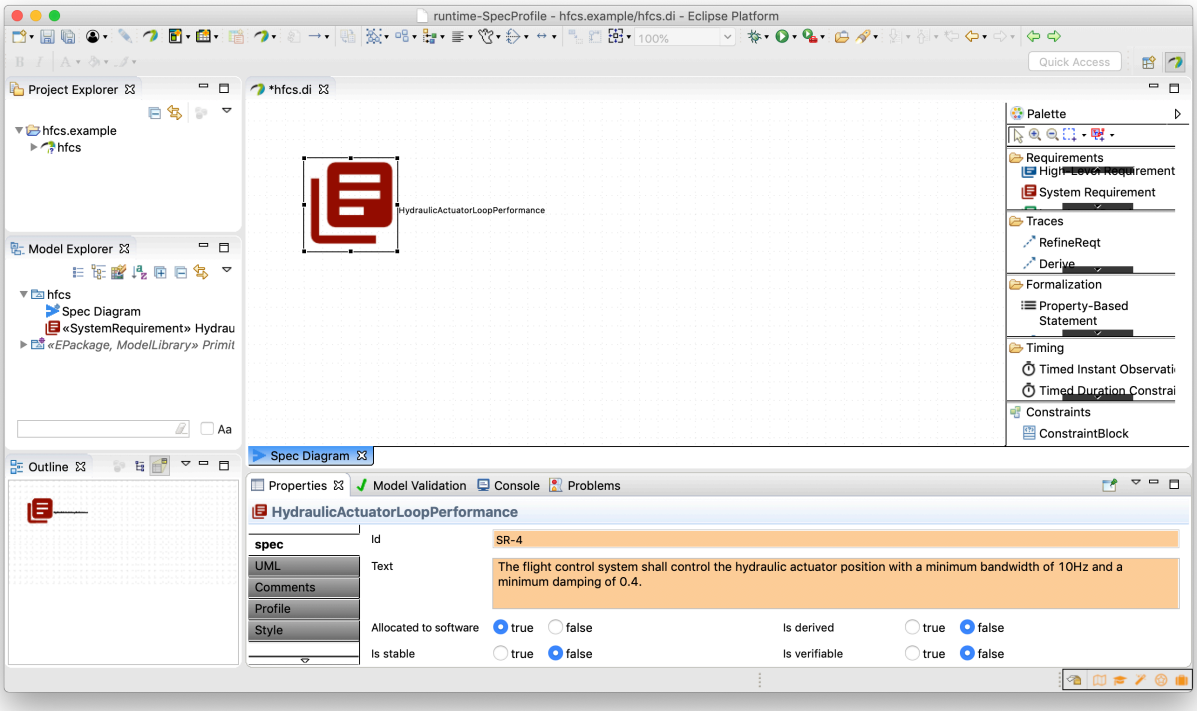
7. Drag and drop elements into the model by using the palette on the right. For instance, select the **System Requirement** element.



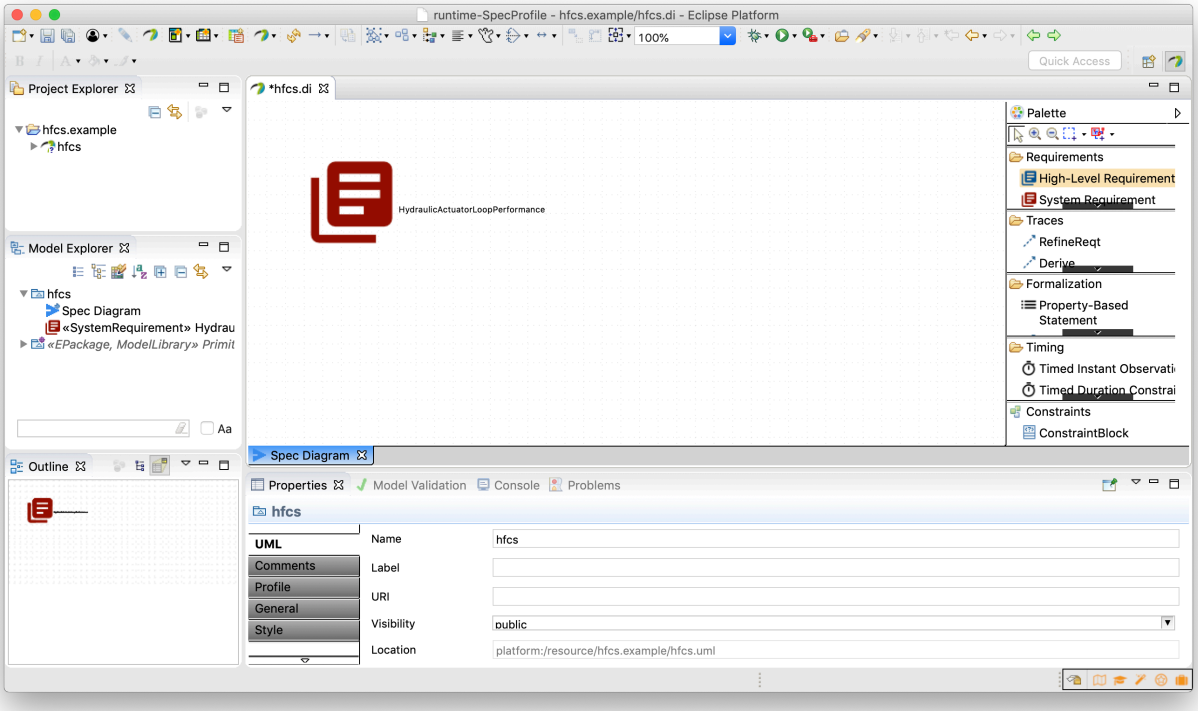
8. Drop the element anywhere in the model and give it a name.



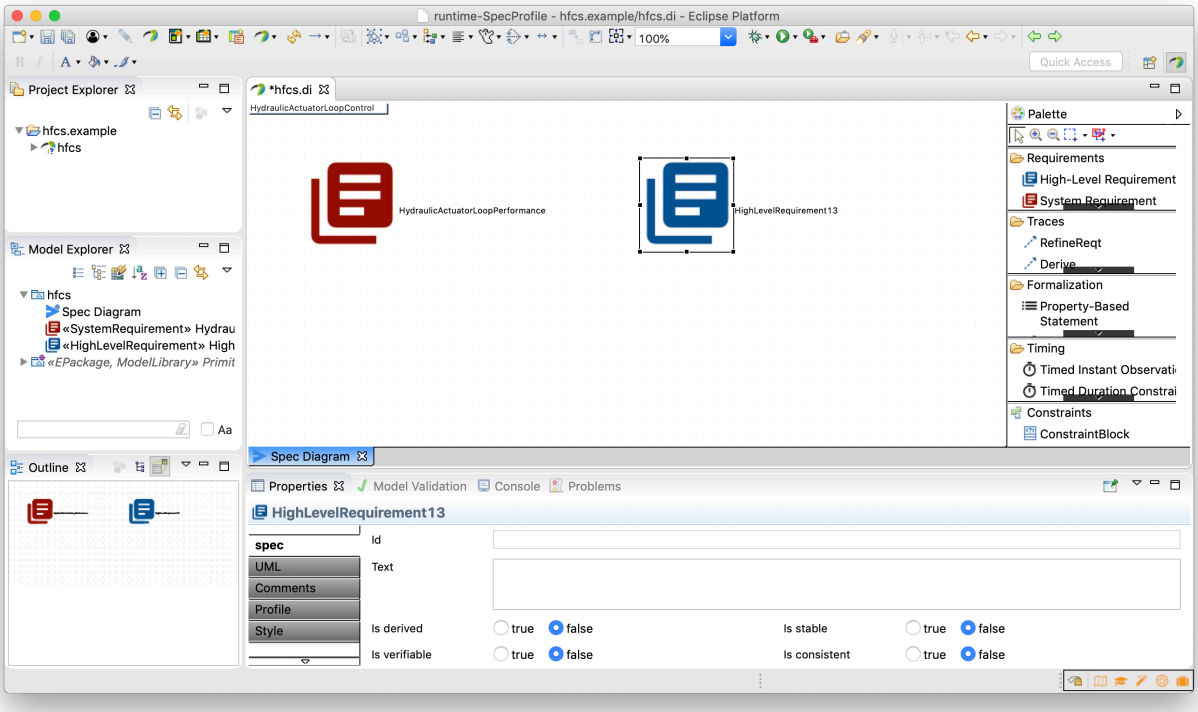
9. Fill out the properties of the element in the **Properties** view (bottom of the screen).



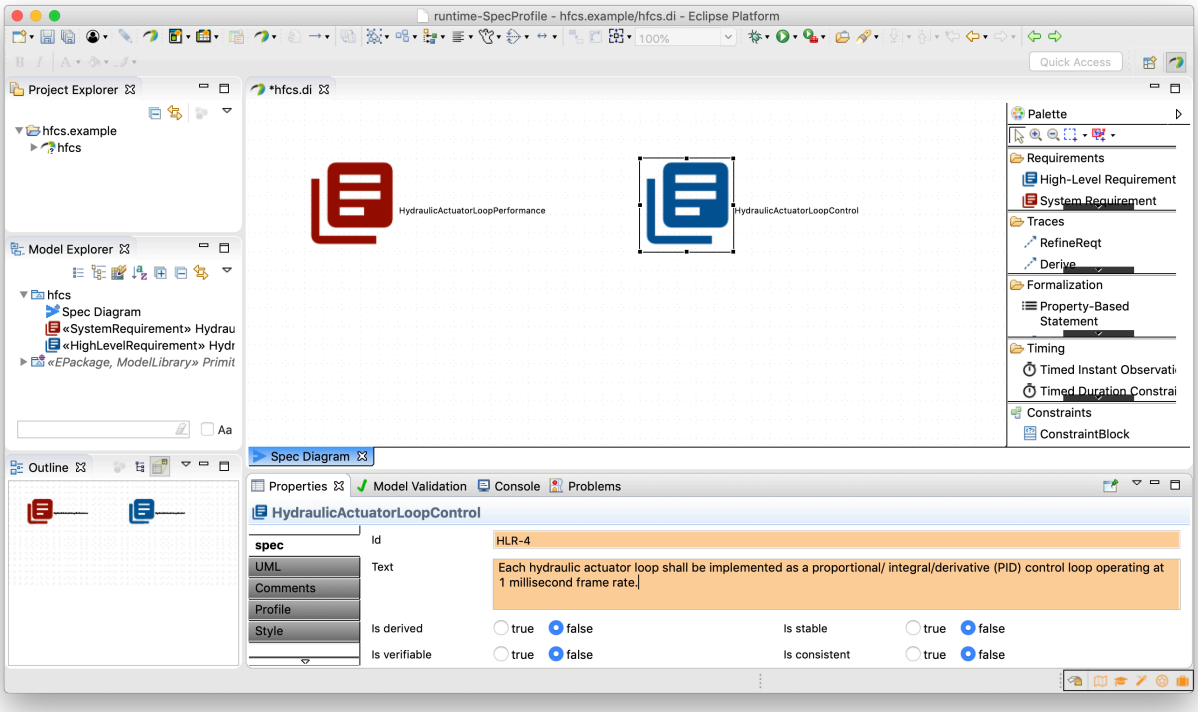
10. Add more elements into the model in the same way. For instance, select the **High-Level Requirement** element.



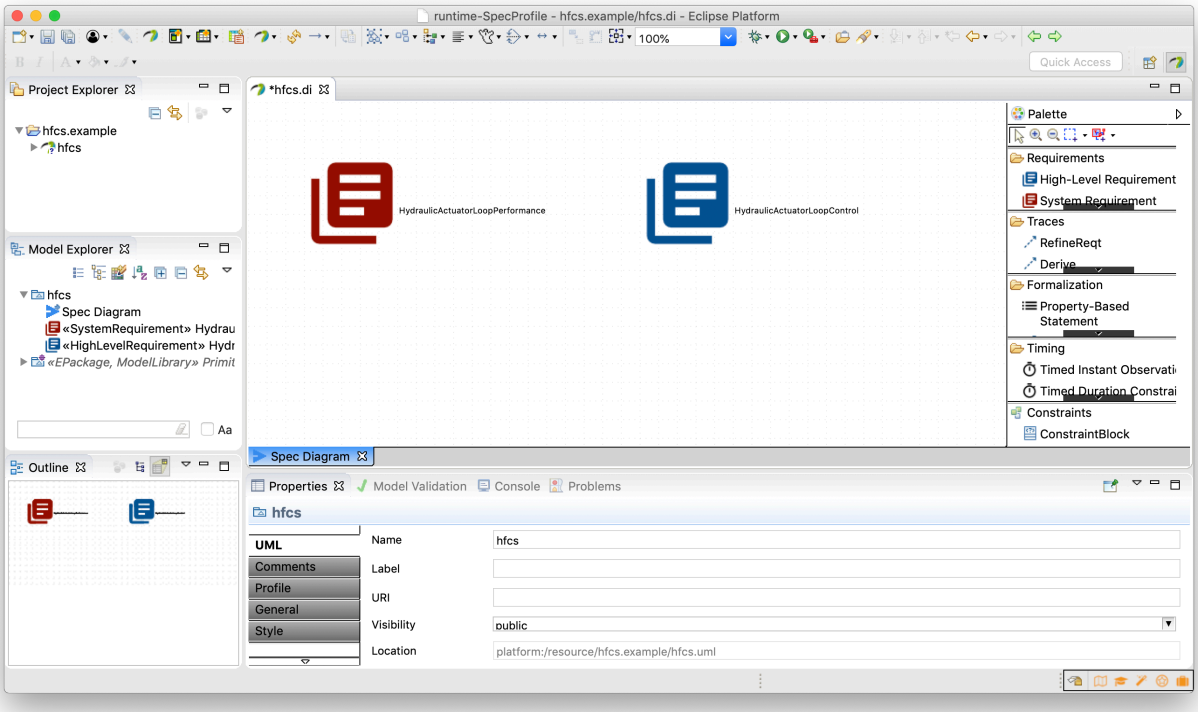
11. Drop the element anywhere in the model and give it a name.



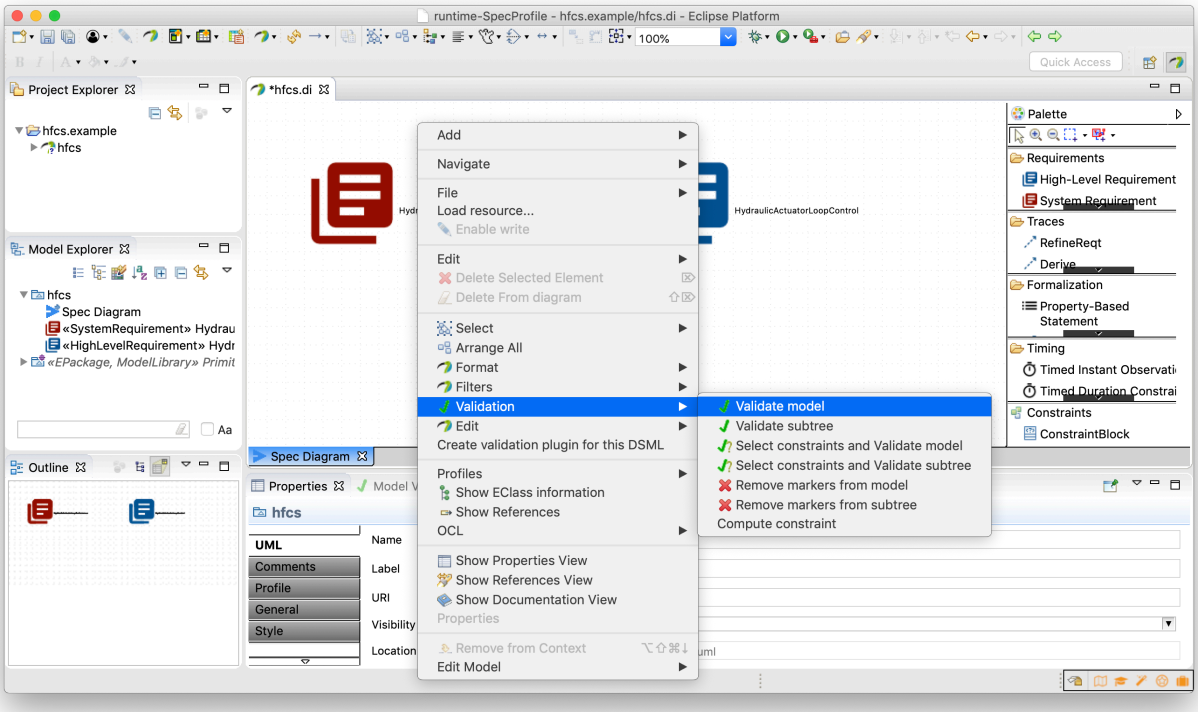
12. Fill out the properties of the element in the **Properties** view.



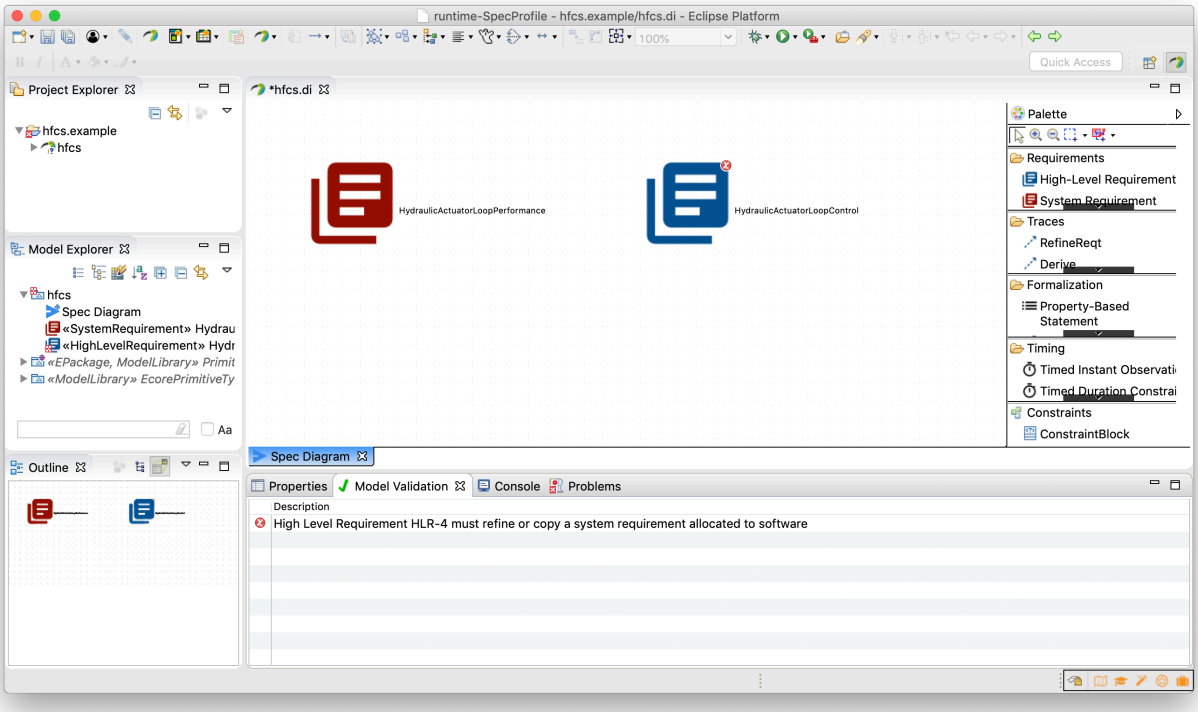
13. Continue adding more elements into the model.



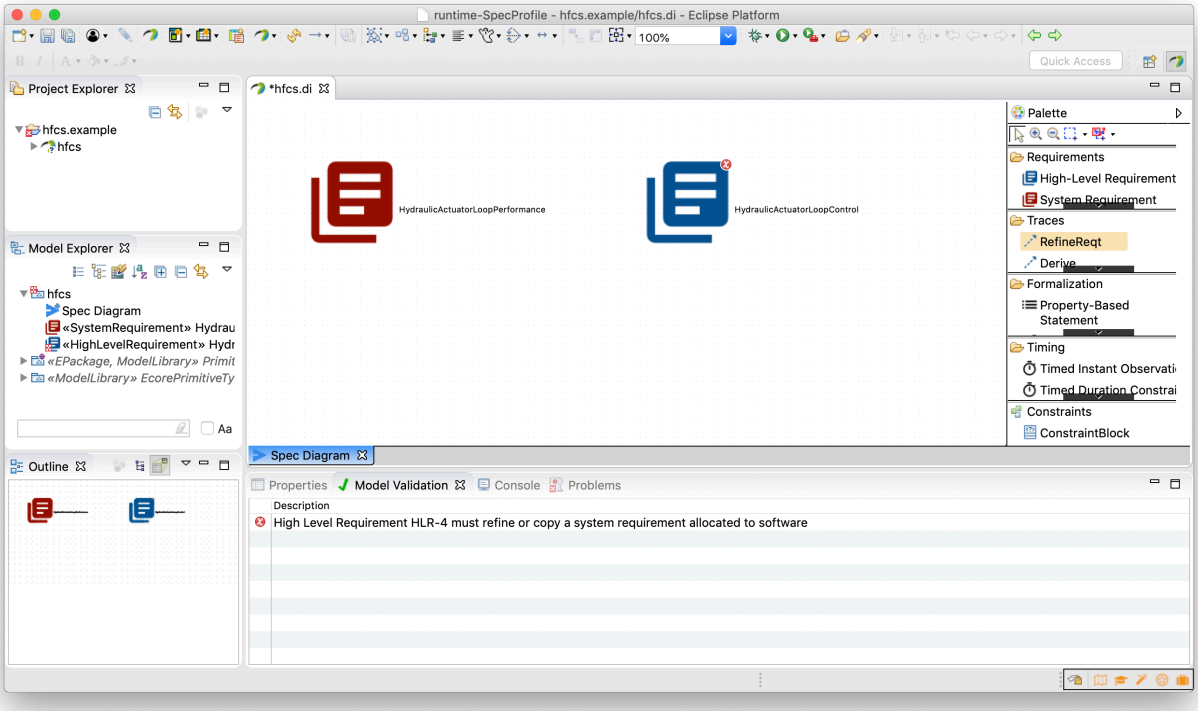
14. A verification of well-formedness of the model can be carried out at anytime. Right click anywhere in the model and select **Validation** » **Validate model**.



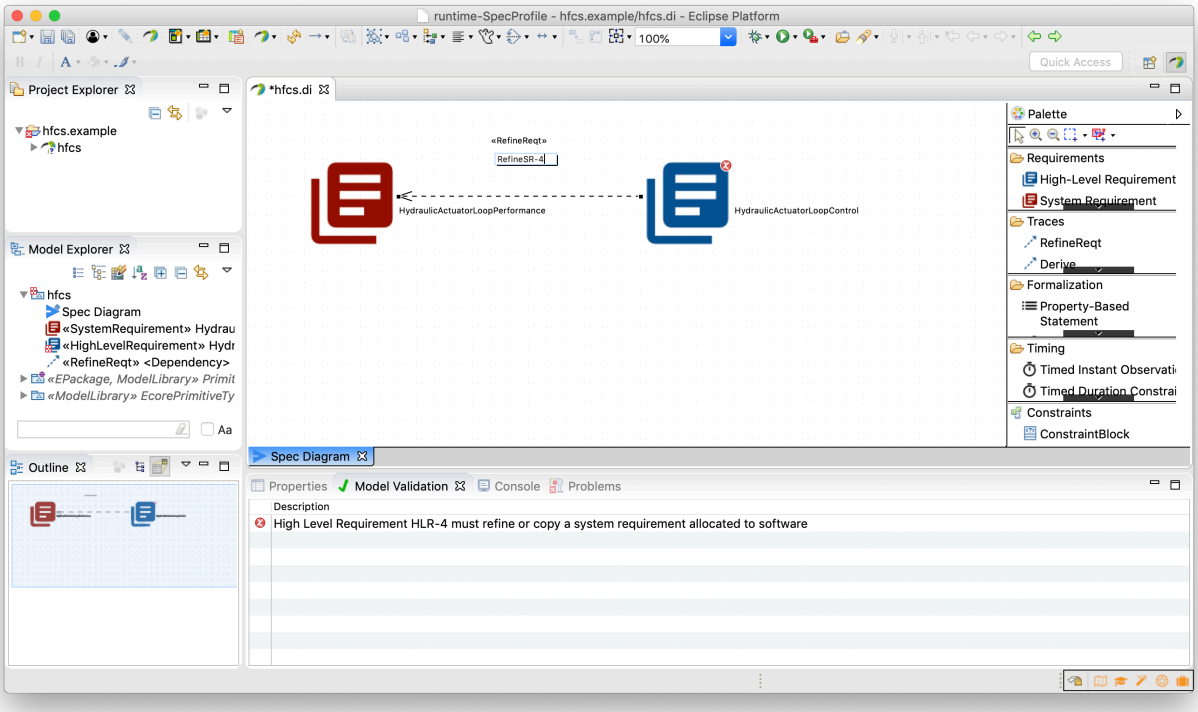
15. Violations will be presented in the **Model Validation** view (bottom of the screen). Any violating element will be marked in the editor view as well. In this case, the HLR (in blue) is not refining or copying an SRATS. To fix a violation, follow the indications given in the violation message.



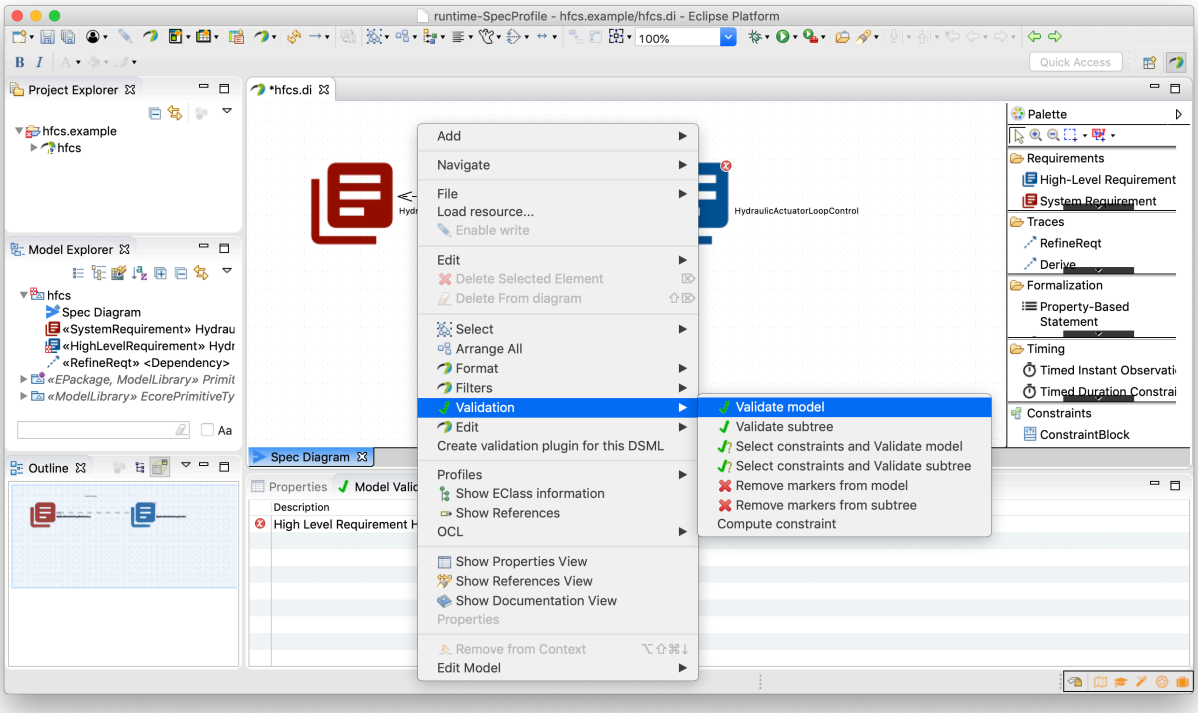
16. In this case, the violation will be fixed by defining a **RefineReqt** trace between the HLR (in blue) and the SRATS (in red).



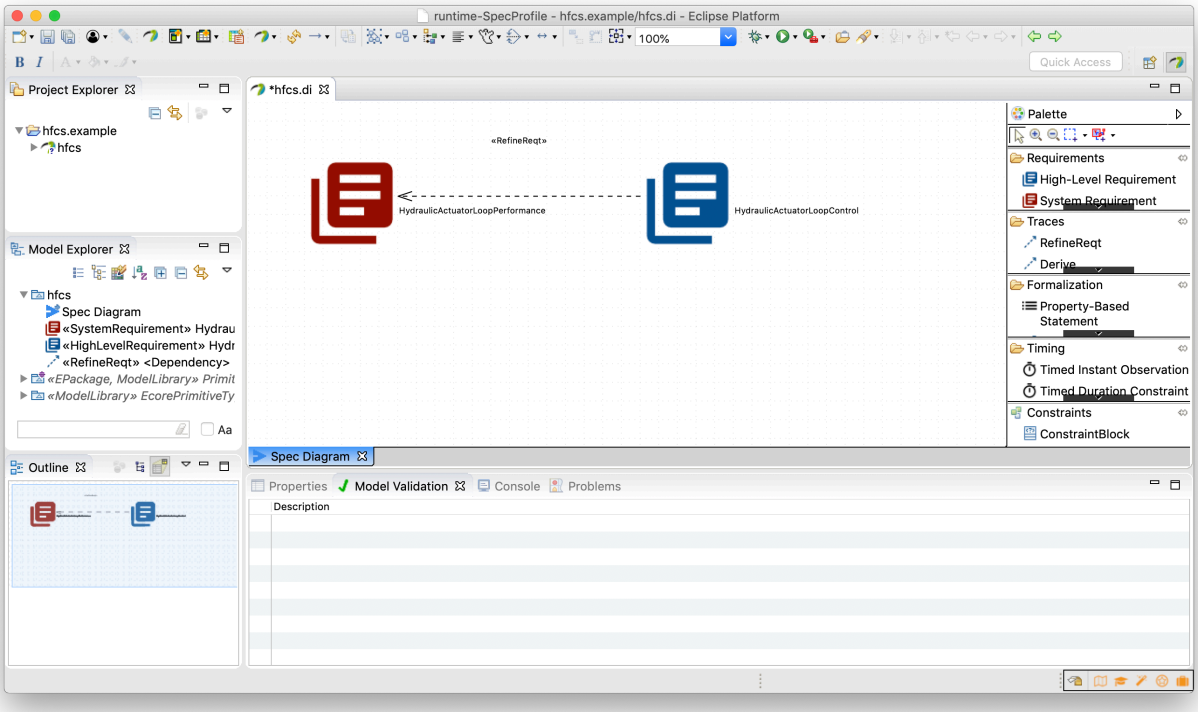
17. Click on the HLR to set the trace's client element. Then, click on the SRATS to set the trace's supplier element. Give the trace a name.



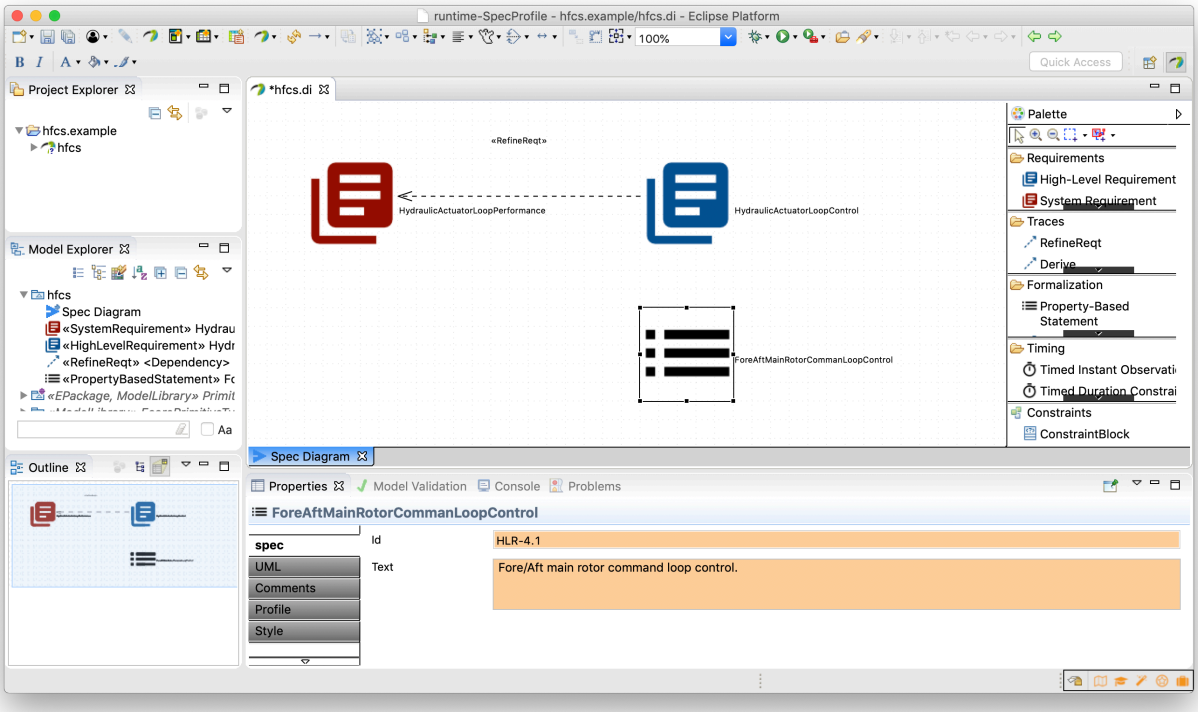
18. The violation was fixed. Re-verify the model by right clicking anywhere in the model and select **Validation** » **Validate model**.



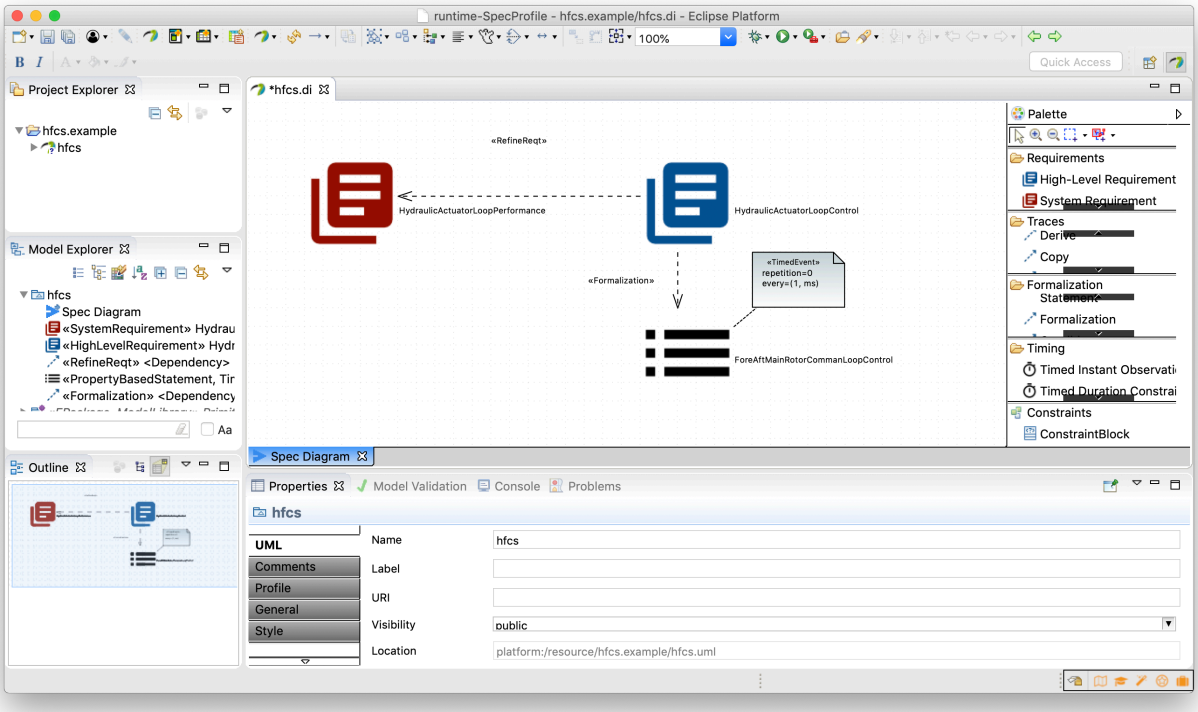
19. The violation should disappear from the **Model Validation** view.



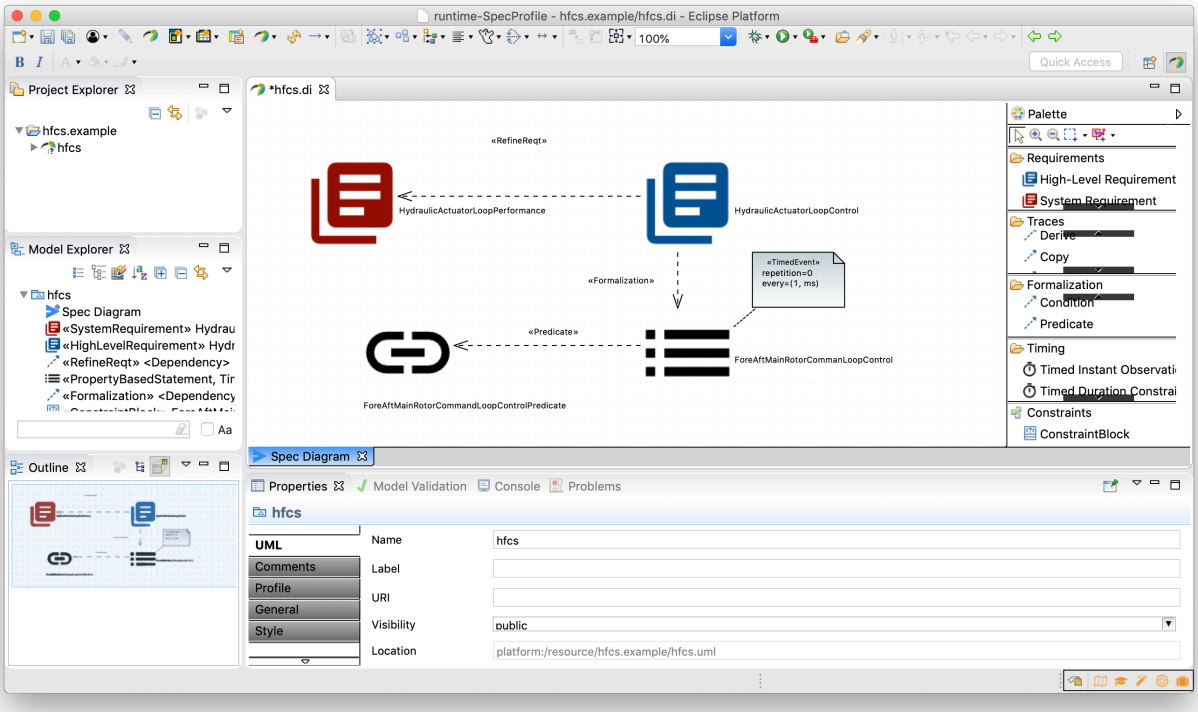
20. Continue adding more elements into the model. For instance, drag and drop a **Property-Based Statement** element and give it a name. Fill out its properties in the **Properties** view.



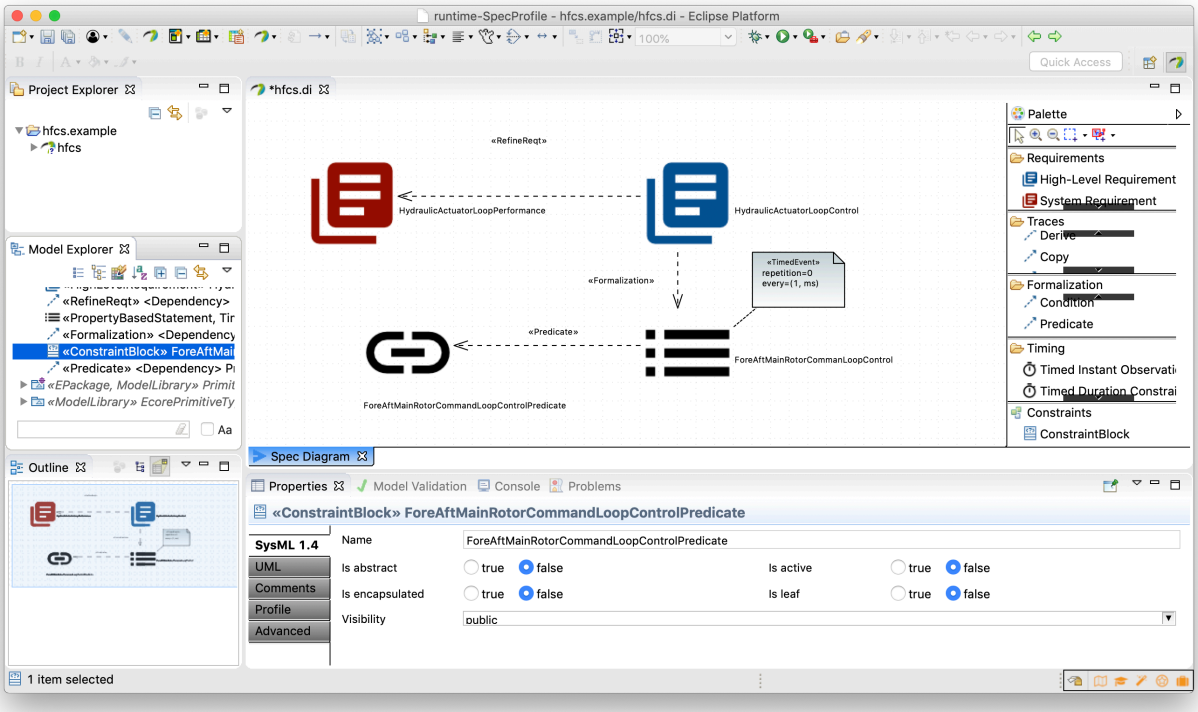
21. Define time-dependent behaviour and constraints by applying MARTE stereotypes onto existing property-based statements or dragging and dropping standalone elements from the palette. In this case, the **TimedEvent** stereotype is applied on the **Property-Based Statement** and displayed as a comment. This is done in the **Profile** and **Appearance** tabs of the **Properties** view (respectively).



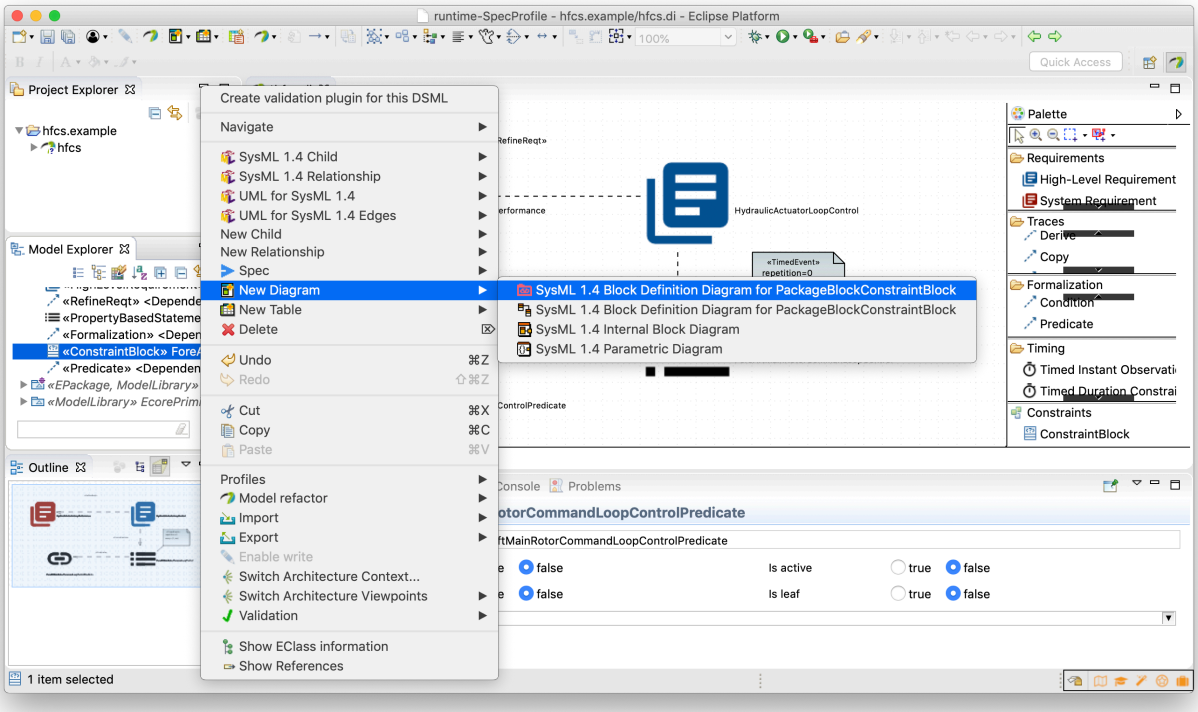
22. Add a **ConstraintBlock** and define as the predicate of the **Property-Based Statement** with the **Predicate** formalization trace.



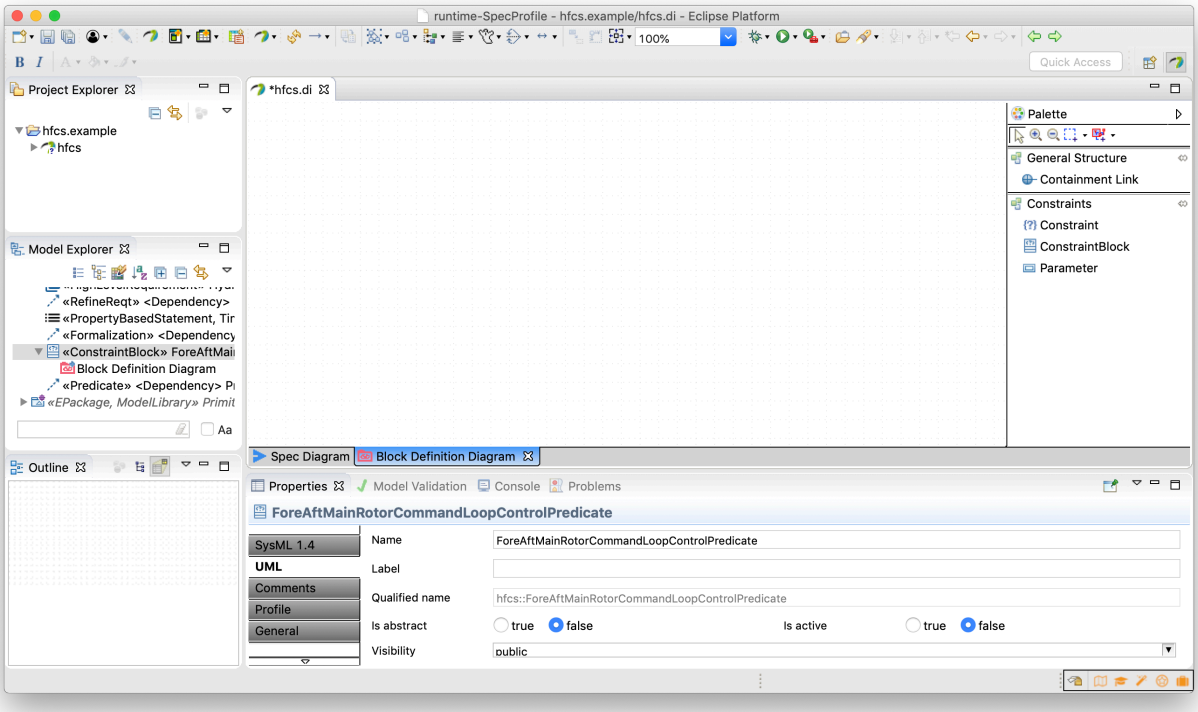
23. In this case, the predicate of this statement has nested SysML constraint blocks. This constraint blocks need to be defined in a SysML block diagram. Select the predicate constraint block in the **Model Explorer** view (left center of the screen).



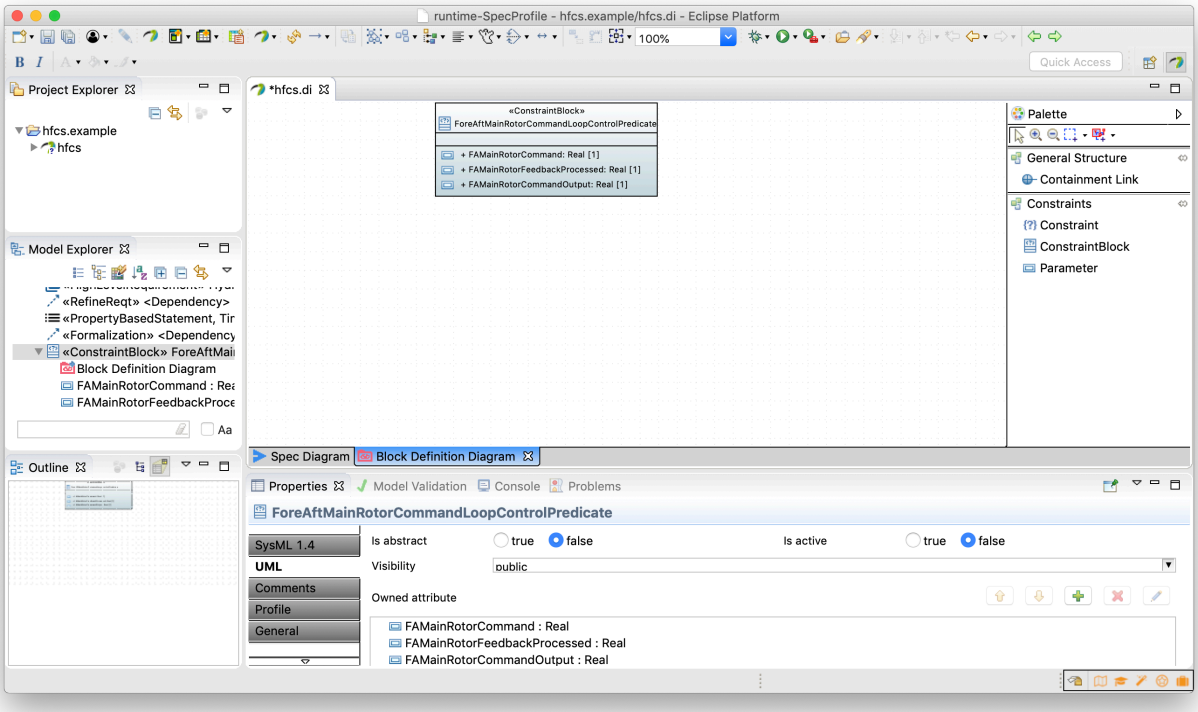
24. Right click and select **New Diagram** » **SysML 1.4 Block Definition Diagram** (first option).



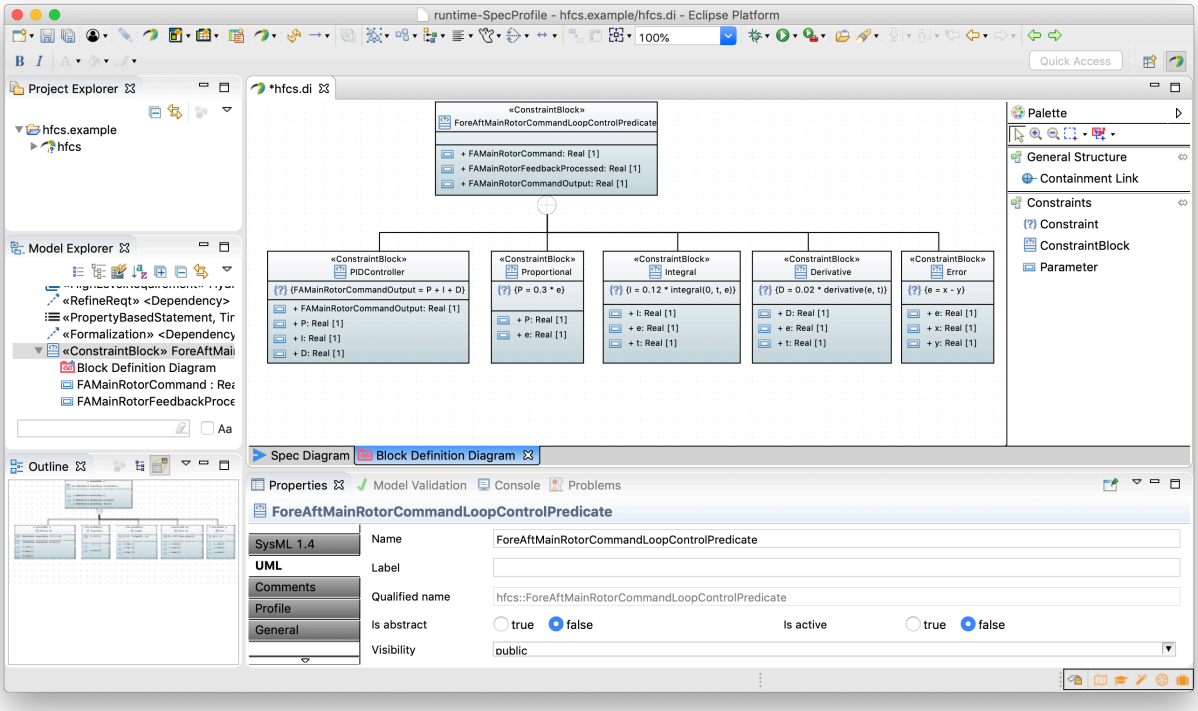
25. A new tab in the model is created.



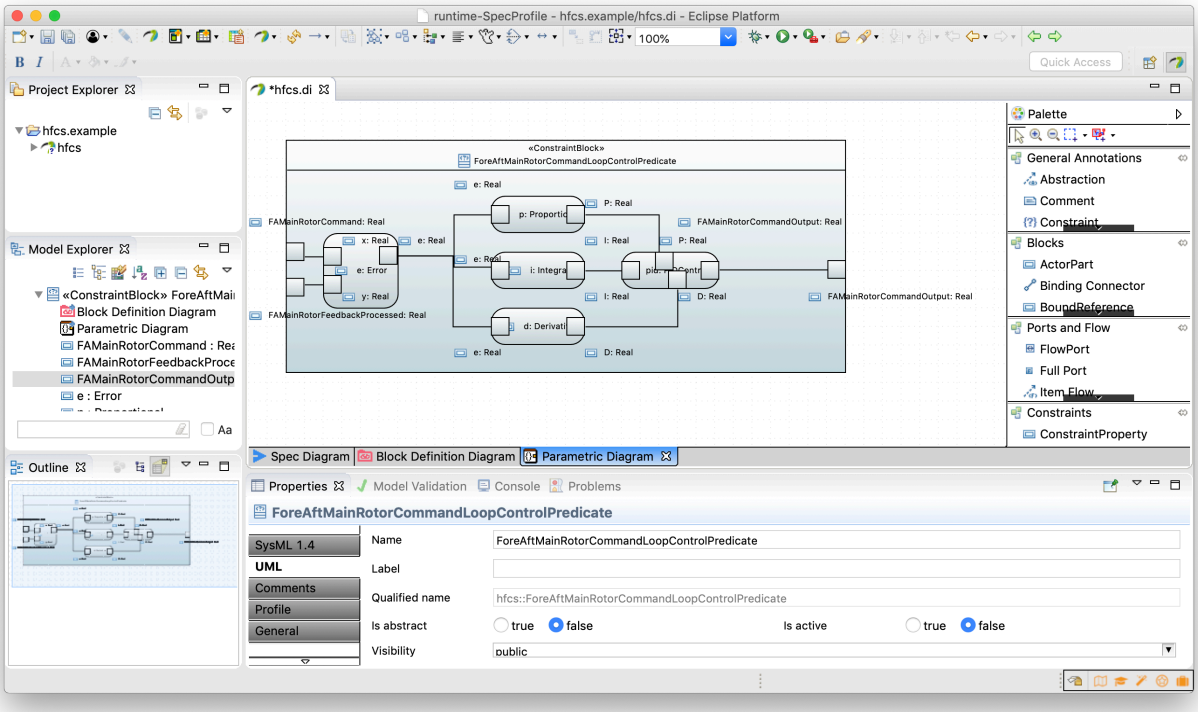
26. Drag and drop the selected constraint block of the predicate from the **Model Explorer** view.



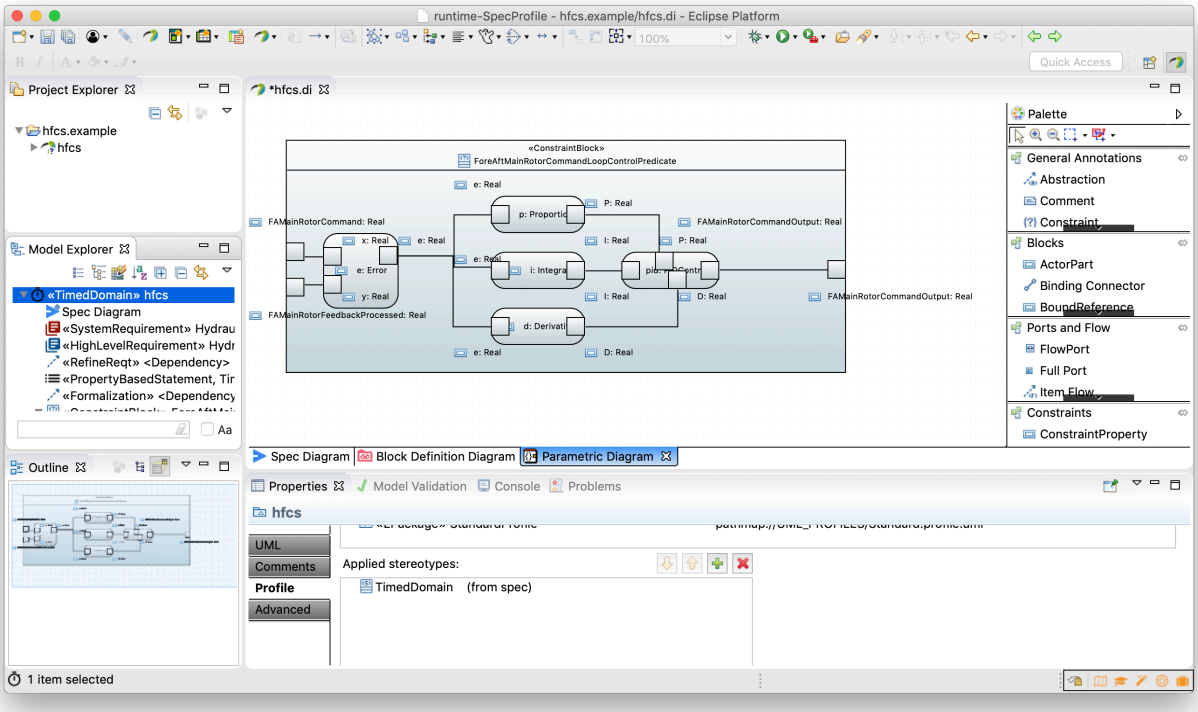
27. Add the nested constraint blocks by dragging and dropping from the palette and connecting them with the **Containment Link**.



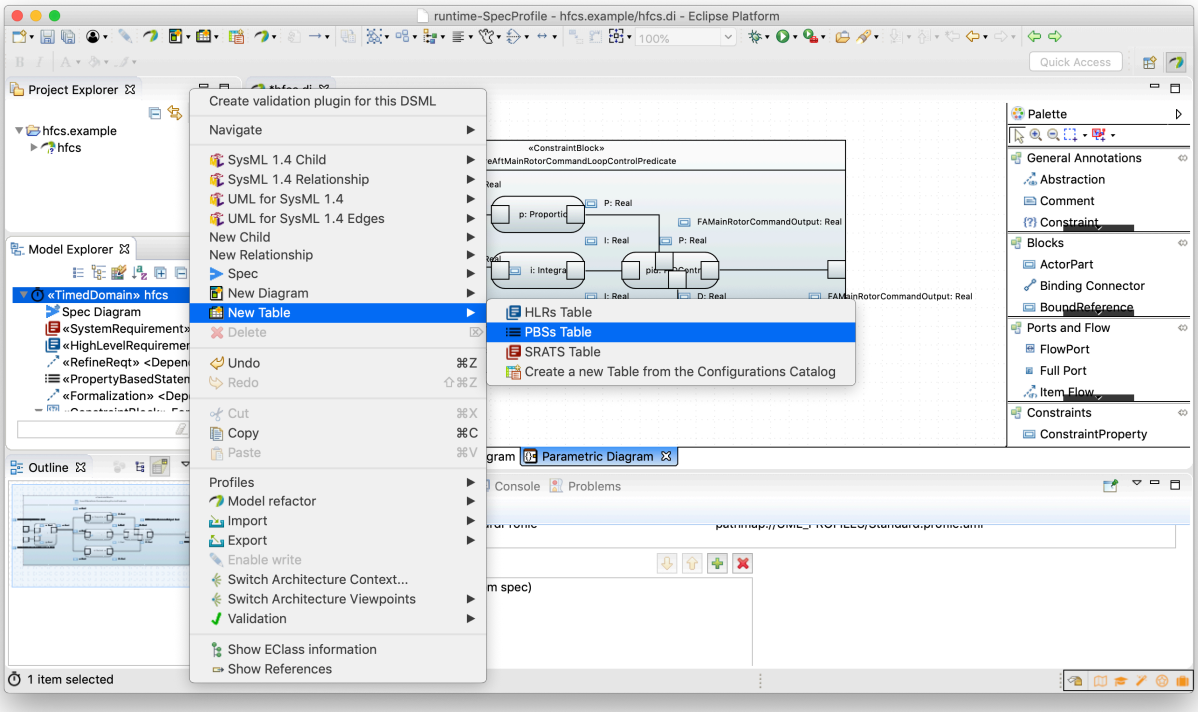
28. In a similar way, a parametric diagram can be added to detail the usage of the nested constraints.



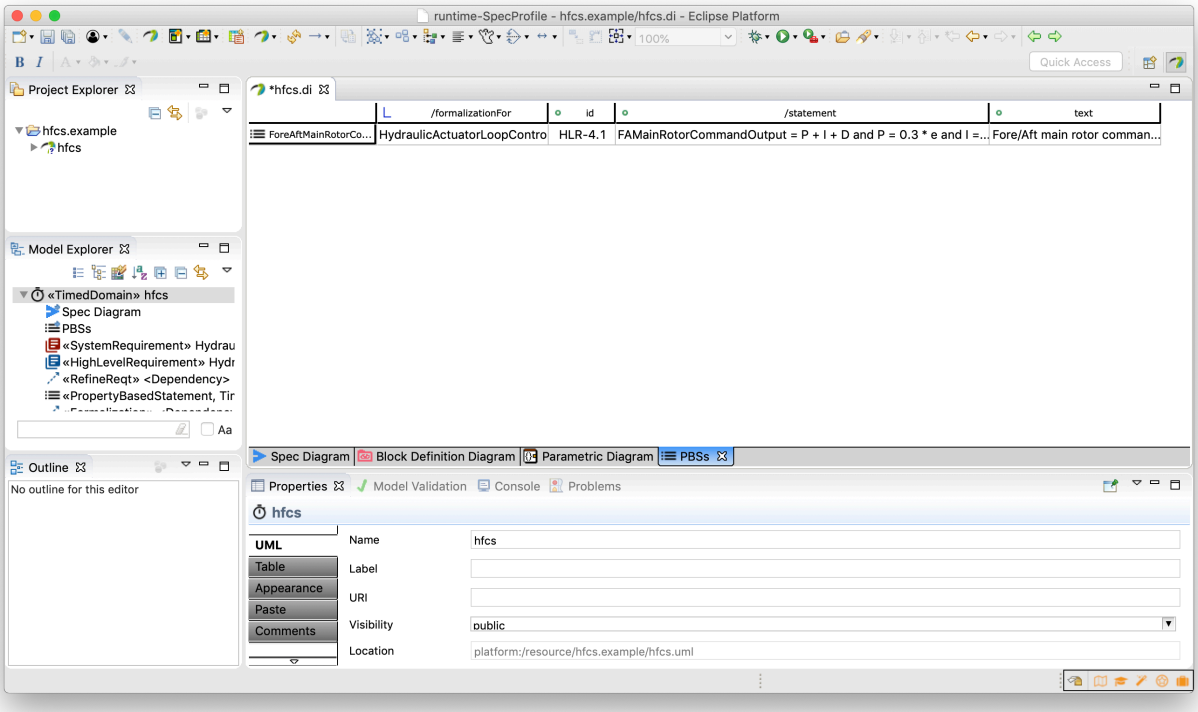
29. If there is time-dependent behaviour and constraints, the specification model must be stereotyped with **TimedDomain**. This is done in the **Profile** tab of the **Properties** view with the SpecML diagram selected in the **Model Explorer** view.



30. The specified requirements and their formalizations can be viewed in tables. Right click and select, for instance, **New Table** » **PBSs Table**.



31. In this case, a table with the requirement formalizations is created.



4.3.3 More information

More information about the Eclipse Papyrus modelling environment and how to use it is available at https://help.eclipse.org/oxygen/nav/79_0.

References

[1] "Software considerations in airborne systems and equipment certification", RTCA, Inc., Std DO-178C, 2011.

[2] "OMG Unified Modeling Language (OMG UML)", OMG, Std, 2017. [Online]. Available: http://www.omg.org/spec/UML.

[3] "Systems Modeling Language (SysML)", Object Management Group (OMG), Std, 2017. [Online]. Available: http://www.omg.org/spec/SysML.

[4] P. Micouin, "Toward a property based requirements theory: System requirements structured as a semilattice", *Systems Engineering*, vol. 11, no. 3, pp. 235–245, Aug. 2008, ISSN: 1098-1241. DOI: 10.1002/sys.v11:3. [Online]. Available: http://dx.doi.org/10.1002/sys.v11:3.

[5] ——, *Model Based Systems Engineering: Fundamentals and Methods*, ser. FOCUS Series. Wiley, 2014, ISBN: 9781118579534.

[6] "UML profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems", OMG, Std, 2011. [Online]. Available: http://www.omg.org/spec/MARTE.

[7] The Eclipse Foundation. (2017). Papyrus Modeling Environment, [Online]. Available: https://www.eclipse.org/papyrus/ (visited on 12/29/2018).

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

[9] 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.

[10] B. Potter. (2016). DO-178 Case Study, [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/56056-do178_case_study (visited on 11/13/2018).