International Council on Systems Engineering
A better world through a systems approach

Case Studies for Querying the Model - SysML V2

Sean Densford

Osvaldas Jankauskas

Hello.

Sean Densford

Industry Process Consultant

Sean Densford is a MBSE with a decade of experience working in the aerospace and defense industry. He holds a Master's Degree in Systems Engineering from Johns Hopkins University and is an OMG-Certified Systems Modeling Language™ (SysML®) Professional (OCSMP) and a Certified Systems Engineering Professional (CSEP). He currently works as an Industry Process Expert for MBSE helping all industries using systems engineering to better utilize engineering tools to develop their systems architectures.

Osvaldas Jankauskas

Senior System Analyst Lead

Osvaldas Jankauskas is Senior System Analyst Lead at Dassault Systems. He holds a BS in Applied Physics from Kaunas University of Technology and is an OMG-Certified Systems Modeling Language™ (SysML®) Professional (OCSMP). He has been working as an Analyst for over 10 years developing tool features and is the original author of Case Studies for Querying the Model for CATIA Magic in SysML V1.

Today's Agenda

Introduce Case Studies for Querying the Model in V1

Overview of SysML V2

V2 Candidate Case Studies

Future Work

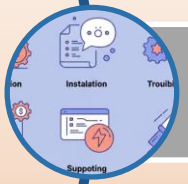
Introduce Case Studies for Querying the Model in V1

- History
- Examples

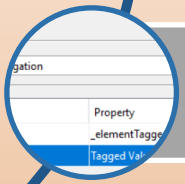
History

Case Studies for Querying the Model

Compiled commonly asked questions



Started with 6 cases in 21x



Expanded to a total of 16 cases in 24x

https://docs.nomagic.com/display/MD2024x/Case+Studies+for+Querying+the+Model

Examples

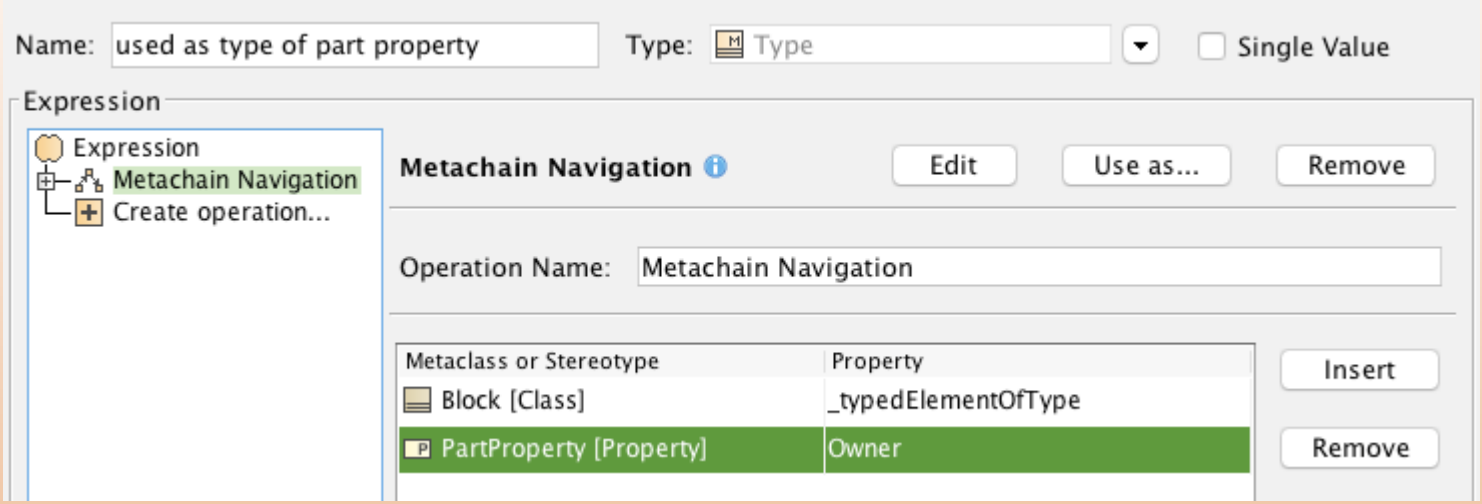
Case Studies for Querying the Model

- Case 1. Coloring Ports by Type using Contains operation
- Case 2. Finding Element Usages as Types
- Case 3. Requirements from the Owning Package of a Smart Package
- Case 4. Properties that Satisfy System Requirements
- Case 5. Validating Elements Not Used in Diagram
- Case 6. Requirements Without Text
- Case 7. Sum of Default Values of Recursively Collected Properties Redefined by Specific Property
- Case 8. Showing Parameter Direction and Type in Single Table Cell using StringConcat
- Case 9. Requirements Derivation and Satisfaction using Table Hierarchy
- Case 10. Filtering Diagrams by Modification Date
- Case 11. Server Project Version in Diagram
- Case 12. Activity Decomposition Table
- Case 13. Coloring Associations between Block and Use Case
- Case 14. Incoming Association of Class
- Case 15. Requirements Coverage by Design Elements
- Case 16. Using the Opposite Reference to find element usages as Tag Values

Examples

Case Studies for Querying the Model

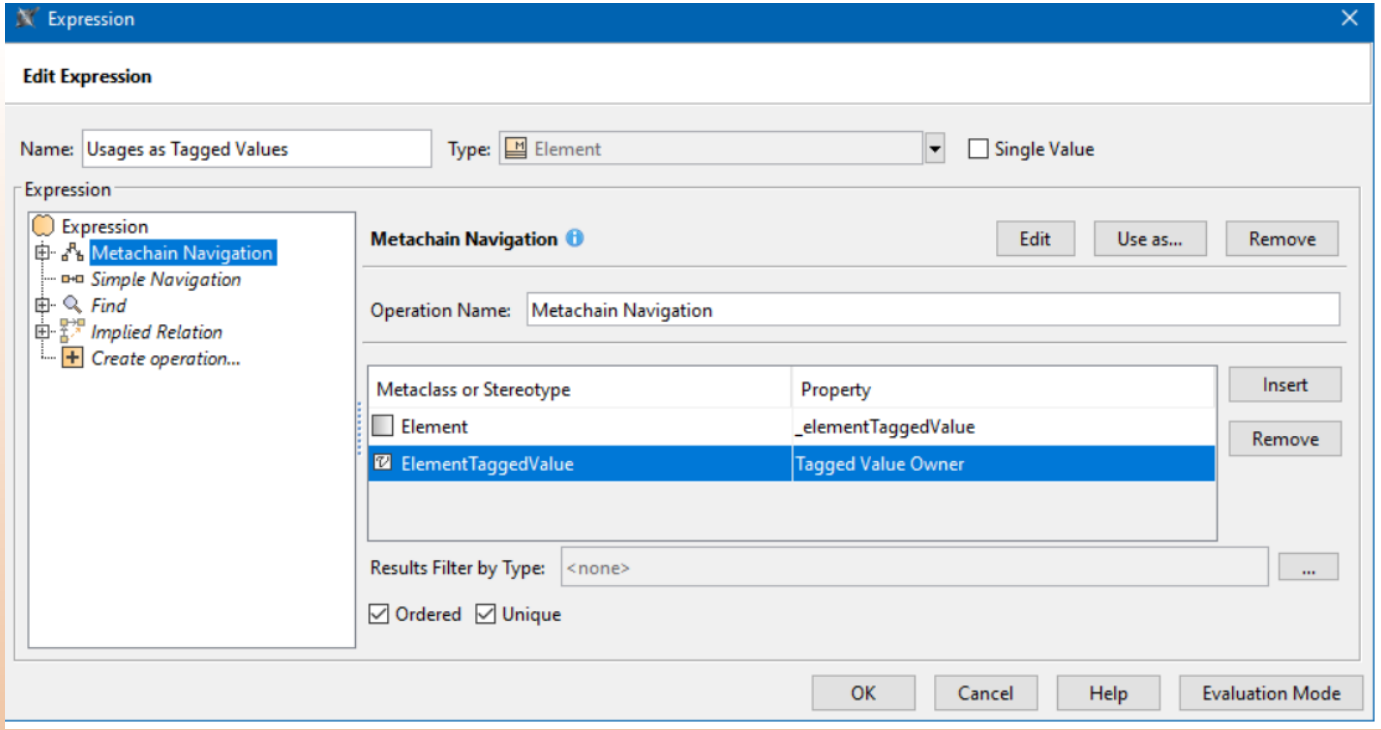
- Case 2. Finding Element Usages as Types



Examples (cont)

Case Studies for Querying the Model

- Case 16. Using the Opposite Reference to find element usages as Tag Values



Query Capabilities of SysML V2

- Textual Notation
- Evaluation

Textual Notation in SysML V2

SysML Libraries:

- **ScalarQuantityValue Operations**
  - Basic Math: Addition, Subtraction, Multiplication, Division, and Exponents
  - Basic Functions: Abs, Max, Min, Sqrt, Floor, Sum, and Product
  - Basic Logic: ==, !=, <, >, and <=
- **VectorQuantityValue Operations**
  - Same as above with more
  - Inner, Outer, Norm, Angle, Transform
- **TensorQuantityValue Operations**

Great Start for Calculations, but Not Enough to Query the Model

Textual Notation in SysML V2

KerML Libraries:

- Function Library
  - Base Functions
    - ==,!=, ===, ToString, all, istype, hastype, meta
  - Math Functions
    - Trig, Natural, Integer, Rational, Real
  - Sequence Functions
    - Equals, Size, Includes, Excludes, Intersection
  - Collection Functions
    - Same as above + Contains
  - Control Functions
    - Collect, Implies, Select, Reduce

Now We Have Everything We Need to Query

Textual Notation in SysML V2

```
verification ProductionQuality {
    private import VerificationCases::*;
    private import NumericalFunctions::*;
    private import SequenceFunctions::*;
    private import RealFunctions::*;
    subject Hinges [*] : HingeDef::Hinge = (1..numberOfRuns)->ControlFunctions::collect {
        generateHinge()
    };
    doc /* Verifies if large scale of production parts can be assembled at required rate */
    attribute numberOfRuns : Integer = Scripts::input("Number of assemblies to analyze");
    attribute negatives : Real [*] = filterNegatives(Hinges);
    attribute outOfSpec : R
    attribute mean : Real =
    attribute variance : Re                                    llect {
        in x : Real;
        (x - mean) ^ 2
    }) / numberOfRuns;
    attribute deviation : Real = sqrt(variance);
    attribute exportCSV = Scripts::exportCSV(Hinges);
    objective obj {
        verify ProductionRequirement {
            subject;
            in :>> actualPercentage = outOfSpec;
        }
    }
    return verdict : VerdictKind = PassIf(obj());
}
```

Import Capability

Built in Functions

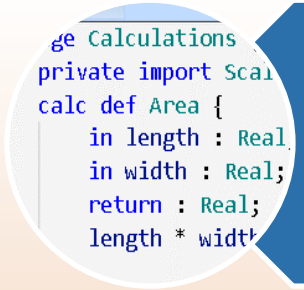
Built in Math Expression Capability

Custom Function Calls

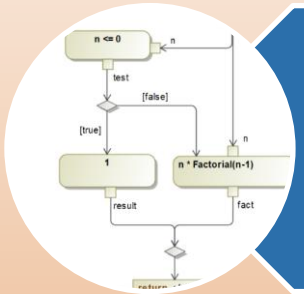
V2 Candidate Case Studies

- Getting Started
- Number of Requirements
- % of Satisfied Requirements
- % of Performed Actions

Metric Methods



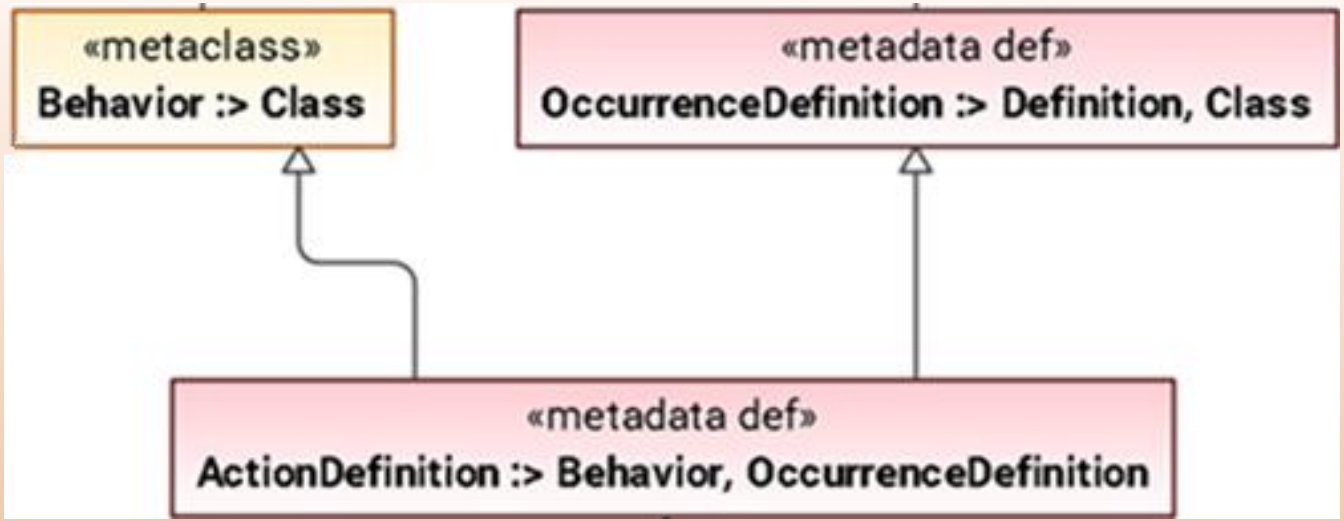
Calc Definitions



Opaque Actions

Opaque Action

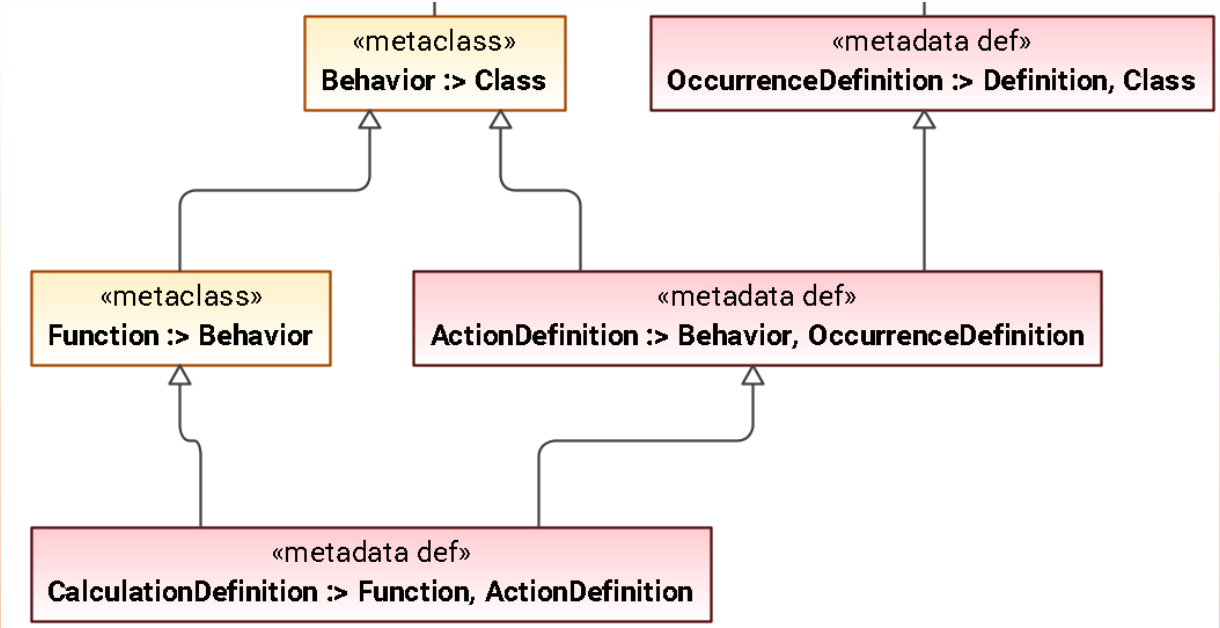
- Action Def/Usage using another language than sysml



More Suitable to Represent Behaviors

Calculation Definition - Functions

- Calculation Definitions are Functions and Actions
- Function – a behavior that returns one result parameter



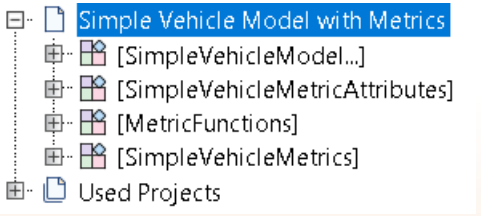
Could Represent Math or Behavior

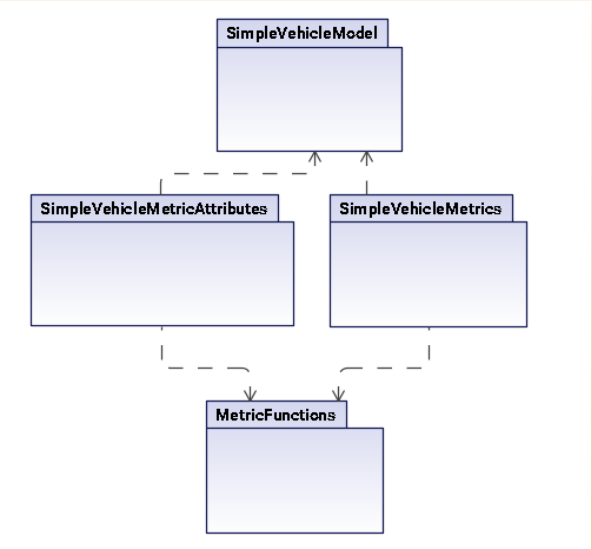
Model-Level Evaluable Expressions

- **Model-Level Evaluable Expressions** are expressions that refer to metadata
- **Metadata** is data about model elements; it is not about what is being modeled

Metric Infrastructure

- Separate Namespaces
  - Model: SimpleVehicleModel
  - Metric Attributes: SimpleVehicleMetricAttributes
  - Metrics Applied to Simple Vehicle Model: SimpleVehicleMetrics
  - Metric Functions Defined: MetricFunctions





Simple Vehicle Model

- From the Specification Annex A: Example Model
- Provides illustration of how SysML can be used to model a sample system with both Textual and Graphical notation
- Provided by OMG

Metric Attributes: SimpleVehicleMetricAttributes

```
package SimpleVehicleMetricAttributes {

    part SimpleVehicleModelMetrics {

        attribute countOfRequirements : ScalarValues::Integer =
MetricFunctions::Satisfy::RequirementsCount(SimpleVehicleModel.metadata);

        attribute countOfSatisfyRequirements : ScalarValues::Integer =
MetricFunctions::Satisfy::OwnedSatisfyRequirementsCount(SimpleVehicleModel.metadata);

        attribute allOwnedRequirementsRecursive : ScalarValues::Integer =
MetricFunctions::Satisfy::OwnedRequirementsAll(SimpleVehicleModel.metadata);

        attribute allOwnedSatisfyRequirementsRecursive : ScalarValues::Integer =
MetricFunctions::Satisfy::OwnedSatisfyRequirementsAll(SimpleVehicleModel.metadata);

        attribute satisfiedRequirements : ScalarValues::Integer =
MetricFunctions::Satisfy::notSatisfiedRequirements(SimpleVehicleModel.metadata);

        attribute satisfiedRequirementsPercent : ScalarValues::Integer =
MetricFunctions::Satisfy::satisfiedRequirementsPercentage(SimpleVehicleModel.metadata);

    }

}
```

Metrics Applied to Simple Vehicle Model: SimpleVehicleMetrics

- Create a custom table view: Metric Table
- Expose the System of Interest
- Define the columns and rows

Metric Functions Defined: MetricFunctions

Imports:

- `private import SequenceFunctions::*;`
- `private import ScalarValues::*;`
- `private import SequenceFunctions::*;`
- `private import ScalarValues::*;`
- `private import ControlFunctions::*;`
- `private import NumericalFunctions::sum;`

MetricFunctions: Calc Defs

Satisfy Functions:

- RequirementsCount
- OwnedRequirementsAll
- OwnedSatisfyRequirementsAll
- OwnedSatisfyRequirementsCount
- satisfiedRequirementsPercentage

```
calc def RequirementsCount {
    in scope : KerML::Kernel::Package [1];
    size(OwnedRequirementsAll(scope))
}
calc def OwnedRequirementsAll {
    in scope : KerML::Kernel::Package [1];
    ownedMembers = scope.ownedMember.? {
        in x;
        (x as SysML::Element).owningRelationship
        hastype OwningMembership |
        (x as SysML::Element).owningRelationship
        hastype FeatureMembership
    };
    requirements = ownedMembers.? {
        in x;
        x hastype SysML::RequirementUsage
    };
    nested = ownedMembers.? {
        in x;
        x @SysML::Namespace & not (@SysML::VerificationCaseUsage)
    };
    numIndirect = nested->collect OwnedRequirementsAll;
    union(numIndirect, requirements)
}
```

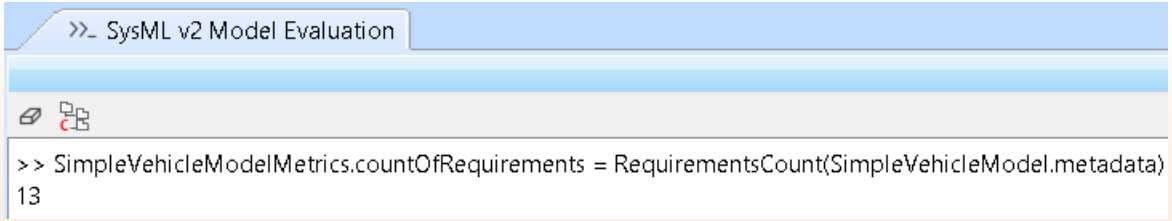
MetricFunctions: Calc Defs

Perform Functions:

- ActionsCount
- ActionsAll
- OwnedPerformedActionsRequirementsAll
- PerformActionsCount
- onlyActionsBeingPerformed
- performedActionsPercentage

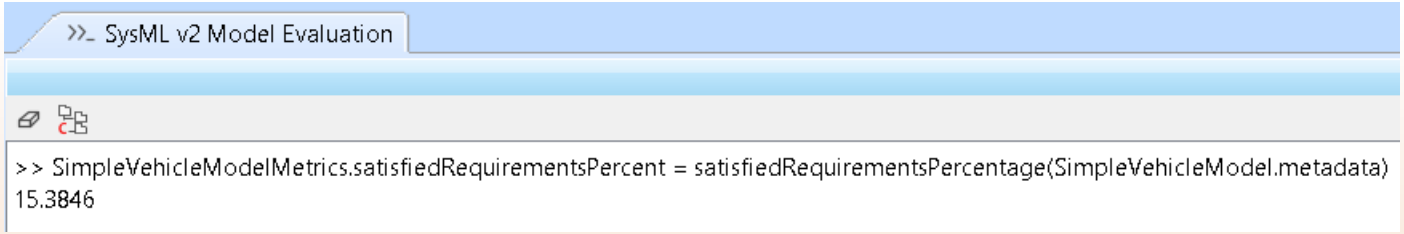
```
calc def ActionsCount {
    in scope : KerML::Kernel::Package [1];
    size(Perform::ActionsAll(scope))
}
calc def ActionsAll {
    in scope : KerML::Kernel::Package [1];
    ownedMembers = scope.ownedMember;
    directActions = ownedMembers.? {
        in x;
        x hastype SysML::Systems::ActionUsage &
        x hastype SysML::VerificationCaseUsage == false
    };
    nestedNamespaces = ownedMembers.? {
        in x;
        x istype SysML::Namespace & x hastype
        SysML::VerificationCaseUsage == false
    };
    indirectActions =
    nestedNamespaces->ControlFunctions::collect ActionsAll;
    SequenceFunctions::union(indirectActions, directActions)
}
```

Number of Requirements



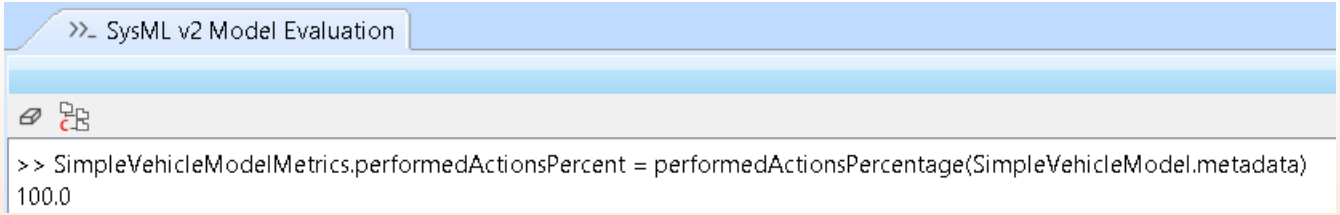
Confirmed with Manual Search of Textual Notation

% of Satisfied Requirements



- Find total requirements
- Find not satisfied requirements
- Find not satisfied requirements percentage
- Take 100%-not satisfied requirements percentage

% of Performed Actions



- Find total actions
- Find performed actions
- Find performed actions %

Confirmed with Manually by Checking Each ActionUsage

Future Work

- Recreating Other Queries Based on Customer Feedback
- Extending to Other Metrics
- Building Validation Suites

Backup

Operator Mapping

Table 5. Operator Mapping

Operator	Library Function	Description	Model-Level Evaluable?
`all`	`BaseFunctions::'all'`	Type extent	No
`istype`	`BaseFunctions::'istype'`	All argument values are directly or indirectly instances of a type	Yes
`hastype`	`BaseFunctions::'hastype'`	All argument values are directly instances of a type	Yes
`@`	`BaseFunctions::'@'`	Any argument value is directly or indirectly an instance of a type	Yes
`@@`	`BaseFunctions::'@@'`	Any argument value is directly or indirectly an instance of a metaclass	Yes

- *Classification expressions.* The *classification operators* are syntactically similar to binary operators, but, instead of an expression as their second operand, they take a type name. The classification operators **`istype`** and **`hastype`** test whether all of the values of their first operand is classified by the named type (either including or not including subtypes, respectively). The `@` operator is similar to **`istype`**, but tests whether at least one of the values of its first operand is classified by the named type. Note that this means that **`istype`** and **`hastype`** evaluate to true on a **`null`** (empty list) value, while `@` evaluates to false.

```
sensors istype ThermalSensor // Are all sensors ThermalSensors?
sensors @ ThermalSensor // Is any sensor a ThermalSensor?
person hastype Administrator
```

The classification operator **`as`**, known as the *cast operator,* performs an **`isType`** test of whether each of the values of its first operand is classified by the named type, and then it selects only those values that pass the test to include in its result. The result values of such a cast expression (if any) are always guaranteed to be instances of the named type.

```
sensors as ThermalSensor
person as Administrator
```

The classification operators may also be used without a first operand, in which case the first operand is implicitly `Anything::self` (see 9.2.2.2.1). This is useful, in particular, when used as a test within an element filter condition expression (see 7.4.14).

```
istype ThermalSensor
@ThermalSensor
hastype Administrator
as Supervisor
```

Metaclassification Expressions

- *Metaclassification expressions.* The metaclassification operators `@@` and **`meta`** take the qualified name of any kind of element as their first operand and a metaclass (see 7.4.13) as their second operand. They are shorthands for classification expressions with the operators `@` and **`as`**, respectively, and a metadata access expression (see 7.4.9.4) as their first operand. As such, `@@` tests whether any metadata associated with an element are classified by the given metaclass, while **`meta`** filters the metadata associated with an element and evaluates to those that are classified by the given metaclass.

```
// Shorthand for designModel.metadata @ ApprovalAnnotation
designModel @@ ApprovalAnnotation

// Shorthand for sensors.metadata as KerML::Feature
sensors meta KerML::Feature

// Evaluates to the string "sensors".
(sensors meta KerML::Feature).name
```

35th Annual INCOSE international symposium

hybrid event

Ottawa, Canada
26 - 31 July 2025