tmforum

TM Forum Technical Report

Function Definition Ontology

TR292C

Maturity Level: **General availability (GA)**	Team Approved Date: 04-Jul-2024
Release Status: Production	Approval Status: TM Forum Approved
Version 3.6.0	IPR Mode: RAND

Notice

of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

Direct inquiries to the TM Forum office:

181 New Road, Suite 304
Parsippany, NJ 07054, USA
Tel No. +1 862 227 1648
TM Forum Web Page: www.tmforum.org

Table of Contents

Executive Summary

This document introduces function notation as elements in intent expression. It improves intuition when working with the intent models.

Functions are a versatile concept. They can be used, for example, for defining logical or numerical operators or to define the creation and modification of data structure. This document provides an ontology and vocabulary for definition and formal description of functions within the TM Forum Intent Ontology (TIO).

Introduction

The concept of a function has multiple slightly different definitions. In mathematics, for example, a function from a set X to a set Y assigns to each element of X exactly one element of Y. In engineering, a function is interpreted as a specific process, action or task that a system is able to perform. In computer programming, a function or subroutine is a sequence of program instructions that performs a specific task, packaged as a unit.

The combined essence of these definitions is that a function represents an operation that delivers a result based on a defined input. The operation is typically a mapping or transformation of input into the resulting output. The words map, mapping, transformation, correspondence, and operator are often used synonymously when describing what a function does and represents. As this concept is used in many fields of science and engineering it is intuitive to use even for users without deeper specialist background and training. Functions are also quite flexible and can represent a broad range of operations as well as input formats and result types. This makes them a useful concept to be used in intent modeling.

Functions are a versatile concept that can be used for a great number of purposes. The models in the TM Forum Intent Ontology including extension models define a notation and format of properties that resembles functions. This addressing many common use cases of intent expression in an intuitive way. This document specifies an ontology for defining and describing functions. This ontology is designed to be used by a great variety of models to introduce and communicate functions consistently.

Scope

Introduce generic classes and properties for defining and describing functions.

Revision Information

This revision v3.6.0 of the function definition ontology model is part of the TM Forum Intent Ontology (TIO) v3.6.0.

The revision v3.6.0 of this document replaces v.3.5.0 with the following changes:

- Minor editorial corrections.

1. Notation and Dependencies

The function definition ontology model depends on the following models.

Model	Prefix	Namespace	Published by	Purpose in the model
Function Definition Ontology	fun	`http://tio.models.tmforum.org/tio/v3.6.0/FunctionOntology`	TM Forum	Model for formal description of functions (This document)
Quantity Ontology	quan	`http://tio.labs.tmforum.org/tio/v3.6.0/QuantityOntology`	TM Forum	Introduction of quantities and related operators
RDF version 1.1	`rdf`	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`	W3C	Providing fundamental modeling artifacts
RDF Schema 1.1	`rdfs`	`http://www.w3.org/2000/01/rdf-schema#`	W3C	Providing fundamental modeling artifacts
XML Schema	`xsd`	`http://www.w3.org/2001/XMLSchema#`	W3C	Providing data types for literal objects
Examples	ex	http://www..example.org/	IANA	Reserved domain name for examples

Table 1.1: Model references

The function definition ontology model is based on the Resource Description Framework (RDF) [rdf, rdf_mt, rdf_primer] and the Resource Description Framework Schema (RDFS) [rdfs] published by the World Wide Web Consortium (W3C).

1.1. Notation and Conventions

The examples provided in this document are presented using the terse RDF triple language (TURTLE) syntax [turtle]. For referring to objects defined in other models and namespaces, the examples use prefixes as defined in Table 1.1. without introducing them with explicit statements in every example. Additionally, the generic namespace prefix "`ex:`" is used for defining example objects. Examples also use XML Schema with the prefix "`xsd:`"

2. Function definition

Functions are introduced as RDF properties. This means a function is an instance of rdf:Property. More specifically, the function definition ontology model introduces the property `fun:function` as sub-property of `rdf:value`. Every distinct function is meant to be introduced as sub property of fun:function.

The range of functions is always `rdf:List`. This means the object of a function is a collection. The elements of the collection are interpreted as the arguments of the function. The function defines an operation with these arguments and the result is assigned to the subject.

In TURTLE notation it is possible to use a syntax with round brackets for specifying collections. This means functions have an intuitive syntax in TURTLE with a format like:

```
<subject> <function> ( <argument1> <argument2> ... )
```

This notation is similar to mathematical functions or functions in many programming languages. They typically consist of a function name, a list of arguments in brackets and a result value that is generated by the function. As most users of the model are familiar with mathematical functions and many would have some experience with programming languages, the function notion for properties and in particular the TURTLE syntax for list representation through brackets is considered to be intuitive and easily understandable for human users. It also provides a compact syntax when expressing nested statements and operators with result values depending on arguments.

2.1. Describing Function Arguments

An essential part of a function definition is the specification of its arguments. This refers to the number of arguments as well as to their types.

`fun:arityMin, fun:arityMax`
The `fun:arityMin` and `fun:arityMax` properties define the number of arguments a function must and can have.

For example, `fun:arityMin` set to 1 and `fun:arityMax` set to 2 means that the function can have one or two arguments. Setting both to "2" means that the function always needs two arguments.

Providing the arity by omitting one or both of the arity properties would indicate that all respective values are possible. For example, setting `fun:arityMin` to "1" and leaving `fun:arityMax` out would mean that the function has at least one argument, but it also can have any number of arguments greater than one. If minimum and maximum arity are not defined, the function is allowed to be used without arguments, which is equivalent to an empty list as object.

`fun:argumentTypes`
The property `fun:argumentTypes` defines the type of each argument of a function. It uses a collection of types expressing the type of each respective argument of the function.

For example, if the first element of the collection of `fun:argumentTypes` is `rdf:Container` and the second member is `icm:Quantity`, this means that the first argument of the function needs to be a container and the second argument needs to be a quantity. If the function can have more arguments than matched by elements in fun:argumentTypes, the last defined type determines the type of all additional arguments. For example, a function with minimum arity of 2 and unlimited maximum arity can have an argument type statement that only contains one member icm:Quantity. This means that all arguments of the function are quantities including the minimum mandatory two arguments and the optional additional ones.

If a function is defined with multiple specifications of argument types, they are considered to be supported alternatives.

2.2. Function specific properties

A function definition can have properties that further steer the function's meaning and the generation of result values.

For example, a function in the ontology model is defined to provide the value of a polynomial. This is a mathematical function. It has an input value x, which would typically be the function's argument. However, a polynomial function is defined by a set of coefficients. The coefficients and their values can be specified as properties to the function and therefore become part of the function definition.

For example:

```
ex:poly1
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  rdfs:label "polynomial" ;
  rdfs:comment "the value of a polynomial according to the input value"
  fun:resultType quan:Quantity ;
  fun:argumentTypes ( quan:Quantity ) ;
  ex:coefficient0 [ a quan:Quantity ;
                    rdf:value 1.0 ] ;
  ex:coefficient1 [ a quan:Quantity ;
                    rdf:value 2.2 ] ;
  ex:coefficient2 [ a quan:Quantity ;
                    rdf:value 3.0 ] ;
  fun:arityMin 1 ;
  fun:arityMax 1 ;
  rdf:range rdf:List
 .
```

This example introduces a function with one argument of type quantity. The function is delivering the resulting value of a polynomial based on the quantity used as input. The function definition specifies the coefficients of the polynomial. Coefficients are provided as properties of the function. Here the first three coefficients are provided, which indicates that the polynomial defined here is a quadratic function. The function is used by only providing the input value as argument, for example: "`ex:poly1 ( 3.45 )`"

Another way to specify this function is by stating all coefficients in the arguments. This would eliminate the need to define a function with additional and function specific properties, and it would make the function more generic, but it also means that all coefficients need to be stated every time the function is used.

For example:

```
ex:poly2
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  rdfs:label "polynomial" ;
  rdfs:comment "the value of a polynomial according to the input value and
coefficients as arguments"
  fun:resultType quan:Quantity ;
  fun:argumentTypes ( quan:Quantity ) ;
  fun:arityMin 1 ;
  rdf:range rdf:List
.
```

This example introduces a function with at least one argument. The first argument is the input value and all other arguments would be interpreted as the coefficients of the polynomial. For example, "`ex:poly2 ( 3.45 1.0 2.2 3.0 )`" would be equivalent to "`ex:poly1 ( 3.45 )`" from the previous example.

2.3. Named arguments

Using the property `fun:argumentNames` it is possible to connect each argument of a function to a property of the function. The range of `fun:argumentNames` is a list which contains one entry per function argument in the order of function arguments. Each entry refers to a property and is interpreted as a property of the function. The properties can be provided directly in the function definition and outside the arguments as well. The values provided for a property in the arguments has preference and would override the values assigned directly to the function. This means the equivalent properties used directly in the function definition are defaults.

For example:

```
ex:poly3
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  rdfs:label "polynomial" ;
  rdfs:comment "the value of a polynomial according to the input value and
named coefficients as arguments"
  fun:resultType quan:Quantity ;
  fun:argumentNames ( ex:x ex:coefficient0 ex:coefficient1 ex:coefficient2 )
  fun:argumentTypes ( quan:Quantity ) ;
  ex:coefficient0 [ a quan:Quantity ;
                    rdf:value 0 ] ;
  ex:coefficient1 [ a quan:Quantity ;
                    rdf:value 0 ] ;
  ex:coefficient2 [ a quan:Quantity ;
                    rdf:value 0 ] ;
```

```
  fun:arityMin 1 ;
  rdf:range rdf:List
.
```

This example defines the polynomial function with named arguments. The values provided when the function is used would define the coefficients of the polynomial. The example also specifies the coefficients as individual properties of the function outside the function arguments. These are default values applied when arguments are left out from the argument list when the function is used. This means, for example, when using "`ex:poly3 ( 3.45 )`" the defaults would apply with all coefficients set to 0. But it is also possible to specify other values for the coefficients such as "`ex:poly3 ( 3.45 1.0 2.2 3.0)`", overriding the defaults with other values.

2.4. Unsorted named arguments

The usage of functions as described in previous chapters depends on order of arguments. This means, each argument is identified by its position in the argument list. With named arguments it is possible to provide argument values by name. For example, the function `ex:poly3` in chapter 2.3 has named arguments. This function can be used, by specifying its arguments with an argument list

```
ex:poly3 ( 3.45 1.0 2.2 3.0)
```

Alternatively the same function call can be expressed using a blind node that contains the function arguments as properties:

```
ex:poly3 [ ex:coefficient1 1.0 ;           ex:coefficient3 3.0
;           ex:x 3.45 ;           ex:coefficient2 2.2          ]
```

Note, that the order of providing the named arguments does not matter as each entry is identified by the argument name.

3. Generating function result values

A function delivers a result. This result is often implicitly generated by a specific implementation of the function according to its definition and purpose. The properties of the function including its arguments are input to the backend process that generates the function result. This result is represented by the property `rdf:value`. The type of the result is specified in the function definition using the `fun:resultType` property.

It is also possible to directly assign a function result to a function instance by adding a statement involving the `fun:result` property. This can, for example, include other functions from which the value of the newly defined function is derived.

For example:

```
ex:poly4
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  rdfs:label "polynomial" ;
  rdfs:comment "the value of a polynomial defined by another function" ;
  fun:resultType quan:Quantity ;
  rdf:value [ ex:poly3 ( _x
                         [ rdf:value 1.0 ]
                         [ rdf:value 2.2 ]
                         [ rdf:value 3.0 ]
                       )
            ]
  fun:argumentTypes ( quan:Quantity ) ;
  fun:arityMin 1 ;
  fun:arityMax 1 ;
  rdf:range rdf:List
.
```

This example assumes there is a function `ex:poly3` already defined, and it allows four arguments. The first is the input value x and the other ones are the coefficients of a polynomial. Based on this the example defines a new function ex:poly4 with only the input value x as single argument. This function defines a specific polynomial characterized by constant coefficient values. The result value of the new function `ex:poly4` is defined using `ex:poly3` with the wanted coefficient values.

4. Applying function result values

Some classes allow their individuals to have associated values. This concept is described in the TM forum Intent ontology TR292A. It means that an individual can have a value associated with it. This allows to use the respective individual wherever a value of this datatype is allowed in the model. The individual embodies the data with respect to type and value. This includes the values of literals. The concept is described with examples in TR292A.

Functions associate their result with their subject. This means they contribute an associated value. This value is the result of evaluating the function with its arguments. The type of the generated result is therefore an essential property of a function specification.

`icm:resultType`
The property `icm:resultType` defines the type of the result. For example, a function that represents a logical operator would provide a boolean truth value. Therefore, the result type of this function is `xsd:boolean`. Another example is the function `quan:biggest`. Its argument is a container of quantities and the result is the quantity that represents the biggest value after considering scale from unit prefix. The result type of this function is therefore `icm:Quantity`.

For example:

```
ex:LatestSliceLatency
  met:lastValue ( ex:SliceLatency )
.
```

This example uses the function `met:lastValue`. Its result type is `quan:quantity`. Its subject is `ex:LatestSliceLatency`. This means `ex:LatestSliceLatency` is of type `quan:quantity` inherited from the function `met:lastValue`. Furthermore, it inherits the properties of the function result. It would at least inherit the rdf:value and potentially the `quan:unit` properties of the function result. Therefore, and assuming the result is a quantity of "10ms", the following definition is equivalent to the one above.

```
ex:LatestSliceLatency
  a quan:Quantity ;
  rdf:value [ rdf:value "10" ;
              quan:unit "ms" ]
.
```

This expression uses rdf:value two times. This is also equivalent in the interpretation of the graph to directly specifying the quantity value and unit as properties of the object.

```
ex:LatestSliceLatency
  a quan:Quantity ;
  rdf:value "10" ;
  quan:unit "ms"
.
```

5. Validity Conditions

Validity context according to the validity model specified in TR291A can be assigned to a function. This means the function would not contribute the result to its subject if the validity context and its conditions evaluate to "false". Its result value would be undefined for literal results or resulting in an instance of the result type without any further properties. This also means that no associated value or result property is provided.

The property `fun:resultIfInvalid` allows specifying a result value to be used in case the function is not valid.

For example:

```
ex:poly4
  a fun:Function ;
  fun:resultType quan:Quantity ;
  fun:argumentNames ( _x ex:coefficient0 ex:coefficient1 ex:coefficient2 )
  fun:argumentTypes ( quan:Quantity ) ;
  fun:arityMin 1 ;
  iv:validIf [ a iv:Validity ;
               quan:greater( _x [ rdf:value 0 ] )
               log:match ( [ imo:Now ]
                           t:inside
                           [ t:hasBeginning [ t:inXSDDateTimeStamp "2023-07-
05T01:00+01:00"^^xsd:dateTimeStamp ] ;
                             t:hasXSDDuration "PT10M"^^xsd:duration ]
                         )
             ]
.
```

This example shows the definition of a function including validity. Here the first argument of the function is also used in the condition expression of the validity context. The function is valid for values of a dimensionless quantity as input, which is greater than "0". This practically defines a value range for the function.

6. Composite Functions

A composite function is a function that derives its result from the combination of one or several other functions with a defined operation to combine the partial results into the function result.

For example:

```
ex:polyX
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  fun:resultType quan:Quantity ;
  fun:argumentNames ( _x )
  fun:argumentTypes ( quan:Quantity ) ;
  ex:coefficients ( 1.0 2.2 3.0 ) ;
  fun:arityMin 1 ;
  fun:arityMax 1 ;
.
ex:polyY
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  fun:resultType quan:Quantity ;
  fun:argumentNames ( _Y )
  fun:argumentTypes ( quan:quantity ) ;
  ex:coefficients ( 1.3 2.3 ) ;
  fun:arityMin 1 ;
  fun:arityMax 1 ;
.
ex:polyAB
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  fun:resultType quan:Quantity ;
  fun:argumentNames ( _CX _CY )
  fun:argumentTypes ( quan:quantity ) ;
  ex:coefficients ( 1.3 2.3 ) ;
  rdf:value [ quan:sum ( ex:polyX (_CX)
                         ex:polyY (_CY)
                       )
            ] ;
  fun:arityMin 2 ;
  fun:arityMax 2 ;
.
```

This example shows a possible definition of the multidimensional function `ex:polyAB`. Its result value is the sum of two one dimensional functions. The result value of the multidimensional function `ex:polyAB` is determined by the sum of the contributing functions `ex:polyX` and `ex:polyY`.

7. Alternative Results

Validity can also be used in composite functions to define alternative results based on conditions. The modeling technique would involve defining a result based on a combination function. The parameters of the combination function would then have individual validity, which includes them in the combination or not.

For example:

```
ex:F1
  a fun:Function ;
  fun:argumentTypes ( quan:Quantity ) ;
  fun:argumentNames ( _x ) ;
  fun:resultType xsd:boolean ;
  rdf:value [ log:anyOf ( [ tio:associatedValue "true" ;
                      iv:validIf [ log:match ( [ imo:Now ]
                                          t:inside
                                          [ t:hasBeginning [ t:inXSDDateTimeStamp
"2023-07-05T01:00+01:00"^^xsd:dateTimeStamp ] ;
                                            t:hasXSDDuration "PT10M"^^xsd:duration ]
                                        )
                                ]
                    ]
                    [ log:smaller ( [ ex:F2 ( _x ) ]
                                "10ms"^^quan:Quantity ) ;
                      iv:validIf [ quan:atLeast ( _x "0"^^quan:Quantity ] ) ]
                    ]
                  )
          ]
.
```

This example shows the definition of a function with boolean result. The result is determined by the function `log:anyOf`. This is used as combination function for partial results with individual validity. The first argument of `log:anyOf` is specified as always true by setting an associated value. Due to the logic "or" relationship of this partial results with other partial results, the entire function `ex:F1` would result in "`true`". However, this partial result is only valid for a specific period of time. This means the function `ex:F1` would always result in "`true`" within this time period irrespective of other result contributions.

The second argument of log:anyOf becomes "`true`" if the result of function `ex:F2` is smaller than "`10ms`" with the input argument of `ex:F1` is passed to `ex:F1`. However, this contribution is only valid if the argument passed to `ex:F1` is at least "0".

If both arguments of `log:anyOf` are invalid, it has no arguments and according to the definition of `log:anyOf` it would default to "`false`".

This example shows that function results can be specified directly in the function with relatively complex conditions and multiple contributors and side conditions. Other functions can be used and combined to produce the result. This also means that not every new function needs a backend implementation to generate its result.

7.1. Value ranges with validity

The creation of a composite functions can also be combined with validity to define that several functions determine the result in different input value ranges. This is mainly relevant for functions with arguments based on numerical data types such as quantities.

A default value to be used if the function is outside its validity range can be specified using iv:valueIfNotValid as specified in the intent validity model TR291A.

For example:

```
ex:poly1
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  fun:resultType quan:Quantity ;
  fun:argumentNames ( _x1 ) ;
  fun:argumentTypes ( quan:Quantity ) ;
  ex:coefficients ( 1.0 2.2 3.0 ) ;
  fun:arityMin 1 ;
  fun:arityMax 1 ;
  iv:validIf [ quan:atMost ( _x1 [ rdf:value 0 ] ) ] ;
  iv:valueIfNotValid [ a quan:Quantity ;
                       rdf:value "0"
                     ]
.
ex:poly2
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  fun:resultType quan:Quantity ;
  fun:argumentNames ( _x2 ) ;
  fun:argumentTypes ( quan:quantity ) ;
  ex:coefficients ( 1.3 2.3 ) ;
  fun:arityMin 1 ;
  fun:arityMax 1 ;
  iv:validIf [ quan:atLeast ( _x2 [ rdf:value 0 ] ) ] ;
  iv:valueIfNotValid [ a quan:Quantity ;
                       rdf:value "0"
                     ]
.
ex:polyCombined
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  fun:resultType quan:Quantity ;
  fun:argumentNames ( _cx ) ;
  fun:argumentTypes ( quan:quantity ) ;
  ex:coefficients ( 1.3 2.3 ) ;
  rdf:value [ quan:mean ( ex:poly1 (_cx)
                          ex:poly2 (_cx)
                        )
            ] ;
  fun:arityMin 1 ;
```

```
  fun:arityMax 1 ;
  iv:validIf [ quan:atLeast ( _cx [ rdf:value -10 ] ) ;
               quan:atMost  ( _cx [ rdf:value  10 ] ) ] ;
  iv:valueIfNotValid "0"^^quan:Quantity
.
```

This example shows a possible definition of a function that is a composite of two other functions. Value ranges are applied using validity context directly to each contributing function. The value of the function `ex:polyCombined` is determined as the arithmetic mean of values contributed from functions `ex:poly1` and `ex:poly2`. Each of these contributing function has individual validity. Here the validity is used to establish value ranges. Note that the value ranges of both contributing functions overlap for the input value "0". This means the result at "0" would be the mean value calculated from both functions. For other input values either `ex:poly1` or `ex:poly2` is invalid and thus removed from the argument list of `quan:mean`. The result is that either the result of `ex:poly1` or the result of `ex:poly2` determines the result of `ex:polyCombined` except for the point "0", where the mean across both is applied.

Alternatively this can be modeled in the following way:

```
ex:poly1
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  fun:resultType quan:Quantity ;
  fun:argumentNames ( _x1 ) ;
  fun:argumentTypes ( quan:Quantity ) ;
  ex:coefficients ( 1.0 2.2 3.0 ) ;
  fun:arityMin 1 ;
  fun:arityMax 1 ;
.
ex:poly2
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  fun:resultType quan:Quantity ;
  fun:argumentNames ( _x2 ) ;
  fun:argumentTypes ( quan:quantity ) ;
  ex:coefficients ( 1.3 2.3 ) ;
  fun:arityMin 1 ;
  fun:arityMax 1 ;
.
ex:polyCombined
  a fun:Function ;
  rdfs:subClassOf quan:Quantity ;
  fun:resultType quan:Quantity ;
  fun:argumentNames ( _cx ) ;
  fun:argumentTypes ( quan:quantity ) ;
  ex:coefficients ( 1.3 2.3 ) ;
  rdf:value [ quan:mean ( [ ex:poly1 ( _cx ) ;
                            iv:validIf [ quan:atLeast ( _cx [ rdf:value 0 ] ) ] ;
                          ]
                          [ ex:poly2 ( _cx ) ;
                            iv:validIf [ quan:atLeast ( _cx [ rdf:value 0 ] ) ] ;
```

```
                    ]
                  )
          ] ;
  fun:arityMin 1 ;
  fun:arityMax 1 ;
  iv:validIf [ quan:atLeast ( _cx [ rdf:value -10 ] ) ;
               quan:atMost  ( _cx [ rdf:value  10 ] ) ] ;
  iv:valueIfNotValid [ a quan:Quantity ;
                       rdf:value "0"
                     ]
.
```

This example shows that the validity of contributions to a function result can be directly applied to the statements that specify how the result is generated from multiple contributions. Each contribution has its individual validity. Here this was used to define value ranges.

8. Function evaluation errors, warnings and information.

The execution of functions can lead to an error. For example a mathematical function for division can lead to the undefined division by zero if the argument representing the denominator assumes the value "0". Also, the use of arguments with unsupported types would lead to errors.

Function related errors are instances of the class **fun:Error**.

Function related warnings are instances of class **fun:Warning**. They can represent if something happens in the use of a function that is not clearly an error but suspicious.

Function related information is represented by instances of class fun:Info. Function related information objects allow providing additional insights in the function evaluation process.

Distinct types of errors, warnings and information are not defined by the function description ontology. They shall be introduced by the model specification that defines a function and together with the function definition. An error type would be a specialization and therefore a subclass of fun:Error. Accordingly, types of warnings and information are specializations of the classes fun:Warning or fun:Info.

Errors:

- No value due to invalid function. This can happen if a validity context is assigned to a function without defining a default result in case the function is invalid.
- Argument type mismatch. An argument was provided that does not have the correct type.

9. Examples

Examples for the use of functions are:

Quantity based conditions: A boolean truth value is derived from evaluating a condition with quantities and their numerical values.
Examples are `quan:greater`, `quan:smaller` or `quan:exactly`.

Logical operators: A boolean truth value is derived from the truth value of its arguments.
Examples are `log:allOf`, `log:oneOf` and `log:none`. They derive a result from conjunction (logical "AND"), disjunction (logical "OR") or negation of their arguments.

Container based conditions: A boolean truth value is derived from evaluating conditions about containers.
Examples are `set:includedIn`, `set:intersectsWith` or `set:elementOf`.

Mathematical operations: A quantity value is derived from its arguments.
Examples are functions for addition, subtraction, multiplication and division.

Container construction: The elements of a container are derived from the arguments.
Examples are `set:union`, `set:intersection` or `set:difference`.

The following chapters define these and further functions in detail.

9.1. Function definition examples

9.1.1. Example 1: Definition of a logical operator

```
ex:oneOf
  a fun:Function ;
  rdfs:subClassOf rdf:Literal ;
  rdfs:label "oneOf" ;
  rdfs:comment "evaluates logical disjunction of arguments" ;
  fun:resultType xsd:boolean ;
  fun:arityMin 0 ;
  fun:argumentTypes ( xsd:boolean ) ;
  rdf:range rdf:List
.
```

This example introduces the logical operator by defining the function ex:oneOf. The function evaluates as logical disjunction, aka logical "OR". This means it has arguments that contribute boolean truth values. If any of the values of the arguments is "true", the result of ex:oneOf assumes also the value "true". The values are of type xsd:boolean using a datatype from XML Schema.

The property fun:resultType states, that the evaluation result of the function assigned to its subject is of type xsd:boolean. A boolean value is a literal and therefore the entire function represents a literal. This is specified by defining the function as subclass of rdf:Literal.

The property fun:argumentTypes states that the arguments also must be of type xsd:boolean.

9.1.2. Example 2: Function for comparing quantities

```
ex:smaller
  a fun:Function ;
  rdfs:subClassOf rdf:Literal ;
  rdfs:label "smaller" ;
  rdfs:comment "evaluates to boolean true if the quantity value of the first
argument is smaller than the quantity value of the second argument" ;
  fun:resultType xsd:boolean ;
  fun:arityMin 2 ;
  fun:arityMax 2 ;
  fun:argumentTypes ( quan:Quantity quan:Quantity ) ;
  rdf:range rdf:List
.
```

This example introduces a function that compares exactly two quantities. The evaluation result is a boolean truth value.

9.1.3. Example 3: Function for intersection of sets

```
ex:intersection
  a fun:Function ;
  rdfs:subClassOf rdfs:Container ;
  rdfs:label "intersection" ;
  rdfs:comment "provides a container that contains only those elements that
the argument containers have in common" ;
  fun:resultType rdfs:Container ;
  fun:argumentTypes ( rdfs:Container ) ;
  rdf:range rdf:List
.
```

This example introduces a function that has containers as arguments. The function represents a new container that contains only the elements that all argument containers have in common. Because of that, the function is defined as subclass of rdfs:Container.

The function is defined without arity specified. This means any number of arguments is possible. However, if arguments are given, they must be of type rdfs:Container. If no argument is given, the resulting container is empty. If only one argument container is given, the resulting container is identical to the single argument container.

10.Administrative Appendix

10.1. Document History

10.1.1. Version History

Version Number	Date Modified	Modified by:	Description of changes
1.0.0	07-Feb-2023	Alan Pope	Final edits prior to publication
3.0.0	11-Apr-2023	Alan Pope	Final edits prior to publication
3.2.0	15-Aug-2023	Alan Pope	Final edits prior to publication
3.4.0	29-Feb-2024	Alan Pope	Final edits prior to publication
3.5.0	03-May-2024	Alan Pope	Final edits prior to publication
3.6.0	04-Jul-2024	Alan Pope	Final edits prior to publication

10.1.2. Release History

Release Status	Date Modified	Modified by:	Description of changes
Pre-production	07-Feb-2023	Alan Pope	Initial release
Pre-production	17-Mar-2023	Adrienne Walcott	Updated to Member Evaluated status
Pre-production	11-Apr-2023	Alan Pope	Updated to v3.0.0
Pre-production	15-May-2023	Adrienne Walcott	Updated to Member Evaluated status
Pre-production	15-Aug-2023	Alan Pope	Updated to v3.2.0
Pre-production	18-Sep-2023	Adrienne Walcott	Updated to Member Evaluated status
Pre-production	29-Feb-2024	Alan Pope	Updated to v3.4.0
Production	26-Apr-2024	Adrienne Walcott	Updated to reflect TM Forum Approved status
Pre-production	03-May-2024	Alan Pope	Updated to v3.5.0
Production	28-Jun-2024	Adrienne Walcott	Updated to reflect TM Forum Approved status
Pre-production	04-Jul-2024	Alan Pope	Updated to v3.6.0
Production	30-Aug-2024	Adrienne Walcott	Updated to reflect TM Forum Approved status

10.2. Acknowledgments

Team Member (@mention)	Company	Role*
Jörg Niemöller	Ericsson	Author, Project Co-Chair
Kevin McDonnell	Huawei	Project Co-Chair

Team Member (@mention)	Company	Role*
Yuval Stein	Amdocs	Project Co-Chair
Kamal Maghsoudlou	Ericsson	Key Contributor
Leonid Mokrushin	Ericsson	Key Contributor
Marin Orlić	Ercisson	Key Contributor
Aaron Boasman-Patel	TM Forum	Additional Input
Alan Pope	TM Forum	Additional Input
Dave Milham	TM Forum	Additional Input
Xiao Hongmei	Inspur	Reviewer

**Select from: Project Chair, Project Co-Chair, Author, Editor, Key Contributor, Additional Input, Reviewer*

11.TR292C Function Definition Ontology v3.6.0 - Vocabulary

Appendix A: Vocabulary reference

This chapter contains a reference definition of all model vocabulary. It is sorted alphabetically:

11.1. argumentNames

The property `fun:argumentNames` is used in function definitions. It allows connecting each argument of the function to a property of the function. The range of `fun:argumentNames` is a list which contains one entry per function argument in the order of function arguments. Each entry refers to a property and is interpreted as a property of the function. The properties can be provided directly in the function definition and outside the arguments as well. The values provided for a property in the arguments has preference and would override the values assigned directly to the function. This means the equivalent properties used directly in the function definition are defaults.

Instance of: `rdf:Property`
Domain: `fun:function`
Range: `rdfs:List`

11.2. argumentTypes

The property `fun:argumentTypes` defines the type of each argument of a function. It uses a collection of types expressing the type of each respective argument of the function. For example, if the 1st member of the collection of `fun:argumentTypes` is `rdf:Container` and the second member is `quan:Quantity`, this means that the first argument of the function needs to be a container and the second argument needs to be a quantity. If the function can have more arguments than matched by elements in `fun:argumentTypes`, the last defined type determines the type of all additional arguments.

Instance of: `rdf:Property`
Range: `rdfs:List`

11.3. arityMax

Defines the maximum number of arguments the function would consider. If the argument collection of the function contains more entries, only the entries up to the max arity are considered, and further entries are ignored.

Instance of: `rdf:Property`
Domain: `fun:Function`
Range: `xsd:nonNegativeInteger`

11.4. arityMin

Defines the minimum number of arguments the function requires.

Instance of: `rdf:Property`
Domain: `fun:Function`
Range: `xsd:nonNegativeInteger`

11.5. Error

Instances of class `fun:Error` are types of errors related to functions and function evaluation. It is a specialization of class imo:Error, which refers to all errors happening in intent handling.

Instance of: `rdfs:Class`
Subclass of: `imo:Error`

11.6. Function

An instance of `fun:Function` represents a function. It is a sub-property of `rdf:Property`. Therefore, instances of fun:function can be used as properties in statements. The range of all instances of `fun:Function` is a list. It represents the list of function arguments.

Sub property of: `rdf:Property`
Range: `rdfs:List`

11.7. resultProperty

The property `fun:resultProperty` specifies the property represented by the function. The function is interpreted as this property of the function subject assigning the function result. The default if not specified is `tio:assumedValue`.

Instance of: `rdf:Property`
Domain: `fun:FunctionRange: rdf:Property`

11.8. resultType

Defines the type of the function evaluation result. This is synonymous to the type of value associated with the subject from function evaluation.

Instance of: `rdf:Property`
Domain: `fun:Function`

11.9. Vocabulary

The object fun:Vocabulary is a container of all model elements.

Instance of: rdfs:Container

11.10. Warning

Instances of class fun:Warning are types of warnings related to functions and function evaluation. It is a specialization of class imo:Warning, which refers to all warnings generated in intent handling.

Instance of: rdfs:Class
Subclass of: imo:Warning