

TM Forum Technical Report

Quantity Ontology

TR292D

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

Copyright © TM Forum 2024. All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to TM FORUM, except as needed for the purpose of developing any document or deliverable produced by a TM FORUM Collaboration Project Team (in which case the rules applicable to copyrights, as set forth in the [TM FORUM IPR Policy](#), must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by TM FORUM or its successors or assigns.

This document and the information contained herein is provided on an “AS IS” basis and TM FORUM DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

TM FORUM invites any TM FORUM Member or any other party that believes it has patent claims that would necessarily be infringed by implementations of this TM Forum Standards Final Deliverable, to notify the TM FORUM Team Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the TM FORUM Collaboration Project Team that produced this deliverable.

The TM FORUM invites any party to contact the TM FORUM Team Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this TM FORUM Standards Final Deliverable by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the TM FORUM Collaboration Project Team that produced this TM FORUM Standards Final Deliverable. TM FORUM may include such claims on its website but disclaims any obligation to do so.

TM FORUM takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this TM FORUM Standards Final Deliverable or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on TM FORUM's procedures with respect to rights in any document or deliverable produced by a TM FORUM Collaboration Project Team can be found on the TM FORUM website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this TM FORUM Standards Final Deliverable, can be obtained from the TM FORUM Team Administrator. TM FORUM makes no representation that any information or list

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

Notice	2
Table of Contents	4
Executive Summary.....	6
Introduction.....	7
Scope.....	7
Revision Information.....	7
1. Notation and Dependencies	8
2. Quantity Objects.....	9
2.1. Quantity value expression.....	9
2.2. Quantity equivalence with literal data types	10
2.3. International System of Units (SI).....	10
2.4. Composite units	11
2.5. Unit Prefixes	12
2.6. Quantity as datatype	13
3. Monetary Quantities	15
4. Custom Unit Expressions	16
5. Quantity operators.....	17
5.1. Quantity arithmetic.....	17
5.2. Quantity comparison	19
6. Potential future additions and ideas (not specified yet in the model).....	21
6.1. Errors.....	21
7. Administrative Appendix.....	22
7.1. Document History	22
7.1.1. Version History.....	22
7.1.2. Release History.....	22
7.2. Acknowledgments.....	22
8. Appendix A: Vocabulary Reference	24
8.1. atLeast.....	24
8.2. atMost.....	24
8.3. difference.....	24
8.4. division.....	24
8.5. exactly	25
8.6. greater	25
8.7. greatest.....	25
8.8. greatestInSet	25

8.9.	inRange	26
8.10.	MonetaryQuantity	26
8.11.	mean.....	26
8.12.	meanOfSet	26
8.13.	median.....	27
8.14.	medianOfSet.....	27
8.15.	multiplication	27
8.16.	multiplicationOfSet	27
8.17.	Quantity	28
8.18.	quantity	28
8.19.	smaller	28
8.20.	smallest	28
8.21.	smallestInSet	29
8.22.	sum.....	29
8.23.	sumOfSet.....	29
8.24.	unit.....	29
8.25.	Vocabulary.....	30

Executive Summary

This document defines quantities and a set of operator functions that allow us to define, modify, combine and compare quantity objects. Quantities are defined with units according to the International System of Quantities as part of the International system of measurement, also known as the metric system, specified by ISO/IEC 80000. Furthermore, this document specifies the monetary quantities with currencies according to ISO 4217 as units.

Introduction

Defining goals based on KPIs and metrics is a common task in intent expression. They often consist of a numerical value in combination with a unit. Quantities, as defined in this document are exactly that: a class with properties representing a numerical literal value in combination with a unit expression. This document also defines operators for constructing, modifying and combining quantities. Further operators allow comparing and derive inference from quantities.

Scope

Defining quantity objects and data types for representing numerical values with a unit. Also, defining a set of operator functions that use quantities as arguments or evaluate quantities.

Revision Information

This revision v3.6.0 of the quantity 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 quantity ontology model depends on the following models:

Model	Prefix	Namespace	Published by	Purpose in the model
Quantity Ontology	quan	http://tio.models.tmforum.org/tio/v3.6.0/QuantityOntology	TM Forum	Specification of quantities and related operators (This document)
Function Definition Ontology	fun	http://tio.models.tmforum.org/tio/v3.6.0/FunctionOntology	TM Forum	Basic expression of functions
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 of data types for literal objects
Examples	ex	http://www..example.org/	IANA	Reserved domain name for examples

Table 1.1: Model references

The quantity 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).

The quantity ontology introduces functions that use quantities as attributes or produce quantities as result of their evaluation. Therefore, the function definition ontology is used.

2. Quantity Objects

A quantity is used to express a numerical value that can also have a unit associated with the value. Quantities represent a structured datatype of class `quan:Quantity`. It has distinct properties to assign a numerical value and a string that represents the unit.

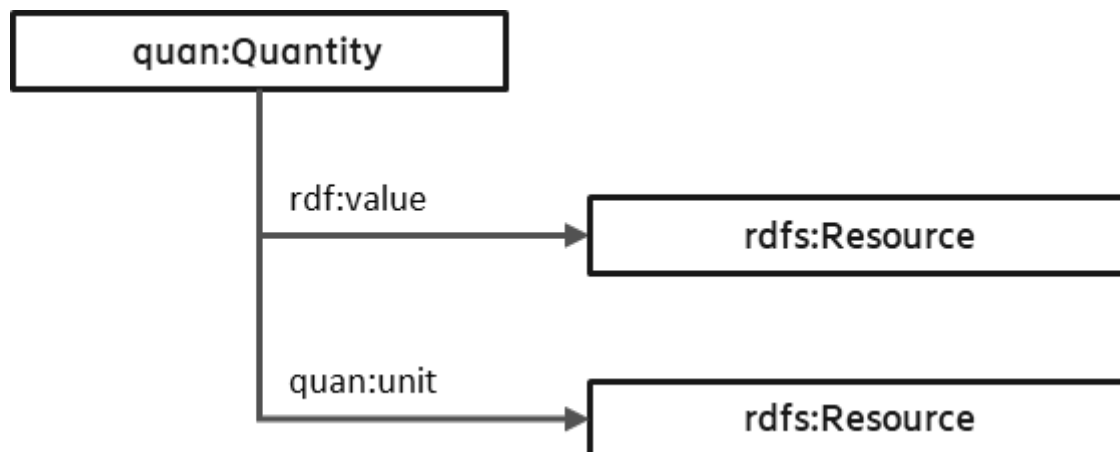


Figure 1: Quantity and its properties

The value of a quantity is assigned using the `rdf:value` property. The range of `rdf:value` is defined in RDF Schema [rdfs] to be `rdfs:Resource`. However, in the context of quantities the range would be limited to a numerical according to XML schema [xsd-2]. Further specialization of the datatype and allowed value ranges can be implied by the metric a quantity is used for. For example, a latency would imply non-negative decimal values.

The unit of measure of a quantity is assigned using the property `quan:unit`. Its range is defined as `rdfs:Resource`. As such it allows the use of any primitive or structured description of a unit.

It is possible to assign multiple units to a quantity. However, all unit expressions assigned to a quantity need to be consistent and equivalent. For example, the unit of electrical power can be expressed as Watt (W). It can also be expressed using base units as "kg·m²·s⁻³". Both are equivalent. They express the same unit dimension and have the same scale through unit prefix. If a unit mismatch is found, an error is reported by the intent manager.

2.1. Quantity value expression

Quantity values in the quantity ontology model are expressed by and interpreted as literals of type `xsd:integer`, `xsd:decimal` or `xsd:double` following the lexical expression defined by XML Schema for these data types. For example, the expressions "100", "100.0" or "1E+2" are alternative representations of the same value.

2.2. Quantity equivalence with literal data types

If no unit is provided for a quantity, it represents a value without unit dimension. This is allowed, and a quantity can therefore be used to express dimensionless metrics such as ratios or the count of things. This also means that expressions with numerical data types such as `xsd:integer`, `xsd:decimal`, `xsd:double` and quantities can be used interchangeably, because they are equivalent in what they represent. This means practically that literals of type `xsd:integer`, `xsd:decimal` or `xsd:double` can be used in the intent when quantities are required according to the TIO model. They would be implicitly interpreted and converted into unitless quantities. This equivalence works the other way around as well. Quantities without unit can be used instead of respective literals.

For example:

```
ex:NumberOfUsers
  a quan:Quantity ;
  rdf:value "10"^^xsd:integer ;
.

ex:C1
  a log:Condition ;
  quan:smaller ( ex:NumberOfUsers
                 "20"^^xsd:integer
               )
.
```

The quantity `ex:NumberOfUsers` is unitless. It represents an integer value of "10". In this example it is used as argument of the `quan:smaller` operator function. All arguments of this function are specified to be of type `quan:Quantity`. This includes also the second argument, which is provided using a literal value "20" of type `xsd:integer`. Due to the equivalence of dimensionless quantities and numeric literals, the second argument is also implicitly interpreted as dimensionless quantity.

2.3. International System of Units (SI)

Units within a quantity object of class `quan:Quantity` are primarily expressed according to the International System of Units (SI) as specified in the ISO/IEC 80000 standard. The quantity ontology model specifically follows the International System of Quantities (ISQ) which is defined in the ISO/IEC 80000 standard. It underlies the International System of Units (SI), which is historically also referred to as the metric system. The SI specification defines symbols to be used for units. For example, the symbol "m" is used to represent meter, the unit of length. Other examples are the symbol "W" for watt, the unit of power, the symbol "s" for seconds, the unit of time. Using these symbols, the unit can be represented as string.

The following example shows how to define an individual quantity:

```
ex:LatencyValue1
  a quan:Quantity ;
  rdf:value "10.35"^^xsd:decimal ;
  quan:unit "s"
.
```

This example defines an individual of type `quan:Quantity`. It has a numerical value of 10.35. Here the data types of XML schema are used to specify that it is a decimal value. The unit of the quantity is milliseconds. According to the ISO/IEC 80000 specification the symbol for seconds is "s".

2.4. Composite units

Quantities often require units that are composed of other units. For example, the unit for velocity is length divided by time as in meter per second. Basic mathematical expressions apply to express these units:

Multiplication

Multiplication is represented by the "*" symbol or by not using a symbol at all. Unit symbols can simply be concatenated. For example Watt multiplied by hours is a common unit for Energy and would be represented by "W*h" or "Wh" using the individual symbols "W" for Watt expressing power and "h" for hour expressing time.

Division

Division can be expressed using the "/" symbol. For example the unit for velocity is length divided by time. A commonly used unit for a velocity quantity is meter per seconds represented by the string "m/s". It contains the unit symbols "m" representing meter and "s" representing second concatenated by "/" indicating a division.

Powers

Powers are represented by the symbol "^" followed by a positive or negative number. For example the unit of volume is length squared. It can be represented by "m^2" expressing meter to the power of two. Volume is length cubed and can therefore be represented by "m^3".

The notation using powers can also express division. The unit expression for velocity "m/s" can alternatively be expressed as "ms^-1".

Brackets are allowed and needed to clarify the meaning of the expression. For example "ms^-1" is equivalent to "m(s)^-1" and "m/s", but different from "(ms)^-1", which would be equivalent to "1/ms".

The following example shows how to define an individual quantity with a composite unit.

```
ex:Energy1
  a quan:Quantity ;
  rdf:value "20.0"^^xsd:decimal ;
  quan:unit "Wh"
```

This example defines a quantity with the unit watt-hours. Its unit symbol string is a multiplication of the unit watt "W" and hours "h".

2.5. Unit Prefixes

The ISO/IEC 80000 standard specifies also the symbols to be used as unit prefix. A unit prefix scales the unit with a defined factor that is represented by a prefix symbols. The quantity ontology proposes to use prefix symbols according to ISO/IEC 80000 according to the following table.

prefix name	prefix symbol	factor
quetta	Q	10^{30}
ronna	R	10^{27}
yotta	Y	10^{24}
zetta	Z	10^{21}
exa	E	10^{18}
peta	P	10^{15}
tera	T	10^{12}
giga	G	10^9
mega	M	10^6
kilo	k	10^3
hecto	h	10^2
deca	da	10^1
—	—	10^0
deci	d	10^{-1}
centi	c	10^{-2}
milli	m	10^{-3}
micro	μ	10^{-6}
nano	n	10^{-9}
pico	p	10^{-12}
femto	f	10^{-15}
atto	a	10^{-18}
zepto	z	10^{-21}
yocto	y	10^{-24}
ronto	r	10^{-27}
quecto	q	10^{-30}

Table 1: Unit prefixes according to ISO/IEC 80000

Unit strings can use unit prefixes in combination with unit symbols. For example, the quantity of 1000 meter can be expressed with the value "1000" in combination with the unit expression "m" or it can be expressed as 1 kilometer using the value "1" and the unit expression "km". Here the symbol "k" is the unit prefix kilo combined with the unit symbols "m".

The prefixes in Table 1 are defined using base 10 factors. However, in information technology often base 2 is preferred and ISO/IEC 80000 defines respective binary prefixes:

prefix factor		prefix symbol	prefix name
1024	2^{10}	Ki	kibi
1024^2	2^{20}	Mi	mebi
1024^3	2^{30}	Gi	gibi
1024^4	2^{40}	Ti	tebi
1024^5	2^{50}	Pi	pebi
1024^6	2^{60}	Ei	exbi
1024^7	2^{70}	Zi	zebi
1024^8	2^{80}	Yi	yobi

Table 2: Binary prefixes

Typical usage for binary prefix are quantities expressing amount of memory in a computer system or data rates. For example a quantity of 1 kbit means 1000 bits, because the base 10 prefix k for kilo was used. The quantity 1 Kibit is equivalent to 1024 bits as the base 2 prefix kibi is used.

2.6. Quantity as datatype

A quantity has a value and unit and is therefore a complex datatype represented by an object with value and unit properties. An individual quantity can therefore be specified using the properties `rdf:value` and `quan:unit` explicitly. Alternatively a string can be used that contains a combination of value and unit expression. This string consists of the concatenation of value expression first followed by the unit expression. A space in between value and unit is allowed, but not mandatory.

The combined value and unit expression string can be explicitly identified as quantity datatype by using the datatype class `quan:quantity` as its data type identifier. This syntax allows a considerably more compact specification of quantities.

For example:

```
ex:C1
  a log:Condition ;
  quan:smaller ( [ a quan:Quantity ;
                  rdf:value "10.1"^^xsd:decimal ;
                  quan:unit "ms"
                ]
                "20.2ms"^^quan:quantity
              )
.
```

This example specifies a condition based on the evaluation of the `quan:smaller` function. Its arguments are both quantities. The first argument uses an explicit definition of an individual quantity object with separate specifications for value and unit. The second quantity uses a combined value and unit expression and identifies its data type as `quan:quantity`. This example also shows the use of the unit prefix "m" for "milli" in both unit expressions.

3. Monetary Quantities

Monetary quantities have a unit expression that identifies a currency. The TM Forum Intent Ontology uses three letter currency identifiers as unit of monetary quantities. These identifiers are defined in ISO 4217. Examples are "EUR" for Euro, "USD" for US Dollar or "SEK" for Swedish Krona.

To avoid ambiguity with SI unit expressions, monetary quantities are represented by individuals of class `quan:MonetaryQuantity`. It is a subclass of `quan:Quantity`.

For example a quantity specified with a monetary unit can look like this:

```
ex:ServicePrice1
  a quan:MonetaryQuantity ;
  rdf:value "10"^^xsd:decimal ;
  quan:unit "USD"
```

This example specifies an individual of type `quan:MonetaryQuantity`. It uses the three letter currency denominator "USD" of the United States Dollar as unit. This quantity represents the amount of 10 US dollars. An alternative expression of this quantity would be `"10 USD"^^quan:MonetaryQuantity`.

4. Custom Unit Expressions

Quantities with units based on the international system of quantities or currencies as introduced in this document cover already a broad range of use cases for intent expression. However, there are still units that are not covered here. Examples are units from other systems of units of measurements, such as the US Customary Unit System or the British Imperial System. Introducing additional units can be done through an intent extension model. It is recommended to define a specialization (subclass) of `quan:Quantity` for the additional units. This avoids accidental ambiguities with SI units associated with `quan:Quantity` directly.

5. Quantity operators

The quantity model defines operations with quantities. This means that quantities are used as result of the function, as arguments or both.

The unit prefix is considered. This means factors are applied to the values according to the prefix if necessary for the operation. Some operations also require that the units of arguments have the same dimension. If this is not the case for the units of the provided arguments, an error event is issued.

A detailed description of the quantity operators, their definition as functions and behavior is provided in the vocabulary reference.

5.1. Quantity arithmetic

The quantity model defines functions for arithmetic operations. The calculations consider units, which means that unit conversions are applied implicitly and consider unit prefixes.

`quan:sum`

Provides the sum of all arguments.

`quan:sumOfSet`

provides a sum of all quantities that are elements of a container. Multiple containers can be provided as arguments and the sum is done across all elements that are quantities across all containers.

`quan:differencehas` has two arguments, and it evaluates the difference of the first argument value minus the second argument value.

`quan:multiplication`

provides a result from the multiplication over all attributes.

`quan:multiplicationOfSet`

provides a result from the multiplication over all member elements from the union of containers specified as arguments. Multiple containers can be provided as arguments and the multiplication is done across all elements that are quantities across all containers.

`quan:division`

divides the first argument by the second. It has exactly two arguments.

`quan:mean`

provides the arithmetic mean value calculated from all provided arguments. It is the sum of the arguments divided by the number of arguments provided.

`quan:meanOfSet`

provides the arithmetic mean value calculated from the member elements of the union of containers provided as arguments. Only members of the containers of type quantity are considered.

`quan:median`

provides the median of the quantities provided as arguments.

quan:medianOfSet

provides the median of the quantities calculated from the member elements of the union of containers provided as arguments. Only members of the containers of type quantity are considered.

quan:smallestprovides the smallest quantity across the provided arguments.

quan:smallestInSetprovides the smallest quantity from a container of quantities.

quan:greatest

provides the largest quantity across the provided arguments.

quan:greatestInSet

provides the largest quantity from a container of quantities.

For example, a sum of quantities would be expressed as:

```
ex:LatencyValue1
  a quan:Quantity ;
  rdf:value "10.35"^^xsd:decimal ;
  quan:unit = "ms"
.
ex:LatencyValue2
  a quan:Quantity ;
  rdf:value "0.0205"^^xsd:decimal ;
  quan:unit "s"
.
ex:LatencySum
  quan:sum ( ex:LatencyValue1
             ex:LatencyValue2
             [ rdf:value 10 ;
               quan:unit "ms" ]
           )
.
```

This example specifies a quantity individual ex:LatencySum as sum of three quantities. It is using the function quan:sum with three arguments. The first two arguments are explicitly defined quantities. The third argument is defined inline with a black node.

The second argument has the unit seconds, while the other arguments use milliseconds. As the unit dimension is compatible this is handled by applying factors to compensate for the unit prefix. By default, the result inherits the unit of the first function argument, which is milliseconds or "ms". The new value is calculated with the factor 0.001 applied to the second argument. The result of the operation is the value 40.85 and the unit set to "ms".

The following example shows a division of two latencies to represent a ratio:

```
ex:LatencyValue3
  a quan:Quantity ;
  rdf:value "10"^^xsd:decimal ;
  quan:unit = "ms"
.
ex:LatencyValue4
  a quan:Quantity ;
  rdf:value "0.02"^^xsd:decimal ;
```

```

    quan:unit "s"
  .
  ex:LatencyRatio
    quan:division ( ex:LatencyValue3
                    ex:LatencyValue4
                    )
  .

```

Also, in this example the underlying calculation would apply a factor to the arguments in order to compensate for unit prefixes. The resulting quantity has the value of 0.5 and no unit. The units of arguments have the same unit dimension and therefore cancel out in a division.

The following example applies a division to arguments with different units:

```

ex:Distance1
  a quan:Quantity ;
  rdf:value "10"^^xsd:decimal ;
  quan:unit = "km"
.
ex:Time1
  a quan:Quantity ;
  rdf:value "2"^^xsd:decimal ;
  quan:unit "h"
.
ex:Speed1
  quan:division ( ex:Distance1
                  ex:Time1
                  )
.

```

The result of the division represents a speed. The unit is formed accordingly from the unit of the arguments. Resulting quantity can have a value of 5.0 with a unit of "km/h". Depending on the implementation other equivalent representations of the result with different units can be chosen. For example, it can have a value of 18.0 with a unit of "m/s". This means the implementation did fall back to base units without prefix. Both representations of the result are equivalent.

5.2. Quantity comparison

Quantity comparison functions provide a boolean result depending on the evaluation of quantities provided as arguments. All arguments are quantities. Unit prefix is considered and factors are applied to the values as needed for comparable units. Argument unit dimensions must match, otherwise an error event is issued in the intent manager. If unit dimensions do not match, the evaluation defaults to "false".

quan:atLeast

is evaluated as "true" if the value of the first argument is equal or bigger than the value of the second argument

quan:atMost

is evaluated as "true" if the value of the first argument is equal or smaller than the value of the second argument

quan:exactly

is evaluated as "true" if the values of all arguments are exactly the same.

quan:greater

is evaluated as "true" if the value of the first argument is bigger than the value of the second argument.

quan:smaller

is evaluated as "true" if the value of the first argument is smaller than the value of the second argument.

quan:inRange

is evaluated as "true" if the value of quantity in the first argument is equal to any of the quantity values the second and third argument or in-between them.

The following example shows the specification of a condition using quantity comparison operators:

```
ex:LatencyValue1
  a quan:Quantity ;
  rdf:value "10.35"^^xsd:decimal ;
  quan:unit "ms"
.
ex:Condition1
  quan:atMost ( ex:LatencyValue1
    [ rdf:value 0.01 ;
      quan:unit "s" ]
  )
.
```

The example specifies that the truth value of an individual condition is derived by a comparison of two quantities. More specifically, the condition is true if and while the quantity `ex:LatencyValue1` is smaller than or equal to 0.01 seconds. In this example the value of `ex:LatencyValue1` is explicitly set, and the evaluation result is therefore always "false", because after adjusting the values for unit prefix factors it is bigger than 0.01 seconds.

In practice, a latency would be a measured metric and its quantity value would be regularly updated in the knowledge base. This means the condition evaluation can lead to different results over time. This demonstrates how condition statements with quantity comparison operators would be essential tools for specifying requirements within an intent.

6. Potential future additions and ideas (not specified yet in the model)

6.1. Errors

Argument unit mismatch

Quantities are used in an operation that requires them to have the same unit dimension. For example, the units in quantity comparison operations must match

Division by zero

A quantity with value 0 is used as denominator in a division operation.

Quantity has no value

A quantity instance is created without a value property. This error is raised once the quantity is used for evaluations and fails due to missing value.

Inconsistent units

multiple unit expressions were assigned to a quantity, and they do not match.

7. Administrative Appendix

7.1. Document History

7.1.1. 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

7.1.2. 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 version 3.0.0
Pre-production	15-May-2023	Adrienne Walcott	Updated to Member Evaluated status
Pre-production	15-Aug-2023	Alan Pope	Updated to version 3.2.0
Pre-production	18-Sep-2023	Adrienne Walcott	Updated to Member Evaluated status
Pre-production	29-Feb-2024	Alan Pope	Updated to version 3.4.0
Production	26-Apr-2024	Adrienne Walcott	Updated to reflect TM Forum Approved status
Pre-production	03-May-2024	Alan Pope	Updated to version 3.5.0
Production	28-Jun-2024	Adrienne Walcott	Updated to reflect TM Forum Approved status
Pre-production	04-Jul-2024	Alan Pope	Updated to version 3.6.0
Production	30-Aug-2024	Adrienne Walcott	Updated to reflect TM Forum Approved status

7.2. 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	Ericsson	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*

8. Appendix A: Vocabulary Reference

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

8.1. atLeast

The function `quan:atLeast` defines a comparison of quantities. It provides a boolean result and is evaluated as "true" if the quantity of the first argument is equal to or bigger than the quantity of the second argument. Unit prefixes are considered.

Instance of: `fun:Function`

Result type: `xsd:boolean`

Arity: exactly 2

Argument types: both arguments are of type `quan:Quantity`

8.2. atMost

The function `quan:atMost` defines a comparison of quantities. It provides a boolean result and is evaluated as "true" if the quantity of the first argument is equal to or smaller than the quantity of the second argument. Unit prefixes are considered.

Instance of: `fun:Function`

Result type: `xsd:boolean`

Arity: exactly 2

Argument types: both arguments are of type `quan:Quantity`

8.3. difference

The function `quan:difference` defines a mathematical operation that provides the numerical difference between two quantities. It has two arguments, and it evaluates to the difference of the first argument value minus the second argument value. Unit prefixes are considered in the calculation. The unit expression of the first argument is used for the result.

Instance of: `fun:Function`

Result type: `quan:Quantity`

Arity: exactly 2

Argument types: both arguments are of type `quan:Quantity`

8.4. division

The function `quan:division` defines a mathematical operation that provides the result of a division of two quantities. The first argument is the numerator and the second the denominator of the division. The function therefore has exactly two arguments. Unit prefixes are considered in the calculation. Both arguments can have different units and the unit expression of the result is generated based on the units of the arguments.

Instance of: `fun:Function`
Result type: `xsd:boolean`
Arity: exactly 2
Argument types: both arguments are of type `quan:Quantity`

8.5. **exactly**

The function `quan:exactly` defines a comparison of quantities. It provides a boolean result and is evaluated as "true" if the quantity of all arguments are the same. Unit prefixes are considered.

Instance of: `fun:Function`
Result type: `xsd:boolean`
Arity: at least two arguments
Argument types: all arguments are of type `quan:Quantity`

8.6. **greater**

The function `quan:greater` defines a comparison of quantities. It provides a boolean result and is evaluated as "true" if the quantity of the first argument is greater than the quantity of the second argument. Unit prefixes are considered.

Instance of: `fun:Function`
Result type: `xsd:boolean`
Arity: exactly 2
Argument types: both arguments are of type `quan:Quantity`

8.7. **greatest**

The function `quan:greatest` provides a result that is the largest quantity across all quantities provided as arguments. If no argument is provided the result has value 0 without unit. Units and unit prefixes are considered.

Instance of: `fun:Function`
Result type: `quan:Quantity`
Arity: any
Argument types: all arguments are of type `quan:Quantity`

8.8. **greatestInSet**

The function `quan:greatestInSet` provides a result that is the largest quantity across all quantities across member elements of the union of all containers that are provided as arguments. If no argument is provided the result has value 0 without unit. Units and unit prefixes are considered.

Instance of: `fun:Function`
Result type: `quan:Quantity`
Arity: any
Argument types: all arguments are of type `rdfs:Container`

8.9. inRange

The function `quan:inRange` defines a comparison of quantities. It provides a boolean result and is evaluated as "true" if the value of the quantity of the first argument is equal to or greater than the quantity of the second argument and if it is equal to or smaller than the quantity of the third argument. Unit prefixes are considered.

Instance of: `fun:Function`

Result type: `xsd:boolean`

Arity: exactly 3

Argument types: all arguments are of type `quan:Quantity`

8.10. MonetaryQuantity

The class `quan:MonetaryQuantity` refers to a quantity with currency based units.

Instance of: `rdfs:Class`

Subclass of: `quan:quantity`

8.11. mean

The function `quan:mean` defines a mathematical operation that provides the arithmetic mean of quantities. The function can have any number of arguments of type `quan:Quantity` and the result is the arithmetic mean across all of them. The result is 0 with no unit if no argument is provided. If only one argument is given, then the result is identical with the argument. Units and unit prefixes are considered in the calculation. The arguments must have units that are of the same dimension and the unit assigned to the result is the unit of the first argument.

Instance of: `fun:Function`

Result type: `quan:Quantity`

Arity: at least two arguments

Argument types: all arguments are of type `quan:Quantity`

8.12. meanOfSet

The function `quan:meanOfSet` defines a mathematical operation that provides the arithmetic mean of quantities. The function can have any number of arguments of type `rdfs:Container` and the result is the arithmetic mean across all member elements of the union of these containers. The result is 0 with no unit if no member element of type `quan:quantity` is provided by any of the argument containers. Units and unit prefixes are considered in the calculation. The arguments must have units that are of the same dimension and the unit assigned to the result is the unit of the first member element processed from the first argument container.

Instance of: `fun:Function`

Result type: `quan:Quantity`

Arity: at least one argument

Argument types: all arguments are of type `rdfs:Container`

8.13. median

The function `quan:median` defines a mathematical operation that provides the median value of quantities. The function can have any number of arguments of type `quan:Quantity` and the result is the arithmetic median value across all of them. The result is 0 with no unit if no argument is provided. If only one argument is given, then the result is identical with the argument. Units and unit prefixes are considered in the calculation. The arguments must have units that are of the same dimension and the unit assigned to the result is the unit of the first argument.

Instance of: `fun:Function`

Result type: `quan:Quantity`

Arity: at least two arguments

Argument types: all arguments are of type `quan:Quantity`

8.14. medianOfSet

The function `quan:medianOfSet` defines a mathematical operation that provides the median value of quantities. The function can have any number of arguments of type `rdfs:Container` and the result is the median value across all member elements of the union of these containers. The result is 0 with no unit if no member element of type `quan:quantity` is provided by any of the argument containers. Units and unit prefixes are considered in the calculation. The arguments must have units that are of the same dimension and the unit assigned to the result is the unit of the first member element processed from the first argument container.

Instance of: `fun:Function`

Result type: `quan:Quantity`

Arity: at least one argument

Argument types: all arguments are of type `rdfs:Container`

8.15. multiplication

The function `quan:multiplication` defines a mathematical operation that provides the result of a multiplication of quantities. The function can have any number of arguments of type `quan:Quantity` and the result is the multiplication of all of them. The result is 0 with no unit if no argument is provided. If only one argument is given, then the result is identical with the argument. Units and unit prefixes are considered in the calculation. The result unit depends on the unit of contributing quantities and a suitable unit will be assigned.

Instance of: `fun:Function`

Result type: `quan:Quantity`

Arity: at least two arguments

Argument types: all arguments are of type `quan:Quantity`

8.16. multiplicationOfSet

The function `quan:multiplicationOfSet` defines a mathematical operation that provides the result of a multiplication of quantities. The function can have any number of arguments of type `rdfs:Container` and the result is the multiplication of all member

elements of the union of these containers. The result is 0 with no unit if no member element of type `quan:Quantity` is provided by any of the argument containers. Units and unit prefixes are considered in the calculation. The result unit depends on the unit of contributing quantities and a suitable unit will be assigned.

Instance of: `fun:Function`

Result type: `quan:Quantity`

Arity: at least one argument

Argument types: all arguments are of type `rdfs:Container`

8.17. Quantity

The class `quan:Quantity` defines a structured datatype for representing a numerical value in combination with a unit expression.

Instance of: `rdfs:Class`

8.18. quantity

The datatype `quan:quantity` is a string that contains a combination of numerical value and unit. A literal of this datatype is equivalent to an individual of class `quan:Quantity` with value and unit specified separately through the properties `rdf:value` and `quan:unit`.

Instance of: `rdfs:Datatype`

8.19. smaller

The function `quan:smaller` defines a comparison of quantities. It provides a boolean result and is evaluated as "true" if the quantity of the first argument is smaller than the quantity of the second argument. Unit prefixes are considered.

Instance of: `fun:Function`

Result type: `xsd:boolean`

Arity: exactly 2

Argument types: both arguments are of type `quan:Quantity`

8.20. smallest

The function `quan:smallestInSet` provides a result that is the smallest quantity across all quantities provided as arguments. If no argument is provided the result has value 0 without unit. Units and unit prefixes are considered.

Instance of: `fun:Function`

Result type: `quan:Quantity`

Arity: any

Argument types: all arguments are of type `quan:Quantity`

8.21. `smallestInSet`

The function `quan:smallestInSet` provides a result that is the smallest quantity across all quantities across member elements of the union of all containers that are provided as arguments. If no argument is provided the result has value 0 without unit. Units and unit prefixes are considered.

Instance of: `fun:Function`

Result type: `quan:Quantity`

Arity: any

Argument types: all arguments are of type `rdfs:Container`

8.22. `sum`

The function `quan:sum` defines a mathematical operation that provides the result of an addition of quantities. The function can have any number of arguments of type `quan:Quantity` and the result is the sum of all of them. The result is 0 with no unit if no argument is provided. If only one argument is given, then the result is identical with the argument. Units and unit prefixes are considered in the calculation. The arguments must have units that are of the same dimension and the unit assigned to the result is the unit of the first argument.

Instance of: `fun:Function`

Result type: `quan:Quantity`

Arity: at least two arguments

Argument types: all arguments are of type `quan:Quantity`

8.23. `sumOfSet`

The function `quan:sumOfSet` defines a mathematical operation that provides the result of an addition of quantities. The function can have any number of arguments of type `rdfs:Container` and the result is the sum of all member elements of the union of these containers. The result is 0 with no unit if no member element of type `quan:quantity` is provided by any of the argument containers. Units and unit prefixes are considered in the calculation. The arguments must have units that are of the same dimension and the unit assigned to the result is the unit of the first member element processed from the first argument container.

Instance of: `fun:Function`

Result type: `quan:Quantity`

Arity: at least one argument

Argument types: all arguments are of type `rdfs:Container`

8.24. `unit`

The property `quan:unit` assigns a unit to a quantity.

Instance of: `rdf:Property`

Domain: `quan:QuantityRange` `xsd:string`

8.25. Vocabulary

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

Instance of: `rdfs:Container`