# TM Forum Technical Report

# Conditions and Logical Operators

## TR292E

| | |
|---|---|
| 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

The ontology model specified in this document defines logical operators that can be used to concatenate atomic conditions into a logical tree. Evaluating this tree determines the compliance of a system to partial requirements and ultimately to an intent.

# Introduction

The expression of requirements by the TM Forum Intent Ontology is based on specifying the conditions a system must meet for being considered compliant. Usually an intent contains a set of related requirements. In simple cases the system is compliant if it meets all requirements. This constitutes a logical conjunction. However, often there are alternatives or a scoped validity of some requirements needs to be considered. This means the compliance of the system to an entire intent depends not only on compliance to each atomic condition, but on potentially complex logical relationships to be evaluated. The TM Forum Intent Ontology approaches this concern by introducing logical operators that can be used to express the relationship of partial requirements and therefore how to evaluate overall system compliance. The conditions are logically aggregated through logical operators leading to the conclusion if the system is compliant to the entire intent.

Conditions and logical operators are used for intent compliance conditions, but they are not limited to this concern only. Intent extension models can use the same vocabulary for expressing conditions with a different interpretation. For example the intent validity model uses the same condition and logical operator vocabulary for expressing if and when an intent or parts of it are valid or not.

The logical operator's ontology model specified in this document introduces a set of logical operators available for expressing logical relationships in intent modeling. It is a mandatory component of the intent common model. This means these operators are available in intent manager that supports intents according to the TMF Intent Ontology with at least version 3 of the intent common model.

## Revision Information

This revision v3.6.0 of the conditions and logical operators ontology is part of the TM Forum Ontology (TIO) version 3.6.0.

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 logical operators ontology model depends on the following models:

| Model | Prefix | Namespace | Published by | Purpose in the model |
|---|---|---|---|---|
| Conditions and Logical Operators Ontology | log | http://tio.models.tmforum.org/tio/v3.6.0/LogicalOperators | TM Forum | (this model) Specifying how conditions are represented and definition of logical operators |
| Quantity Ontology | quan | http://tio.models.tmforum.org/tio/v3.6.0/QuantityOntology | TM Forum | Defintion of quantities and quantity operators |
| Function Definition Ontology | fun | http://tio.models.tmforum.org/tio/v3.6.0/FunctionOntology | TM Forum | Basic expression of functions |
| Intent Common Model | icm | http://tio.models.tmforum.org/tio/v3.6.0/IntentCommonModel | TM Forum | General ontology model of intent and intent report expression. Used in this document for examples |
| 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 conditions and logical operators 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 logical operators ontology introduces functions that use boolean truth values as attributes and produce boolean truth values as result of their evaluation. Therefore, the function definition ontology is used.

# 2. Logical Operator Functions

Logical operators are introduced as functions following the function definition ontology. All the specified logical operators have a result type of boolean. All arguments of the function also need to be of type boolean.

If arguments are provided that have a different type than boolean an error would be issued and the respective argument would be ignored. This means the evaluation would be performed with an argument list from which all non-boolean arguments are removed.

The logical operators ontology defines the following functions.

`log:allOf`
Logical conjunction operator. It evaluated to "true" only if all arguments are "true".

`log:anyOf`
Logical disjunction operator. It evaluates to "true" if any of the arguments is "true".

`log:oneOf`
logical exclusive disjunction. It evaluates to "true" if exactly one of the arguments is "true".

`log:noneOf`
Logical negation operator. It evaluates to "true" only if all arguments are "false".

This example shows how to define the truth value of a condition based on other conditions.

```
ex:Condition1
  quan:smaller ( ex:SliceLatency1
                 [ rdf:value "10"^^xsd:decimal ;
                   quan:unit "ms" ]
               )
.
ex:Condition2
  quan:atMost ( ex:CEMLatency1
                [ rdf:value "20"^^xsd:decimal ;
                  quan:unit "ms" ]
              )
.
ex:Condition3
  log:anyOf ( ex:Condition1 ex:Condition2 )
.
```

In this example two conditions are defined that specify distinct requirements using two different latency metrics. The overall requirement is that the system shall meet either of the conditions. This is expressed by `ex:Condition3` using the `log:anyOf` operator to logically combine the results of `ex:Condition1` and `ex:Condition2`.

# 3. Statement Evaluation Functions

The TM Forum Intent Ontology allows inference from the truth of RDF statements. A statement in RDF is an instance of class `rdf:Statement`. An RDF statement is the statement made by a token of an RDF triple. The subject of an RDF statement is the instance of `rdfs:Resource` identified by the subject of the triple. The predicate of an RDF statement is the instance of `rdf:Property` identified by the predicate of the triple. The object of an RDF statement is the instance of `rdfs:Resource` identified by the object of the triple [rdf11]. An instance of this class has therefore three properties: `rdf:subject`, `rdf:predicate`, `rdf:object`.

The following functions derive their truth value from statements.

`log:matchStatement ( <rdf:Statement> ... )`

The function `log:matchStatement` represents the boolean truth value of the statement in the function argument. It is true, if and only if all statements in its arguments are true in the current knowledge base. For example:

```
ex:Condition1
  a log:Condition ;
  log:matchStatement ( [ rdf:subject   ex:User1 ;
                         rdf:predicate ex:userClass ;
                         rdf:object    ex:UserClassGold ]
                     )
.
```

This example specifies the condition `ex:Condition1` using the statement evaluation function `log:matchStatement`. The function and therefore the condition evaluate to true if there is evidence in the knowledge base that shows that `ex:User1` has `ex:userClass` `ex:UserClassGold`. The evaluation can lead to this result either directly if the knowledge base contains a direct assertion of this statement or indirectly if the truth of this statement can be logically derived by the reasoner from other knowledge. For example if the user is part of a user group for which it is known that all members of the group have the `ex:userClass` `ex:UserClassGold`.

`log:match ( <subject> <predicate> <object> )`

The function `log:match` represents the boolean truth value of the statement triple in the function arguments. The first argument represents the subject, the second argument represents the predicate and the third argument represents the object of a statement. It is true, if and only if the statement represented by this triple is true in the current knowledge base. For example:

```
ex:Condition1
  a log:Condition ;
  log:match ( ex:User1 ex:userClass ex:UserClassGold )
.
```

This example specifies the condition `ex:Condition1` using the statement evaluation function `log:match`. The function has three arguments representing the subject, predicate and object of the statement to be evaluated. This example is logically equivalent to the example given for the `ex:matchStatement` function.

`log:matchAny ( <subject container> <predicate> <object> )`

The function `log:matchAny` represents a boolean truth value from multiple statement triples. A constituent statement is formed by using a member of the container in the first function argument as subject. The constituent statement is completed with the predicate from the second argument and the resource from the third argument. Therefore, the function needs to evaluate as many statements as there are members in the subject container. The function is evaluated with a logical disjunction of constituent statements. The result is boolean "`true`", if any of the constituent statements is "`true`". For example:

```
ex:Condition1
  a log:Condition ;
  log:matchAny ( [ a rdfs:Container ;
                  rdfs:member ex:User1 ;
                  rdfs:member ex:User2
                ]
                ex:userClass
                ex:UserClassGold
              )
.
```

This example specifies the condition `ex:Condition1` using the statement evaluation function `log:matchAny`. The function has three arguments. The first argument is a container of statement subjects. Here, `ex:User1` and `ex:User2` are therefore statement subjects. The second argument is the statement predicate and the third argument is the statement object. The function therefore evaluates two distinct statements with the same predicate and object, but with the two users from the subject container as statement subjects. If any of the statements is true according to the current knowledge base, the result of the function is boolean "`true`". This means it is true if any of the two users in the subject are of user class `ex:UserClassGold`.

`log:matchAll ( <subject container> <predicate> <object> )`

The function `log:matchAll` represents a boolean truth value from multiple statement triples. A constituent statement is formed by using a member of the container in the first function argument as subject. The constituent statement is completed with the predicate from the second argument and the resource from the third argument. Therefore, the function needs to evaluate as many statements as there are members in the subject container. The function is evaluated with a logical conjunction of constituent statements. The result is boolean "`true`", if all the constituent statements are "`true`".

For example:

```
ex:Condition1
  a log:Condition ;
  log:matchAll ( [ a rdfs:Container ;
                  rdfs:member ex:User1 ;
                  rdfs:member ex:User2
                ]
                ex:userClass
```

```
                ex:UserClassGold
              )
.
```

This example specifies the condition `ex:Condition1` using the statement evaluation function `log:matchAll`. The function has three arguments. The first argument is a container of statement subjects. Here, `ex:User1` and `ex:User2` are therefore statement subjects. The second argument is the statement predicate and the third argument is the statement object. The function therefore evaluates two distinct statements with the same predicate and object, but with the two users from the subject container as statement subjects. Only if all the statements are true according to the current knowledge base, the result of the function is boolean "`true`". This means it is true if both users in the subject are of user class `ex:UserClassGold`.

`log:matchOne ( <subject container> <predicate> <object> )`
The function `log:matchOne` represents a boolean truth value from multiple statement triples. A constituent statement is formed by using a member of the container in the first function argument as subject. The constituent statement is completed with the predicate from the second argument and the resource from the third argument. Therefore, the function needs to evaluate as many statements as there are members in the subject container. The function is evaluated with a logical exclusive disjunction of constituent statements. The result is boolean "`true`", if exactly one of the constituent statements is "`true`". For example:

```
ex:Condition1
  a log:Condition ;
  log:matchOne ( [ a rdfs:Container ;
                   rdfs:member ex:User1 ;
                   rdfs:member ex:User2
                 ]
                 ex:userClass
                 ex:UserClassGold
               )
.
```

This example specifies the condition `ex:Condition1` using the statement evaluation function `log:matchOne`. The function has three arguments. The first argument is a container of statement subjects. Here, `ex:User1` and `ex:User2` are therefore statement subjects. The second argument is the statement predicate and the third argument is the statement object. The function therefore evaluates two distinct statements with the same predicate and object, but with the two users from the subject container as statement subjects. Only if exactly one of the statements is true according to the current knowledge base, the result of the function is boolean "`true`". This means it is true if either `ex:User1` or `ex:User2` are of user class `ex:UserClassGold`, but not if both are.

`log:matchNone ( <subject container> <predicate> <object> )`
The function `log:matchNone` represents a boolean truth value from multiple statement triples. A constituent statement is formed by using a member of the container in the first function argument as subject. The constituent statement is completed with the predicate from the second argument and the resource from the third argument. Therefore, the function needs to evaluate as many statements as there are members in the subject container. The function is evaluated with a logical negation of constituent

statements. The result is boolean "`true`", if none of the constituent statements are "`true`". For example:

```
ex:Condition1
  a log:Condition ;
  log:matchNone ( [ a rdfs:Container ;
                    rdfs:member ex:User1 ;
                    rdfs:member ex:User2
                  ]
                  ex:userClass
                  ex:UserClassGold
                )
.
```

This example specifies the condition `ex:Condition1` using the statement evaluation function `log:matchNone`. The function has three arguments. The first argument is a container of statement subjects. Here, `ex:User1` and `ex:User2` are therefore statement subjects. The second argument is the statement predicate and the third argument is the statement object. The function therefore evaluates two distinct statements with the same predicate and object, but with the two users from the subject container as statement subjects. Only if none one of the statements is true according to the current knowledge base, the result of the function is boolean "`true`". This means, none of the users `ex:User1` or `ex:User2` are of user class `ex:UserClassGold`.

# 4. Conditions

In intent expressions, conditions are used to specify requirements. The class `log:Condition` represents a condition. An instance of this class contains statements to be logically evaluated in order to get the truth value of the condition. This means that every condition object is associated with a boolean type truth value.

The truth value of the condition is typically determined by functions with a truth value result.

For example:

```
ex:C1
  a log:Condition ;
  quan:smaller ( ex:Latency1
               [ rdf:value "10"^^xsd:decimal ;
                 unit "ms" ]
              )
.
```

This example determines the boolean truth value from the comparison of quantities. The comparison function `quan:smaller` has a boolean result contributed to its subject, which is the condition `ex:C1`.

If multiple properties or functions are used within a condition, they are by default in a logical conjunction. The truth value of the condition gets "true" if and only if all contributing properties provide the value "true" as well. For example:

```
ex:C1
  a log:Condition ;
  quan:smaller ( ex:Latency1
               [ rdf:value "10"^^xsd:decimal ;
                 unit "ms" ]
              ) ;
  quan:greater ( ex:Availability1
               [ rdf:value "0.999"^^xsd:decimal ]
              )
.
```

This example shows two functions contributing to the truth value of the condition `ex:C1`. Consequently, the condition only assumes the value "true", if and while the referenced latency is smaller than 10 milliseconds and the referenced availability is greater than 0.999.

Because conditions represent a truth value, they can be used as parameters in logical statements. This allows us to apply logical operators to aggregate conditions into combined results. For example:

```
ex:C1
  a log:Condition ;
  quan:smaller ( ex:Latency1
               [ rdf:value "10"^^xsd:decimal ;
                 unit "ms" ]
              )
```

```
ex:C2
  a log:Condition ;
  quan:greater ( ex:Availability1
                [ rdf:value "0.999"^^xsd:decimal ]
              )
.
ex:C3
  a log:Condition ;
  log:anyOf ( ex:C1 ex:C2 )
.
```

This example defines that the truth value of the condition ex:C1 shall be determined by the logical disjunction of conditions ex:C1 and ex:C2. This also demonstrates how a logical operator function such as `log:anyOf` can be used if something other than the default logical conjunction is needed for combining partial conditions.

A condition without properties that determine its value would be assumed to have the value "false" by default.

An instance of class log:Condition carries a boolean truth value without assuming a particular interpretation of the value. Other elements of the intent common model, such as intent and expectation not only bear a truth value, but they interpret the value as compliance to requirements. The truth value of a condition can be used to determine the truth value of intents and expectations, and it is therefore indirectly interpreted as compliance.

Conditions are not exclusively used for compliance. They can also contribute truth values into other contexts with a different interpretation. The validity of requirements is an example of a different concern that is based on an interpretation of a truth value. Conditions are universal and can also contribute to this other interpretation of their truth value. Intent extension models can utilize conditions as building blocks for a great variety of use cases.

# 5. Administrative Appendix

## 5.1. Document History

### 5.1.1. 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 |

### 5.1.2. 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 |

## 5.2. 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*

# 6. Appendix A: Vocabulary Reference

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

## 6.1. allOf

The function `log:allOf` represents a logical conjunction also known as AND logical operator. It evaluates to boolean "true" if all arguments are individually true, otherwise the result is "false". If the list of arguments is empty the result is also "false".

Instance of: `fun:Function`
Result type: `xsd:boolean`
Arity: None or any number of arguments
Argument types: `xsd:boolean` for all arguments provided.

## 6.2. anyOf

The function `log:anyOf` represents a logical disjunction also known as OR logical operator if there are only two arguments. It evaluates to boolean "true" if any (at least one) of the arguments are true, otherwise the result is "false". If the list of arguments is empty the result is also "false"

Instance of: `fun:Function`
Result type: `xsd:boolean`
Arity: None or any number of arguments
Argument types: `xsd:boolean` for all arguments provided.

## 6.3. Condition

The class `log:Condition` expresses that its instance is specifying a condition statement with a boolean result.

Instance of: `rdfs:Class`Subclass of: `icm:IntentElement`

## 6.4. match

The function `log:match` represents the boolean truth value of the statement triple in the function arguments. The first argument represents the subject, the second argument represents the predicate and the third argument represents the object of a statement. It is true, if and only if the statement represented by this triple is true in the current knowledge base. Closed world assumption applies. If no statement is given as arguments or if it is incomplete, the result is "false".

Instance of: `fun:Function`
Result type: `xsd:boolean`
Arity: Exactly three arguments
Argument types: The first argument is an instance of class `rdfs:Resource`, the second

argument is an instance of class `rdf:Property` and the third argument is an instance of `rdfs:Resource` or a literal.

### 6.5. matchAll

The function `log:matchAll` represents a boolean truth value from multiple statement triples. A constituent statement is formed by using a member of the container in the first function argument as subject. The constituent statement is completed with the predicate from the second argument and the resource from the third argument. Therefore, the function needs to evaluate as many statements as there are members in the subject container. The function is evaluated with a logical conjunction of constituent statements. The result is boolean "`true`", if all the constituent statements is "`true`". Closed world assumption applies. If no complete statement is given by the function arguments, the result is "false". This is also the case if the subject container in the first argument is empty, thus no subject is specified.

Instance of: `fun:Function`Result type: `xsd:boolean`Arity: Exactly three arguments Argument types: The first argument is an instance of class `rdfs:Container`, with members of class `rdfs:Resource`. The second argument is an instance of class `rdf:Property` and the third argument is an instance of `rdfs:Resource`.

### 6.6. matchAny

The function `log:matchAny` represents a boolean truth value from multiple statement triples. A constituent statement is formed by using a member of the container in the first function argument as subject. The constituent statement is completed with the predicate from the second argument and the resource from the third argument. Therefore, the function needs to evaluate as many statements as there are members in the subject container. The function is evaluated with a logical disjunction of constituent statements. The result is boolean "`true`", if any of the constituent statements is "`true`". Closed world assumption applies. If no complete statement is given by the function arguments, the result is "false". This is also the case if the subject container in the first argument is empty, thus no subject is specified.

Instance of: `fun:Function`Result type: `xsd:boolean`Arity: Exactly three arguments Argument types: The first argument is an instance of class `rdfs:Container`, with members of class `rdfs:Resource`. The second argument is an instance of class `rdf:Property` and the third argument is an instance of `rdfs:Resource`.

### 6.7. matchNone

The function `log:matchNone` represents a boolean truth value from multiple statement triples. A constituent statement is formed by using a member of the container in the first function argument as subject. The constituent statement is completed with the predicate from the second argument and the resource from the third argument. Therefore, the function needs to evaluate as many statements as there are members in the subject container. The function is evaluated with a logical negation of constituent statements. The result is boolean "`true`", if none of the constituent statements are "`true`". Closed world assumption applies. If no complete statement is given by the function arguments, the result is "false". This is also the case if the subject container in the first argument is empty, thus no subject is specified.

Instance of: `fun:Function`Result type: `xsd:boolean`Arity: Exactly three arguments Argument types: The first argument is an instance of class `rdfs:Container`, with members of class `rdfs:Resource`. The second argument is an instance of class `rdf:Property` and the third argument is an instance of `rdfs:Resource`.

## 6.8. matchOne

The function `log:matchOne` represents a boolean truth value from multiple statement triples. A constituent statement is formed by using a member of the container in the first function argument as subject. The constituent statement is completed with the predicate from the second argument and the resource from the third argument. Therefore, the function needs to evaluate as many statements as there are members in the subject container. The function is evaluated with a logical exclusive disjunction of constituent statements. The result is boolean "`true`", if exactly one of the constituent statements is "`true`". Closed world assumption applies. If no complete statement is given by the function arguments, the result is "false". This is also the case if the subject container in the first argument is empty, thus no subject is specified.

Instance of: `fun:Function`Result type: `xsd:boolean`Arity: Exactly three arguments Argument types: The first argument is an instance of class `rdfs:Container`, with members of class `rdfs:Resource`. The second argument is an instance of class `rdf:Property` and the third argument is an instance of `rdfs:Resource`.

## 6.9. matchStatement

The function `log:matchStatement` represents the boolean truth value of the statement in the function argument. It is true, if and only if all statements in its arguments are true in the current knowledge base. Closed world assumption applies. If no statement is given as argument, the result is "false".

Instance of: `fun:Function`
Result type: `xsd:boolean`
Arity: None or any number of arguments
Argument types: all arguments are instances of `rdf:Statement`.

## 6.10. noneOf

The function `log:noneOf` represents an evaluation similar to logical negation. It evaluates to boolean "true" if all the arguments are "false" and to "false" is any of the arguments are "true". If the list of arguments is empty the result is "true".

Instance of: `fun:Function`
Result type: `xsd:boolean`
Arity: None or any number of arguments
Argument types: `xsd:boolean` for all arguments provided

## 6.11. oneOf

The function `log:oneOf` represents an exclusive disjunction also known as XOR logical operator if there are only two arguments. It evaluates to boolean "true" if exactly one of the arguments is true, otherwise the result is "false". If the list of arguments is empty the result is also "false"

Instance of: `fun:Function`
Subclass of: `rdf:Literal`
Result type: `xsd:boolean`
Arity: None or any number of arguments
Argument types: `xsd:boolean` for all arguments provided

## 6.12. Vocabulary

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

Instance of: `rdfs:Container`