Enhancements to the TOSCA data filtering abilities

Contact: anatoly.katzman@att.com

# Introduction

The TOSCA constraint clause (3.6.3) has numerous applications in the TOSCA grammar. It is an important building block for such constructs as:

* Data type definitions
* Property, attribute and input definitions
* Node filters
* Imperative workflow conditions
* Policy triggers

Unfortunately, the TOSCA constraint-based definitions in their present form fall short of expectations in too many very practical use cases. Most notably, they are currently unable to constrain values of complex-type and collection-type values, thus diminishing the overall value of the TOSCA typing system.

Another limitation of the current TOSCA constraint grammar is that it does not go beyond a very limited list of basic assertions, practically excluding the possibility to combine these assertions into a Boolean expression. The only exception of this rule is that a list of assertions can implicitly work as an AND. Besides this “accidentally” provided AND, there is absolutely no way to use in a constraint clause any other known Boolean function, like OR or even NOT.

This document presents a suite of proposals with the objective to overcome these limitations. The proposed changes will significantly improve the TOSCA ability to specify advanced constraints and to apply these constraints to values of practically any types.

Other facts standing to the proposals’ credit:

* They leverage already existing TOSCA syntax constructs rather than inventing new ones
* They preserve backward compatibility with the legacy TOSCA grammar. None of the existing constructs are going to be re-defined or deprecated in result of the proposed changes
* They promote a more coherent syntax for constraints and conditions across various TOSCA grammar clauses, including type definitions, node filters, workflows and policies

It is also worth mentioning that the suite comprises several individual proposals. They are all related and will achieve the maximum effect when accepted together. However, each of them is in fact independent, brings its own value and can therefore be discussed and accepted (or rejected) separately of others.

The figure below is an attempt to visualize the impact of the proposed changes to the TOSCA grammar.



# Proposal #1: Allow the Boolean negation operator NOT in the condition clauses

The current TOSCA condition definition grammar (3.6.23 Condition clause definition) recognizes the keynames *and*, *or*, and *assert*. I propose to extend this list with the *not* keyword that would stand for the Boolean negation operator. Adding this clause will make the TOSCA Condition definition [functionally complete](https://en.wikipedia.org/wiki/Functional_completeness).

A *not* clause should have one nested condition clause: an *and*, an *or*, an *assert*, or even another *not*.

## Examples

The following condition yields TRUE when the attribute *my\_attribute1* takes any value other than *value1*:

|  |
| --- |
| condition: - not: - assert: - my\_attribute1: [{equal: value1}]} |

If we allow the ‘assert’ keyname to be optional, then a more readable equivalent would look as follows:

|  |
| --- |
| condition: - not: - my\_attribute1: [{equal: value1}]} |

The following condition yields TRUE when none of the attributes *my\_attribute1* and *my\_attribute2* is equal to *value1*.

|  |
| --- |
| condition: - not: - and: - assert: - my\_attribute1: [{equal: value1}]} - assert: - my\_attribute2: [{equal: value1}]} |

Assuming again that the ‘assert’ keyname is optional, and that a list of clauses is implicitly ‘and’-ed, this could get simplified as follows:

|  |
| --- |
| condition: - not: - my\_attribute1: [{equal: value1}]} - my\_attribute2: [{equal: value1}]} |

The following condition is a functional equivalent of the previous example:

|  |
| --- |
| condition: - or: - not: - assert: - my\_attribute1: [{equal: value1}]} - not: - assert: - my\_attribute2: [{equal: value1}]} |

Or:

|  |
| --- |
| condition: - or: - not: - my\_attribute1: [{equal: value1}]} - not: - my\_attribute2: [{equal: value1}]} |

TODO: translate this description into an exact list of changes to the TOSCA document

# Proposal #2: Add Boolean condition operators to the constraint clause

The list of operators which are currently supported by the TOSCA Constraint clauses (3.6.3) should be extended by adding a new operator, *condition*.

This operator specifies a complex Boolean expression that should yield TRUE when applied to a value under the constraint.

As a shorthand notation for the modified constraint clause, it should be allowed to omit the *condition* keyname in the context of a constraint clause and use the condition operators *and*, *or*, and *not* as directly included keynames.

Note: Given the efficiency of the shorthand notation, the “longhand” variant with its explicit *condition* operator may seem redundant. However, allowing this more verbose syntax will establish a certain degree of symmetry between the updated Constraint clause (3.6.3) and the existing Condition clause definition (3.6.23) with its omittable operator *assert*. The longhand notation also provides a better ground for future extensions.

## Examples

In the example below, we create a new string-based data type. In order to be valid, a value of this type should belong to either of the two disjoint ranges AND not be longer than 7 characters.

|  |
| --- |
| data\_types: MyType: derived\_from: string constraints: - condition: - or: - and: - greater\_than: aaa - less\_than: ccc - and: - greater\_than: kkk - less\_than: mmm - max\_length: 7 |

Same type as above, defined using the shorthand notation:

|  |
| --- |
| data\_types: MyType: derived\_from: string constraints: - or: - and: - greater\_than: aaa - less\_than: ccc - and: - greater\_than: kkk - less\_than: mmm - max\_length: 7 |

# Proposal #3: Enable expressions to target sub-properties and list items

The TOSCA grammar in its current form allows only a whole property/attribute/input value as an argument for the Boolean expressions (i.e., conditions and assertions), even when this value is of a complex or collection type. That is, a TOSCA expression cannot address a nested sub-property or an individual list item.

I propose to eliminate this limitation by allowing a YAML list to be used for the <property\_name>/<attribute\_name> part of all TOSCA syntax constructs that specify a Boolean expression to be applied to a target property/attribute, such as:

* property filter definition (3.6.4),
* policy trigger definition (3.6.20),
* workflow precondition definition (3.6.24),
* workflow step definition (3.6.25)

Such a list should be interpreted as *a path* to the target sub-property. Each item in the path specifies the name of a nested sub-property (for complex properties) or the index of an item (for list properties).

Please note that TOSCA already uses a similar *path* syntax in substitution mappings (3.8.12) and functions like *get\_input* (4.4.1), *get\_property* (4.4.2), and *get\_attribute* (4.5.1).

## Example

SOL001 defines a rich set of complex types, including the selected types below:

|  |
| --- |
| capability\_types: tosca.capabilities.nfv.VirtualCompute: derived\_from: tosca.capabilities.Root properties: logical\_node: type: map entry\_schema: type: tosca.datatypes.nfv.LogicalNodeData required: false requested\_additional\_capabilities: type: map entry\_schema: type: tosca.datatypes.nfv.RequestedAdditionalCapability required: false virtual\_memory: type: tosca.datatypes.nfv.VirtualMemory required: true virtual\_cpu: type: tosca.datatypes.nfv.VirtualCpu required: true virtual\_local\_storage: type: list entry\_schema: description: virtual system disk definition type: tosca.datatypes.nfv.VirtualStorageData required:FFSdata\_types: tosca.datatypes.nfv.VirtualCpu: derived\_from: tosca.datatypes.Root properties: cpu\_architecture: type: string required: false num\_virtual\_cpu: type: integer required: true virtual\_cpu\_clock: type: scalar-unit.frequency required: false virtual\_cpu\_oversubscription\_policy: type: string required: false vdu\_cpu\_requirements type: map entry\_schema: type: string required: false virtual\_cpu\_pinning: type: tosca.datatypes.nfv.VirtualCpuPinning required: false tosca.datatypes.nfv.VirtualCpuPinning: derived\_from: tosca.datatypes.Root properties: cpu\_pinning\_policy: type: string # CpuPinningPolicy constraints: - valid\_values: [ static, dynamic ] required: false cpu\_pinning\_rule: type: list entry\_schema: type: string required: false |

Unfortunately, the complex nature of these types makes them practically useless under the current TOSCA grammar’s limitations.

The proposed enhancements would bring these types back to life. Below is an example of a node filter that is only looking for a CPU with static pinning while disregarding all other CPU characteristics:

|  |
| --- |
| Node\_templates: function\_01: type: MyFunction requirements: - compute: node\_filter: capabilities: - tosca.capabilities.nfv.VirtualCompute: properties: [virtual\_cpu, virtual\_cpu\_pinning, cpu\_pinning\_policy]:  - equal: static |

TODO: for better consistency of the TOSCA Specs document, formally introduce the Path clause, refer to it across the TOSCA doc (not only for condition targets of all kinds, but also for other areas – mappings, functions, etc).

# Proposal #X: Future development

The proposals described in this document will make TOSCA a much more powerful and precise tool for handling data values. However, even with all the proposed changes included into its grammar, TOSCA will still be lacking many useful data selection abilities.

For example, the proposed changes do not provide a solution for following problems:

* matching a list item when its index is unknown
* matching a map entry by key only, ignoring its value part
* matching a map entry by a part of its value
* defining custom constraints with implementation in an external language
* etc.

These and other limitations will be addressed by the next series of proposals.