**Notifications Proposal**

This is a revised version of the EventInterface2018\_06\_20.docx proposal, NotificationInterface2018\_10\_02.docx, and NotificationInterface2018\_10\_15.docx

**Introduction to notifications**

TOSCA allows service template designers to define operation interfaces that allows the execution of lifecycle management operations using external artefacts. While executing these operations within workflows, attributes of the different nodes might be changed. Nevertheless, there are many situations when the information modeled by an attribute changes as a consequence of external events (load changes, failures, mode changes) and not as a consequence of workflow execution.

As of today, there is no way by which such information can be communicated to the orchestrator (i.e. for updating the instance model) in a TOSCA-defined formal way. Moreover, while the concept of an event is used in policies to trigger executions (of either single operations or entire workflows, see section 3.6.18) the language does not define yet how such events come to be. This leaves implementations to employ ad-hoc solutions to change attribute value and/or generate trigger events that are not standardized or even visible to the template creator.

To this end we propose the definition of “notifications” in TOSCA. They allow service template designers to specify how to receive external events and/or attribute changes in a way that is asynchronous to the lifecycle management workflows and operations. Even more, these *notifications* can be used to define a standardized way of asynchronous interaction between the TOSCA orchestrator and the outside world (that implements the managed objects).

The major difference between *notifications* and *operations* is that the former are called from the outside world to on the orchestrator, and not the other way around. For ease of use, we propose a format similar to operations, however no *inputs* are defined (as it is called asynchronously from the outside), information is pushed to the orchestrator via *outputs* (similarly to operation outputs). In addition, we can associate different *event\_types* with *notifications*, that is these event types are generated when the *notification* is called.

We recommend that TOSCA should be extended as follows:

* Allow service template designers to define *notifications* associated with nodes or relationships.
* Each *interface* may expose several *notifications* that can be called by the outside world on the orchestrator.
* Within an *interface*, to differentiate *operations* from notifications we should gather the *operations* under the “operations” keyname and *notifications* under the “notifications” keyname. For backward compatibility if an *interface* does not specify the *operations* or *notifications* keyname the definitions should be regarded as *operations*, but this usage should be deprecated.
* Each reception of a *notification* also generates a homonymous event type that can be used to set off triggers as specified in policies.
* Each *notification* can have several *outputs* specified, these are a way to introduce information from the outside world, Similar to the *outputs* of *operations*, the *output* values of notifications are mapped into *attributes* in the scope of SELF for *nodes* or SELF, SOURCE or TARGET for *relationships*.

**Interface definition**

Within an interface we can either define operations or notifications. Thus, we change the grammar to gather the *operations* under the operations keyname and the *notifications* under the notifications keyname. For back compatibility if neither the operations or notifications are specified then we assume the symbolic names in the interface definition to mean operations, but this use is deprecated. The rest stays the same. Operations and notifications names should not overlap.

**Interface grammar**

Interface definitions have the following grammar:

|  |
| --- |
| <interface\_definition\_name>    type: <interface\_type\_name>  inputs:  <property\_definitions or property\_assignments>  operations:  <operation\_definitions>  notifications:  <notification\_definitions> |

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

* **interface\_definition\_name**: represents the required symbolic name of the interface as a string.
* **interface\_type\_name**: represents the required name of the interface type for the interface definition.
* **property\_definitions**: represents the optional list of property definitions (i.e., parameters) which the TOSCA orchestrator would make available (i.e. will pass) to all defined operations. This means these properties and their values would be accessible to the implementation artifacts (e.g., scripts) of any interface operation during their execution. The property definitions should be used within Node or Relationship type definitions, including as part of a Requirement definition in a Node Type.
* **property\_assignments**: represents the optional list of property assignments that should be used for passing parameters to operations within Node or Relationship Template providing values for properties defined in their respective type definitions.
* **operation\_definitions**: represents the required name of one or more operation definitions.
* **notification\_definitions**: represents the required name of one or more notification definitions.

**Notification definition**

A notification definition defines a named notification that can be associated with an interface. The notification is a way for an external event to be transmitted to the TOSCA orchestrator. Parameter values can be sent together with a notification and we can map them to node/relationship attributes similarly to operation outputs. The artifact that the orchestrator is registering with in order to receive the notification is specified using the “implementation” keyname in a similar way to operations.

When the notification is received an event is generated within the orchestrator that can be associated to triggers in policies to call other internal operations and workflows. The notification name (the unqualified full name) itself identifies the event type that is generated and can be textually used when defining the associated triggers.

**Notification definition keynames**

|  |  |  |  |
| --- | --- | --- | --- |
| **Keyname** | **Required** | **Type** | **Description** |
| description | no | description | The optional description string for the associated named notification. |
| implementation | no | notification implementation definition | The optional definition of the notification implementation. |
| outputs | no | list of attribute mappings | The optional list of property mappings that specify named notification output values and their mappings onto attributes of the node or relationship that contains the interface within which the notification is defined. |

**Grammar**

The following multi-line grammar may be used in Node or Relationship Template or Type definitions:

|  |
| --- |
| <notification\_name>:     description: <notification\_description>  implementation: <notification\_implementation\_definition>     outputs:  <attribute\_mappings> |

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

* notification\_name: represents the required symbolic name of the notification as a string.
* notification\_description: represents the optional description string for the corresponding notification\_name.
* notification\_implementation\_definition: representes the optional specification of the notification implementation (i.e. the external artifact that is may send notifications)
* attribute\_mappings: represents the optional list of attribute assignments for mapping the outputs values to the respective attributes of the node or relationship.

**Notification implementation definition**

Notification implementations definition specifies one or more artifacts to be used by the orchestrator to subscribe for that particular notification. We use the *primary* and *dependencies* keynames as in the operation implementation definition. The *operation\_host* and *timeout* are not used as they make no sense for notifications.

**Discussions:**

**Denomination of notification:**

After a discussion on email, we decided to use the name “notification” for this construct, and have it defined as part of an interface. Thus, an interface can define a mix of zero or more operations and zero or more notifications.

**Identifying the right notification:**

We are specifying an implementation for the notification, that is, using this information the orchestrator is registering to an external artifact that will send the notifications. Note that there are several ways to implement this (out of scope of this definition):

* the template designer may specify different implementation artifacts for each template node
* the template designer may specify the same implementation artifact for similar nodes, and then relies on the orchestrator to communicate the node name and the notification name to the external artifact when registering (during template parsing/ instance model creation).

**Examples:**

The following example shows how an interface can be set up to receive an external notification, and then set up a policy that triggers on it.

**Preliminaries: already existing event-condition-action specification in TOSCA:**

There are already two constructs in TOSCA to deal with event-condition-action patterns: policies and triggers. As part of a policy definition, we can combine the existence of one or more triggers with a policy target group (i.e. a set of nodes or relationships). Each trigger further defines a combination of an event\_type with a certain action (the action can be used either to invoke a workflow or a specific operation). The way I interpret the standard, is that events of the event\_type defined in the trigger and which in addition are associated with the entities in the target group trigger the specified actions.

What has not been specified before is how these events are generated. Here, this proposes that one way to generate such an event is when receiving a notification. Thus, besides updating the attributes associated with the notification outputs, the reception of a notification also generates a homonymous event type that can be used to set off triggers as specified in policies.

Now, getting back to the policy, the following semantics is assumed:

If

* a trigger triggers on the specific event type, and
* the node on which the notification has been called is in the target list

the associated action will be performed.

**Notification example:**

In the case below the notifications are defined in the *org.ego.interfaces.StayingAlive* interface, the policy is *recover\_after\_crash\_policy,* the action is to call the *failure\_recovery\_workflow*, the event it will react on is the *org.ego.interfaces.StayingAlive.failure\_report* notification.

**Templates:**

Note: only the highlighted text uses the proposed extensions.

tosca\_definitions\_version: tosca\_simple\_yaml\_1\_3\_x

description: Example template of notification usage

interface\_types:

org.ego.interfaces.Upgrade:

derived\_from: tosca.interfaces.Root

description: A simple interface

operations:

upgrade:

implementation: upgrade\_artifact

inputs:

name:

type: string

mode:

type: string

outputs:

status:

type: string

cancel\_upgrade:

implementation: cancel\_upgrade\_artifact

outputs:

message:

type: string

upgrade\_cleanup:

implementation: upgrade\_cleanup\_artifact

org.ego.interfaces.StayingAlive:

derived\_from: tosca.interfaces.Root

description: >

A simple interface to receive heartbeat, fault and failure events

(only notifications)

notifications:

heartbeat:

implementation: staying\_alive\_artifact

outputs:

tick:

type: boolean

failure\_report:

implementation: staying\_alive\_artifact

outputs:

level:

type: integer

time:

type: timestamp

environment:

type: string

topology\_template:

node\_templates:

db\_1:

type: org.ego.nodes.Database

interfaces:

org.ego.interfaces.Upgrade:

...

org.ego.interfaces.StayingAlive:

notifications:

heartbeat:

outputs:

tick: [ SELF, still\_alive ]

failure\_report:

outputs:

level: [SELF, failure\_level]

time: [SELF, failure\_time]

environment: [SELF, failure\_context]

...

policies:

- recover\_policy:

type: org.ego.policies.trigger.RecoveryPolicy

description: >

A trigger type policy.

Kicks in if either the database or app\_logic nodes fail.

Triggers the execution of the failure\_recovery workflow.

targets: [ db\_1, app\_logic\_1 ]

triggers:

- mon\_fail\_trigger:

event\_type: org.ego.interfaces.StayingAlive.failure\_report

action: failure\_recovery\_workflow

...

workflows:

failure\_recovery\_workflow:

description: Recover the end-to-end service after failure

steps:

stepA:

target:

db\_1

filter:

- assert:

- failure\_level: [{greater\_than: 0}]

activities:

- call\_operation: Standard.create

on\_success:

- stepB

- stepC

stepB:

target:

app\_logic\_1

filter:

- assert:

- failure\_level: [{greater\_than: 0}]

activities:

- call\_operation: Standard.create

on\_success:

- stepC

stepC:

target:

app\_logic\_1

activities:

- call\_operation: Standard.configure