OOF-HAS Homing Specification Guide

Homing Specification Guide

Updated: 27 Mar 2018

This document describes the Homing Template format, used by the Homing service. It is a work in progress and subject to frequent revision.

Template Structure

Homing templates are defined in YAML and follow the structure outlined below.

```
homing_template_version: 2018-02-01
parameters:
  PARAMETER_DICT
locations:
  LOCATION_DICT
demands:
  DEMAND_DICT
constraints:
  CONSTRAINT_DICT
reservations:
  RESERVATION_DICT
optimization:
  OPTIMIZATION
```

- `homing_template_version`: This key with value 2016-11-01 (or a later date) indicates that the YAML document is a Homing template of the specified version.
- `parameters`: This section allows for specifying input parameters that have to be provided when instantiating the homing template. Typically, this section is used for providing runtime parameters (like SLA thresholds), which in turn is used in the existing homing policies. The section is optional and can be omitted when no input is required.
- `locations`: This section contains the declaration of geographic locations. This section is optional and can be omitted when no input is required.
- `demands`: This section contains the declaration of demands. This section with at least one demand should be defined in any Homing template, or the template would not really do anything when being instantiated.
- `constraints`: This section contains the declaration of constraints. The section is optional and can be omitted when no input is required.
- `reservations`: This section contains the declaration of required reservations. This section is optional and can be omitted when reservations are not required.
- `optimization`: This section allows the declaration of an optimization. This section is optional and can be omitted when no input is required.

Homing Template Version

The value of `homing_template_version` tells HAS not only the format of the template but also features that will be validated and supported. The following values are supported: "2016-11-01" or "2018-02-01" in the initial release of HAS.

```
homing_template_version: 2018-02-01
```

Parameters

The **parameters** section allows for specifying input parameters that have to be provided when instantiating the template. Such parameters are typically used for providing runtime inputs (like SLA thresholds), which in turn is used in the existing homing policies. This also helps build reusable homing constraints where these parameters can be embedded design time, and it corresponding values can be supplied during runtime.

Each parameter is specified with the name followed by its value. Values can be strings, lists, or dictionaries.

Example

In this example, `provider_name` is a string and `service_info` is a dictionary containing both a string and a list (keyed by `base_url` and `nod_config`, respectively).

```
parameters:
  provider_name: multicloud
  service_info:
    base_url: http://serviceprovider.sdngc.com/
    nod_config:
    - http://nod/config_a.yaml
    - http://nod/config_b.yaml
    - http://nod/config_c.yaml
    - http://nod/config_d.yaml
```

A parameter can be referenced in place of any value. See the **Intrinsic Functions** section for more details.

Locations

One or more **locations** may be declared. A location may be referenced by one or more `constraints`. Locations may be defined in any of the following ways:

Coordinate

A geographic coordinate expressed as a latitude and longitude.

Key	Value
latitude	Latitude of the location.
longitude	Longitude of the location.

| Key | Value | |----------|----------| | `latitude` | Latitude of the location. | | `longitude` | Longitude of the location. |

Host Name

An opaque host name that can be translated to a coordinate via an inventory provider (e.g., A&AI).

Key	Value
host_name	Host name identifying a location.

| Key | Value | |----------|----------| | `host_name` | Host name identifying a location. |

CLLI

Common Language Location Identification (CLLI) code(https://en.wikipedia.org/wiki/CLLI_code).

Key	Value
clli_code	8 character CLLI.

| Key | Value | |----------|----------| | `clli_code` | 8 character CLLI. |

Questions

- Do we need functions that can convert one of these to the other? E.g., CLLI Codes to a latitude/longitude

Placemark

An address expressed in geographic region-agnostic terms (referred to as a *placemark*).

Support for this schema is deferred.

Key	Value
iso_country_code	The abbreviated country name associated with the placemark.

postal_code	The postal code associated with the placemark.
administrative_area	The state or province associated with the placemark.
sub_administrative_area	Additional administrative area information for the placemark.
locality	The city associated with the placemark.
sub_locality	Additional city-level information for the placemark.
thoroughfare	The street address associated with the placemark.
sub_thoroughfare	Additional street-level information for the placemark.

Key	Value
`iso_country_code`	The abbreviated country name associated with the placemark.
`postal_code`	The postal code associated with the placemark.
`administrative_area`	The state or province associated with the placemark.
`sub_administrative_area`	Additional administrative area information for the placemark.
`locality`	The city associated with the placemark.
`sub_locality`	Additional city-level information for the placemark.
`thoroughfare`	The street address associated with the placemark.
`sub_thoroughfare`	Additional street-level information for the placemark.

Questions

- What geocoder can we use to convert placemarks to a latitude/longitude?

Examples

The following examples illustrate a location expressed in coordinate, host_name, CLLI, and placemark, respectively.

```
locations:
  location_using_coordinates:
    latitude: 32.897480
    longitude: -97.040443

  host_location_using_host_name:
    host_name: USESTCDLLSTX55ANZ123

  location_using_clli:
    clli_code: DLLSTX55

  location_using_placemark:
    sub_thoroughfare: 1
    thoroughfare: ATT Way
    locality: Bedminster
    administrative_area: NJ
    postal_code: 07921-2694
```

Demands

A **demand** can be satisfied by using candidates drawn from inventories. Each demand is uniquely named. Inventory is considered to be opaque and can represent anything from which candidates can be drawn.

A demand's resource requirements are determined by asking an **inventory provider** for one or more sets of **inventory candidates** against which the demand will be made. An explicit set of candidates may also be declared, for example, if the only candidates for a demand are predetermined.

Demand criteria is dependent upon the inventory provider in use.

Provider-agnostic Schema

Key	Value
inventory_provider	A HAS-supported inventory provider.
inventory_type	The reserved word `cloud` (for cloud regions) or the reserved word `service` (for existing service instances). Exactly one inventory type may be specified.
`attributes` (Optional)	A list of key-value pairs, that is used to select inventory candidates that match *all* the specified attributes. The key should be a uniquely identifiable attribute at the inventory provider.
`service_type` (Optional)	If `inventory_type` is `service`, a list of one or more provider-defined service types. If only one service type is specified, it may appear without list markers (`[]`).
`service_id` (Optional)	If `inventory_type` is `service`, a list of one or more provider-defined service ids. If only one service id is specified, it may appear without list markers (`[]`).

`default_cost` (Optional)	The default cost of an inventory candidate, expressed as currency. This must be specified if the inventory provider may not always return a cost.
`required_candidates` (Optional)	A list of one or more candidates from which a solution will be explored. Must be a valid candidate as described in the **candidate schema**.
`excluded_candidates` (Optional)	A list of one or more candidates that should be excluded from the search space. Must be a valid candidate as described in the **candidate schema**.
`existing_placement` (Optional)	The current placement for the demand. Must be a valid candidate as described in the **candidate schema**.

| Key | Value | |----------|----------| | `inventory_provider` | A HAS-supported inventory provider. | | `inventory_type` | The reserved word `cloud` (for cloud regions) or the reserved word `service` (for existing service instances). Exactly one inventory type may be specified. | | `attributes` (Optional) | A list of key-value pairs, that is used to select inventory candidates that match *all* the specified attributes. The key should be a uniquely identifiable attribute at the inventory provider. | | `service_type` (Optional) | If `inventory_type` is `service`, a list of one or more provider-defined service types. If only one service type is specified, it may appear without list markers (`[]`). | | `service_id` (Optional) | If `inventory_type` is `service`, a list of one or more provider-defined service ids. If only one service id is specified, it may appear without list markers (`[]`). | | `default_cost` (Optional) | The default cost of an inventory candidate, expressed as currency. This must be specified if the inventory provider may not always return a cost. | | `required_candidates` (Optional) | A list of one or more candidates from which a solution will be explored. Must be a valid candidate as described in the **candidate schema**. | | `excluded_candidates` (Optional) | A list of one or more candidates that should be excluded from the search space. Must be a valid candidate as described in the **candidate schema**. | | `existing_placement` (Optional) | The current placement for the demand. Must be a valid candidate as described in the **candidate schema**. |

Examples

The following example helps understand a demand specification using Active & Available Inventory (A&AI), the inventory provider-of-record for ONAP.

Inventory Provider Criteria

Key	Value
`inventory_provider`	Examples: `aai`, `multicloud`.
`inventory_type`	The reserved word `cloud` (for new inventory) or the reserved word `service` (for existing inventory). Exactly one inventory type may be specified.
`attributes` (Optional)	A list of key-value pairs to match against inventory when drawing candidates.
`service_type` (Optional)	Examples may include `vG`, `vG_MuxInfra`, etc.
`service_id` (Optional)	Must be a valid service id. Examples may include `vCPE`, `VoLTE`, etc.
`default_cost` (Optional)	The default cost of an inventory candidate, expressed as a unitless number.
`required_candidates` (Optional)	A list of one or more valid candidates. See **Candidate Schema** for details.
`excluded_candidates` (Optional)	A list of one or more valid candidates. See **Candidate Schema** for details.
`existing_placement` (Optional)	A single valid candidate, representing the current placement for the demand. See **candidate schema** for details.

| Key | Value | |----------|----------| | `inventory_provider` | Examples: `aai`, `multicloud`. | | `inventory_type` | The reserved word `cloud` (for new inventory) or the reserved word `service` (for existing inventory). Exactly one inventory type may be specified. | | `attributes` (Optional) | A list of key-value pairs to match against inventory when drawing candidates. | | `service_type` (Optional) | Examples may include `vG`, `vG_MuxInfra`, etc. | | `service_id` (Optional) | Must be a valid service id. Examples may include `vCPE`, `VoLTE`, etc. | | `default_cost` (Optional) | The default cost of an inventory candidate, expressed as a unitless number. | | `required_candidates` (Optional) | A list of one or more valid candidates. See **Candidate Schema** for details. | | `excluded_candidates` (Optional) | A list of one or more valid candidates. See **Candidate Schema** for details. | | `existing_placement` (Optional) | A single valid candidate, representing the current placement for the demand. See **candidate schema** for details. |

Candidate Schema

The following is the schema for a valid `candidate`: *`candidate_id` uniquely identifies a candidate. Currently, it is either a Service Instance ID or Cloud Region ID.* `candidate_type` identifies the type of the candidate. Currently, it is either `cloud` or `service`. *`inventory_type` is defined as described in* *I nventory Provider Criteria `inventory_provider` identifies the inventory from which the candidate was drawn. *`host_id` is an ID of a specific host (used only when referring to service/existing inventory).* `cost` is expressed as a unitless number. *`location_id` is always a location ID of the specified location type (e.g., for a type of `cloud` this will be an Cloud Region ID).* `location_type` is an inventory provider supported location type. *`latitude` is a valid latitude corresponding to the* location*id. `longitude` is a valid longitude corresponding to the *location*id.* `city` (Optional) city corresponding to the *location _id*. *`state` (Optional) state corresponding to the* location*id. `country` (Optional) country corresponding to the *location*id.* `region` (Optional) geographic region corresponding to the *location_id*. *`complex_name` (Optional) Name of the complex corresponding to the* location*id. ``cloud*owner`(Optional) refers to the *cloud owner* (e.g.,*azure,aws,att, etc.). *cloud*region*version(Optional) is an inventory provider supported version of the cloud region. *physical*location*id`` (Optional) is an inventory provider supported CLLI code corresponding to the cloud region.

Examples

service candidate

```
{
        "candidate_id": "1ac71fb8-ad43-4e16-9459-c3f372b8236d",
        "candidate_type": "service",
        "inventory_type": "service",
        "inventory_provider": "aai",
        "host_id": "vnf_123456",
        "cost": "100",
        "location_id": "DLLSTX55",
        "location_type": "azure",
        "latitude": "32.897480",
        "longitude": "-97.040443",
        "city": "Dallas",
        "state": "TX",
        "country": "USA",
        "region": "US",
        "complex_name": "dalls_one",
        "cloud_owner": "att-aic",
        "cloud_region_version": "1.1",
        "physical_location_id": "DLLSTX55",
}
```

cloud candidate

```
{
      "candidate_id": "NYCNY55",
      "candidate_type": "cloud",
      "inventory_type": "cloud",
      "inventory_provider": "aai",
      "cost": "100",
      "location_id": "NYCNY55",
      "location_type": "azure",
      "latitude": "40.7128",
      "longitude": "-74.0060",
      "city": "New York",
      "state": "NY",
      "country": "USA",
      "region": "US",
      "complex_name": "ny_one",
      "cloud_owner": "att-aic",
      "cloud_region_version": "1.1",
      "physical_location_id": "NYCNY55"
}
```

Questions * Currently, candidates are either service instances or cloud regions. As new services are on-boarded, this can be evolved to represent different types of resources.

Examples

The following examples illustrate two demands:

- `vGMuxInfra`: A vGMuxInfra service, drawing candidates of type *service* from the inventory. Only candidates that match the customer_id and orchestration-status will be included in the search space.
- `vG`: A vG, drawing candidates of type *service* and *cloud* from the inventory. Only candidates that match the customer_id and provisioning-status will be included in the search space.

```
demands:
  vGMuxInfra:
  - inventory_provider: aai
    inventory_type: service
    service_type: vG_Mux
    attributes:
      customer-id: some_company
      orchestration-status: Activated
  vG:
  - inventory_provider: aai
    inventory_type: service
    service_type: vG
    attributes:
      customer-id: some_company
      provisioning-status: provisioned
  - inventory_provider: aai
    inventory_type: cloud
```

Questions * Do we need to support cost as a function ?

Constraints

A **Constraint** is used to *eliminate* inventory candidates from one or more demands that do not meet the requirements specified by the constraint. Since reusability is one of the cornerstones of HAS, Constraints are designed to be service-agnostic, and is parameterized such that it can be reused across a wide range of services. Further, HAS is designed with a plug-in architecture that facilitates easy addition of new constraint types.

Constraints are denoted by a `constraints` key. Each constraint is uniquely named and set to a dictionary containing a constraint type, a list of demands to apply the constraint to, and a dictionary of constraint properties.

Considerations while using multiple constraints *Constraints should be treated as a unordered list, and no assumptions should be made as regards to the order in which the constraints are evaluated for any given demand.* All constraints are effectively AND-ed together. Constructs such as "Constraint X OR Y" are unsupported. * Constraints are reducing in nature, and does not increase the available candidates at any point during the constraint evaluations.

Schema

Key	Value
`CONSTRAINT_NAME`	Key is a unique name.
`type`	The type of constraint. See **Constraint Types** for a list of currently supported values.
`demands`	One or more previously declared demands. If only one demand is specified, it may appear without list markers (`[]`).
`properties` (Optional)	Properties particular to the specified constraint type. Use if required by the constraint.

| Key | Value | |---|---| | `CONSTRAINT_NAME` | Key is a unique name. | | `type` | The type of constraint. See **Constraint Types** for a list of currently supported values. | | `demands` | One or more previously declared demands. If only one demand is specified, it may appear without list markers (`[]`). | | `properties` (Optional) | Properties particular to the specified constraint type. Use if required by the constraint. |

```
constraints:
  CONSTRAINT_NAME_1:
    type: CONSTRAINT_TYPE
    demands: DEMAND_NAME | [DEMAND_NAME_1, DEMAND_NAME_2, ...]
    properties: PROPERTY_DICT

  CONSTRAINT_NAME_2:
    type: CONSTRAINT_TYPE
    demands: DEMAND_NAME | [DEMAND_NAME_1, DEMAND_NAME_2, ...]
    properties: PROPERTY_DICT

  ...
```

Constraint Types

Type	Description
`attribute`	Constraint that matches the specified list of Attributes.
`distance_between_demands`	Geographic distance constraint between each pair of a list of demands.

`distance_to_location`	Geographic distance constraint between each of a list of demands and a specific location.
`instance_fit`	Constraint that ensures available capacity in an existing service instance for an incoming demand.
`inventory_group`	Constraint that enforces two or more demands are satisfied using candidates from a pre-established group in the inventory.
`region_fit`	Constraint that ensures available capacity in an existing cloud region for an incoming demand.
`zone`	Constraint that enforces co-location/diversity at the granularities of clouds/regions/availability-zones.
hpa	Constraint that recommends optimal flavor and cloud region based on required hardware platform capabilities for an incoming demand.
vim_fit	Constraint that ensures capacity check with available capacity of VIMs based on incoming request.
`license` (Deferred)	License availability constraint.
`network_between_demands` (Deferred)	Network constraint between each pair of a list of demands.
`network_to_location` (Deferred)	Network constraint between each of a list of demands and a specific location/address.

| Type | Description | |---|---| | `attribute` | Constraint that matches the specified list of Attributes. | | `distance_between_demands` | Geographic distance constraint between each pair of a list of demands. | | `distance_to_location` | Geographic distance constraint between each of a list of demands and a specific location. | | `instance_fit` | Constraint that ensures available capacity in an existing service instance for an incoming demand. | | `inventory_group` | Constraint that enforces two or more demands are satisfied using candidates from a pre-established group in the inventory. | | `region_fit` | Constraint that ensures available capacity in an existing cloud region for an incoming demand. | | `zone` | Constraint that enforces co-location/diversity at the granularities of clouds/regions/availability-zones. | | `hpa` | Constraint that recommends optimal flavor and cloud region based on required hardware platform capabilities for an incoming demand. | | vim_fit | Constraint that ensures capacity check with available capacity of VIMs based on incoming request. | | `license` (Deferred) | License availability constraint. | | `network_between_demands` (Deferred) | Network constraint between each pair of a list of demands. | | `network_to_location` (Deferred) | Network constraint between each of a list of demands and a specific location/address. |

Note: Constraint names marked "Deferred" *will not

Threshold Values

Constraint property values representing a threshold may be an integer or floating point number, optionally prefixed with a comparison operator: =, <, >, <=, or >=. The default is = and optionally suffixed with a unit.

Whitespace may appear between the comparison operator and value, and between the value and units. When a range values is specified (e.g., `10-20 km`), the comparison operator is omitted.

Each property is documented with a default unit. The following units are supported:

Unit	Values	Default
Currency	USD	USD
Time	`ms`, `sec`	`ms`
Distance	`km`, `mi`	`km`
Throughput	`Kbps`, `Mbps`, `Gbps`	`Mbps`

| Unit | Values | Default | |---|---|---| | Currency | USD | USD | | Time | `ms`, `sec` | `ms` | | Distance | `km`, `mi` | `km` | | Throughput | `Kbps`, `Mbps`, `Gbps` | `Mbps` |

Attribute

Constrain one or more demands by one or more attributes, expressed as properties. Attributes are mapped to the **inventory provider** specified properties, referenced by the demands. For example, properties could be hardware capabilities provided by the platform (flavor, CPU-Pinning, NUMA), features supported by the services, etc.

Schema

Property	Value

`evaluate`	Opaque dictionary of attribute name and value pairs. Values must be strings or numbers. Encoded and sent to the service provider via a plugin.

| Property | Value | |--------------|--| | `evaluate` | Opaque dictionary of attribute name and value pairs. Values must be strings or numbers. Encoded and sent to the service provider via a plugin. |

Note: Attribute values are not detected/parsed as thresholds by the Homing framework. Such interpretations and evaluations are inventory provider-specific and delegated to the corresponding plugin

```
constraints:
  sriov_nj:
    type: attribute
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      evaluate:
        cloud_version: 1.1
        flavor: SRIOV
        subdivision: US-TX
        vcpu_pinning: True
        numa_topology: numa_spanning
```

Proposal: Evaluation Operators

To assist in evaluating attributes, the following operators and notation are proposed:

Operator	Name	Operand
`eq`	`==`	Any object (string, number, list, dict)
`ne`	`!=`	
`lt`	<	A number (strings are converted to float)
`gt`	>	
`lte`	<=	
`gte`	>=	
`any`	Any	A list of objects (string, number, list, dict)
`all`	All	
`regex`	RegEx	A regular expression pattern

| Operator | Name | Operand | |--------------|-----------|--| | `eq` | == | Any object (string, number, list, dict) | | `ne` | `!=` | | | `lt` | < | A number (strings are converted to float) | | `gt` | > | | | `lte` | <= | | | `gte` | >= | | | `any` | Any | A list of objects (string, number, list, dict) | | `all` | `All` | | | `regex` | `RegEx` | A regular expression pattern |

Example usage:

```
constraints:
  sriov_nj:
    type: attribute
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      evaluate:
        cloud_version: {gt: 1.0}
        flavor: {regex: /^SRIOV$/i}
        subdivision: {any: [US-TX, US-NY, US-CA]}
```

Distance Between Demands

Constrain each pairwise combination of two or more demands by distance requirements.

Schema

Name	Value
`distance`	Distance between demands, measured by the geographic path.

| Name | Value | |----------|----------| | `distance` | Distance between demands, measured by the geographic path. |

The constraint is applied between each pairwise combination of demands. For this reason, at least two demands must be specified, implicitly or explicitly.

```
constraints:
  distance_vnf1_vnf2:
    type: distance_between_demands
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      distance: < 250 km
```

Distance To Location

Constrain one or more demands by distance requirements relative to a specific location.

Schema

Property	Value
`distance`	Distance between demands, measured by the geographic path.
`location`	A previously declared location.

| Property | Value | |----------|----------| | `distance` | Distance between demands, measured by the geographic path. | | `location` | A previously declared location. |

The constraint is applied between each demand and the referenced location, not across all pairwise combinations of Demands.

```
constraints:
  distance_vnf1_loc:
    type: distance_to_location
    demands: [my_vnf_demand, my_other_vnf_demand, another_vnf_demand]
    properties:
      distance: < 250 km
      location: LOCATION_ID
```

Instance Fit

Constrain each demand by its service requirements.

Requirements are sent as a request to a **service controller**. Service controllers are defined by plugins in Homing (e.g., `sdn-c`).

A service controller plugin knows how to communicate with a particular endpoint (via HTTP/REST, DMaaP, etc.), obtain necessary information, and make a decision. The endpoint and credentials can be configured through plugin settings.

Schema

Property	Description
`controller`	Name of a service controller.
`request`	Opaque dictionary of key/value pairs. Values must be strings or numbers. Encoded and sent to the service provider via a plugin.

| Property | Description | |----------|----------| | `controller` | Name of a service controller. | | `request` | Opaque dictionary of key /value pairs. Values must be strings or numbers. Encoded and sent to the service provider via a plugin. |

```
constraints:
  check_for_availability:
    type: instance_fit
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      controller: sdn-c
      request: REQUEST_DICT
```

Region Fit

Constrain each demand's inventory candidates based on inventory provider membership.

Requirements are sent as a request to a **service controller**. Service controllers are defined by plugins in Homing (e.g., `sdn-c`).

A service controller plugin knows how to communicate with a particular endpoint (via HTTP/REST, DMaaP, etc.), obtain necessary information, and make a decision. The endpoint and credentials can be configured through plugin settings.

Schema

Property	Description
`controller`	Name of a service controller.
`request`	Opaque dictionary of key/value pairs. Values must be strings or numbers. Encoded and sent to the service provider via a plugin.

| Property | Description | |----------|-------------| | `controller` | Name of a service controller. | | `request` | Opaque dictionary of key /value pairs. Values must be strings or numbers. Encoded and sent to the service provider via a plugin. |

```
constraints:
  check_for_membership:
    type: region_fit
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      controller: sdn-c
      request: REQUEST_DICT
```

Zone

Constrain two or more demands such that each is located in the same or different zone category.

Zone categories are inventory provider-defined, based on the demands being constrained.

Schema

Property	Description
`qualifier`	Zone qualifier. One of `same` or `different`.
`category`	Zone category. One of `disaster`, `region`, `complex`, `time`, or `maintenance`.

| Property | Value | |----------|-------------| | `qualifier` | Zone qualifier. One of `same` or `different`. | | `category` | Zone category. One of `disaster`, `region`, `complex`, `time`, or `maintenance`. |

For example, to place two demands in different disaster zones:

```
constraints:
  vnf_diversity:
    type: zone
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      qualifier: different
      category: disaster
```

Or, to place two demands in the same region:

```
constraints:
  vnf_affinity:
    type: zone
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      qualifier: same
      category: region
```

Notes

- These categories could be any of the following: `disaster_zone`, `region`, `complex`, `time_zone`, and `maintenance_zone`. Really, we are talking affinity/anti-affinity at the level of DCs, but these terms may cause confusion with affinity/anti-affinity in OpenStack.

HPA & Cloud Agnostic Intent

Constrain each demand's inventory candidates based on available Hardware platform capabilities (HPA) and also intent support. Note that currently HPA and the cloud agnostic contraints will use the same schema.

Requirements mapped to the **inventory provider** specified properties, referenced by the demands. For example, properties could be hardware capabilities provided by the platform through flavors or cloud-region eg:(CPU-Pinning, NUMA), features supported by the services, etc.

Schema

Property	Description
`evaluate`	List of id, type, directives and flavorProperties of each VM of the VNF demand.

| Property | Description | |----------|-------------| | evaluate| List of id, type, directives and flavorProperties of each VM of the VNF demand. |

```
constraints:
  hpa_constraint:
    type: hpa
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      evaluate:
        - [ List of {id: {vdu Name},
                     type: {type of VF}
                     directives: {DIRECTIVES LIST},
                     flavorProperties: HPACapability DICT} ]

HPACapability DICT :
  hpa-feature: basicCapabilities
  hpa-version: v1
  architecture: generic
  directives:
    - DIRECTIVES LIST
  hpa-feature-attributes:
    - HPAFEATUREATTRIBUTES LIST

DIRECTIVES LIST
  type: String
  attributes:
    - ATTRIBUTES LIST

ATTRIBUTES LIST
  attribute_name: String
  attribute_value: String

HPAFEATUREATTRIBUTES LIST
  hpa-attribute-key: String
  hpa-attribute-value: String
  operator: One of OPERATOR
  unit: String

OPERATOR : ['=', '<', '>', '<=', '>=', 'ALL']
```

VIM Fit

Constrain each demand's inventory candidates based on capacity check for available capacity of a list of VIMs

Requirements are sent as a request to a **vim controller**. vim controllers are defined by plugins in Homing (e.g., `multicloud`).

A vimcontroller plugin knows how to communicate with a particular endpoint (via HTTP/REST, DMaaP, etc.), obtain necessary information, and make a decision. The endpoint and credentials can be configured through plugin settings.

Schema

Property	Description
`controller`	Name of a vim controller. (e.g., `multicloud`)
`request`	Opaque dictionary of key/value pairs. Values must be strings or numbers. Encoded and sent to the vim controller via a plugin.

| Property | Description | |----------|-------------| | `controller` | Name of a vim controller. | | `request` | Opaque dictionary of key/value pairs. Values must be strings or numbers. Encoded and sent to the vim controller via a plugin. |

```
constraints:
  check_cloud_capacity:
    type: vim_fit
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      controller: multicloud
      request: REQUEST_DICT
```

Inventory Group

Constrain demands such that inventory items are grouped across two demands.

This constraint has no properties.

```
constraints:
  my_group:
    type: inventory_group
    demands: [demand_1, demand_2]
```

Note: Only pair-wise groups are supported at this time. If three or more demands are specified, only the first two will be used.

License

Constrain demands according to license availability.

Support for this constraint is deferred.

Schema

Property	Value
`id`	Unique license identifier.
`key`	Opaque license key, particular to the license identifier.

| Property | Value | |---|---| | `id` | Unique license identifier | | `key` | Opaque license key, particular to the license identifier |

```
constraints:
  my_software:
    type: license
    demands: [demand_1, demand_2, ...]
    properties:
      id: SOFTWARE_ID
      key: LICENSE_KEY
```

Network Between Demands

Constrain each pairwise combination of two or more demands by network requirements.

Support for this constraint is deferred.

Schema

Property	Value
`bandwidth` (Optional)	Desired network bandwidth.
`distance` (Optional)	Desired distance between demands, measured by the network path.
`latency` (Optional)	Desired network latency.

| Property | Value | |---|---| | `bandwidth` (Optional) | Desired network bandwidth. | | `distance` (Optional) | Desired distance between demands, measured by the network path. | | `latency` (Optional) | Desired network latency. |

Any combination of `bandwidth`, `distance`, or `latency` must be specified. If none of these properties are used, it is treated as a malformed request.

The constraint is applied between each pairwise combination of demands. For this reason, at least two demands must be specified, implicitly or explicitly.

```
constraints:
  network_requirements:
    type: network_between_demands
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      bandwidth: >= 1000 Mbps
      distance: < 250 km
      latency: < 50 ms
```

Network To Location

Constrain one or more demands by network requirements relative to a specific location.

Support for this constraint is deferred.

Schema

Property	Value
`bandwidth`	Desired network bandwidth.
`distance`	Desired distance between demands, measured by the network path.
`latency`	Desired network latency.
`location`	A previously declared location.

| Property | Value | |---|---| | `bandwidth` | Desired network bandwidth. | | `distance` | Desired distance between demands, measured by the network path. | | `latency` | Desired network latency. | | `location` | A previously declared location. |

Any combination of `bandwidth`, `distance`, or `latency` must be specified. If none of these properties are used, it is treated as a malformed request.

The constraint is applied between each demand and the referenced location, not across all pairwise combinations of Demands.

```
constraints:
  my_access_network_constraint:
    type: network_to_location
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      bandwidth: >= 1000 Mbps
      distance: < 250 km
      latency: < 50 ms
      location: LOCATION_ID
```

Capabilities

Constrain each demand by its cluster capability requirements. For example, as described by an OpenStack Heat template and operational environment.

Support for this constraint is deferred.

Schema

Property	Value
`specification`	Indicates the kind of specification being provided in the properties. Must be `heat`. Future values may include `tosca`, `Homing`, etc.
`template`	For specifications of type `heat`, a single stack in OpenStack Heat Orchestration Template (HOT) format. Stacks may be expressed as a URI reference or a string of well-formed YAML/JSON. Templates are validated by the Heat service configured for use by HAS. Nested stack references are unsupported.
`environment` (Optional)	For specifications of type `heat`, an optional Heat environment. Environments may be expressed as a URI reference or a string of well-formed YAML/JSON. Environments are validated by the Heat service configured for use by Homing.

| Property | Value | |---|---| | `specification` | Indicates the kind of specification being provided in the properties. Must be `heat`. Future values may include `tosca`, `Homing`, etc. | | `template` | For specifications of type `heat`, a single stack in OpenStack Heat Orchestration Template (HOT) format. Stacks may be expressed as a URI reference or a string of well-formed YAML/JSON. Templates are validated by the Heat service configured for use by HAS. Nested stack references are unsupported. | | `environment` (Optional) | For specifications of type `heat`, an optional Heat environment. Environments may be expressed as a URI reference or a string of well-formed YAML/JSON. Environments are validated by the Heat service configured for use by Homing. |

```
constraints:
  check_for_fit:
    type: capability
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      specification: heat
      template: http://repository/my/stack_template
      environment: http://repository/my/stack_environment
```

Reservations

A **Reservation** allows reservation of resources associated with candidate that satisfies one or more demands.

Similar to the *instance_fit* constraint, requirements are sent as a request to a **service controller** that handles the reservation. Service controllers are defined by plugins in Homing (e.g., `sdn-c`).

The service controller plugin knows how to make a reservation (and initiate rollback on a failure) with a particular endpoint (via HTTP/REST, DMaaP, etc.) of the service controller. The endpoint and credentials can be configured through plugin settings.

Schema

Property	Description
`controller`	Name of a service controller.
`request`	Opaque dictionary of key/value pairs. Values must be strings or numbers. Encoded and sent to the service provider via a plugin.

| Property | Description | |----------|-------------| | `controller` | Name of a service controller. | | `request` | Opaque dictionary of key /value pairs. Values must be strings or numbers. Encoded and sent to the service provider via a plugin. |

```
resource_reservation:
  type: instance_reservation
  demands: [my_vnf_demand, my_other_vnf_demand]
  properties:
    controller: sdn-c
    request: REQUEST_DICT
```

Optimizations

An **Optimization** allows specification of a objective function, which aims to maximize or minimize a certain value that varies based on the choice of candidates for one or more demands that are a part of the objective function. For example, an objective function may be to find the *closest* cloud-region to a customer to home a demand.

Optimization Components

Optimization definitions can be broken down into three components:

Component	Key	Value
Goal	`minimize`	A single Operand (usually `sum`) or Function.
Operator	`sum`, `product`	Two or more Operands (Numbers, Operators, Functions)
Function	`distance_between`	A two-element list consisting of a location and demand.

| Component | Key | Value | |-----------|----------------------|---| | Goal | `minimize` | A single Operand (usually `sum`) or Function | | Operator | `sum`, `product` | Two or more Operands (Numbers, Operators, Functions) | | Function | `distance_between` | A two-element list consisting of a location and demand. |

Example

Given a customer location `cl`, two demands `vG1` and `vG2`, and weights `w1` and `w2`, the optimization criteria can be expressed as:

```
minimize(weight1 * distance_between(cl, vG1) + weight2 * distance_between(cl, vG2))
```

This can be read as: "Minimize the sum of weighted distances from cl to vG1 and from cl to vG2."

Such optimizations may be expressed in a template as follows:

```
parameters:
  w1: 10
  w2: 20

optimization:
  minimize:
    sum:
    - product:
      - {get_param: w1}
      - {distance_between: [cl, vG1]}
    - product:
      - {get_param: w2}
      - {distance_between: [cl, vG2]}
```

Or without the weights as:

```
optimization:
  minimize:
    sum:
    - {distance_between: [cl, vG1]}
    - {distance_between: [cl, vG2]}
```

Template Restriction

While the template format supports any number of arrangements of numbers, operators, and functions, HAS's solver presently expects a very specific arrangement.

Until further notice:

- Optimizations must conform to a single goal of `minimize` followed by a `sum` operator.
- The sum can consist of two `distance_between` function calls, or two `product` operators.
- If a `product` operator is present, it must contain at least a `distance_between` function call, plus one optional number to be used for weighting.
- Numbers may be referenced via `get_param`.
- The objective function has to be written in the sum-of-product format. In the future, HAS can convert product-of-sum into sum-of-product automatically.

The first two examples in this section illustrate both of these use cases.

Inline Operations

If desired, operations can be rewritten inline. For example, the two `product` operations from the previous example can also be expressed as:

```
parameters:
  w1: 10
  w2: 20

optimization:
  minimize:
    sum:
    - {product: [{get_param: w1}, {distance_between: [cl, vG1]}]}
    - {product: [{get_param: w2}, {distance_between: [cl, vG2]}]}
```

In turn, even the `sum` operation can be rewritten inline, however there is a point of diminishing returns in terms of readability!

Notes

- In the first version, we do not support more than one dimension in the optimization (e.g., Minimize distance and cost). For supporting multiple dimensions we would need a function the normalize the unit across dimensions.

Intrinsic Functions

Homing provides a set of intrinsic functions that can be used inside templates to perform specific tasks. The following section describes the role and syntax of the intrinsic functions.

Functions are written as a dictionary with one key/value pair. The key is the function name. The value is a list of arguments. If only one argument is provided, a string may be used instead.

```
a_property: {FUNCTION_NAME: [ARGUMENT_LIST]}

a_property: {FUNCTION_NAME: ARGUMENT_STRING}
```

Note: These functions can only be used within "properties" sections.

get_file

The `get_file` function inserts the content of a file into the template. It is generally used as a file inclusion mechanism for files containing templates from other services (e.g., Heat).

The syntax of the `get_file` function is:

```
{get_file: <content key>}
```

The `content` key is used to look up the `files` dictionary that is provided in the REST API call. The Homing client command (`Homing`) is `get_file` aware and populates the `files` dictionary with the actual content of fetched paths and URLs. The Homing client command supports relative paths and transforms these to the absolute URLs required by the Homing API.

Note: The `get_file` argument must be a static path or URL and not rely on intrinsic functions like `get_param`. The Homing client does not process intrinsic functions. They are only processed by the Homing engine.

The example below demonstrates the `get_file` function usage with both relative and absolute URLs:

```
constraints:
  check_for_fit:
    type: capacity
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      template: {get_file: stack_template.yaml}
      environment: {get_file: http://hostname/environment.yaml}
```

The `files` dictionary generated by the Homing client during instantiation of the plan would contain the following keys. Each value would be of that file's contents.

- `file:///path/to/stack_template.yaml`
- `http://hostname/environment.yaml`

Questions

- If Homing will only be accessed over DMaaP, files will need to be embedded using the Homing API request format.

get_param

The `get_param` function references an input parameter of a template. It resolves to the value provided for this input parameter at runtime.

The syntax of the `get_param` function is:

```
{get_param: <parameter name>}

{get_param: [<parameter name>, <key/index1> (optional), <key/index2> (optional), ...]}
```

parameter name is the parameter name to be resolved. If the parameters returns a complex data structure such as a list or a dict, then subsequent keys or indices can be specified. These additional parameters are used to navigate the data structure to return the desired value. Indices are zero-based.

The following example demonstrates how the `get_param` function is used:

```
parameters:
  software_id: SOFTWARE_ID
  license_key: LICENSE_KEY
  service_info:
    provider: dmaap:///full.topic.name
    costs: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

constraints:
  my_software:
    type: license
    demands: [demand_1, demand_2, ...]
    properties:
      id: {get_param: software_id}
      key: {get_param: license_key}

  check_for_availability:
    type: service
    demands: [my_vnf_demand, my_other_vnf_demand]
    properties:
      provider_url: {get_param: [service_info, provider]}
      request: REQUEST_DICT
      cost: {get_param: [service_info, costs, 4]}
```

In this example, properties would be set as follows:

Key	Value
`id`	SOFTWARE*ID*
key	*LICENSE*KEY
`provider_url`	dmaap:///full.topic.name
`cost`	50

| Key | Value | |----------|----------| | `id` | SOFTWARE*ID* | | *key* | *LICENSE*KEY | | `provider_url` | dmaap:///full.topic.name | | `cost` | 50 |

Contact

Shankar Narayanan (shankarpnsn@gmail.com)