

Towards a formal reference computational model for cloud configuration management

Philippe Merle, Souha Ben Rayana, Lionel Seinturier, Roger Pissard-Gibollet, Jean-Bernard Stefani, Adja Ndeye Sylla

▶ **To cite this version:**

Philippe Merle, Souha Ben Rayana, Lionel Seinturier, Roger Pissard-Gibollet, Jean-Bernard Stefani, et al.. Towards a formal reference computational model for cloud configuration management. [Research Report] RR-9317, INRIA. 2020. hal-02940938

HAL Id: hal-02940938
https://inria.hal.science/hal-02940938

Submitted on 16 Sep 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Towards a formal reference computational model for cloud configuration management

Philippe Merle, Souha Ben Rayana, Lionel Seinturier, Roger Pissard-Gibollet, Jean-Bernard Stefani, Adja Ndeye Sylla

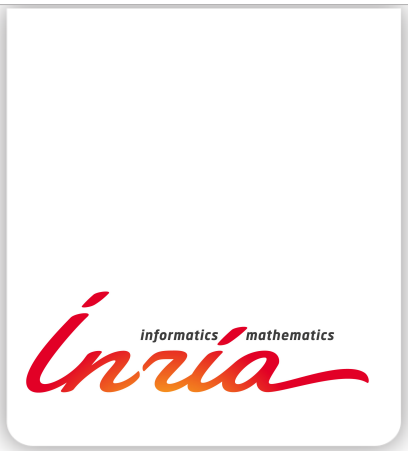
RESEARCH REPORT

N° 9317

January 2020

Project-Teams Spirals and Spades

ISSN 0249-6399 ISRN INRIA/RR--9317--FR+ENG



Towards a formal reference computational model for cloud configuration management

Philippe Merle*, Souha Ben Rayana†, Lionel Seinturier‡, Roger Pissard-Gibollet§, Jean-Bernard Stefani¶, Adja Ndeye Sylla‖

Project-Teams Spirals and Spades

Research Report n° 9317 — January 2020 — 152 pages

Abstract: The multiplication of models, languages, APIs and tools for cloud and network configuration management raises heterogeneity issues that can be tackled by introducing a reference model. A reference model provides a common basis for interpretation for various models and languages, and for bridging different APIs and tools. This report formally specifies, in the Alloy specification language, a reference model for cloud configuration management, we call the Cloudnet Computational Model. We show how to formally interpret several configuration languages in it, including the TOSCA configuration language, the OpenStack Heat Orchestration Template, the Docker Compose configuration language, and the Aeolus cloud deployment model. We show in particular how the formal operational semantics of our Cloudnet computation model allows us to extend the TOSCA standard with Aeolus concepts for deployment lifecycle, and how the Alloy formalization allowed us to discover several classes of errors in the OpenStack HOT specification.

Key-words: configuration management, cloud computing, computational model, Alloy specification

* INRIA, Spirals project team, Lille, France
† INRIA, Spades project team, Grenoble, France
‡ INRIA, Spirals project team, Lille, France
§ INRIA, Spades project team, Grenoble, France
¶ INRIA, Spades project team, Grenoble, France
‖ Orange Labs, Rennes, France

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Towards a formal reference computational model for cloud configuration management

Résumé : The multiplication of models, languages, APIs and tools for cloud and network configuration management raises heterogeneity issues that can be tackled by introducing a reference model. A reference model provides a common basis for interpretation for various models and languages, and for bridging different APIs and tools. This report formally specifies, in the Alloy specification language, a reference model for cloud configuration management, we call the Cloudnet Computational Model. We show how to formally interpret several configuration languages in it, including the TOSCA configuration language, the OpenStack Heat Orchestration Template, the Docker Compose configuration language, and the Aeolus cloud deployment model. We show in particular how the formal operational semantics of our Cloudnet computation model allows us to extend the TOSCA standard with Aeolus concepts for deployment lifecycle, and how the Alloy formalization allowed us to discover several classes of errors in the OpenStack HOT specification.

Mots-clés : gestion de configuration, informatique en nuage, modèle de programmation, spécification Alloy

Contents

1 Introduction **5**
1.1 Motivations . . . 5
1.2 Contributions . . . 6
1.3 Organization . . . 7

2 A reference computational model for configuration management **8**
2.1 Location graph concepts . . . 8
2.2 Operational semantics . . . 13

3 The Fractal model **19**

4 The OCCI model **23**
4.1 OCCI Core . . . 23
4.2 OCCI Infrastructure . . . 25
4.3 OCCI Platform . . . 25

5 The TOSCA model **27**
5.1 TOSCA Core . . . 27
5.2 TOSCA Types . . . 29
5.3 Relating TOSCA and OCCI . . . 39

6 The Docker Compose model **43**
6.1 An overview of Docker Compose . . . 43
6.2 A Location-Graph Based Specification of Docker Compose Configurations . . . 43
6.3 The Docker Compose to Alloy-LG Translator . . . 52

7 The HOT model **59**
7.1 Overview and interpretation of HOT . . . 59
7.1.1 Overview of the HOT core . . . 59
7.1.2 Interpretation of the HOT core . . . 60
7.1.3 Overview of the HOT types . . . 73
7.1.4 Interpretation of the HOT types . . . 74
7.2 Limitations of the HOT specification . . . 101
7.2.1 Errors identified by reading the HOT specification . . . 101
7.2.2 Errors identified by deploying HOT templates . . . 102
7.2.3 Errors identified by formally analyzing the HOT specification . . . 105
7.3 The implemented verification tool . . . 107
7.3.1 Architecture of the implemented tool . . . 107
7.3.2 Illustration of the implemented tool . . . 108
7.3.3 Comparison with other verification tools . . . 119

8 The Aeolus Model **120**
8.1 An overview on the Aeolus component model . . . 120
8.2 Location-Graph interpretation of the Aeolus model concepts . . . 123
8.3 Location-Graph interpretation of the Aeolus model operational semantics . . . 129
8.3.1 State Change . . . 129
8.3.2 Bind . . . 131
8.3.3 Unbind . . . 133

8.3.4 New 136
8.3.5 Delete 137
8.3.6 Correctness of the Location Graph interpretation of the Aeolus model 139
8.4 Extending the TOSCA model with Aeolus concepts 146

9 Conclusion **149**

1 Introduction

1.1 Motivations

Automating configuration management in large scale computing and communication infrastructures has attracted a lot of attention in the past decade, due to the growth of cloud computing services, and the appeal of continuous software development and operation. Just concerning automating cloud software deployment and the management of associated cloud configurations, a number of resource management application programming interfaces (e.g. APIs provided by Amazon AWS [2], Microsoft Azure [13], OpenStack [14], CloudStack [6], Docker [7]), as well as configuration languages and tools (e.g. Amazon CloudFormation and associated AWS tools [4], OpenStack Heat [15], Chef/Puppet [5], Juju [12], Ansible [3], Engage [33], ConfSolve [38], TOSCA [19], OCCI [47, 44, 45], Aeolus [31], SmartFrog [36]) have been proposed, both from industry and academia.

These multiple resource management APIs and configuration languages and tools target various parts of the problem space, from low-level file-based software deployment, to higher-level component-based software orchestration, but with large overlaps in concepts, constructs and functionalities. In situations involving large software deployments and multiple cloud providers, this heterogeneity becomes a problem for it hampers software reuse and portability, as well as systems interoperability and integration. These issues can classically be addressed by identifying core common concepts among the various proposals, and defining a *reference* model that can encompass them. Interoperability issues between these different models and languages, such as e.g. dealing with heterogeneous configuration descriptions, can be alleviated by using the reference model as a common target for tools such as translators and orchestrators. Proposals towards a reference model for configuration management have already been made, notably the consolidated ontology of [54] and the Essential Deployment Metamodel of [53]. The work reported in [54] builds a set of ontologies to represent common concepts from the TOSCA, OCCI and CIMI [11] industry standards, and shows how to use the resulting common semantic knowledge base to translate resource descriptions from one standard to the other. The Essential Deployment Metamodel introduced in [53], gets its inspiration from the core concepts of the TOSCA standard, and shows how configurations described in this metamodel can be mapped to thirteen configuration management and deployment technologies, chosen for their online popularity.

The reference models which have been proposed so far, however, remain mostly informal as do most configuration language and standards mentioned above. This hampers the verification and analysis of configuration descriptions, deployment processes and configuration management tools. Several works have addressed different verification issues in cloud computing by means of formal methods. The recent survey [50] covers a good number of them, and the recent survey on TOSCA-related works [24] provides additional ones. What we find missing, however, is a formally defined reference model:

1. That is expressive enough to serve as a pivot model between the different configuration languages and standards that have been proposed.

2. That allows the formal definition of translations and of hybridizations, between selected formalisms.

3. That allows the formal verification of configuration descriptions and deployment processes, with the support of appropriate formal specification and verification technology.

With respect to objective 2, an example can illustrate specific integration and hybridization questions that may arise. The Topology and Orchestration Specification for Cloud Applications (TOSCA) [19], from the OASIS consortium, is an industry standard that defines a modeling language for describing cloud configurations, i.e. the software components that constitute an application destined to run in a cloud environment, together with the physical and/or virtual resources needed to support them. Although the key TOSCA concepts for cloud configurations are fairly general and versatile, key information required

for the deployment of large cloud configurations such as the deployment and activation lifecycle of components is not readily available in a TOSCA configuration description. The TOSCA specification does include the notions of deployment plan, and of operations that can be provided by components for use at deployment time, but the former merely points to the possible use of business process management facilities, and the latter exact interpretation is essentially left to supporting tools. In contrast, the Aeolus component model [31], developed in academia, allows the formal description of cloud configurations together with their component deployment and activation lifecycles. Aeolus uses a different ontology than TOSCA, and takes the view that configuration deployment constitutes a planning problem that can be automatically solved by analyzing the component lifecycles in a target configuration. Suppose one were interested in using TOSCA e.g. for its support by an industrial standard body, and in using Aeolus notion of deployment lifecycle and its accompanying planification tools. How do we proceed to hybridize TOSCA with Aeolus concepts ? Certainly, we need (i) to map the different TOSCA and Aeolus ontologies, but we also need (ii) to clarify the relationships between the deployment aspects of the two, and (iii) to extend the TOSCA configurations with Aeolus deployment lifecycle elements. Finally, we need (iv) to correlatively extend a TOSCA deployment and orchestration toolchain to benefit from Aeolus deployment planning tools. Of course, the question of importing Aeolus concepts can be relevant with other formalisms and standards. For instance, one may contemplate the hybridization of the OCCI industrial standard with Aeolus, if we want to benefit from the former unified cloud resource management API for operating in a multi-cloud environment.

Items (ii) and (iii) above suggest that the reference model we are looking for needs to go beyond the mere alignment of concepts between formalisms for configuration descriptions, and allow reasoning about architectural invariants, configuration behaviors, deployment plans and processes. In turn this suggests a reference model for configuration management should not just be a data model for configuration descriptions, but ought to be a computational model endowed with a formal operational semantics, able to describe configurations and their associated behaviors, including configuration deployment and activation lifecycles. Being able to define and reason about configuration behavior is useful beyond deployment time: it is necessary e.g. for describing dynamic adaptation in scale out scenarios, or in fault or security management for describing fault occurrences in configurations as well as fault management processes or component fall back behaviors. More generally, moving towards continuous software deployment and operation (in their many forms, e.g. Continuous Delivery, DevOps, ArchOps), and autonomic computer systems management, requires the identification of computational models to allow the description, and reasoning about, running and continuously upgraded software architectures. A reference computational model such as the one we advocate in this paper can serve as a useful first step in this matter.

Finally, objective 3 suggests our reference computational model ought to be embedded in some formal specification language able to express structural properties such as the architectural invariants identified for HOT deployments in Section 7, as well as dynamical properties such as the execution invariant characterizing correct Aeolus deployment plans or processes in Section 8.

1.2 Contributions

In this report, we introduce a formal reference computational model for the description of cloud software configurations. Its broad goal is to provide a uniform and flexible basis for describing and analyzing the execution, operation and management of cloud applications, services and systems. More specifically, it aims to provide: to system and software architects, a meta-model for describing and analyzing their system models and software architectures; to configuration management tool builders, a pivot model for the interoperability and integration of different languages and tools for the description, deployment and orchestration of cloud applications.

In more detail, the report makes the following contributions:

1. We formally define, using the Alloy specification language [39], a reference model for configura-

tion management and its operational semantics. We show that it subsumes the Fractal component model by specifying an interpretation of the Fractal component model, as formally defined in [43].

2. We illustrate the pivot role our reference model can play, by providing an interpretation of two recent standard models for configuration management in cloud computing systems, the OCCI [47, 44, 45] and TOSCA [19] specifications, and of two popular configuration languages for cloud orchestration and container deployment, the OpenStack Heat Orchestration Template [9] and Docker Compose [8].

3. We show with the OpenStack Heat Orchestration Template (HOT) the benefits of our formal approach, highlighting three classes of errors that were discovered in the process of writing the formal specifications and of checking them with the Alloy Analyzer, including errors and missing invariants in the HOT specifications, and inconsistencies between the HOT specification and the behavior of the Heat deployment engine.

4. We formally specify an interpretation of the Aeolus component model for cloud deployment [31] in our reference model, and we show that this interpretation is faithful to the Aeolus operational semantics. We further show how this interpretation can be immediately leveraged to extend the TOSCA specification with Aeolus deployment lifecycle concepts.

All our formal specifications are developed wih the Alloy specification language [39], a lightweight formal specification language based on first-order relational logic. Alloy is interesting because of its simplicity (it should be readable by anyone familiar with object-oriented programming and modeling), and because of the straightforward use of its analyzer, which enables rapid iterations between modeling and analysis when writing a specification (very much akin to debugging a specification). For a brief presentation and motivation of Alloy, we refer the interested reader to the paper [40]. An online tutorial for Alloy is also available on the Alloy Analyzer Web site [1].

1.3 Organization

The report is written mostly in a literate programming style: the Alloy specifications are presented in full, with the (informal) commentary on the formal specifications interspersed with excerpts of the Alloy code. All axioms (Alloy *facts*) and theorems (Alloy *assertions*) have been checked with the Alloy analyzer, checking for the existence of finite models in the first case, and for the absence of counter-examples in models below a certain size in the second case[1].

The report is organized as follows. Section 2 specifies the location graph subset we adopt as our core computational model. Section 3 specifies the interpretation of the Fractal model. Section 4 specifies the interpretation of the OCCI model. Section 5 specifies the interpretation of the TOSCA model. Section 6 specifies the interpretation of the Docker Compose configuration language. Section 7 specifies the interpretation of the OpenStack Heat Orchestation Template (HOT). Section 8 specifies the interpretation of the Aeolus model. Section 9 concludes the report.

[1] The checks were performed using the SAT4J and MiniSat SAT solvers embedded in the Alloy analyzer, and using its maximum memory size (4096 Mb) and its maximum stack size (65536 Kb)

2 A reference computational model for configuration management

In this section we formally specify our reference computational model for cloud configuration management. Our reference model is component-based, in the sense of module and component-connector views of system architecture [23], and of software component models as defined in [32]. We have by now ample evidence that a component-based approach provides an excellent basis for building configurable systems and for managing them, as illustrated e.g. by the Fractal component model [27] and its use in systems management [49]. We are in line, in this respect, with the TOSCA specification which comprises essentially a component model at its core.

Our reference computational model is developed as a subset of the hypercell framework [52], itself a generalization of the location graph framework [51]. This framework constitutes a component meta-model, with a formal operational semantics and behavioral theory, that is the first, to our knowledge, to support the modeling of dynamic hardware and software architectures with sharing and encapsulation. Even though it is less well established than other software component models developed over the past twenty years [32], it can be seen as an extension and a generalization of the well-known Fractal [27] and BIP [25, 35] component models.

In this report we use the location graph terminology. We first define the core location graph concepts of our reference model and their static semantics. We then define the operational semantics of our reference computational model. Our reference computational model inherits from the hypercell framework its meta model character: it can be instantiated to yield specific location graph models. A specific location graph model is obtained by defining key datatypes Value, Process, Sort, Role, a set of individual location transitions, and an authorization predicate Auth on individual location transitions.

2.1 Location graph concepts

A configuration in our reference computational model takes the form of a location graph. A LocationGraph is just a set of locations. A Location is a locus of concurrent computation, as in process calculi with localities, such as the Distributed π-calculus [37], Ambient [29, 28] or Kell [48] calculi. From a software engineering point of view, a location can be understood as a (hardware or software) component or as a connector, as in software component models [32] and in component-and-connector views of software architecture [23]. Each location is endowed with a unique name, a sort, a process, as well as required and provided sets of roles.

A Role corresponds to a point of attachment and interaction of a location, similar to the notion of port or interface found in other component models. Intuitively, a *provided* role of a location L corresponds to a point of interaction at which L provides a service to its environment. Conversely, a *required* role of a location L corresponds to a point of interaction at which L expects some service from its environment. The condition **no** (provided & required), in the definition of the Location signature, ensures that required and provided roles of a location form a partition of the set of roles of a location. Intuitively, this is to reflect a distinction between services or functionalities *provided* by a location, and services *required* by the same location. It is possible that, in a location graph, a service required by location h be fulfilled by some service provided by the same location h (think recursive software modules), but this provision will be mediated by at least another location connecting the two roles. When, in a given location graph, a role r appears in provided position in one location, and in required position in another location, we say r is *bound*, and that r *binds* the two locations that offer it. A role in a location graph which only appears in one location, in provided or required position, is said to be *unbound*.

The Process of a location specifies the location behavior. The Sort of a location is a type for the location, which is used in particular to specify encapsulation constraints in a location graph. For instance, the sort of a location L can specify which of the roles of L are meant to bind L to its subcomponents, and which should be understood as public interfaces to the L composite. We will not make much use

of sorts and encapsulation constraints in this report but we refer the interested reader to [52] for further information. We will make use of processes in the Aeolus interpretation in Section 8.

Interactions in a location graph take place over roles, and Values can be exchanged during these interactions. Location names, sorts, processes, and roles are all instances of values and can thus be exchanged during interactions between locations.

```
module LocationGraphs

sig LocationGraph {
  locations : set Location
}

sig Location {
  name : one Name,
  process : one Process,
  sort : one Sort,
  provided : set Role,
  required : set Role
}
{ no ( provided & required ) } // In a location, provided roles and required roles are disjoint.

sig Value {}

sig Name extends Value {}

sig Process extends Value {
  patoms : set Name + Role
}

sig Sort extends Value {
  satoms : set Name + Role
}

sig Role extends Value {}

one sig Null extends LocationGraph {}

fact NullHasNoLocations {
  all c : Null | c.locations = none
}
```

A few additional comments on the Alloy LocationGraphs module above are warranted. Location processes and sorts can embed *atoms*, i.e. location names and roles (think local variables in some procedure code), accessible through the patoms and satoms relations, respectively. They are mostly used to specify some constraints for the reference model operational semantics below. The Null location graph is singled out explicitly as the neutral element of location graph composition, defined in the operational semantics below.

Location graphs can be seen as hypergraphs where vertices are roles, and (hyper)edges are locations, subject to the static semantics constraints given below. We draw locations graphs as illustrated in Figure 1: locations are drawn as circles or ovals, unbound roles as black dots on their boundary, and bound roles by links connecting two black dots in two different locations.

As said above, one can understand a location as an executing software component, with different points of interactions, called roles, to connect it to other components. This interpretation of the location graph ontology aligns well with the standard concepts of component-based software engineering and software architecture, as present e.g. in the ACME component model [34] (locations can be understood as ACME *components* or *connectors*, roles as ACME *ports*), or in the Fractal component model [27, 43] (locations can be understood as Fractal *components*, roles as Fractal *interfaces*, with the same distinction

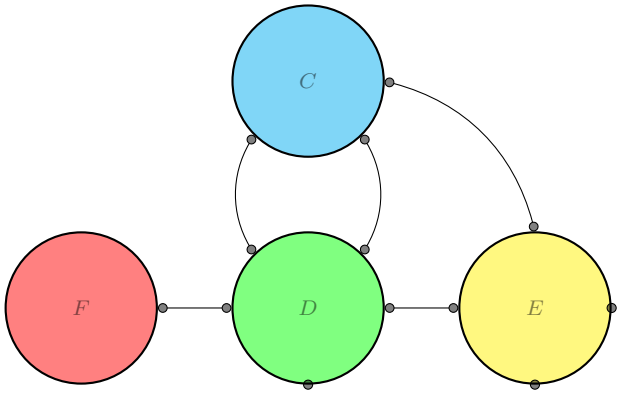


Figure 1: A small location graph

between provided and required roles as between provided and required interfaces, and bound roles as Fractal *primitive bindings*). In line with the above "standard" interpretation, direct interactions between locations take place only at bound roles, in the form of simple point-to-point bidirectional interactions between two locations attached to the same role. In Figure 1, D and F can directly interact because of their shared role, but F cannot directly interact with C or E, for there is no shared role between F and any one of these two locations.

Other interpretations are also valid. Consider the simple location graph depicted in Figure 2. A first reading of the picture is as suggested above: component R is a composite or aggregate with three subcomponents C, D and E. Another reading, in conformity with the BIP component model [35, 25], is that R is a composition operator that glues or synchronizes together C, D, and E. Yet another reading, in conformity with the Reo component model [41], would have R as a connector, connecting C, D and E, each one at a different port. Finally, and more generally, one can interpret the configuration in Figure 2 as depicting an instance of a relationship, where R is the relation and C, D and E are the components participating in the relationship (taking part in different *roles*). Relation R in the latter interpretation, can be an arbitrary relation, and the roles binding R and the participating components need not carry interactions between them. These different interpretations can of course coexist within the same location graph. Figure 3 can depict a relation R connecting three components C, D, and E, as in Figure 2, but which contains two sub-relations R_1 and R_2. We actually use this interpretation of locations as relations in the interpretation of the TOSCA concepts in location graphs in Section 5.

The static semantics constraints that apply to location graphs are as follows:

1. In a location graph, all locations are uniquely named (fact LocationsInLocationGraphAreUniquelyNamed below), i.e. no two locations can have the same name.

2. In a location graph, a role is provided by at most one location (fact InALocationGraphARoleIsProvidedBySingleLocation below).

3. In a location graph, a role is required by at most one location (fact InALocationGraphARoleIsRequiredBySingleLocation below).

The first constraint ensures that locations are uniquely identified by their names. This ensures all components (locations) in a configuration (location graph) can be singled out for management purposes (e.g. removal in case of faulty behavior). The second and third constraints ensure that a role binds at most two locations. In turn, this ensures that the interaction semantics at roles can be defined once and

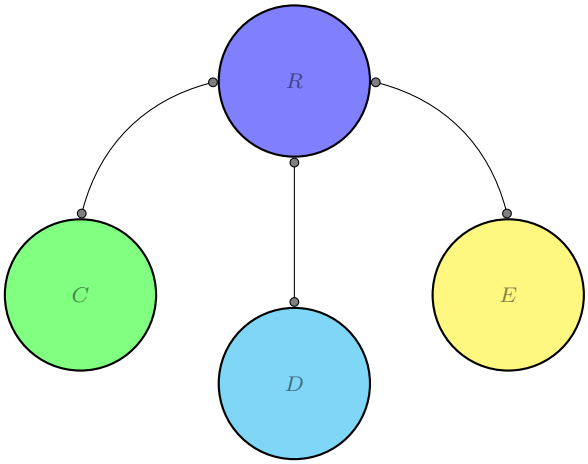


Figure 2: A location graph depicting a relation between three components

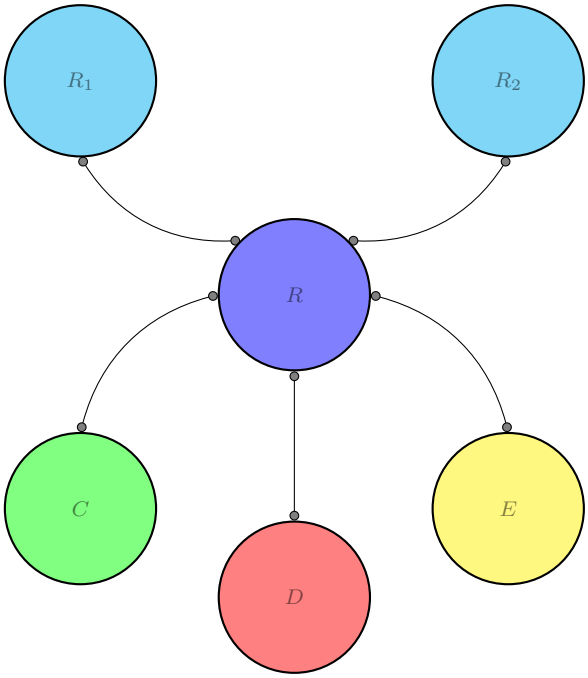


Figure 3: A location graph depicting a composite relation between three components

for all; the alternative would have been to allow an arbitrary number of locations to be bound by a role, but this would have prompted the question of which semantics to apply when an interaction takes place at a multi-bound role (binary rendez-vous between single provider and a randomly chosen requirer? total or partial broadcast from single provider to requirers? publish-subscribe like behavior between multiple providers and requirers? etc.). It also has the benefit of simplifying the operational semantics and the behavioral semantics of the model. The downside of this choice is the necessity to make explicit, by means of locations, any multipoint interaction capability.

```
/*****************************************************************************
 * Location graphs: static semantics
 *****************************************************************************/

/** Locations in a location graph are uniquely named. */

pred UniquelyNamedLocations[c : set Location] {
  no disj c1, c2 : c | c1.name = c2.name
}


/** A role is provided by a single location. */

pred UniquelyProvidedRoles[c : set Location] {
  no disj l, h : c | some l.provided & h.provided
}


/** A role is required by a single location. */

pred UniquelyRequiredRoles[c : set Location] {
  no disj l, h : c | some l.required & h.required
}



/** Well−formedness for location graphs */

pred WellFormedLocationSet[c : set Location]{
  UniquelyNamedLocations[c]
  and
  UniquelyProvidedRoles[c]
  and
  UniquelyRequiredRoles[c]
}


pred WellFormedLocationGraph[g: LocationGraph] {
   WellFormedLocationSet[g.locations]
}


/** Axiom for Location Graphs */

fact All_LocationGraph_Are_Well_Formed {
  all g: LocationGraph | WellFormedLocationGraph[g]
}
```

This concludes the definition of the structure and static semantics of our reference model. In Alloy, one can check that its specification is consistent, i.e. that it has a model (in the logical sense), and evaluate various structural properties on location graphs. We can also check simple consequences of the static semantics constraints, including the fact that, in any location graph, roles bind at most two locations

(assertion ARoleBindsAtMostTwoLocations below).

```
/** Model means that there exists some location graphs. */
pred Model {
  some g: LocationGraph | #(g.locations) > 2 and #(g.locations.required) > 2 and #(g.locations.provided) > 3
}

run Model for 10

/** In a well−formed location set, a role is provided and required by two distinct locations. */

assert RoleIsProvidedAndRequiredByTwoDistinctLocations
{
  all lg : set Location, r : Role, l, h : Location |
    WellFormedLocationSet[lg] implies
    l in lg and h in lg and r in l.provided and r in h.required implies l != h
}

check RoleIsProvidedAndRequiredByTwoDistinctLocations for 40 expect 0


/** In a well−formed location set, a role binds at most two distinct locations. */
assert ARoleBindsAtMostTwoLocations
{
  all lg : set Location, r : Role |
     WellFormedLocationSet[lg] implies
    all ls : set lg | (all h : ls | r in h.required + h.provided) implies #ls < 3
}

check ARoleBindsAtMostTwoLocations for 40 expect 0
```

2.2 Operational semantics

The Hypercell framework comes equipped with an operational semantics, featuring both interactions and priorities [52]. For our reference model in this report, we consider only a simplified operational semantics that comprises only interactions. Priorities are important for expressivity (for instance, to model discrete time and preemption) but we left them out for the time being for they are not needed for the constructions in this report. Should the need arise to take into account behaviors involving timing or preemption aspects, they can easily be added to our reference computational model, with no change needed to the interpretations and constructions in this report. The notion of interaction used in this report is also simplified compared to [52]. The Hypercell framework also comes equipped with a notion of authorization for the definition of encapsulation policies. We leave this aside in this report, but should the need arise to take into account e.g. the definition of configurations with nesting and sharing, authorization can readily be introduced in our reference computational model.

The behavior of a location graph is given by a set of Transitions. A Transition is a 4-tuple $\langle \Delta, G, \Lambda, G' \rangle$, where Δ is a Context, G and G' are sets of locations, and Λ is a Label. It describes the evolution of an initial set of locations G (init) into a terminal set of locations G' (term). During a transition locations in the initial set may interact with other locations not in this initial set. This potential for interaction takes the form of a set of Interactions that form the Label of a transition. Interactions consist in the emission or receipt of some Values. An interaction takes place at a given role, which can be in provided (Plus polarity) or required (Minus polarity) position. The Context of a transition just describes the set of known atoms (eatoms) at the onset of the transition. This is used to define new atoms (names or roles) that may be created during a transition: by definition, a new name or role appearing in the terminal set of locations of a transition will not appear in the Context of this transition.

Notice that the location graph model is a higher order model: the constituents of a location, processes, that embody the computational part of a location, sorts, location names and roles are values and thus can be exchanged during interactions. The exchange of processes allows to model e.g. the introduction of new functionality in a system.

```
module LG_Semantics
open LocationGraphs

/*****************************************************************************
 * Location Graph Transitions
*****************************************************************************/

sig Transition {
  env: one Context,
  init: set Location,
  label: one Label,
  term: set Location,
}

sig Context {
  eatoms: set Name + Role,
}


sig Label {
  signals : set Interaction,
}

sig Interaction {
  irole: one Role,
  polarity: one Polarity,
  payload: one Value,
}

abstract sig Polarity {}

one sig Plus, Minus extends Polarity {}
```

One may wonder why, in the definition of Transition above, we use location sets in place of location graph for the initial an terminal configurations of a transition. This is mostly for technical reasons: using sets of locations simplifies the definition of the operational semantics of composition defined below, simplifies the specification of examples of composition as can be found in the interpretation of the Aeolus model in Section 8, and accelerates the construction by the Alloy analyzer of counterexamples involving transitions.

Before moving to the definition of axioms concerning transitions allowed in our reference model, we define below a few auxiliary functions and predicates on location graphs. Predicate Bound_in_LG determines when a role r is bound in a location graph c (exactly when there are two locations in c, one offering r in provided position, and the other offering r in required position). Function Bound_roles returns all the bound roles of a given location graph c. Functions Unbound_roles_in_LG and Unbound_roles_in_LS return the set of unbound roles in a location graph or in a set of locations, respectively. Function AtomsInGraph returns all the atoms appearing in a location and a location graph, respectively. Predicate MixableLG represents the pre-condition for composing two location graphs $c1$ and c_2 into a location graph whose locations is the the union of the locations of c_1 and c_2 (predicate Compose).

```
/*****************************************************************************
* Auxiliary functions on location graphs and transitions
*****************************************************************************/

pred Bound_in_LS [ r: Role, c : set Location] {
```

```
  some l1, l2: c | r in l1.provided & l2.required
}

pred Bound_in_LG [r: Role, c: LocationGraph] { Bound_in_LS[r, c.locations] }

fun Bound_roles_in_LS[c: set Location]: set Role {
 { r : Role | Bound_in_LS[r,c] }
}

fun Bound_roles_in_LG[c:LocationGraph]: set Role { Bound_roles_in_LS[c.locations] }

fun Unbound_roles_in_LS [c: set Location]: set Role {
 (c.provided + c.required) − Bound_roles_in_LS[c]
}

fun Unbound_roles_in_LG [c:LocationGraph]: set Role { Unbound_roles_in_LS[c.locations] }

fun AtomsInGraph [lg: set Location] : set Name + Role {
   lg.process.patoms + lg.sort.satoms + lg.provided + lg.required + lg.name
}

pred MixableLS[c1,c2: set Location] {
  (no c1.name & c2.name)
  and (no c1.provided & c2.provided)
  and (no c1.required & c2.required)
}

pred MixableLG[c1,c2: LocationGraph] { MixableLS[c1.locations, c2.locations] }

pred Compose[c, c1, c2: LocationGraph] { c.locations = c1.locations + c2.locations }
```

Transitions must obey a number of well-formedness axioms. Axiom TransitionsEnvironmentsGatherAllAtoms states that contexts record all the atoms (location names and roles) that appear in the initial location set of a transition. Axiom ATransitionInitialGraphIsNotEmpty merely states that the initial location set of a transition is not empty: by definition, the empty location set has no transition (hence an empty location graph has not associated transition). Axiom SignalsInTransitionsEmittedOnUnboundRoles ensures any successful interaction taking place on a bound role (see the predicate Matched_signal_pair below for a definition of a successful interaction). in a given location set G is not visible in the transitions of G.

```
/*****************************************************************************
* Well−formedness facts about transitions
*****************************************************************************/

fact TransitionsContextsGatherAllAtoms {
  all t: Transition | AtomsInGraph[t.init] in t.env.eatoms
}

fact ATransitionInitialGraphIsNotEmpty {
  all t: Transition |some t.init
}

fact SignalsInTransitionsEmittedOnUnboundRoles {
  all t : Transition | t.label.signals.irole in Unbound_roles_in_LS[t.init]
}

fact SignalsInTransitionsEmittedOnAGivenRoleAreUnique {
  all t : Transition, r : Role| lone m : t.label.signals | m.irole = r
}
```

We now present the operational semantics of composition in our reference computational model. Specifically, we define the possible transitions of a location graph that is the composition of several

location graphs. The classical way to proceed is by means of inference rules, as is done in [52], but one cannot define inference rules in Alloy so we proceed indirectly, by means of predicates.

Function LocationGraphFromLocation is just an auxiliary function which turns a single Location into a LocationGraph. Predicate Matched_Signal_Pair characterizes a successful interaction pair, where a value is exchanged between locations bound at a role. Intuitively, a successful pair takes the form $\langle r : V, \overline{r} : V \rangle$, where r is the role at which the exchange takes place (the irole of an interaction), and V is the value exchanged (the payload of an interaction). The exchange at role r is successful if one of the involved location offers r in provided position ($r : V$ – Plus polarity), and the other offers r in required position ($\overline{r} : V$ – Minus polarity). Predicate Matched_Signals generalizes predicate Matched_Signal_Pair to a pair of sets of interaction. Intuitively, a interactions in a given interaction set are matched if each interaction in the set can be matched with another interaction in the set.

```
/*****************************************************************************
* Operational semantics: composition
*****************************************************************************/


fun LocationGraphFromLocation[l: Location]: LocationGraph {
  {c : LocationGraph | c.locations = l }
}

pred Matched_Polarities[p1,p2: Polarity] {
   (p1 in Plus and p2 in Minus) or (p1 in Minus and p2 in Plus)
}

pred Matched_Signal_Pair[m1,m2: Interaction] {
   (m1.irole = m2.irole) and (m1.payload = m2.payload)
   and Matched_Polarities[m1.polarity,m2.polarity]
}

pred WF_Signals[s : set Interaction] {
  all m1,m2 : s | m1.irole = m2.irole and m1.polarity = m2.polarity implies m1 = m2
}

pred Matched_Signals [s : set Interaction] {
  WF_Signals[s]
  and (all m1 : s | one m2: s | Matched_Signal_Pair[m1,m2])
}
```

There are essentially two possibilities to consider when computing a possible transition t of a location graph consisting of at least two locations: either the transition t is the result of the synchronized transitions of a several subsets of the initial location graph of t, or it is the result of the transition of a subset of the initial location graph of t. The two predicates Synchronizing_Transition and Uncoupling_Transition below deal with these two cases, respectively.

Predicate Synchronized_Transitions characterizes the ondition for sets of transitions denoting the evolution of several parts of a location graph to synchronize properly. Essentially, the initial location sets of the different transitions must be Mixable, and their interactions and actions on bound roles in the composition of their initial configurations should match (condition with Matched_Signals), and their terminal configurations are Mixable.

When transitions in a set are synchronized, they give rise to a synchronizing transition t, characterized by the predicate Synchronizing_Transition. The synchronizing transition t has the same environment than the synchronized transitions, has as initial and terminal configuration, respectively, the composition of the initial, resp. terminal, configurations of t_1 and t_2, and has as interactions the union of the interactions in t_1 and t_2 minus the interactions that take place on bound roles.

```
pred Synchronized_Transitions[ts: set Transition] {
     let sb = { m : ts.label.signals | m.irole in Bound_roles_in_LS[ts.init] } |
```

```
        one ts.env
        and ( all disj t1, t2: ts | MixableLS[t1.init, t2.init] )
        and WellFormedLocationSet[ts.init]
        and Matched_Signals[ sb ]
        and WellFormedLocationSet[ts.term]
}

pred Synchronizing_Transition[t: Transition, ts: set Transition] {
   let sb = { m : ts.label.signals | m.irole in Bound_roles_in_LS[ts.init] } |
        Synchronized_Transitions[ts]
        and t.env = ts.env
        and t.init = ts.init
        and t.label.signals = ts.label.signals − sb
        and t.term = ts.term
}
```

Predicate Uncoupled_Transition characterizes a transition t and a location graph c such that t represents the evolution of a subset of the location graph that is the union of c and of the initial location of t. For this to be the case, there must be no required synchronization to carry out with c: no interaction in the label of transition t must take place on a role of c. When transition t and location graph c are uncoupled, as stipulated by Uncoupled_Transition, they give rise to a transition of the location graph that is the union of c and of the initial location graph of t. This transition is characterized by predicate Uncoupling_Transition.

```
pred Uncoupled_Transition [t: Transition, c: set Location] {
    MixableLS[t.init, c]
    and MixableLS[t.term,c]
    and no t.label.signals.irole & (c.provided + c.required)
}

pred Uncoupling_Transition[tt, t: Transition, c: set Location] {
     tt.env = t.env
     and Uncoupled_Transition[t, c]
     and tt.init = t.init + c
     and tt.label = t.label
     and tt.term = t.term + c
}
```

We can check that our definitions and predicates are consistent, together with a few simple properties, that are easy consequences of our definitions. In particular, instances for ModelSynchronizing and ModelUncouplong show that there are synchronizing transitions and uncoupling transitions.

```
run Model { some t: Transition | t.label.signals != none
} for 10 but exactly 5 Transition, exactly 3 Interaction, exactly 3 Label expect 1

run ModelMatchedSignals {
  some s1,s2:Interaction | Matched_Signals[s1 + s2]
} for 10


run ModelMixableLS {
  some c1,c2: set Location |
          #c1 > 1 and #c2 >1
        and #c1.provided >1 and #c2.required > 2 and #c2.provided > 1
        and MixableLS[c1, c2]
        and WellFormedLocationSet[c1] and WellFormedLocationSet[c2]
} for 10


run ModelSynchronized {
    some t1,t2: Transition | Synchronized_Transitions[t1 + t2] and #(t1 + t2).label.signals > 3
} for 10
```

```
run ModelSynchronizing {
    some t: Transition, ts: set Transition | Synchronizing_Transition[t,ts] and #ts >2 and #ts.label.signals > 3
} for 10


run ModelUncoupling {
    some t, tt:Transition, c: set Location | Uncoupling_Transition[tt,t,c] and #t.label.signals >1
} for 10


assert UnboundRoleIsNotBound {
  all c:LocationGraph |
    no ( Unbound_roles_in_LG[c] & Bound_roles_in_LG[c] )
}

check UnboundRoleIsNotBound for 20 expect 0


assert MixableLGsAreComposable {
  all c1, c2: LocationGraph | MixableLG[c1,c2]
                                          implies WellFormedLocationSet[c1.locations + c2.locations]
}

check MixableLGsAreComposable for 20 expect 0


assert MixableLS_Symmetric {
  all c,d : set Location | MixableLS[c,d] iff MixableLS[d,c]
}

check MixableLS_Symmetric for 20 expect 0


assert Roles_In_Synchronizing_Transition_Labels_Are_Unbound {
  all t, t1, t2: Transition |
     Synchronizing_Transition[t,t1 + t2] implies no ( t.label.signals.irole & Bound_roles_in_LS[t.init] )
}

check Roles_In_Synchronizing_Transition_Labels_Are_Unbound for 20 expect 0
```

3 The Fractal model

We give in this section a structure-preserving interpretation of the core concepts of the Fractal model in our pivot model. We follow the formal specification of the Fractal model in Alloy [43] and its terminology. The core concept that underlies the notion of component in Fractal is called a *kell* in [43], and is defined simply in Alloy as follows:

```
module fractal/foundations

sig Id {}
sig Gate { gid: Id }

sig Kell {
      gates: set Gate,
      sc: set Kell,
      kid: Id
}

fact GatesInKellHaveUniqueIds
{ all c:Kell | all i,j:c.gates | i.gid = j.gid => i = j }
```

A kell is interpreted as a location graph with a distinguished location, top, which bears the identity kid of the kell. Locations in a location graph representing a kell are called KellComps, so named because they constitute in effect a composition operator for their subkells. The points of attachment for subkells of a KellComp k are roles gathered in the subrequired set of roles of k. Conversely, the points of attachment for a KellComp k to its parent kell are roles gathered in the subprovided set of roles of k. Other roles of a KellComp k in grequired and gprovided correspond to gates of the kell whose k is top, i.e. base notions of Fractal component interfaces in [43]. We use here the notion of role in two forms: as a mere point of attachment for locations in a given relation (here of composition), or as points of interaction between two locations. The different axioms about kells interpreted in location graphs enforce some simple consistency constraints: any kell has a single top location; all locations that form a kell are KellComps; the locations of a kell comprise its top location and all locations of its subkells; all top locations of the immediate subkells of a kell k, given by the sc relation, are linked to the top of k by some bound role in its subrequired set of roles; all gates of a kell are in the grequired or grovided set of roles of its top kell; and the identifier id of a kell corresponds to the name of its top location.

```
module Fractal
open LocationGraphs


abstract sig KellComp extends Location {
  grequired : set Role,
  gprovided : set Role,
  subrequired : set Role,
  subprovided : set Role,
}

fact Gates_are_not_subs
{ all k:KellComp |no (k.grequired + k.gprovided) & (k.subrequired + k.subprovided) }

fact Gates_and_subs_are_location_roles
{ all k:KellComp | (k.grequired + k.subrequired = k.required) and (k.gprovided + k.subprovided = k.provided) }


abstract sig Kell extends LocationGraph {
  top : one KellComp,
  sc : set Kell,
  gates : set Role,
```

```
  kid : one Name
}

fact Top_is_functional
{ all h,k : Kell | h.top = k.top implies h = k }

fact All_locations_in_a_kell_are_kellcomps
{ all k:Kell | k.locations in KellComp }

fact All_locations_in_a_subkell_are_also_in_top_graph
{ all k : Kell | k.^sc.locations + k.top = k.locations }

fact All_top_subkells_locations_are_bound_to_top
{ all k : Kell | all h : k.sc.top | some r : h.subprovided | r in k.top.subrequired }

fact All_bound_sub_roles_are_to_subkells_top_location
{ all k : Kell | k.top.subrequired in k.sc.top.subprovided }

fact All_gates_are_grprovided_or_grequired_roles_of_its_top_location
{ all k : Kell | k.gates = k.top.gprovided + k.top.grequired }

fact A_kell_id_is_the_name_of_its_top_location
{ all k : Kell | k.kid = k.top.name }
```

With this interpretation of the kell concept we can found all the development of the Fractal model as done in [43]. In fact, modulo the identification of Name with Id, and of Role with Gate, all the development of the Fractal component model in [43] can be carried out by relying on the present Fractal module in place of the Foundations module that defines the kell concept.

At this point it is interesting to ask what are the differences between location graphs and Fractal components (and their kell foundations). The key difference, as is visible in the specification above, is that the location graph framework has no pre-built concept of sub-component, in contrast to kells that are defined with the sc relation between kell and subkells. Being a subcomponent of a composite is manifested by role bindings in location graphs, as are relations of interactions between components. Said otherwise, the Fractal model makes a distinction between *vertical composition* (composite and sub-components) and *horizontal composition* (component to component interactions) [32], which the location graph framework ignores. The main benefit of lifting this distinction in location graphs is a better understanding of component composition and component binding (they are the same thing), and a well-defined operational semantics and behavioral theory, which have never been fully obtained for the Fractal model. In fact, the equivalent of a location graph in the Fractal model, would be what we define below as a KellSystem, namely a set of kells which do not necessarily stand in a relation of aggregation with one another. With this notion, we can verify that relations of aggregation between components (being a subcomponent of another) in an arbitrary Fractal system are indeed captured by role bindings as the propositions All_tlinked_kells_in_kell_system_are_descendant_subkells and All_descendant_subkells_in_a_kell_system_are_tlinked attest.

```
abstract sig KellSystem extends LocationGraph {
  kells : set Kell,
}

fact All_locations_in_kells_of_kell_system_are_in_kell_system_graph
{ all ks: KellSystem | ks.locations = ks.kells.locations }

fact All_subkells_in_kell_system_are_in_kell_system_kells
{ all ks: KellSystem | ks.kells.^sc in ks.kells }

/*****************************************************************************
 * Properties and non-properties.
```

```
*****************************************************************************/

assert Top_for_every_kell
{ all k: Kell | some kc: KellComp | k.top = kc }

pred linked [h,k : Kell]
{ some r : h.top.subrequired | r in k.top.subprovided }

assert All_immediate_subkells_linked
{ all k : Kell | all h: k.sc | linked[k,h] }

assert All_linked_kells_are_subkells
{ all h,k : Kell | linked[h,k] implies k in h.sc }

assert All_linked_kells_in_kell_system_are_subkells
{ all ks: KellSystem | all h,k : ks.kells | linked[h,k] implies k in h.sc }

assert All_subkells_in_kell_system_are_linked
{ all ks : KellSystem | all h,k : ks.kells | k in h.sc implies linked [h,k] }

pred tlinked [h,k : Kell]
{ some p : h.*sc | linked[p,k] }

assert All_tlinked_kells_in_kell_system_are_descendant_subkells
{ all ks: KellSystem | all h,k: ks.kells | tlinked[h,k] implies k in h.^sc }

assert All_descendant_subkells_in_a_kell_system_are_tlinked
{ all ks : KellSystem | all h,k: ks.kells | k in h.^sc implies tlinked[h,k] }

assert Well_foundedness
{ no p: Kell | p in p.^sc }

assert Sharing_not_possible
{
   all k : Kell | no p, q, h : k.^sc |
      h in p.sc and h in q.sc and p != q and (not p in q.sc) and (not q in p.sc)
}

assert Sharing_not_possible_bis
{
  all ks: KellSystem | all k: ks.kells | no p,q,h: ks.kells |
     tlinked[k,p] and tlinked[k,q] and tlinked[k,h] and p != q and (not tlinked[q,p]) and (not tlinked[p,q])
}
```

Remark 1 *In a kell system, the linked predicate and the sc relation coincide, as attested by the two propositions: All_immediate_subkells_linked and All_linked_kells_are_subkells. One could have thought to actually the tlinked predicate without reference to the sc relation but unfortunately predicates cannot be defined recursively in Alloy, hence the use of the reflexive and transitive closure of sc in the definition of predicate tlinked.*

This interpretation of the Fractal model core is sound, and the propositions above hold as expected. To fix the ideas we also check that two non-properties are indeed not valid: that aggregation is well-founded, and that sharing between aggregates is not possible. The non-well-foundedness property may seem surprising, but in location graphs this merely says that it is possible to have cycles in the transitive closure of the linked relation, which is not more surprising than the statement that one can have cycles in the interaction relation between components, or in the dependency graph between code modules.

```
run Model {} expect 1
```

```
check Top_for_every_kell for 20 expect 0
check All_immediate_subkells_linked for 20 expect 0
check All_linked_kells_are_subkells for 15 expect 0
check All_linked_kells_in_kell_system_are_subkells for 15 expect 0
check All_subkells_in_kell_system_are_linked for 15 expect 0
check All_tlinked_kells_in_kell_system_are_descendant_subkells for 11 expect 0
check All_descendant_subkells_in_a_kell_system_are_tlinked for 10 expect 0
check Well_foundedness for 10 expect 1
check Sharing_not_possible for 10 expect 1
check Sharing_not_possible_bis for 10 expect 1
```

4 The OCCI model

We present in this section a location graph interpretation of the OCCI core mode [47], together with its infrastructure and platform concepts. The OCCI specifications aim to provide APIs for the management of (primarily) IaaS services. For an introduction to OCCI (and an alternate formal specification using the OCL language), we refer the reader to [42].

4.1 OCCI Core

We present in this section an interpretation of the key subset of the OCCI core model [47], which consists of three concepts: Entity, Resource, and Link. A Resource represents any hardware or software cloud computing resource, such as a machine, a virtual machine, a container, a network. A Link represents a binary relation between Resources, such as the connection between a machine and a network, the hosting relationship between a machine and a container. Entity is the general abstraction for structural configuration elements, covering both Resource and Link. For the time being, we do not consider in our interpretation the typing concepts of the OCCI Core, Category, Kind, Mixin and Attribute. These will be added in the final version of this work and related to the notion of sort in the location graph framework. We leave aside as well the concept of Action, that describes actions that can be executed by resources, and its relation to the location graph operational semantics.

An Entity is interpreted as a location, with a unique id interpreted as the location name, and REST interfaces modeled as roles. A Resource is an Entity that can be linked to other Resources via Links. A Link is a location which attaches to the two Resources it connects: its source and target. The connection a Link realizes is manifested by roles that attach source and target Resources to the Link that connects them. Note that, as is the case in the Fractal model with the subkell relation sc, the links relation can in fact be derived from the attachments, via roles, between Resources and Links, but it cannot be defined in that way in Alloy (i.e. it must be declared explicitly as a relation links). We also introduce the concept of Configuration, interpreted as a location graph, which is just a set of resources. The notion of configuration is not present in the OCCI Core specification [47], but is an important concept to introduce for the specification and analysis of whole systems, as argued in [42].

```
module OCCI_Core
open LocationGraphs

/** Specific notion of role for OCCI. */
sig Rest extends Role {}

let URI=Name

abstract sig Entity extends Location {
    id : one URI
} { //
    // Mapping OCCI to Location Graphs.
    //
    // The location name is the entity id.
    name = id
    //
    // Provided roles are only Rest atoms.
    Entity.@provided in Rest
    //
    // Required roles are only Rest atoms.
    Entity.@required in Rest
}

sig Resource extends Entity {
```

```
    links : set Link
} { //
    // Mapping OCCI to Location Graphs.
    //
    // One provided role of each link is a required role of this resource.
    all l : links | one r : l.@provided | r in required
}

sig Link extends Entity {
    source : one Resource,
    target : one Resource
} { //
    // OCCI constraints.
    //
    // The source of this link is the resource owning this link.
    source = ~(Resource<:links)[this]
    //
    // Mapping OCCI to Location Graphs.
    //
    // One provided role of the source entity is a required role of this link.
    one r : source.@provided | r in required
    //
    // One provided role of the target entity is a required role of this link.
    one r : target.@provided | r in required
}

sig Configuration extends LocationGraph {
    resources : set Resource
} {
    //
    // Mapping OCCI to Location Graphs.
    //
    // All the resources and their links are locations of this location graph.
    locations = resources + resources.links
}
```

A few remarks are in order. As explained in Section 2, a location can be interpreted as a component or as a relation between components. We use this versatility here in interpreting both Resource and Link as locations. The interpretation of OCCI entities as locations leads to some considerations on the OCCI model. For instance, OCCI links are limited to binary relationships between OCCI resources. In view of the interpretation of links as locations, one could consider lifting this restriction to allow n-ary relationships. It is always possible to represent n-ary relations by means of binary relations and fork algebras [46], but this may entail introducing spurious entities to satisfy the constraint. Another consideration is the runtime status of OCCI links. Any location in a location graph comes with its own behavior (in the form of Transitions, see Section 2). What kind of behavior can links exhibit at runtime, and, in particular, can they fail or disappear? The need for an operational semantics for OCCI concepts has been identified in the OCCIware project, and a proposal for an OCCI Behavioral Model has been put forward [22].

The interpretation of the OCCI core concepts given above is sound, in the sense that it has a model. One can investigate further and check that example small configurations exist and have the expected structure, as manifested by the instances of OneConfiguration, OneLink and OneResourceTargetedByTwoLinks below.

```
/** Consistency means that there exists some OCCI configuration. */
run Model {}

run OneConfiguration {
   one c : Configuration | Resource in c.resources
   Link in Configuration.resources.links
```

```
} for 5 but exactly 1 Configuration

run OneLink {} for 5 but exactly 1 Link

/** When a resource is targeted by two links then the resource has two roles, one for each link */
run OneResourceTargetedByTwoLinks {
  some r : Resource, l1, l2 : Link | l1 != l2 and l1.target = r and l2.target = r
} for 5
```

4.2 OCCI Infrastructure

This section covers the main concepts of the OCCI Infrastructure specification [44], which focuses on the description of IaaS resources. For simplicity, we again ignore typing and action information and focus only on structural elements. The infrastructure specification identifies classical cloud computing resources, in the form of Compute, Storage and Network resources, and the basic relations between them: a Compute resource can be linked to a Network resource via a NetworkInterface, and to a Storage resource via a StorageLink. Curiously, the specification does not consider the possibility of direct connection between storage and network resources. As usual, we check the soundness of the obtained model by verifying that instances exist.

```
module OCCI_Infrastructure
open OCCI_Core

sig Compute extends Resource {}

sig Network extends Resource {}

sig IPNetwork in Network {}

sig Storage extends Resource {}

sig NetworkInterface extends Link {
} { source in Compute
    target in Network }

sig IPNetworkInterface in NetworkInterface {}

sig StorageLink extends Link {
} { source in Compute
    target in Storage }

/** Consistency means that there exists some OCCI Infrastructure configuration. */
run Model {}

run OneConfigurationWithTwoComputeOneNetworkOneStorage {
  Resource in Configuration.resources
} for 25 but exactly 1 Configuration, exactly 2 Compute, exactly 2 Storage,
exactly 2 StorageLink, exactly 1 Network, exactly 2 NetworkInterface
```

4.3 OCCI Platform

This section covers the main concepts of the OCCI Platform specification [45], which focuses on the description of PaaS elements. The main resource introduced here is Application. Each Application can be composed of multiple Components. The composition is manifested by ComponentLinks that connect an Application to each of its Components.

```
module OCCI_Platform
open OCCI_Core

/*****************************************************************************
 * This is a simplification of OCCI Platform, which does not include
 * attributes and actions.
 *****************************************************************************/

sig Application extends Resource {}

sig Component extends Resource {}

sig ComponentLink extends Link {
} {
  source in Application + Component
  target in Component
}

/** Consistency means that there exists some OCCI Platform configuration. */
run Model {}

run OneConfigurationWithOneApplicationTwoComponent {
  Resource in Configuration.resources
  ComponentLink in Application.links
} for 20 but exactly 1 Configuration, exactly 1 Application, exactly 2 Component, exactly 2 ComponentLink
```

5 The TOSCA model

We present in this section an interpretation of a key subset of the TOSCA model in location graphs. The TOSCA specification defines a language for describing the topology and orchestration of cloud applications. For an introduction to, and motivation for TOSCA, the reader is referred to [26].

5.1 TOSCA Core

We present in this section an interpretation of key concepts that underlie the TOSCA language [19]. For the sake of simplicity, we leave aside a number of notions introduced in the TOSCA specification, including its notions of type, property, interface, and template. The core TOSCA concepts are Topology, Node and Relationship. A Topology describes a configuration, that comprises a set of Nodes related by Relationships. A Node corresponds to a component that offers capabilities and has requirements for services offered by other nodes. A Relationship corresponds to a binary relation between components, that maps a Requirement with a Capability. Nodes and Relationships are interpreted as locations, while a Topology is interpreted as a location graph. A Requirements and a Capability are interpreted as roles of a Node, in required and provided positions, respectively.

```
module TOSCA
open LocationGraphs

sig Topology extends LocationGraph {
  nodes : set Node,
  relationships : set Relationship
} {
  // Mapping TOSCA to Location Graphs.
  //
  // nodes and relationships are locations of this location graph.
  locations = nodes + relationships + nodes.requirements.relationship
}

abstract sig Node extends Location {
  requirements : set Requirement,
  capabilities : set Capability
} {
  // Mapping TOSCA to Location Graphs.
  //
  // Requirements are required roles of this location.
  required = requirements
  //
  // Capabilities are provided roles of this location.
  provided = capabilities
}

abstract sig Requirement extends Role {
  relationship: one Relationship
} {
  // TOSCA constraints.
  //
  // The source of the relationship is this requirement.
  relationship.source = this
  //
  // One requirement is owned by only one node.
  one node
}

abstract sig Capability extends Role {}
{
```

```
  // TOSCA constraints.
  //
  // One capability is owned by only one node.
  one node
}

abstract sig Relationship extends Location {
  source : one Requirement,
  target: one Capability
} {
  // Mapping TOSCA to Location Graphs.
  //
  // The source requirement is a provided role of this location.
  provided = source
  //
  // The target capability is a required role of this location.
  required = target
}
```

The core concepts of TOSCA are very similar to those of OCCI, with nodes playing the same role as resources, and relationships corresponding to links. This intuition will be made formal in Section 5.3. There are slight differences in the Alloy formalization, for instance the fact that OCCI Resources have links whereas the reference to Relationships that bind TOSCA nodes is only found in Requirement, via the relationship relation. In large part this results from design choices in the Alloy specifications to more directly reflect the informal ones.

We specify below self-explanatory functions and predicates that will simplify the writing of node types in the next section.

```
/* Return the node owning a given requirement. */
fun node[req: one Requirement] : Node {
  ~(Node<:requirements)[req]
}

/* Return the node owning a given capability. */
fun node[cap: one Capability] : Node {
  ~(Node<:capabilities)[cap]
}

/** Check that the capability targeted by the given requirement is of given capability types. */
pred capability[requirement: Requirement, capabilities: set Capability]
{
  requirement.relationship.target in capabilities
}

/** Check that the capability targeted by the given requirement is owned by given node types. */
pred node[requirement: Requirement, nodes: set Node]
{
  requirement.relationship.target.node in nodes
}

/** Check that the requirements targeting the given capability are owned by given node types. */
pred valid_source_types[cap: Capability, nodes: set Node]
{
  ~(Relationship<:target)[cap].source.node in nodes
}
```

The obtained model is consistent, in the sense that there exist TOSCA topologies, and one can check the existence of topologies with expected features such as OneTopologyWithTwoNodeOneRelationship.

```
/*****************************************************************************
```

```
 * Consistency Property.
 *****************************************************************************/

/** Consistency means that there exists some TOSCA topology. */
run Model {}

run OneTopologyWithTwoNodeOneRelationship
{
  Node in Topology.nodes
  Relationship in Topology.relationships
} for 10 but exactly 1 Topology, exactly 2 Node, exactly 1 Relationship
```

5.2 TOSCA Types

We present in this section a formal specification of the Capability, Relationship, and Node Types defined in the TOSCA specification documenting normative types [21], i.e. various kinds of nodes and relationships to be supported by compliant TOSCA environments. The specification was written in a systematic way, by following the YAML description in [20]. For simplicity, we focus only on certain key structural types, and leave aside datatypes, artefacts, interfaces and properties that appear in the specification. We leave in comments excerpts from the YAML specification to clarify the relation with our Alloy specification. Comments also signal one error in the YAML description (which we fixed), and several redundancies in the original TOSCA specification.

```
module TOSCA_types

open TOSCA

/*****************************************************************************
 * Capability Types.
 *****************************************************************************/
/*
  tosca.capabilities.Root:
    description:
      This is the default (root) TOSCA Capability Type definition that all other TOSCA Capability Types derive from.
*/
sig Capability_Root extends Capability {}

/*
  tosca.capabilities.Node:
    derived_from: tosca.capabilities.Root
    description:
      The Node capability indicates the base capabilities of a TOSCA Node Type.
*/
sig Capability_Node extends Capability_Root {}

/*
  tosca.capabilities.Container:
    derived_from: tosca.capabilities.Root
    description:
      The Container capability, when included on a Node Type or Template definition, indicates that the node can act as a container for (or a host for)
      one or more other declared Node Types.
*/
sig Container extends Capability_Root {}

/*
  tosca.capabilities.Endpoint:
    derived_from: tosca.capabilities.Root
    description:
```

```
        This is the default TOSCA type that should be used or extended to define a network endpoint capability.
        This includes the information to express a basic endpoint with a single port or a complex endpoint with multiple ports.
        By default the Endpoint is assumed to represent an address on a private network unless otherwise specified.
*/
sig Endpoint extends Capability_Root {}

/*
  tosca.capabilities.Endpoint.Public:
    derived_from: tosca.capabilities.Endpoint
    description:
      This capability represents a public endpoint which is accessible to the general internet (and its public IP address ranges).
      This public endpoint capability also can be used to create a floating (IP) address that the underlying network assigns from a pool allocated fro
      the application's underlying public network. This floating address is managed by the underlying network such that can be routed an applicati
      private address and remains reliable to internet clients.
*/
sig Endpoint_Public extends Endpoint {}

/*
  tosca.capabilities.Endpoint.Admin:
    derived_from: tosca.capabilities.Endpoint
    description:
      This is the default TOSCA type that should be used or extended to define a specialized administrator endpoint capability.
*/
sig Endpoint_Admin extends Endpoint {}

/*
  tosca.capabilities.Endpoint.Database:
    derived_from: tosca.capabilities.Endpoint
    description:
      This is the default TOSCA type that should be used or extended to define a specialized database endpoint capability.
*/
sig Endpoint_Database extends Endpoint {}

/*
  tosca.capabilities.Attachment:
    derived_from: tosca.capabilities.Root
    description:
      This is the default TOSCA type that should be used or extended to define an attachment capability of a (logical) infrastructure device node
      (e.g., BlockStorage node).
*/
sig Attachment extends Capability_Root {}

/*
  tosca.capabilities.OperatingSystem:
    derived_from: tosca.capabilities.Root
    description:
      This is the default TOSCA type that should be used to express an Operating System capability for a node.
*/
sig OperatingSystem extends Capability_Root {}

/*
  tosca.capabilities.Scalable:
    derived_from: tosca.capabilities.Root
    description:
      This is the default TOSCA type that should be used to express a scalability capability for a node.
*/
sig Scalable extends Capability_Root {}

/*
  tosca.capabilities.network.Bindable:
    derived_from: tosca.capabilities.Node
*/
```

```
sig Bindable extends Capability_Node {}

/*
   tosca.capabilities.network.Linkable:
      derived_from: tosca.capabilities.Node
*/
sig Linkable extends Capability_Node {}


/*********************************************************************
 * Relationship Types.
 *********************************************************************/
/*
   tosca.relationships.Root:
      description:
         The TOSCA root Relationship Type all other TOSCA base Relationship Types derive from
*/
sig Relationship_Root extends Relationship {}

/*
   tosca.relationships.DependsOn:
      derived_from: tosca.relationships.Root
      description:
         This type represents a general dependency relationship between two nodes.
      valid_target_types: [ tosca.capabilities.Node ]
*/
sig DependsOn extends Relationship_Root {
} {
   target in Capability_Node
}

/*
   tosca.relationships.HostedOn:
      derived_from: tosca.relationships.Root
      description:
         This type represents a hosting relationship between two nodes.
      valid_target_types: [ tosca.capabilities.Container ]
*/
sig HostedOn extends Relationship_Root {
} {
   target in Container
}

/*
   tosca.relationships.ConnectsTo:
      derived_from: tosca.relationships.Root
      description:
         This type represents a network connection relationship between two nodes.
      valid_target_types: [ tosca.capabilities.Endpoint ]
*/
sig ConnectsTo extends Relationship_Root {
} {
   target in Endpoint
}

/*
   tosca.relationships.AttachTo: // BUG -> OASIS: Is AttachesTo, see page 156
      derived_from: tosca.relationships.Root
      valid_target_types: [ tosca.capabilities.Attachment ]
*/
sig AttachesTo extends Relationship_Root {
} {
```

```
  target in Attachment
}

/*
  tosca.relationships.RoutesTo:
    derived_from: tosca.relationships.ConnectsTo
    description:
      This type represents an intentional network routing between two Endpoints in different networks.
// OASIS: Following seems useless as ConnectsTo already restricts valid target types to Endpoint.
    valid_target_types: [ tosca.capabilities.Endpoint ]
*/
sig RoutesTo extends ConnectsTo {
} {
// TO CHECK: Following seems useless as ConnectsTo already restricts valid target types to Endpoint.
  target in Endpoint
}

/*
  tosca.relationships.network.LinksTo:
    derived_from: tosca.relationships.DependsOn
    valid_target_types: [ tosca.capabilities.network.Linkable ]
*/
sig LinksTo extends DependsOn {
} {
  target in Linkable
}

/*
  tosca.relationships.network.BindsTo:
    derived_from: tosca.relationships.DependsOn
    valid_target_types: [ tosca.capabilities.network.Bindable ]
*/
sig BindsTo extends DependsOn {
} {
  target in Bindable
}

/*****************************************************************************
 * Node Types.
 *****************************************************************************/
/*
  tosca.nodes.Root:
    description:
      This is the default (root) TOSCA Node Type that all other TOSCA nodes should extends.
      This allows all TOSCA nodes to have a consistent set of features for modeling and management
      (e.g, consistent definitions for requirements, capabilities, and lifecycle interfaces).
    capabilities:
      feature:
        type: tosca.capabilities.Node
    requirements:
      - dependency:
          capability: tosca.capabilities.Node // OASIS: Is it useless as already defined in DependsOn?
          node: tosca.nodes.Root
          relationship: tosca.relationships.DependsOn
          occurrences: [ 0, UNBOUNDED ]
*/
sig Node_Root extends Node
{
  feature : one Capability_Node,
  /* - dependency: occurrences: [ 0, UNBOUNDED ] */
  dependency: set Requirement
} {
```

```
    feature in capabilities
    dependency in requirements
    /* — dependency: capability: tosca.capabilities.Node */
    dependency.capability[Capability_Node] // TO CHECK: This constraint is useless as already defined in DependsOn.
    /* — dependency: node: tosca.nodes.Root */
    dependency.node[Node_Root]
    /* — dependency: relationship: tosca.relationships.DependsOn */
    dependency.relationship in DependsOn
}

/*
    tosca.nodes.Compute:
      derived_from: tosca.nodes.Root
      description:
        The TOSCA Compute node represents one or more real or virtual processors of software applications or services along with other essential local resources
        Collectively, the resources the compute node represents can logically be viewed as a (real or virtual) "server".
      requirements:
        — local_storage:
            capability: tosca.capabilities.Attachment // OASIS: Is it useless as already defined in AttachesTo?
            node: tosca.nodes.BlockStorage
            relationship: tosca.relationships.AttachesTo
            occurrences: [0, UNBOUNDED]
      capabilities:
        host:
          type: tosca.capabilities.Container
          valid_source_types: [tosca.nodes.SoftwareComponent]
        os:
          type: tosca.capabilities.OperatingSystem
        endpoint:
          type: tosca.capabilities.Endpoint.Admin
        scalable:
          type: tosca.capabilities.Scalable
        binding:
          type: tosca.capabilities.network.Bindable
*/
sig Compute extends Node_Root
{
    /* — local_storage: occurrences: [ 0, UNBOUNDED ] */
    local_storage: set Requirement,
    host: one Container,
    os: one OperatingSystem,
    endpoint: one Endpoint_Admin,
    scalable: one Scalable,
    binding: one Bindable
} {
    local_storage in requirements
    /* — local_storage: capability: tosca.capabilities.Attachment */

    local_storage.capability[Attachment] // TO CHECK: This constraint is useless as already defined in AttachesTo.
    /* — local_storage: node: tosca.nodes.BlockStorage */
    local_storage.node[BlockStorage]
    /* — local_storage: relationship: tosca.relationships.AttachesTo */
    local_storage.relationship in AttachesTo

    host in capabilities
    // host: valid_source_types: [tosca.nodes.SoftwareComponent]
    host.valid_source_types[SoftwareComponent]
    os in capabilities
    endpoint in capabilities
    scalable in capabilities
    binding in capabilities
}
```

```
/*
  tosca.nodes.SoftwareComponent:
    derived_from: tosca.nodes.Root
    description: The TOSCA SoftwareComponent node represents a generic software component that can be managed and run by a TOSCA Compu
    requirements:
      - host:
          capability: tosca.capabilities.Container // OASIS: Is it useless as already defined in HostedOn?
          node: tosca.nodes.Compute
          relationship: tosca.relationships.HostedOn
*/
sig SoftwareComponent extends Node_Root {
  host: one Requirement
} {
  host in requirements

  /* - host: capability: tosca.capabilities.Container */
  host.capability[Container] // TO CHECK: This constraint is useless as already defined in HostedOn.
  /* - host: node: tosca.nodes.Compute */
  host.node[Compute]
  /* - host: relationship: tosca.relationships.HostedOn */
  host.relationship in HostedOn
}

/*
  tosca.nodes.WebServer:
    derived_from: tosca.nodes.SoftwareComponent
    description:
      This TOSCA WebServer Node Type represents an abstract software component or service that is capable of hosting and providing managemer
      for one or more WebApplication nodes.
    capabilities:
      # Private, layer 4 endpoints
      data_endpoint: tosca.capabilities.Endpoint
      admin_endpoint: tosca.capabilities.Endpoint.Admin
      host:
        type: tosca.capabilities.Container
        valid_source_types: [ tosca.nodes.WebApplication ]
*/
sig WebServer extends SoftwareComponent {
  data_endpoint: one Endpoint,
  admin_endpoint: one Endpoint_Admin,
  // host: one Container // host is already a field of sig SoftwareComponent.
  chost: one Container
} {
  data_endpoint in capabilities
  admin_endpoint in capabilities
  chost in capabilities
  /* host: valid_source_types: [ tosca.nodes.WebApplication ] */
  chost.valid_source_types[WebApplication]
}

/*
  tosca.nodes.WebApplication:
    derived_from: tosca.nodes.Root
    description:
      The TOSCA WebApplication node represents a software application that can be managed and run by a TOSCA WebServer node.
      Specific types of web applications such as Java, etc. could be derived from this type.
    capabilities:
      app_endpoint:
        type: tosca.capabilities.Endpoint
    requirements:
      - host:
```

```
        capability: tosca.capabilities.Container // OASIS: Is it useless as already defined in HostedOn?
        node: tosca.nodes.WebServer
        relationship: tosca.relationships.HostedOn
*/
sig WebApplication extends Node_Root {
  app_endpoint: one Endpoint,
  host: one Requirement
} {
  app_endpoint in capabilities
  host in requirements
  /* – host: capability: tosca.capabilities.Container */
  host.capability[Container] // TO CHECK: This constraint is useless as already defined in HostedOn.
  /* – host: node: tosca.nodes.WebServer */
  host.node[WebServer]
  /* – host: relationship: tosca.relationships.HostedOn */
  host.relationship in HostedOn
}

/*
  tosca.nodes.DBMS:
    derived_from: tosca.nodes.SoftwareComponent
    description:
      The TOSCA DBMS node represents a typical relational, SQL Database Management System software component or service.
    capabilities:
      host:
        type: tosca.capabilities.Container
        valid_source_types: [ tosca.nodes.Database ]
*/
sig DBMS extends SoftwareComponent {
  // host: one Container // host is already a field of sig SoftwareComponent.
  chost: one Container
} {
  chost in capabilities
  /* host: valid_source_types: [ tosca.nodes.Database ] */
  chost.valid_source_types[Database]
}

/*
  tosca.nodes.Database:
    derived_from: tosca.nodes.Root
    description:
      The TOSCA Database node represents a logical database that can be managed and hosted by a TOSCA DBMS node.
    requirements:
      – host: tosca.capabilities.Container // OASIS: Is it useless as already defined in HostedOn?
        # node: tosca.nodes.DBMS // OASIS: Why is it commented?
        relationship: tosca.relationships.HostedOn
    capabilities:
      database_endpoint:
        type: tosca.capabilities.Endpoint.Database
*/
sig Database extends Node_Root {
  host: one Requirement,
  database_endpoint: one Endpoint_Database
} {
  host in requirements
  /* – host: tosca.capabilities.Container */
  host.capability[Container] // TO CHECK: This constraint is useless as already defined in HostedOn.
  /* – host: # node: tosca.nodes.DBMS */
  host.node[DBMS]
  /* – host: relationship: tosca.relationships.HostedOn */
  host.relationship in HostedOn
```

```
  database_endpoint in capabilities
}

/*
  tosca.nodes.ObjectStorage:
    derived_from: tosca.nodes.Root
    capabilities:
      storage_endpoint:
        type: tosca.capabilities.Endpoint
*/
sig ObjectStorage extends Node_Root {
  storage_endpoint: one Endpoint
} {
  storage_endpoint in capabilities
}

/*
  tosca.nodes.BlockStorage:
    abstract: true
    derived_from: tosca.nodes.Root
    description:
      The TOSCA BlockStorage node currently represents a server−local block storage device (i.e., not shared)
      offering evenly sized blocks of data from which raw storage volumes can be created.
    tags:
      icon: /images/volume.png
    capabilities:
      attachment:
        type: tosca.capabilities.Attachment
*/
abstract sig BlockStorage extends Node_Root {
  attachment: one Attachment
} {
  attachment in capabilities
}

/*
  tosca.nodes.Container.Runtime:
    derived_from: tosca.nodes.SoftwareComponent
    description:
      The TOSCA Container Runtime node represents operating system−level virtualization technology used to run multiple application services
      single Compute host.
    capabilities:
      host:
        type: tosca.capabilities.Container
      scalable:
        type: tosca.capabilities.Scalable
*/
sig Container_Runtime extends SoftwareComponent {
  // host: one Container // host is already a field of sig SoftwareComponent.
  chost: one Container,
  scalable: one Scalable
} {
  chost in capabilities
  scalable in capabilities
}

/*
  tosca.nodes.Container.Application:
    derived_from: tosca.nodes.Root
    description:
      The TOSCA Container Application node represents an application that requires Container−level virtualization technology.
    requirements:
```

```
        host:
            capability: tosca.capabilities.Container // OASIS: Is it useless as already defined in HostedOn?
            node: tosca.nodes.Container // BUG -> OASIS: Container.Runtime instead of Container? page 173
            relationship: tosca.relationships.HostedOn
*/
sig Container_Application extends Node_Root {
  host: one Requirement
} {
  host in requirements
  /* -- host: capability: tosca.capabilities.Container */
  host.capability[Container] // TO CHECK: It is useless as already defined in HostedOn.
  /* -- host: node: tosca.nodes.Container */
  host.node[Container_Runtime] // Correction: Container_Runtime instead of Container.
 /* -- host: relationship: tosca.relationships.HostedOn */
  host.relationship in HostedOn
}

/*
  tosca.nodes.LoadBalancer:
    derived_from: tosca.nodes.Root
    description:
      The TOSCA Load Balancer node represents logical function that be used in conjunction with a Floating Address to distribute an application's
      traffic (load) across a number of instances of the application (e.g., for a clustered or scaled application).
    properties:
      algorithm:
        type: string
        required: false
        status: experimental
    capabilities:
      client:
        type: tosca.capabilities.Endpoint.Public
        occurrences: [0, UNBOUNDED]
        description: the Floating (IP) client's on the public network can connect to
    requirements:
      - application:
          capability: tosca.capabilities.Endpoint // OASIS: Is it useless as already defined in RoutesTo?
          relationship: tosca.relationships.RoutesTo
          occurrences: [0, UNBOUNDED]
          description: Connection to one or more load balanced applications
*/
sig LoadBalancer extends Node_Root {
  client: set Endpoint_Public,
  application: set Requirement
} {
  client in capabilities
  application in requirements
  /* -- application: capability: tosca.capabilities.Endpoint */
  application.capability[Endpoint] // TO CHECK: This constraint is useless as already defined in RoutesTo.
  /* -- application: relationship: tosca.relationships.RoutesTo */
  application.relationship in RoutesTo
}

/*
  tosca.nodes.network.Network:
    derived_from: tosca.nodes.Root
    capabilities:
      link:
        type: tosca.capabilities.network.Linkable
*/
sig Network extends Node_Root {
  link: one Linkable
} {
```

```
  link in capabilities
}

/*
  tosca.nodes.network.Port:
    derived_from: tosca.nodes.Root
      requirements:
        - link:
            capability: tosca.capabilities.network.Linkable
          relationship: tosca.relationships.network.LinksTo
        - binding:
            capability: tosca.capabilities.network.Bindable
            relationship: tosca.relationships.network.BindsTo
*/
sig Port extends Node_Root {
  link: one Requirement,
  binding: one Requirement
} {
  link in requirements
  /* - link: capability: tosca.capabilities.network.Linkable */
  link.capability[Linkable] // TO CHECK: This constraint is useless as already defined in LinksTo.
  /* - link: relationship: tosca.relationships.network.LinksTo */
  link.relationship in LinksTo

  binding in requirements
  /* - binding: capability: tosca.capabilities.network.Bindable */
  binding.capability[Bindable] // TO CHECK: This constraint is useless as already defined in BindsTo.
  /* - binding: relationship: tosca.relationships.network.BindsTo */
  binding.relationship in BindsTo
}
```

As usual we check for the consistency of the specification. We also check that examples in the TOSCA YAML specification [21] can be obtained as topologies compliant with our Alloy specification.

```
/*****************************************************************************
 * Consistency Property.
 *****************************************************************************/

/** Consistency means that there exists some TOSCA topology. */
run Model {} for 10

run OneTopologyWithTwoNodeOneRelationship {
  Node in Topology.nodes
  Relationship in Topology.relationships
} for 10 but exactly 1 Topology, exactly 2 Node, exactly 1 Relationship

/** TOSCA Simple Profile YAML v1.0 cs01 page 12 */
run TOSCA_Example_2_1 {
  Topology.nodes = Compute
} for 10 but exactly 1 Topology, exactly 1 Compute, exactly 5 Capability

/** TOSCA Simple Profile YAML v1.0 cs01 page 16 */
run TOSCA_Example_2_2 {
  Topology.nodes = Compute + DBMS
  Topology.relationships = HostedOn
} for 20 but exactly 1 Topology, exactly 1 Compute, exactly 1 DBMS,
  exactly 1 HostedOn, exactly 7 Capability

/** TOSCA Simple Profile YAML v1.0 cs01 page 19 */
run TOSCA_Example_2_4 {
  Topology.nodes = Compute + DBMS + Database
  Topology.relationships = HostedOn
```

```
} for 20 but exactly 1 Topology, exactly 1 Compute, exactly 1 DBMS,
  exactly 1 Database, exactly 2 HostedOn, exactly 9 Capability

/** TOSCA Simple Profile YAML v1.0 cs01 page 19 */
run TOSCA_Example_2_5 {
  Topology.nodes = Compute + DBMS + Database + WebServer + WebApplication
  Topology.relationships = HostedOn
} for 40 but exactly 1 Topology, exactly 2 Compute, exactly 1 DBMS, exactly 1 Database,
  exactly 1 WebServer, exactly 1 WebApplication, exactly 4 HostedOn, exactly 20 Capability, exactly 24 Role

/** TOSCA Simple Profile YAML v1.0 cs01 page 178 */
run TOSCA_Example_7_2 {
  Topology.nodes = Compute + Network + Port
  Topology.relationships = LinksTo + BindsTo
} for 20 but exactly 1 Topology, exactly 1 Compute, exactly 1 Port, exactly 1 Network,
  exactly 1 BindsTo, exactly 1 LinksTo, exactly 7 Capability
```

5.3 Relating TOSCA and OCCI

We present in this section a mapping between the TOSCA and OCCI core concepts. This mapping formalizes the intuition expressed above that the TOSCA and OCCI core concepts are similar. It can be defined very easily because of the interpretation of the two models in the location graph pivot model. The mapping establishes a direct relation between TOSCA and OCCI core concepts as defined in Sections 5.1 and 4.1, respectively, which is summarized in Table 1.

TOSCA Concepts	**OCCI Concepts**
Topology - nodes - relationships	Configuration - resources - resources.links
Node - name - requirements	Resource - id - links
Relationship - name - source - target	Link - id - source - target
Requirement	Rest
Capability	Rest

Table 1: Mapping TOSCA and OCCI core concepts

The mapping is formally defined via a base type Mapping, and three predicates that characterize a well-formed Mapping. Mapping comes equipped with the following relations: topology and configuration that identify the TOSCA topology and the OCCI configuration being mapped; node2resource and relationship2link that identify how nodes and relationships in the given TOSCA configuration are mapped to resources and links, respectively, in the given OCCI configuration.

```
module TOSCA2OCCI
open TOSCA
open OCCI_Core


sig Mapping {
```

```
  topology : one Topology,
  configuration : one Configuration,
  node2resource : Node -> Resource,
  relationship2link: Relationship -> Link
}

pred topology2configuration[mapping : one Mapping, top: one Topology, conf: one Configuration]
{
  //
  // Sets up the mapping.
  //
  mapping.topology = top
  mapping.configuration = conf
  //
  // Iterates over all nodes of the topology.
  //
  #top.nodes = #conf.resources
  all node : top.nodes {
    one resource : conf.resources {
      mapping.node2resource[node, resource]
    }
  }
  //
  // Iterates over all relationships of the topology.
  //
  all relationship : top.relationships {
    one link : conf.resources.links {
      mapping.relationship2link[relationship, link]
    }
  }
}

pred node2resource[mapping : one Mapping, node: one Node, resource: one Resource]
{
  //
  // Sets up the mapping.
  //
  (node -> resource) in mapping.node2resource
  //
  // The resource id is the node name.
  //
  resource.id = node.name
  //
  // Iterates over all requirements of the node.
  //
  #node.requirements = #resource.links
  all requirement : node.requirements {
    one link : resource.links {
      mapping.relationship2link[requirement.relationship, link]
    }
  }
}

pred relationship2link[mapping : one Mapping, relationship: one Relationship, link: one Link]
{
  //
  // Sets up the mapping.
  //
  (relationship -> link) in mapping.relationship2link
  //
  // The link id is the node name.
  //
```

```
    link.id = relationship.name
    //
    // Set the resource source and target of the link.
    //
    link.source = mapping.configuration.getResource[relationship.source.node.name]
    link.target = mapping.configuration.getResource[relationship.target.node.name]
}
```

One can check that this definition is consistent, i.e. that there exist Mappings that verify the well-formedness property defined by predicate topology2configuration.

```
/*****************************************************************************
 * Consistency Property
 *****************************************************************************/

run Consistency
{
  Topology.nodes = Node
  one mapping: Mapping, topology : Topology, configuration : Configuration |
    mapping.topology2configuration[topology, configuration]
} for 10 but exactly 1 Mapping, exactly 1 Topology, exactly 1 Configuration

run MappingFiveNodesRelationships
{
  Topology.nodes = Node
  one mapping: Mapping, topology : Topology, configuration : Configuration |
    mapping.topology2configuration[topology, configuration]
} for 20 but exactly 1 Mapping, exactly 1 Topology, exactly 5 Node, exactly 5 Relationship, exactly 1 Configuration
```

Importantly, we can check that the mapping just defined is structure-preserving, i.e. that the different structural elements in one model are mapped to identified structural elements in the other model. In our case, this means in particular that the mapping preserves the number of locations and the location names involved in the mapped TOSCA topology and OCCI configuration. This is an important property to obtain when relating different models: in more pedantic terms, we can say that we expect a mapping between models to be not just an arbitrary relation but a form of *homomorphism*, where what counts as structure (configuration, component) in one model is consistently mapped on what counts as similar structure in the other model. In turn, this "homomorphy" property is crucial to ensure that one can indeed manipulate instances of both models in a consistent way, e.g. that one change induced in a configuration described in one model yields an equivalent change in the mapped configuration in the other model.

```
/*****************************************************************************
 * Homomorphy Properties
 *****************************************************************************/

 /**
   * Each TOSCA Topology maps to an OCCI Configuration.
   */
 assert EachTopologyMapsToConfiguration
 {
   all t: Topology, m: Mapping, c: Configuration {
     m.topology2configuration[t, c] implies m.topology = t and m.configuration = c
   }
 }
 check EachTopologyMapsToConfiguration for 15 expect 0

 /**
   * The TOSCA to OCCI Mapping conserves the number of locations.
   */
 assert MappingPreserveNumberOfLocations
 {
```

```
  all m: Mapping, t: Topology, c: Configuration {
    m.topology2configuration[t, c] implies #t.locations = #c.locations
  }
}
check MappingPreserveNumberOfLocations for 10 expect 0

/**
  * The TOSCA to OCCI Mapping conserves all the location names.
  */
assert MappingPreserveLocationNames
{
  all m: Mapping, t: Topology, c: Configuration {
    m.topology2configuration[t, c] implies t.locations.name = c.locations.name
  }
}
check MappingPreserveLocationNames for 8 expect 0
```

6 The Docker Compose model

We present in this section a formal specification of Docker configurations conforming to our computational model.

6.1 An overview of Docker Compose

The Compose file is a Yaml file defining services, networks, and volumes for a Docker application. The Compose file reference may be found in [16]. There, Docker compose is defined as a tool for defining and running multi-container Docker applications. With Compose, one can use a Yaml file to configure the application's services. Then, with a single command, all services are created and launched according to the configuration. Compose works in all environments: production, staging, development, testing, as well as CI workflows. The structure of the Compose Yaml file itself is described in [16]. There, the user defines the services that make up the applications so they can be run together in an isolated environment. In the following, we show a simple example of docker-compose.yml, where the services making up the application are mysql and wordpress.

```
version: "3.7"

services:
  wordpress:
    image: wordpress
    ports:
      - "8080:80"
    networks:
       overlay
         ipv4_address: 172.16.238.10
         ipv6_address: 2001:3984:3989::10
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: vip

  mysql:
    image: mysql
    volumes:
       - db-data:/var/lib/mysql/data
    networks:
       - overlay
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: dnsrr

volumes:
  db-data:

networks:
  overlay:
```

In the next subsection, we will give a hint on the definition of the different attributes along with the related docker specification.

6.2 A Location-Graph Based Specification of Docker Compose Configurations

We ensure that our Alloy specification is conform to the Docker compose file reference, as it was written in systematic way, by following the grammar expressed in the JSON (JavaScript Object Notation) de-

scription used in the Docker compose toolset. The source code of Docker Compose can be found in [8]. It contains JSON schemas for the different versions of docker compose (from version 1 to version 3.7 up to the date we are writing this document). JSON is a lightweight data-interchange format. It is easy for humans to read and write and is easy for machines to parse and generate. Every docker compose Yaml file should respect the grammar defined in the JSON schema related to its declared version. In this work, we consider the latest version, 3.7, of Docker Compose.

A Docker composition is defined by :

- a version number;
- a set of services;
- a set of networks;
- a set of volumes;
- a set of secrets.
  A secret is *a blob of data, such as a password, SSH private key, SSL certificate, or another piece of data that should not be transmitted over a network or stored unencrypted in a Dockerfile or in the application's source code.*
- a set of configs.
  A config *allows to store non-sensitive information, such as configuration files, outside a service's image or running containers. Configs operate in a similar way to secrets, except that they are not encrypted at rest and are mounted directly into the container's filesystem without the use of RAM disks.*

We note that both secrets and configs are only available to swarm services, not to standalone containers. Therefore, there are two options for the translator: either targeting the swarm mode or the standalone containers. In the latter case, configs and secrets in the compose file are ignored (this is the case for version 3.3 and later).
In an opposite manner, there are some attributes that have not to be translated when targeting swarm mode. This is the case for the build property at service level. This option is ignored when deploying a stack in swarm mode with a (version 3) Compose file. The docker stack command accepts only pre-built images.
We note that the above networks (resp.volumes, secrets, configs) attribute of a composition may be referenced by the services. In the docker community, they are called the **top-level** networks (resp. top-level volumes, top-level secrets, top-level configs).

```
/*************************************************************************
DOCKER composition.
*************************************************************************/
sig Composition extends LG/LocationGraph
{
  dockerComposeVersion: one DockerComposeVersion,
  services : set Service,
  networks: set Network,
  volumes: set Volume,
  secrets: set Secret,
  configs: set Config
} {
  // Locations are services, networks and volumes.
  locations = services + networks + volumes + secrets
distinct_names[services]
}
```

A Docker service has containers, shares networks and volumes with other services, and can depend on other services. In the Docker Compose Yaml files, the networks property of a service refers to a top-level network and contains connection features between this service and the top level volume. For example, in the Yaml composition defined in Subsection 6.1, the two services share the network overlay. This network is defined also as a top level network. Besides, the Yaml description specifies the IP address on which the overlay service is connected to the overlay network. In our LG interpretation, both services and networks are defined as locations. A service is connected to every network on a role, defined in Alloy with the NetworkRole signature.

```
sig Service extends LG/Location
{
  containers: set Container,
  networks: set NetworkRole,
  volumes: set VolumeRole,
  secrets: set SecretRole,
  configs: set ConfigRole,
  depends_on: set Role,
  on_which_depend: set Role, // point of attachement with other services which depend on this
  deploy: lone Deployment,
  build: lone Build,
  cap_add : set string,
  cap_drop:set string ,
  cgroup_parent: lone string,
  command: set string,
  container_name: lone string,
  credential_spec_file: lone string,
  credential_spec_registry: lone string,
  devices: set string,
  dns: set string, //Custom DNS servers. The format of each string is DNS ip address
  dns_search: set string, // the format is domain name
  tmpfs: set string,
  tty: lone boolean,
  ulimit: lone Ulimit,
  security_opt: set string,
  shm_size: lone any,
  entrypoint: set string,
  env_file: set string, // format of each string: Path to a file
  environment: lone Dict,
  expose: lone string, //Format expose
  ports: set string,
  external_links: set string,
  extra_hosts: set string, //format: hostname mappings
  healthcheck: lone Healthcheck,
  hostname: lone string, // hostname where the service sits
  image: lone string, // format: a repository/tag or a partial image ID.

  //the following 3 attributes are relative to restart_policy
  privileged: lone boolean,
  read_only: lone boolean,
  restart: lone string,

  //logging config
  log_driver: lone String,

} {
   #containers >1 implies no container_name

  on_which_depend in provided

  // Required roles link to networks, volumes, depends_on.
  required = networks + volumes + depends_on+secrets

  restart in "no"+"always"+"on-failure"+"unless-stopped"
}
```

We define the characteristics of the connection of the service to the network as attributes of the NetworkRole. It is necessary to model them at role level rather than as attributes of the Network itself since many services may have access to the same Network.

```
sig NetworkRole extends LG/Role
{
  aliases : lone List,
  ipv4_address: lone string,
  ipv6_address: lone string
}
```

The service and the network are bound at the role level. The bind predicate expresses generally that a role belongs to a given location. In the Service.network predicate, it is used to express that the service on which the predicate applies is connected to a network and they are bound on the role named spec.

```
pred bind[roles: set LG/Role, location: one LG/Location]
{
  one role : location.provided {
    role in roles
  }
}


pred Service.network[network: one Network , spec: one NetworkRole]
{
  spec in this.networks and bind[spec, network]
}

// The following predicate expresses that a service is connected to a network without specifying
//the NetworkRole on which they are connected. Therefore, no information about aliases and ip address can be reflected.
//The translator always generates rather the complete above version


pred Service.network[network: one Network]
{
  bind[this.networks, network]
}
```

In a similar manner, a service has access to volumes. Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. The abstract signature Volume_spec models a volume in general. There are two types of volumes: Volumes specific to a given service or top-level volumes. In the first case, the user may choose to mount a host path as part of a definition for a single service, in which case there is no need to define it inside the top level volumes key. The signature ServiceVolume, inheriting from Volume, is used to specify such a volume. But, in the case where a volume is reused across multiple services, a named volume should be defined in the top-level volumes key. For this case, the signature Volume represents the top-level volume. In our specification, a service is connected to every volume by a role, defined in Alloy with the VolumeRole signature.

```
abstract sig Volume_spec extends LG/Location
{} {
  // Provided roles are volume roles.
  provided in VolumeRole
  // At least one provided role.
  some provided
  // No required roles.
  no required
}
sig Volume extends Volume_spec{} // a top−level volume

sig ServiceVolume extends Volume_spec{} // a volume wich is not defined at top level
```

Following Docker terminology, there are two signatures to model the characteristics of volumes at service level, which are VolumeSpec and MountSpec and both extend VolumeRole.

```
abstract sig VolumeRole extends LG/Role{}


sig VolumeSpec extends VolumeRole{
 external: lone string+Volume,
 internal: one string,
 mode:lone string
}
{
mode in ”rw”+”ro”
}

sig MountSpec extends VolumeRole{
 type: lone string,
 source: lone string,
 target: lone string,
 read_only: lone boolean,
 consistency: lone string,
 bind_propagation: lone string,
 volume_noCopy: lone boolean,
 tmpfs_size: lone integer
}
{tmpfs_size >= 0
consistency in ”consistent” + ”cached” + ”delegated”
type in ”volume” + ”bind” + ”tmpfs”
}
```

The Service.volume is used to express that the service on which the predicate is applied has access to a given volume and specifies the VolumeRole spec which carries information about their connection.

```
pred Service.volume[volume: one Volume, spec: one VolumeRole]
{
 spec in this.volumes and bind[spec, volume]
}
```

Following the JSON schema of Docker Compose, the secrets and configurations have common attributes.

```
sig Secret extends LG/Location {
file: one String,
is_external : one boolean,
external_name: one Name, // optional: Actual name if external (usually different from the name in the docker compose Yaml file)
labels: set String
}
{
some external_name implies is_external=true
}


sig Config extends Location {
file: one String,
is_external : lone boolean,
external_name: one Name, // optional: Actual name if external (usually different from the name in the docker compose Yaml file)
labels: set String
}
{
some external_name implies is_external=true
}
```

As for Volume and Network, the services connections to the configs and secrets are defined at role level. They are modeled in Alloy by SecretRole and ConfigRole. Since we noticed that the attributes are the same for both, we introduce an abstract Service_access signature from which they inherit.

```
abstract sig Service_access extends Role{
  source: one Secret, // the source (name) in the long syntax or the direct attribute in the short syntax of Docker Compose Yaml file
  target: one string,
  uid: lone Int,
  gid: lone Int,
  mode: lone Int
}

sig SecretRole extends Service_access{}{}
sig ConfigRole extends Service_access{}{}
```

The following predicates express that a service has access to a given secret (resp. config) at a precise SecretRole (resp. ConfigRole).

```
pred Service.secret[secret: one Secret, spec: one SecretRole]
{
 spec in this.secrets and bind[spec, secret]
}


pred Service.config[config: one Config, spec: one ConfigRole]
{
 spec in this.configs and bind[spec, config]
}
```

A service may depend on one or more other services. In the Service signature, there are two attributes which are defined as sets of roles. One is named depends_on and the other is named on_which_depend. While the first represents the set of roles on which the service connects to the other service(s) on which it depends, the second represents the opposite, that is the set of roles on which the service(s) depending on it are bound. In the facts related to the Service signature, the depends_on roles belong to the set of required roles of the service location and the roles in on_which_depend belong to the set of provided roles. The following predicate links the two services in the dedicated roles.

```
pred Service.depends_on[service: one Service]
{
    one role : service.on_which_depend {
        role in this.depends_on}
}
```

The service has a build attribute. The Build signature reflects all the build properties that may appear in the Yamlfile.

```
sig Build{
 context: one string,
 dockerfile: lone string,
 args: set string,
 label: set string,
 cache_from: set string,
 network: lone string,
 target: lone string,
 shm_size: lone size
}
```

A healthcheck attribute is aimed to configure a check that's run to determine whether or not containers for this service are "healthy". The Healthcheck signature is defined as follows:

```
sig Healthcheck {
 disable: lone boolean,
 interval: lone scalar_unit_time, //format is duration
 retries: lone integer,
 test: set string,
 timeout: lone scalar_unit_time,
 start_period: lone scalar_unit_time,
}
```

The interval, timeout **and** start_period are time durations. The scalar_unit_time is proposed to model them.

```
abstract sig Scalar {
  value: one Int
} {
  value >= 0 // A scalar is a positive integer.
}

enum TimeUnits { d, h, m, s, ms, us, ns }

sig scalar_unit_time extends Scalar {
  unit: one TimeUnits
}

//Its value and unit are initialized using the init predicate.

pred scalar_unit_time.init[v: one Int, u: one TimeUnits]
{
  this.value = v
  this.unit = u
}
```

As mentioned above, there are some properties in the Yaml file which take effect only when deployed to a swarm with docker stack deploy, and which are ignored by docker-compose up and docker-compose run. This is the case for the deploy key in the Yaml file. It specifies configuration related to the deployment and running of services.

```
sig Deployment {
 mode: lone string,
 endpoint_mode: lone string,
 replicas: lone integer,
 labels: set string,
resources_limit: lone Service_resource_spec,
resources_resrevations:lone Reservation_resources_spec,
restart_condition: lone string,
restart_delay: lone scalar_unit_time,
restart_max_attempts: lone integer,
restart_window: lone scalar_unit_time,
 rollback_config: lone Deploy_config,
 update_config: lone Deploy_config
}


sig Reservation_resources_spec extends Service_resource_spec
{
generic_resources: set Discrete_resource_spec
}
{Discrete_resource_spec = generic_resources}

sig Discrete_resource_spec{
kind: one string,
value: one number
}


sig Deploy_config {
 parallelism: lone integer,
 delay: lone scalar_unit_time,
 failure_action: lone string,
 monitor: lone scalar_unit_time,
 max_failure_ratio: lone number,
 order: lone string
}
{
  order in "start-first"+ "stop-first"
}
```

In the following, we show the additional axioms characterizing the docker compositions.

```
fact all_service_net_roles_are_attached_to_some_network{
  all s: Service|
    all vr: s.networks|one vol: Network| s.network[vol,vr] }


fact network_roles_are_attached_to_disjoint_networks{
  all s: Service|
    all disj nr1,nr2: s.networks|all n1,n2: Network|
                          s.network[n1,nr1] and s.network[n2,nr2]
                           implies no n1&n2 }


fact all_service_volume_roles_are_attached_to_some_volume{
  all s: Service|
```

```
    all vr: s.volumes|one vol: Volume| s.volume[vol,vr] }

fact volume_roles_are_attached_to_disjoint_volumes{
  all s: Service|
    all disj vr1,vr2: s.volumes|all vol1,vol2: Volume|
                          s.volume[vol1,vr1] and s.volume[vol2,vr2]
                          implies no vol1&vol2 }
```

In the following we show, for clarity, a simple composition example containing mainly structural elements, that is services along with the volumes and networks to which they are connected. In the following subsection, a more complex example which is automatically generated is detailed.

```
//
// This is a Docker compose example.
//
sig Example extends Composition
{
  web: one Service,
  db: one Service,
  network: one Network,
  volume: one Volume,
  netwebrole: one NetworkRole,
  netdbrole:one NetworkRole,
  volrole: one VolumeRole
} {
  services = web + db
  networks = network
  volumes = volume
  web.replicas[3]
  web.network[network,netwebrole ]
  web.depends_on[db]
  no web.volumes
  db.replicas[1]
  db.network[network,netdbrole]
  db.volume[volume, volrole]
  no db.depends_on
  db.deploy.update_config.delay.init[10,s]
}
```

6.3 The Docker Compose to Alloy-LG Translator

All the signatures and axioms presented in the previous subsection are included in a Docker.als file. Besides,we implemented a tool to translate Docker Compose files to composition which extend the Composition signature in Alloy. The tool is developed in Python. We made the choice of interacting with the Docker Compose project which is open source[8]. The Docker Compose project takes charge of the parsing, validation, and the translation of the Yaml dictionaries into Python objects whose attributes are accessed in our project. This approach allows our tool to comply with Docker Compose which performs the following:

1. the parsing of the Yaml file and the syntax checking according with the Docker Compose JSON schema.
2. checking for the absence of some errors at an early stage. We cite for example the circular dependency between services.
3. building python objects which are instantiated from representative classes of Docker compose CLI source code (service, network, volumeSpec, networkSpec, etc.)

Our project essentially has wrapper classes which refer to this source code, creates instances, and generates Alloy text for the Docker composition. In the following, we show an example of Docker Compose Yaml example together with its Alloy translation.

```
 version: "3.7"
services:

  redis:
    image: redis:alpine
    ports:
      - "6379"
    networks:
      frontend:
        ipv4_address: 172.16.238.10
        ipv6_address: 2001:3984:3989::10

    deploy:
      replicas: 2
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure

      resources:
        limits:
          cpus: '0.50'
          memory: 50M
        reservations:
          cpus: '0.25'
          memory: 20M

    secrets:
      - source: my_secret
        target: redis_secret
        uid: '103'
        gid: '103'
        mode: 0440

  db:
    image: postgres:9.4
    volumes:
      - db-data:/var/lib/postgresql/data
    networks:
      - backend
    deploy:
      placement:
        constraints: [node.role == manager]

  vote:
    image: dockersamples/examplevotingapp_vote:before
    ports:
      - 5000:80
    networks:
      - frontend
    depends_on:
      - redis
    deploy:
      replicas: 2
      update_config:
        parallelism: 2
      restart_policy:
```

```
        condition: on-failure

  result:
    image: dockersamples/examplevotingapp_result:before
    ports:
      - 5001:80
    networks:
      - backend
    depends_on:
      - db
      - vote
    deploy:
      replicas: 1
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
    volumes:
    - type: tmpfs
      target: /foo/bar

  worker:
    image: dockersamples/examplevotingapp_worker
    networks:
      - frontend
      - backend
    deploy:
      mode: replicated
      replicas: 3
      labels: [APP=VOTING]
      restart_policy:
        condition: on-failure
        delay: 10s
        max_attempts: 3
        window: 120s
      placement:
        constraints: [node.role == manager]

  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    stop_grace_period: 1m30s
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]
  web:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - type: volume
        source: mydata
        target: /data
        volume:
          nocopy: true

      - type: bind
        source: ./static
```

```
      target: /opt/app/static
      bind:
        propagation: shared

networks:
  frontend:
  backend:

volumes:
  db-data:
  mydata:

secrets:
  my_secret:
    file: ./my_secret.txt
  my_other_secret:
    external: true
```

In the following, we show the generated Alloy file

```
module composetest
open lib/Docker

sig S_redis_frontend_netrole extends NetworkRole{} {

ipv4_address = "172.16.238.10"
ipv6_address = "2001:3984:3989::10"
}

sig S_redis_my_secret_secrole extends SecretRole{} {
// source = my_secret
target = "redis_secret"
uid = 103
gid = 103
mode = 288
}

sig S_db_backend_netrole extends NetworkRole{} {
}
sig S_db_db_data_volrole extends VolumeSpec{} {
external= "composetest_db-data"
internal= "/var/lib/postgresql/data"
mode= "rw"
}

sig S_vote_frontend_netrole extends NetworkRole{} {
}

sig S_result_backend_netrole extends NetworkRole{} {
}

sig S_worker_backend_netrole extends NetworkRole{} {
}

sig S_worker_frontend_netrole extends NetworkRole{} {
}

sig S_visualizer_default_netrole extends NetworkRole{} {
}
sig S_visualizer_vol1_volrole extends VolumeSpec{} {
external= "/var/run/docker.sock"
internal= "/var/run/docker.sock"
mode= "rw"
```

```
}

sig S_web_default_netrole extends NetworkRole{} {
}
sig S_web_mydata_mountrole extends MountSpec{} {
source= "composetest_mydata"
target= "/data"
volume_noCopy= true
}
sig S_web_vol2_mountrole extends MountSpec{} {
source= "/home/pissard/Dev/Spades/globalrm/Demonstrator/compose2alloy/static"
target= "/opt/app/static"
bind_propagation= "shared"
}

sig configuration extends Composition {

redis: one Service,
db: one Service,
vote: one Service,
result: one Service,
worker: one Service,
visualizer: one Service,
web: one Service,

_frontend: one Network,
_backend: one Network,
_default: one Network,

_db_data: one Volume,
_mydata: one Volume,

my_secret: one Secret,
my_other_secret: one Secret,

redis_frontend_netrole: one S_redis_frontend_netrole,
redis_my_secret_secrole: one S_redis_my_secret_secrole,
redis_deploy: one Deployment,
redis_deploy_update_config: one Deploy_config,

db_backend_netrole: one S_db_backend_netrole,
db_db_data_volrole: one S_db_db_data_volrole,
db_deploy: one Deployment,

vote_frontend_netrole: one S_vote_frontend_netrole,
vote_deploy: one Deployment,
vote_deploy_update_config: one Deploy_config,

result_backend_netrole: one S_result_backend_netrole,
result_deploy: one Deployment,
result_deploy_update_config: one Deploy_config,

worker_backend_netrole: one S_worker_backend_netrole,
worker_frontend_netrole: one S_worker_frontend_netrole,
worker_deploy: one Deployment,

visualizer_default_netrole: one S_visualizer_default_netrole,
visualizer_vol1_volrole: one S_visualizer_vol1_volrole,
visualizer_deploy: one Deployment,

web_default_netrole: one S_web_default_netrole,
web_mydata_mountrole: one S_web_mydata_mountrole,
```

web_vol2_mountrole: **one** S_web_vol2_mountrole,

}
{

services = redis + db + vote + result + worker + visualizer + web
networks = _frontend + _backend + _default

volumes = _db_data + _mydata
redis.image = "redis:alpine"
redis.network[_frontend, redis_frontend_netrole]
no redis.depends_on
redis.privileged = false
redis.ports = "6379/tcp"
redis.deploy = redis_deploy
redis_deploy.update_config = redis_deploy_update_config
redis_deploy_update_config.parallelism = 2
redis_deploy_update_config.delay.init[10, s]

db.image = "postgres:9.4"
db.network[_backend, db_backend_netrole]
db.volume[_db_data, db_db_data_volrole]
no db.depends_on
db.privileged = false
no db.ports
db.deploy = db_deploy

vote.image = "dockersamples/examplevotingapp_vote:before"
vote.network[_frontend, vote_frontend_netrole]
vote.depends_on[redis]
vote.privileged = false
vote.ports = "5000:80/tcp"
vote.deploy = vote_deploy
vote_deploy.update_config = vote_deploy_update_config
vote_deploy_update_config.parallelism = 2

result.image = "dockersamples/examplevotingapp_result:before"
result.network[_backend, result_backend_netrole]
result.depends_on[db]
result.depends_on[vote]
result.privileged = false
result.ports = "5001:80/tcp"
result.deploy = result_deploy
result_deploy.update_config = result_deploy_update_config
result_deploy_update_config.parallelism = 2
result_deploy_update_config.delay.init[10, s]

worker.image = "dockersamples/examplevotingapp_worker"
worker.network[_backend, worker_backend_netrole]
worker.network[_frontend, worker_frontend_netrole]
no worker.depends_on
worker.privileged = false
no worker.ports
worker.deploy = worker_deploy

visualizer.image = "dockersamples/visualizer:stable"
visualizer.network[_default, visualizer_default_netrole]
visualizer_vol1_volrole **in** visualizer.volumes
no visualizer.depends_on
visualizer.privileged = false
visualizer.ports = "8080:8080/tcp"
visualizer.deploy = visualizer_deploy

```
web.image = ”nginx:alpine”
web.network[ _default, web_default_netrole ]
web.volume[_mydata, web_mydata_mountrole]
web_vol2_mountrole in web.volumes
no web.depends_on
web.privileged = false
web.ports = ”80:80/tcp”


my_secret.file = ”/home/pissard/Dev/Spades/globalrm/Demonstrator/compose2alloy/my_secret.txt”
my_other_secret.is_external = true
}
```

We call the run command as follows to show a model.

```
run {
 Service in configuration.services
} for 30 but exactly 1 configuration, exactly 3 Network, exactly 7 Service
, 4 Secret, exactly 1 LocationGraph
```

7 The HOT model

This section first gives an overview of the HOT [9] [10] language and shows its interpretation in our reference computational model. Then, it presents the limitations we found on the HOT specification. Finally, this section describes the tool implemented for the formal verification of HOT templates and compares it with other tools found in the literature.

7.1 Overview and interpretation of HOT

HOT (Heat Orchestration Template) is a YAML-based language used by the Heat orchestration engine of OpenStack [14]. This language allows for describing a system as a set of resources of different types that can be for instance, virtual machines, physical servers, networks, routers, and ports. These resources are defined in a template.

The structure of a template and the supported types of resources are defined in the HOT specification and can be found, respectively, in [9] and [10] for the latest version. This allows consumers of cloud services to know how to design HOT templates that can be automatically deployed with Heat. However, this specification is mostly written in natural language and is, therefore, possibly ambiguous, inconsistent, and error-prone. To handle this challenge, we propose a formal interpretation of the HOT specification in our reference computational model. In the following, we first present the HOT language core and its interpretation. Then, the same is done for the supported resource types.

7.1.1 Overview of the HOT core

The specification [9] that defines the core of concepts of HOT states that a template is structured in seven sections:

- **heat_template_version** defines the version used to write the template and is mandatory. The expected value is a string that must be equal to *2013-05-23*, *2014-10-16*, *2015-04-30*, *2015-10-15*, *2016-04-08*, *2016-10-14*, *2017-02-24*, *2017-09-01*, *2018-03-02*, *2018-08-31*, *2019-04-10*, *newton*, *ocata*, *pike*, *queens*, *rocky*, or *stein*;

- **description** is an optional string that gives an explanation of the template (e.g. the service that is provided);

- **parameters** optionally defines the input parameters that have to be provided at deployment time. This allows for customizing the template and reusing it. A parameter is a 10-tuple $< name, label, description, type, default, value, hidden, constraints, tag, immutable >$. The $hidden$ field specifies if the parameter value must be readable or not (e.g. for a password). The $constraints$ field defines a set of constraints, checked by the orchestrator, on the parameter value. The supported constraints are $length$, $range$, $allowed_values$, $allowed_pattern$ and $custom_constraint$. The later takes values that are defined in Heat (e.g. *ip_addr*, *neutron_network*). When defined for a parameter, a $custom_constraint$ means either that the parameter value must be syntactically valid (e.g. *ip_addr*) or it must be the id of a resource of the specified type (e.g. *neutron_network*). The $immutable$ field specifies if the parameter can be modified or not when updating the deployment of the system. To create a parameter, Heat users have to define at least all the required fields (i.e. $name$, $type$, either $default$ or $value$);

- **parameter_groups** specifies how the input parameters are grouped. For example, all parameters that are related to network (e.g. IP addresses, cidr, DNS name-servers) can belong to the same group. A parameter_group is a 3-tuple $< label, description, parameters >$. The $label$ field is

used for identification purpose, $description$ to give information about the parameter_group and $parameters$ to order and group a set of parameters names;

- **resources** defines a set of resources to be deployed. At least one resource must be specified for the deployment to be performed. A resource is a 9-tuple $< id, type, properties, metadata, depends_on, condition, ext$ $deletion_policy, update_policy >$. This means that a resource has an id, a $type$, a set of $properties$ based on its type, a set of $metadata$ for information, and can $depends_on$ other resources. A resource can also have a $condition$ specifying when it must be created, an $external_id$ when it is an existing resource that will be just used but not created during the deployment. Finally, a resource has a $deletion_policy$ and can have an $update_policy$ depending on its type. The types of resources supported by HOT are presented in Section 7.1.3;

- **outputs** optionally defines the outputs in order to give to users a feedback after the deployment. An example of output is the IP address of a virtual machine that is dynamically allocated through DHCP. An output is a 4-tuple $< name, description, condition, value >$. To create an output, one must define its $name$ and an expression for its $value$. The actual $value$ will be computed by the orchestrator after the deployment and given to the user;

- **conditions** optionally defines a set of conditions based on the values of the input parameters. A condition is a 2-tuple $< name, expression >$, where $expression$ will be evaluated, to true or false, by the orchestrator. The objective is to decide whether the related actions (e.g. creation of a specific resource) can be performed or not.

7.1.2 Interpretation of the HOT core

The HOT core is interpreted in our reference computational model by writing, in a systematic way from the specification [9], an Alloy module named *HOT.als*. This is done as follows. First, an inheritance relationship and a mapping are defined between the concepts of HOT core and those of Location Graphs. A HOT Template inherits from LocationGraph as both can be viewed as a component-based system in software engineering. Resource inherits from Location, both can be seen as components of the system. Hence, the resources of a Template are mapped to the Locations of a LocationGraph. A Resource Property inherits from Value. Parameter_group, Parameter, ParameterConstraint, Output and Condition also inherit from Value. Then, a set of functions (**fun**) are written, in Alloy, to implement the functions that are defined in the HOT specification (e.g. get_param, get_resource). A set of predicates (**pred**) are also written. They allow to fill the sections of a template (e.g. state that a parameter belong to a specific template). Finally, a set of facts (**fact**) and predicates are written in order to ensure the HOT invariants that are specified, in English, in [9]. Examples of such invariants are:

- the length parameter constraint applies to parameters of type string, comma_delimited_list and json;
- the allowed_values parameter constraint applies to parameters of type string or number;
- a resource id must be unique within the resources section of the template;
- external resources can not depend on other resources that are not external ones;
- an output name must be unique within the outputs section of a template.

Other invariants were added by us because they were omitted in the HOT specification. It was necessary to write a set of facts related to these invariants in order to prevent from unrealistic behaviors. Examples of invariants we add are:

- for a length and a range parameter constraint, max must be greater or equal to min if both are specified;
- a parameter name must be unique within the parameters section of a template;
- there must be no loop between resource dependencies.

To check the consistency of the HOT core interpretation, a **run** command is first written. Then, this command is executed and the Alloy analyzer finds several instances of HOT templates, meaning that the interpretation is consistent.

```
/*****************************************************************************
 * A formal interpretation of HOT with Location Graphs in Alloy
 * Hand written
 *****************************************************************************/
module HOT
open LocationGraphs as LG
open map[String] as map_string
open map[Int] as map_int
open map[Boolean] as map_boolean
open map[ResourceRole] as map_resourceRole

/*****************************************************************************
 * Template
*****************************************************************************/

sig Template extends LG/LocationGraph
{
  heat_template_version: one HeatTemplateVersion, // the version of HOT used for the template
  description: lone String, // the description of the template
  parameter_groups: seq ParameterGroup, // the parameter groups of the template
  parameters: set Parameter, // the input parameters of the template
  resources: some Resource, // the resources to deploy
  outputs: set Output, // the output values for users
  conditions: set Condition // the conditions for testing parameters or resources properties
}{
 // defining the locations of the LocationGraph
    locations = resources

 // resources have distinct ids (in the spec)
    distinct_resources_id[resources]

 // parameter groups must be distinct (not in the spec)
    distinct_labels[parameter_groups] // parameter groups have distinct labels
    not parameter_groups.hasDups // each parameter group is defined once

 // for each group, parameter names must be names of defined parameters (in the spec)
    group_param_names_bind_names_of_defined_params[parameter_groups,parameters]

 // parameters have distinct names (not in the spec)
    distinct_param_names[parameters]

 // each parameter is in 0 or 1 group (in the spec)
    param_in_zero_or_one_group[parameters,parameter_groups]

 // outputs have distinct names (in the spec)
    distinct_output_names[outputs]

 // conditions have distinct names (in the spec)
    distinct_cond_names[conditions]
```

```
 // no loop depencency between resources (not in the spec but handled by Heat)
    no_loop_dependency[resources] //
}


/*****************************************************************************
* Heat Template Version
*****************************************************************************/

let HeatTemplateVersion = "2013-05-23" + "2014-10-16" + "2015-04-30" + "2015-10-15"
                        + "2016-04-08" + "2016-10-14" + "2017-02-24" + "2017-09-01"
                        + "2018-03-02" + "2018-08-31" + "2019-04-10" + "newton" + "ocata"
                        + "pike" + "queens" + "rocky" + "stein"


/*****************************************************************************
 * HOTValue
*****************************************************************************/

sig HOTValue extends LG/Value
{
}

/*****************************************************************************
 * Resource
*****************************************************************************/

abstract sig Resource extends LG/Location
{
  id: one String, // id for identifying a resource
  properties : set Property + PropertyValueType, // a set of properties describing the resource
  attributes: set Attribute, // the attributes that can be obtained when the resource is deployed
  support_status: lone Support_status, // information about the resource support status
  depends_on: set ResourceRole, // the list of resources on which a resource depends on
  deletion_policy: lone "delete" + "retain" + "snapshot", // applied when a resource is removed from the template
  external_id: lone String, // specifies that a resource already exists and has an external id
  condition: lone String, // specifies if to create or not a resource
  metadata: lone MapValue // give information about the resource
}{
  // defining required and provided roles
      // a resource has one or more provided
        some provided

     // required roles contain depends_on elements
        depends_on in required

  // an external resource must not depend on another resource
    no_depends_on_for_external_resource[this]
 }

/* Resource Role */
sig ResourceRole extends LG/Role
{
}


/* Properties */

// Property
abstract sig Property extends HOTValue
{
```

```
  prop_name: one String,
  prop_immutable: lone Boolean,
  prop_required: one Boolean,
  prop_update_allowed: lone Boolean,
  prop_description: lone String,
  prop_type: one String,
  prop_default: set PropertyValueType
}

// Flat property
sig FlatProperty extends Property
{
  prop_value: set PropertyValueType
}{
  // the type of a property
   prop_type in "list" + "map" + "integer" + "number" + "boolean" + "string"
   required_implies_at_least_one_value[this]
}

// Type of a property value
let PropertyValueType = String + Boolean + Integer + Number + MapValue + ResourceRole
let Any = PropertyValueType − ResourceRole


// Composed property
abstract sig ComposedProperty extends Property
{
}{
 prop_type in "list" + "map"
}

abstract sig MapOfProperties extends ComposedProperty
{
}{
 prop_type = "map"
}

abstract sig ListOfProperties extends ComposedProperty
{
 elements: set Property
}{
 prop_type = "list"
}

/* Attributes */
sig Attribute extends HOTValue
{
  name: one String, // name of the attribute
  type: one AttributeType, // the type of the attribute
  description: lone String // the description of the attribute
}

// AttributeType
let AttributeType = "string" + "map" + "list"+ "integer" + "boolean"

/* Support Status */
sig Support_status extends HOTValue
{
  message: lone String, // an information about the status of a resource
  status: one "SUPPORTED" + "DEPRECATED" + "UNSUPPORTED", // specfies the current status of a resource
  version: lone String, // the version of a resource
  previous_status: lone Support_status
```

```
}

/*****************************************************************************
 * Parameter group
*****************************************************************************/

sig ParameterGroup extends HOTValue
{
  label: one String, // the label for identifying a parameter group
  description: lone String, // the description of a parameter group
  parameters: seq String // the list of parameter names of a group
}{
 // a parameter is once in a group (specified in the spec)
    not parameters.hasDups

 // a parameter group must not be empty (not specified in the spec)
    not_empty_params[this]
}

/*****************************************************************************
 * Parameter
*****************************************************************************/

sig Parameter extends HOTValue
{
  name: one String, // the name for identifying a parameter
  type: one ParameterType, // the type of a parameter
  label: lone String, // the label of a parameter
  description: lone String, // the description of a parameter
  default: lone univ, // the default value for a parameter
  value: one univ, // the value of the parameter
  hidden: lone Boolean, // specifies if a parameter is hidden (like a password) or not
  constraints: set ParameterConstraint, // the list of constraints for a parameter
  immutable: lone Boolean, // specifies if a parameter is updable
  tags: set String // used to group parameters
}{
 // depending on the parameter type, specific constraints can be used (in the spec)
    validate_constraints[this]

 // a constraints must be defined once (not in the spec)
    no_duplicated_constraints[this]
}


/* Parameter Type */
let ParameterType = "string" + "number" + "json" + "comma_delimited_list" + "boolean"


/* Pameter Constraints */

abstract sig ParameterConstraint
{
  description: lone String
}

// Length constraint
sig Length extends ParameterConstraint
{
  definition: one Length_interval
}

// Range constraint
```

```
sig Range extends ParameterConstraint
{
  definition: one Range_interval
}

// Modulo constraint
sig Modulo extends ParameterConstraint
{
  definition: one StepOffset
}

// Allowed_values constraint
sig Allowed_values extends ParameterConstraint
{
  definition: set univ
}

// Allowed pattern constraint
sig Allowed_pattern extends ParameterConstraint
{
  definition: one String
}

// Custom constraint
sig Custom_constraint extends ParameterConstraint
{
  definition: one CustomConstraintName
}

/* Interval */
abstract sig Interval
{
  i_min: lone Int,
  i_max: lone Int
}{
 // min alone or max alone or both (in the spec)
   min_or_max_or_both[this]

 // if they are both specified, max >= min (not in the spec)
   all min, max: Int | (min = i_min and max = i_max) implies max >= min
}

// Interval for the length constraint
sig Length_interval extends Interval
{
}{
 // if they are specified, both are >= 0 (not in the spec)
   all min: Int| min = i_min implies min >= 0
   all max: Int| max = i_max implies max >= 0
}

// Interval for the range constraint
sig Range_interval extends Interval
{
}

// Step offset for the modulo constraint
sig StepOffset
{
  step: one Int,
  offset: one Int
}{
```

```
 // step must not be equal to 0 because division by 0 cannot be done (not in the spec)
    step != 0
}

// custom constriant name for the custom constraint
let CustomConstraintName = ”barbican.container” + ”barbican.secret” + ”cinder.backup” + ”cinder.qos_specs”
+ ”cinder.snapshot” + ”cinder.volume” + ”cinder.vtype” + ”cron_expression”
+ ”designate.domain” + ”designate.zone” + ”dns_domain” + ”dns_name” + ”expiration”
+ ”glance.image” + ”ip_addr” + ”iso_8601” + ”keystone.domain” + ”keystone.group”
+ ”keystone.project” + ”keystone.region” + ”keystone.role” + ”keystone.service”
+ ”keystone.user” + ”mac_addr” + ”magnum.baymodel” + ”magnum.cluster_template”
+ ”manila.share_network” + ”manila.share_snapshot” + ”manila.share_type”
+ ”mistral.workflow” + ”monasca.notification” + ”net_cidr” + ”neutron.address_scope”
+ ”neutron.flow_classifier” + ”neutron.lb.provider” + ”neutron.lbaas.listener”
+ ”neutron.lbaas.loadbalancer” + ”neutron.lbaas.pool” + ”neutron.lbaas.provider”
+ ”neutron.network” + ”neutron.port” + ”neutron.port_pair” + ”neutron.port_pair_group”
+ ”neutron.qos_policy” + ”neutron.router” + ”neutron.security_group” + ”neutron.segment”
+ ”neutron.subnet” + ”neutron.subnetpool” + ”nova.flavor” + ”nova.host” + ”nova.keypair”
+ ”nova.network” + ”nova.server” + ”octavia.l7policy” + ”octavia.listener” + ”octavia.loadbalancer”
+ ”octavia.pool”+ ”rel_dns_name” + ”sahara.cluster” + ”sahara.cluster_template” + ”sahara.data_source”
+ ”sahara.image” + ”sahara.job_binary” + ”sahara.job_type” + ”sahara.plugin” + ”senlin.cluster”
+ ”senlin.policy” + ”senlin.policy_type” + ”senlin.profile” + ”senlin.profile_type”
+ ”test_constr” + ”timezone” + ”trove.flavor” + ”zaqar.queue”

/*************************************************************************
 * Output
*************************************************************************/

sig Output extends HOTValue
{
  name: one String,
  description: lone String,
  value: one String,
  condition: lone String
}


/*************************************************************************
 * Condition
*************************************************************************/

sig Condition extends HOTValue
{
 name: one String,
 expression: one String
}

/*************************************************************************
                    Functions
*************************************************************************/

/* get_param: returns the provided roles of the resource with the id that is
    equal to the value of the parameter given as input of the function*/

fun get_param_role[param: one Parameter]: set ResourceRole
{
   param.value.~id.provided
}

/* get param: returns the value of a parameter*/

fun get_param[param: one Parameter]: one Any
```

```
{
 param.value
}

/* get_resource: returns the provided roles of the resource
    given as input of the function*/


fun get_resource[rsc: one Resource]: set ResourceRole
{
  rsc.provided
}

/*****************************************************************************
 * Predicates for filling the sections of a template
*****************************************************************************/

// a parameter is part of this template
pred Template.parameter[param: Parameter]
{
  param in this.parameters
}


// a parameter group is part of this template
pred Template.parameter_group[paramGroup: ParameterGroup]
{
  paramGroup in this.parameter_groups.elems
}

// a resource is part of this template
pred Template.resource[rsc: Resource]
{
  rsc in this.resources
}

// an output is part of this template
pred Template.output[output: Output]
{
 output in this.outputs
}

// a condition is part of this template
pred Template.condition[condition: Condition]
{
  condition in this.conditions
}

/*****************************************************************************
* Predicates for filling the required roles of a resource
*****************************************************************************/

/* append a resource role */

// add a role of a resource to a list of roles
pred Resource.append[rsc: one Resource, roles: set ResourceRole]
{
    one role: rsc.provided{
        role in roles
    }
}
```

```
/* add a role to a set of roles */

pred bind_one_role[target_roles: one ResourceRole, roles: set ResourceRole]
{
  one role: roles {
  target_roles = role
 }
}

pred binding_roles[target_roles: one ResourceRole, roles: set ResourceRole]
{
  target_roles = roles
}

/* resource dependency */

pred Resource.depends_on[resource: one Resource]
{
  this.append[resource, this.depends_on]
}

/*****************************************************************************
* Predicates for filling a constraint for a parameter
*****************************************************************************/

//length
pred Parameter.length[a_min: lone Int, a_max: lone Int, descr: lone String]
{
  one l: Length{
   l.definition.i_min = a_min
   l.definition.i_max = a_max
   l.description = descr
   l in this.constraints
  }
}

// range
pred Parameter.range[a_min: lone Int, a_max: lone Int, descr: lone String]
{
  one r: Range{
   r.definition.i_min = a_min
   r.definition.i_max= a_max
   r.description = descr
   r in this.constraints
  }
}

// modulo
pred Parameter.modulo[a_step: one Int, an_offset: one Int, descr: lone String]
{
  one m: Modulo{
      m.definition.step = a_step
      m.definition.offset = an_offset
      m.description = descr
      m in this.constraints
  }
}

// allowed_values
pred Parameter.allowed_values[def: set univ, descr: lone String]
{
```

```
 one a: Allowed_values{
        a.definition = def
        a.description = descr
        a in this.constraints
 }
}

// allowed_pattern
pred Parameter.allowed_pattern[def: one String, descr: lone String]
{
 one a : Allowed_pattern{
          a.definition = def
          a.description = descr
          a in this.constraints
      }
}

// custom_constraint
pred Parameter.custom_constraint[def: one CustomConstraintName, descr: lone String]
{
 one c : Custom_constraint{
          c.definition = def
          c.description = descr
          c in this.constraints
      }
}

/*****************************************************************************
  * Predicates for filling a parameter name for a parameter group
*****************************************************************************/
// a parameter name is part of a parameter group
pred ParameterGroup.parameter[paramName: one String]
{
 paramName in this.parameters.elems
}

/*****************************************************************************
  * Predicates for specifying an attribute and a property of a resource
*****************************************************************************/

// an attribute is part of a resource
pred Resource.attribute[attr: one Attribute]
{
 attr in this.attributes
}

// a property is part of a resource
pred Resource.property[prop: one Property + PropertyValueType]
{
 prop in this.properties
}

/*****************************************************************************
  * Predicates for filling a tag for a parameter
*****************************************************************************/

// a tag is part of a parameter
pred Parameter.tag[tag: one String]
{
 tag in this.tags
}
```

```
/*****************************************************************************
 * Predicates for enforcing constraints in a template
*****************************************************************************/

// distinct labels for prameter groups
pred distinct_labels[parameterGroups: seq ParameterGroup]
{
  all g1, g2 : parameterGroups.elems | g1.label = g2.label implies g1 = g2
}

// distinct ids for resources
pred distinct_resources_id[resources: set Resource]
{
 all r1, r2: resources | r1.id = r2.id implies r1 = r2
}

// distinct names for parameters
pred distinct_param_names[params: set Parameter]
{
  all p1, p2 : params | p1.name = p2.name implies p1 = p2
}

// distinct names for outputs
pred distinct_output_names[outpts: set Output]
{
  all o1, o2 : outpts | o1.name = o2.name implies o1 = o2
}

// distinct names for conditions
pred distinct_cond_names[conds: set Condition]
{
  all c1, c2 : conds | c1.name = c2.name implies c1 = c2
}

// parameter names in parameter groups must correspond to the names of defined parameters
pred group_param_names_bind_names_of_defined_params[prGroups: seq ParameterGroup, params: set Parameter]
{
 all g1: prGroups.elems | all paramNameIng1: g1.parameters.elems | one nameOfDefinedPararm: params.name |
 paramNameIng1 = nameOfDefinedPararm
}

// a parameter name is in 0 or 1 group
pred param_in_zero_or_one_group[params: set Parameter, prGroups: seq ParameterGroup]
{
  all param: params, g1, g2: prGroups.elems |(param.name in g1.parameters.elems and
                                              param.name in g2.parameters.elems) implies g1 = g2
}

// the constraint that can be defined for a parameter depends on its type
pred validate_constraints[param: Parameter]
{
 (some l: Length | l in param.constraints) implies (param.type = "string" or param.type = "json"
                                                     or param.type = "comma_delimited_list")

 (some r: Range | r in param.constraints) implies param.type = "number"
 (some m: Modulo | m in param.constraints) implies param.type = "number"
 (some a_v: Allowed_values | a_v in param.constraints) implies (param.type = "string" or param.type = "number")
 (some a_p: Allowed_pattern | a_p in param.constraints) implies param.type = "string"
 /*in the spec, the types of parameters for which custom constraint can be applied is not specified.
 */
}
```

```
// a constraint cannot be specified more than once for the same parameter
pred no_duplicated_constraints[param: Parameter]
{
  all l1, l2: Length | (l1 in param.constraints and l2 in param.constraints) implies l1 = l2
  all r1, r2: Range | (r1 in param.constraints and r2 in param.constraints) implies r1 = r2
  all m1, m2: Modulo | (m1 in param.constraints and m2 in param.constraints) implies m1 = m2
  all a_v1, a_v2: Allowed_values | (a_v1 in param.constraints and a_v2 in param.constraints) implies a_v1 = a_v2
  all a_p1, a_p2: Allowed_pattern | (a_p1 in param.constraints and a_p2 in param.constraints) implies a_p1 = a_p2
  all c1, c2: Custom_constraint | (c1 in param.constraints and c2 in param.constraints) implies c1 = c2
}

// for the constraint length, either min or max is given or both
pred min_or_max_or_both[interval1: Interval]
{
  (no min: Int | min = interval1.i_min) implies (one max: Int| max = interval1.i_max)
}


// an external resource must not depends on another resource
pred no_depends_on_for_external_resource[rsc: one Resource]
{
 (one externalId: String | externalId = rsc.external_id) implies (#rsc.depends_on = 0)
}

// a parameter group must not be empty
pred not_empty_params[g: ParameterGroup]
{
  #(g.parameters.elems) > 0
}

// a flat property that is required must have a value
pred required_implies_at_least_one_value[prop: one FlatProperty]
{
 (prop.prop_required = true) implies (#prop.prop_value >= 1)
}

/*****************************************************************************
* Predicates and fact for avoiding loops in resources dependencies
*****************************************************************************/

pred dependsOn[r1,r2: one Resource]
{
  one roleR2: r2.provided| roleR2 in r1.depends_on
}

fact dependsOn_is_transitive
{
  all r1, r2, r3: Resource | (r1.dependsOn[r2] and r2.dependsOn[r3]) implies r1.dependsOn[r3]
}

pred no_loop_dependency[resources: set Resource]
{
 all r1, r2: resources| (r1.dependsOn[r2]) implies (not r2.dependsOn[r1])
}

/* Predicates for values of properties*/

// allowed_values
pred FlatProperty.allowed_values[values: set PropertyValueType]
{
 (this.prop_type in "string" + "number" + "integer") implies this.prop_value in values
}
```

```
pred ListOfProperties.allowed_values[values: set PropertyValueType]
{
 all prop: FlatProperty | (prop in this.elements and prop.prop_type in "string" + "number" + "integer") implies prop.prop_value in values
}

// length
pred FlatProperty.length[min: lone Int, max: lone Int]
{
 (this.prop_type = "list" and one min) implies #(this.prop_value) >= min
 (this.prop_type = "list" and one max) implies #(this.prop_value) <= max
}

pred ListOfProperties.length[min: lone Int, max: lone Int]
{
 (this.prop_type = "list" and one min) implies #(this.elements) >= min
 (this.prop_type = "list" and one max) implies #(this.elements) <= max
}

// range
pred FlatProperty.range[min: lone Int, max: lone Int]
{
 (this.prop_type in "integer" + "number" and one min and min = -1) implies (#(this.prop_value) = -1 or #(this.prop_value) >= 0)
 (this.prop_type in "integer" + "number" and one min and min >= 0) implies #(this.prop_value) >= min
 (this.prop_type in "integer" + "number" and one max) implies #(this.prop_value) <= max
}

/*****************************************************************************
 * HOT primitive types
*****************************************************************************/
// boolean
enum Boolean { true, false }

// number (integer or float but Alloy does not support float)
let Number = Int

// integer
let Integer = Int

// map for values of properties
let MapValue = map_string/Map + map_int/Map + map_boolean/Map + map_resourceRole/Map

run Show_HOT_Template
{
// resources must belong to the template (i.e. no isolated resource)
    Resource in Template.resources
    ResourceRole in Resource.required + Resource.provided

// parameter groups must belong to the template
    ParameterGroup in Template.parameter_groups.elems

// parameters must belong to the template
    Parameter in Template.parameters
    ParameterConstraint in Parameter.constraints // parameter constraints must belong to the parameters

// outputs must belong to the template
    Output in Template.outputs

// conditions must belong to the template
    Condition in Template.conditions

// LG names must belong to resources names
```

```
    LG/Name in Template.resources.name

// LG sorts, processes and roles must belong to the resources
    LG/Sort in Template.resources.sort
    LG/Process in Template.resources.process
    LG/Role in ResourceRole

// no isolated LG values
   LG/Value in LG/Name + LG/Sort + LG/Process + LG/Role

// no isolated length interval, range interval and stepOffset
    Length_interval in Length.definition
    Range_interval in Range.definition
    StepOffset in Modulo.definition

} for 15 but
            5 Int,
            exactly 1 Template,
            exactly 3 Resource,
            exactly 0 ParameterGroup,
            exactly 1 Parameter,
            exactly 1 ParameterConstraint,
            exactly 0 Condition,
            exactly 0 Output
            expect 1
```

7.1.3 Overview of the HOT types

The HOT resource types are provided by the OpenStack services and are grouped in domains (e.g. computing, networking, storage and monitoring). For instance, the Nova service provides computing resource types. Examples of these types are OS::Nova::Server (virtual machine) and OS::Nova::Keypair (pairs of keys for remote connection). The Neutron service provides networking resource types such as OS::Neutron::Net, OS::Neutron::Port and OS::Neutron::SecurityGroup. The latter allows for filtering the network traffic by disabling/enabling some protocols.

The specification related to the supported resource types can be found, in English, in [10] for the latest version of HOT (stein). A YAML version of this specification can be retrieved from a running OpenStack platform. This version is readable by machines (cf. Fig. 4) and can thus be used for generation purpose. This specification states that each resource type has one support status, a set of attributes, and a set of properties. The support status defines both the current and the previous statuses of the resource type (e.g. deprecated, supported). An attribute is a 4-tuple $< name, description, type, value >$. The first three fields are defined in the specification. The value is computed by the orchestrator after the deployment is performed. A property is either flat or composed of other nested properties.

A flat *property* is defined in the specification as a 9-tuple $< name, description, required, immutable, type, update_allowed, value, default, constraints >$. To define a property of a resource in a template, one has to specify only its $value$. The other fields are defined by the specification (cf. Fig 4). The $required$ field is a boolean that specifies if the property is mandatory or not. The $immutable$ field specifies, when it is true, that the property cannot be modified when updating the deployment. In this case, modifying the property leads to the failure of the deployment. The $update_allowed$ field means, when it is true, that the property can be updated without consequence on the resource. When it is false, it means that updating the property will be done by deleting the resource and creating a new one. Finally, the $constraints$ of a property allows the orchestrator for controlling its value. The possible constraints are the same than those of a parameter ($length$, $range$, $allowed_values$, $allowed_pattern$ and $custom_constraint$). The difference is that for a parameter, the constraints are defined if necessary by the HOT template designer whereas for

a property the constraints are imposed by the specification. For instance, the specification presented in Fig. 4 states that the floating_network property of an OS::Neutron::FloatingIP has a $custom_constraint$ named $neutron.network$.

Each $custom_constraint$ is tied to an OpenStack service and is associated to a plug-in that validates the value of the related properties. For instance, $neutron.network$ is defined by Neutron and is associated to the plug-in *os.neutron.neutron_constraints:NetworkConstraint* in Heat for validation purposes. Some $custom_constraints$ are characterized by the fact that they include one or several resource type in their name and/or plug-in. When associated to a property, such $custom_constraint$ specifies that the expected value for this property must be the id of a resource with the specified type. In this case, the property defines an interaction between the resource it belongs to and one or more other resources of the specified type. Hence, a property is either a simple value or a relation between several resources.

```
resource_type: OS::Neutron::FloatingIP
properties:
  floating_network:
    constraints:
    - custom_constraint: neutron.network
    description: Network to allocate floating IP from.
    immutable: false
    required: true
    type: string
    update_allowed: false
  # other properties
# other types of resources
```

Figure 4: *HOT types* specification in YAML

7.1.4 Interpretation of the HOT types

This interpretation is done in two steps. In the first step, an inheritance relationship is established between the concepts of HOT types and those that are defined in the HOT core interpretation. A Resource type (e.g. OS::Neutron::Net) inherits from Resource (defined in the HOT core interpretation) which inherits from Location. Therefore, a resource type is a Location. It has a set of required and a set of provided roles that must be specified. For each resource type, the set of resources it depends on and its properties that define interaction with other resources are mapped to the required roles of the Location. The provided roles inherit from ResourceRole (in the HOT core interpretation) which is derived from Role of LocationGraph. In the second step, we implement a python script for the generation of the HOT types interpretation.

This script takes as input the YAML version of the HOT types specification and generates for each resource type:

- a signature, derived from ResourceRole, for the provided roles of the resource type (e.g. OS_Neutron_Net_Role);
- a signature (**sig**), derived from Resource (e.g. OS_Neutron_Net), with a set of fields for its attributes and properties;
- a set of signatures for the properties, of the resource type, that are composed of other nested properties;
- a set of facts that fill the different fields of the resource type attributes (i.e. $name$, $description$, $type$);
- a set of facts that enforce the constraints (e.g. $length$, $allowed_values$) of the resource type properties;

- a set of predicates and/or functions that will be used to fill the value of the properties as specified in a template;
- a **run** command that can be executed to find one or more instances of the resource type with the Alloy analyzer.

Then, after the HOT types interpretation is generated, we manually add on it a set of facts. The objective is to ensure the invariants of the HOT types that are described in the specification in [10]. Three examples of such invariants are:

- for a router, the distributed property must not be used in conjunction with l3_agent_ids;
- for a router interface, either the port property or the subnet one must be specified;
- for a security group, if the protocol is set to TCP or UDP, the value of the port_range_min property must be less than or equal the one of port_range_max. If the protocol is ICMP, the port_range_min must be an ICMP type.

We also write facts for the invariants that are omitted in the HOT types specification. These invariants must be taken into account to prevent from unrealistic behaviors between different resource types. Examples of such invariants are:

- distinct routers of the same subnet or net cannot have the same IP address;
- for a port, if both network and subnet are specified, the subnet must belong to the network;
- distinct servers cannot be attached to the same port.

```
/*************************************************************************
* A formal interpretation of HOT types with Location Graphs in Alloy
* Generated by HOT_Types2AlloyLG script
*************************************************************************/

module HOT_types
open HOT
open LocationGraphs as LG

/*************************************************************************
*   OS::Neutron::Net
*************************************************************************/

// Resource Role

sig OS_Neutron_Net_Role extends HOT/ResourceRole
{
}

// Resource

sig OS_Neutron_Net extends HOT/Resource
{

    /** Attributes */

    // YAML admin_state_up: {'description': 'The administrative status of the network.', 'type': 'string'}
    attr_admin_state_up: lone Attribute,
    // YAML l2_adjacency: {'description': 'A boolean value for L2 adjacency,
        //True means that you can expect L2 connectivity throughout the Network.', 'type': 'boolean'}
```

```
    attr_l2_adjacency: lone Attribute,
    // YAML mtu: {'description': 'The maximum transmission unit size(in bytes) for the network.', 'type': 'integer'}
    attr_mtu: lone Attribute,
    // YAML name: {'description': 'The name of the network.', 'type': 'string'}
    attr_name: lone Attribute,
    // YAML port_security_enabled: {'description': 'Port security enabled of the network.', 'type': 'boolean'}
    attr_port_security_enabled: lone Attribute,
    // YAML qos_policy_id: {'description': 'The QoS policy ID attached to this network.', 'type': 'string'}
    attr_qos_policy_id: lone Attribute,
    // YAML segments: {'description': 'The segments of this network.', 'type': 'list'}
    attr_segments: lone Attribute,
    // YAML show: {'description': 'Detailed information about resource.', 'type': 'map'}
    attr_show: lone Attribute,
    // YAML status: {'description': 'The status of the network.', 'type': 'string'}
    attr_status: lone Attribute,
    // YAML subnets: {'description': 'Subnets of this network.', 'type': 'list'}
    attr_subnets: lone Attribute,
    // YAML tenant_id: {'description': 'The tenant owning this network.', 'type': 'string'}
    attr_tenant_id: lone Attribute,

    /** Properties */

    // YAML admin_state_up: {'default': True, 'description': 'A boolean value specifying the administrative
        //status of the network.', 'immutable': False, 'required': False, 'type': 'boolean', 'update_allowed': True}
    prop_admin_state_up: lone Boolean,
    // YAML dhcp_agent_ids: {'description': 'The IDs of the DHCP agent to schedule the network.
        // Note that the default policy setting in Neutron restricts usage of this property to administrative users only.
        //', 'immutable': False, 'required': False, 'type': 'list', 'update_allowed': True}
    prop_dhcp_agent_ids: set Any,
    // YAML dns_domain: {'constraints': [{'custom_constraint': 'dns_domain'}], 'description':
        //'DNS domain associated with this network.', 'immutable': False, 'required': False, 'type': 'string', 'update_allowed': True}
    prop_dns_domain: lone String,
    // YAML name: {'description': 'A string specifying a symbolic name for the network, which is not
        //required to be unique.', 'immutable': False, 'required': False, 'type': 'string', 'update_allowed': True}
    prop_name: lone String,
    // YAML port_security_enabled: {'description': 'Flag to enable/disable port security on the network.
        //It provides the default value for the attribute of the ports created on this network.', 'immutable': False,
        //'required': False, 'type': 'boolean', 'update_allowed': True}
    prop_port_security_enabled: lone Boolean,
    // YAML qos_policy: {'constraints': [{'custom_constraint': 'neutron.qos_policy'}], 'description':
        //'The name or ID of QoS policy to attach to this network.', 'immutable': False, 'required': False, 'type': 'string', 'update_allowed': True}
    prop_qos_policy: lone OS_Neutron_QoSPolicy_Role,
    // YAML shared: {'default': False, 'description': 'Whether this network should be shared across all tenants.
        //Note that the default policy setting restricts usage of this attribute to administrative users only.',
        //'immutable': False, 'required': False, 'type': 'boolean', 'update_allowed': True}
    prop_shared: lone Boolean,
    // YAML tags: {'description': 'The tags to be added to the network.', 'immutable': False,
        //'required': False, 'schema': {'*': {'immutable': False, 'required': False, 'type': 'string', 'update_allowed': False,
        //'visited': True}, 'visited': True}, 'type': 'list', 'update_allowed': True, 'visited': True}
    prop_tags: lone OS_Neutron_Net_tags,
    // YAML tenant_id: {'constraints': [{'custom_constraint': 'keystone.project'}], 'description':
        //'The ID of the tenant which will own the network. Only administrative users can set the tenant identifier;
        //this cannot be changed using authorization policies.', 'immutable': False, 'required': False,
        //'type': 'string', 'update_allowed': False}
    prop_tenant_id: lone OS_Keystone_Project_Role,
    // YAML value_specs: {'default': {}, 'description': 'Extra parameters to include in the request.
        // Parameters are often specific to installed hardware or extensions.', 'immutable':
        //False, 'required': False, 'type': 'map', 'update_allowed': True}
    prop_value_specs: lone MapValue
}{ /** Attributes */

    // YAML admin_state_up: {'description': 'The administrative status of the network.', 'type': 'string'}
```

```
attribute[attr_admin_state_up]
attr_admin_state_up.name = ''admin_state_up''
attr_admin_state_up.description = ''The administrative status of the network.''
attr_admin_state_up.type = ''string''

// YAML l2_adjacency: {'description': 'A boolean value for L2 adjacency,
    //True means that you can expect L2 connectivity throughout the Network.', 'type': 'boolean'}
attribute[attr_l2_adjacency]
attr_l2_adjacency.name = ''l2_adjacency''
attr_l2_adjacency.description = ''A boolean value for L2 adjacency,'' +
    '' True means that you can expect L2 connectivity throughout the Network.''
attr_l2_adjacency.type = ''boolean''

// YAML mtu: {'description': 'The maximum transmission unit size(in bytes) for the network.',
    //'type': 'integer'}
attribute[attr_mtu]
attr_mtu.name = ''mtu''
attr_mtu.description = ''The maximum transmission unit size(in bytes) for the network.''
attr_mtu.type = ''integer''

// YAML name: {'description': 'The name of the network.', 'type': 'string'}
attribute[attr_name]
attr_name.name = ''name''
attr_name.description = ''The name of the network.''
attr_name.type = ''string''

// YAML port_security_enabled: {'description': 'Port security enabled of the network.', 'type': 'boolean'}
attribute[attr_port_security_enabled]
attr_port_security_enabled.name = ''port_security_enabled''
attr_port_security_enabled.description = ''Port security enabled of the network.''
attr_port_security_enabled.type = ''boolean''

// YAML qos_policy_id: {'description': 'The QoS policy ID attached to this network.', 'type': 'string'}
attribute[attr_qos_policy_id]
attr_qos_policy_id.name = ''qos_policy_id''
attr_qos_policy_id.description = ''The QoS policy ID attached to this network.''
attr_qos_policy_id.type = ''string''

// YAML segments: {'description': 'The segments of this network.', 'type': 'list'}
attribute[attr_segments]
attr_segments.name = ''segments''
attr_segments.description = ''The segments of this network.''
attr_segments.type = ''list''

// YAML show: {'description': 'Detailed information about resource.', 'type': 'map'}
attribute[attr_show]
attr_show.name = ''show''
attr_show.description = ''Detailed information about resource.''
attr_show.type = ''map''

// YAML status: {'description': 'The status of the network.', 'type': 'string'}
attribute[attr_status]
attr_status.name = ''status''
attr_status.description = ''The status of the network.''
attr_status.type = ''string''

// YAML subnets: {'description': 'Subnets of this network.', 'type': 'list'}
attribute[attr_subnets]
attr_subnets.name = ''subnets''
attr_subnets.description = ''Subnets of this network.''
attr_subnets.type = ''list''
```

```
// YAML tenant_id: {'description': 'The tenant owning this network.', 'type': 'string'}
attribute[attr_tenant_id]
attr_tenant_id.name = ''tenant_id''
attr_tenant_id.description = ''The tenant owning this network.''
attr_tenant_id.type = ''string''

/** Properties */

// YAML admin_state_up: {'default': True, 'description': 'A boolean value specifying
    //the administrative status of the network.', 'immutable': False, 'required': False,
    //'type': 'boolean', 'update_allowed': True}
property[prop_admin_state_up]

// YAML dhcp_agent_ids: {'description': 'The IDs of the DHCP agent to schedule the network.
    //Note that the default policy setting in Neutron restricts usage of this property to administrative users only.
    //', 'immutable': False, 'required': False, 'type': 'list', 'update_allowed': True}
property[prop_dhcp_agent_ids]

// YAML dns_domain: {'constraints': [{'custom_constraint': 'dns_domain'}], 'description':
    //'DNS domain associated with this network.', 'immutable': False, 'required': False,
    //'type': 'string', 'update_allowed': True}
property[prop_dns_domain]

// YAML name: {'description': 'A string specifying a symbolic name for the network,
    //which is not required to be unique.', 'immutable': False, 'required': False,
    //'type': 'string', 'update_allowed': True}
property[prop_name]

// YAML port_security_enabled: {'description': 'Flag to enable/disable port security on the network.
    //It provides the default value for the attribute of the ports created on this network.',
    //'immutable': False, 'required': False, 'type': 'boolean', 'update_allowed': True}
property[prop_port_security_enabled]

// YAML qos_policy: {'constraints': [{'custom_constraint': 'neutron.qos_policy'}],
    //'description': 'The name or ID of QoS policy to attach to this network.', 'immutable': False,
    //'required': False, 'type': 'string', 'update_allowed': True}
property[prop_qos_policy]

// YAML shared: {'default': False, 'description': 'Whether this network should be shared across all tenants.
    //Note that the default policy setting restricts usage of this attribute to administrative users only.',
    //'immutable': False, 'required': False, 'type': 'boolean', 'update_allowed': True}
property[prop_shared]

// YAML tags: {'description': 'The tags to be added to the network.', 'immutable': False,
    //'required': False, 'schema': {'*': {'immutable': False, 'required': False, 'type': 'string',
    //'update_allowed': False, 'visited': True}, 'visited': True}, 'type': 'list', 'update_allowed': True, 'visited': True}
property[prop_tags]

// YAML tenant_id: {'constraints': [{'custom_constraint': 'keystone.project'}],
    //'description': 'The ID of the tenant which will own the network. Only administrative users can
    //set the tenant identifier; this cannot be changed using authorization policies.',
    //'immutable': False, 'required': False, 'type': 'string', 'update_allowed': False}
property[prop_tenant_id]

// YAML value_specs: {'default': {}, 'description': 'Extra parameters to include in the request.
    //Parameters are often specific to installed hardware or extensions.', 'immutable': False,
    //'required': False, 'type': 'map', 'update_allowed': True}
property[prop_value_specs]

/** Provided roles */

provided in OS_Neutron_Net_Role
```

```
    /** Required roles */

    required = depends_on + prop_qos_policy + prop_tenant_id
}

// Composed Properties

sig OS_Neutron_Net_tags extends HOT/ListOfProperties
{
}{ // YAML OS_Neutron_Net_tags: {'description': 'The tags to be added to the network.',
        //'immutable': False, 'required': False, 'schema': {'*': {'immutable': False, 'required': False,
        // 'type': 'string', 'update_allowed': False}}, 'type': 'list', 'update_allowed': True}
    prop_name = ''OS_Neutron_Net_tags''
    prop_required = false
    no prop_immutable
    no prop_update_allowed
    prop_description = ''The tags to be added to the network.''
    prop_type = ''list''
    no prop_default
    elements in OS_Neutron_Net_tags_elements
}

sig OS_Neutron_Net_tags_elements extends HOT/FlatProperty
{
}{ // YAML *: {'immutable': False, 'required': False, 'type': 'string', 'update_allowed': False}
    prop_name = ''*''
    prop_type = ''string''
    no prop_description
    prop_required = false
    no prop_immutable
    no prop_update_allowed
    no prop_default
    #(prop_value) <= 1
    prop_value in String
}

// Predicates and Functions

pred OS_Neutron_Net.prop_dhcp_agent_ids[elem: lone Any]
{
  elem in this.prop_dhcp_agent_ids
}

pred OS_Neutron_Net.prop_qos_policy[values: set ResourceRole]
{
  one value: values {
    value in this.prop_qos_policy
  }
}

pred OS_Neutron_Net.prop_tags[elem: lone String]
{ one p: OS_Neutron_Net_tags_elements {
    one elem implies (p.@prop_value = elem) else no p.@prop_value
  p in this.prop_tags.elements }
}

pred OS_Neutron_Net.prop_tenant_id[values: set ResourceRole]
{
  one value: values {
    value in this.prop_tenant_id
  }
```

```
}

/** There exists some OS_Neutron_Net*/

run Show_OS_Neutron_Net{
} for 0 but
  8 Int,
  exactly 1 OS_Neutron_Net,
  exactly 1 HOT/Resource,
  exactly 0 HOT/Template,
  exactly 0 HOT/Parameter,
  exactly 0 HOT/ParameterGroup,
  exactly 0 HOT/ParameterConstraint,
  exactly 0 HOT/Output,
  exactly 0 HOT/Condition,
  exactly 0 HOT/Property,
  exactly 11 HOT/Attribute,
  exactly 1 HOT/Support_status,
  exactly 1 HOT/ResourceRole,
  exactly 0 HOT/map_resourceRole/Map,
  exactly 0 HOT/map_string/Map,
  exactly 0 HOT/map_int/Map,
  exactly 0 HOT/map_boolean/Map,
  exactly 1 LG/Location,
  exactly 0 LG/LocationGraph,
  exactly 1 LG/Sort,
  exactly 1 LG/Process,
  exactly 1 LG/Name,
  exactly 1 LG/Role,
  exactly 16 LG/Value
  expect 1

/*****************************************************************************
*   OS::Neutron::Router
*****************************************************************************/

// Resource Role

sig OS_Neutron_Router_Role extends HOT/ResourceRole
{
}

// Resource

sig OS_Neutron_Router extends HOT/Resource
{

    /** Attributes */

    // YAML admin_state_up: {'description': 'Administrative state of the router.', 'type': 'string'}
    attr_admin_state_up: lone Attribute,
    // YAML external_gateway_info: {'description': 'Gateway network for the router.', 'type': 'map'}
    attr_external_gateway_info: lone Attribute,
    // YAML name: {'description': 'Friendly name of the router.', 'type': 'string'}
    attr_name: lone Attribute,
    // YAML show: {'description': 'Detailed information about resource.', 'type': 'map'}
    attr_show: lone Attribute,
    // YAML status: {'description': 'The status of the router.', 'type': 'string'}
    attr_status: lone Attribute,
    // YAML tenant_id: {'description': 'Tenant owning the router.', 'type': 'string'}
    attr_tenant_id: lone Attribute,
```

```
    /** Properties */

    // YAML admin_state_up: {'default': True, 'description': 'The administrative state of the router.',
        // 'immutable': False, 'required': False, 'type': 'boolean', 'update_allowed': True}
    prop_admin_state_up: lone Boolean,
    // YAML distributed: {'description': 'Indicates whether or not to create a distributed router.
        // NOTE: The default policy setting in Neutron restricts usage of this property to administrative users only.
        //This property can not be used in conjunction with the L3 agent ID.', 'immutable': False, 'required': False,
        //'type': 'boolean', 'update_allowed': False}
    prop_distributed: lone Boolean,
    // YAML external_gateway_info: {'description': 'External network gateway configuration for a router.', 'immutable': False,
        //'required': False, 'schema': {'enable_snat': {'description': 'Enables Source NAT on the router gateway. NOTE: The default
        //policy setting in Neutron restricts usage of this property to administrative users only.', 'immutable': False, 'required': False,
        //'type': 'boolean', 'update_allowed': True, 'visited': True}, 'external_fixed_ips': {'description':
        //'External fixed IP addresses for the gateway.', 'immutable': False, 'required': False,
        //'schema': {'*': {'immutable': False, 'required': False, 'schema': {'ip_address': {'constraints': [{'custom_constraint': 'ip_addr'}],
        //'description': 'External fixed IP address.', 'immutable': False, 'required': False, 'type': 'string', 'update_allowed': False, 'visited': True},
        //'subnet': {'constraints': [{'custom_constraint': 'neutron.subnet'}], 'description': 'Subnet of external fixed IP address.',
        //'immutable': False, 'required': False, 'type': 'string', 'update_allowed': False, 'visited': True}, 'visited': True},
        //'type': 'map', 'update_allowed': False, 'visited': True}, 'visited': True}, 'type': 'list', 'update_allowed': True,
        //'visited': True}, 'network': {'constraints': [{'custom_constraint': 'neutron.providernet'}], 'description':
        //'ID or name of the external network for the gateway.', 'immutable': False, 'required': True, 'type': 'string',
        //'update_allowed': True, 'visited': True}, 'visited': True}, 'type': 'map', 'update_allowed': True, 'visited': True}
    prop_external_gateway_info: lone OS_Neutron_Router_external_gateway_info,
    // YAML ha: {'description': 'Indicates whether or not to create a highly available router.
        //NOTE: The default policy setting in Neutron restricts usage of this property to administrative users only.
        //And now neutron do not support distributed and ha at the same time.', 'immutable': False,
        //'required': False, 'type': 'boolean', 'update_allowed': False}
    prop_ha: lone Boolean,
    // YAML l3_agent_ids: {'description': 'ID list of the L3 agent. User can specify multi-agents
        //for highly available router. NOTE: The default policy setting in Neutron restricts usage of
        //this property to administrative users only.', 'immutable': False, 'required': False,
        //'schema': {'*': {'immutable': False, 'required': False, 'type': 'string', 'update_allowed': False, 'visited': True},
        //'visited': True}, 'type': 'list', 'update_allowed': True, 'visited': True}
    prop_l3_agent_ids: lone OS_Neutron_Router_l3_agent_ids,
    // YAML name: {'description': 'The name of the router.', 'immutable': False,
        //'required': False, 'type': 'string', 'update_allowed': True}
    prop_name: lone String,
    // YAML tags: {'description': 'The tags to be added to the router.', 'immutable': False,
        //'required': False, 'schema': {'*': {'immutable': False, 'required': False, 'type': 'string',
        //'update_allowed': False, 'visited': True}, 'visited': True}, 'type': 'list', 'update_allowed': True, 'visited': True}
    prop_tags: lone OS_Neutron_Router_tags,
    // YAML value_specs: {'default': {}, 'description': 'Extra parameters to include in the creation request.',
        //'immutable': False, 'required': False, 'type': 'map', 'update_allowed': True}
    prop_value_specs: lone MapValue
}{ /** Attributes */

    // YAML admin_state_up: {'description': 'Administrative state of the router.', 'type': 'string'}
    attribute[attr_admin_state_up]
    attr_admin_state_up.name = ''admin_state_up''
    attr_admin_state_up.description = ''Administrative state of the router.''
    attr_admin_state_up.type = ''string''

    // YAML external_gateway_info: {'description': 'Gateway network for the router.', 'type': 'map'}
    attribute[attr_external_gateway_info]
    attr_external_gateway_info.name = ''external_gateway_info''
    attr_external_gateway_info.description = ''Gateway network for the router.''
    attr_external_gateway_info.type = ''map''

    // YAML name: {'description': 'Friendly name of the router.', 'type': 'string'}
    attribute[attr_name]
    attr_name.name = ''name''
```

```
attr_name.description = ''Friendly name of the router.''
attr_name.type = ''string''

// YAML show: {'description': 'Detailed information about resource.', 'type': 'map'}
attribute[attr_show]
attr_show.name = ''show''
attr_show.description = ''Detailed information about resource.''
attr_show.type = ''map''

// YAML status: {'description': 'The status of the router.', 'type': 'string'}
attribute[attr_status]
attr_status.name = ''status''
attr_status.description = ''The status of the router.''
attr_status.type = ''string''

// YAML tenant_id: {'description': 'Tenant owning the router.', 'type': 'string'}
attribute[attr_tenant_id]
attr_tenant_id.name = ''tenant_id''
attr_tenant_id.description = ''Tenant owning the router.''
attr_tenant_id.type = ''string''

/** Properties */

// YAML admin_state_up: {'default': True, 'description': 'The administrative state of the router.',
    //'immutable': False, 'required': False, 'type': 'boolean', 'update_allowed': True}
property[prop_admin_state_up]

// YAML distributed: {'description': 'Indicates whether or not to create a distributed router.
    //NOTE: The default policy setting in Neutron restricts usage of this property to administrative users only.
    //This property can not be used in conjunction with the L3 agent ID.', 'immutable': False,
    //'required': False, 'type': 'boolean', 'update_allowed': False}
property[prop_distributed]

    // YAML external_gateway_info: {'description': 'External network gateway configuration for a router.',
    //'immutable': False, 'required': False, 'schema': {'enable_snat': {'description': 'Enables Source NAT
    //on the router gateway. NOTE: The default policy setting in Neutron restricts usage of this property
    //to administrative users only.', 'immutable': False, 'required': False, 'type': 'boolean', 'update_allowed': True, 'visited': True},
    //'external_fixed_ips': {'description': 'External fixed Iaddresses for the gateway.', 'immutable': False,
    //'required': False, 'schema': {'*': {'immutable': False, 'required': False,
    //'schema': {'ip_address': {'constraints': [{'custom_constraint': 'ip_addr'}], 'description': 'External fixed IP address.',
    //'immutable': False, 'required': False, 'type': 'string', 'update_allowed': False, 'visited': True},
    //'subnet': {'constraints': [{'custom_constraint': 'neutron.subnet'}], 'description':
    //'Subnet of external fixed IP address.', 'immutable': False, 'required': False, 'type': 'string',
    //'update_allowed': False, 'visited': True}, 'visited': True}, 'type': 'map', 'update_allowed': False,
    //'visited': True}, 'visited': True}, 'type': 'list', 'update_allowed': True, 'visited': True},
    //'network': {'constraints': [{'custom_constraint': 'neutron.providernet'}], 'description':
    //'ID or name of the external network for the gateway.', 'immutable': False, 'required': True,
    //'type': 'string', 'update_allowed': True, 'visited': True}, 'visited': True},
    //'type': 'map', 'update_allowed': True, 'visited': True}
property[prop_external_gateway_info]

// YAML ha: {'description': 'Indicates whether or not to create a highly available router.
    //NOTE: The default policy setting in Neutron restricts usage of this property to administrative
    //users only. And now neutron do not support distributed and ha at the same time.',
    //'immutable': False, 'required': False, 'type': 'boolean', 'update_allowed': False}
property[prop_ha]

// YAML l3_agent_ids: {'description': 'ID list of the L3 agent. User can
    //specify multi−agents for highly available router. NOTE: The default policy setting in
    //Neutron restricts usage of this property to administrative users only.', 'immutable': False,
    //'required': False, 'schema': {'*': {'immutable': False, 'required': False, 'type': 'string',
    //'update_allowed': False, 'visited': True}, 'visited': True}, 'type': 'list',
```

```
        //'update_allowed': True, 'visited': True}
    property[prop_l3_agent_ids]

    // YAML name: {'description': 'The name of the router.', 'immutable': False,
        //'required': False, 'type': 'string', 'update_allowed': True}
    property[prop_name]

    // YAML tags: {'description': 'The tags to be added to the router.', 'immutable': False,
        //'required': False, 'schema': {'*': {'immutable': False, 'required': False, 'type': 'string',
       //'update_allowed': False, 'visited': True}, 'visited': True}, 'type': 'list',
       //'update_allowed': True, 'visited': True}
    property[prop_tags]

    // YAML value_specs: {'default': {}, 'description': 'Extra parameters to
        //include in the creation request.', 'immutable': False, 'required': False,
        //'type': 'map', 'update_allowed': True}
    property[prop_value_specs]

    /** Provided roles */

    provided in OS_Neutron_Router_Role

    /** Required roles */

    required = depends_on +
        prop_external_gateway_info.(OS_Neutron_Router_external_gateway_info <: external_fixed_ips)
        .elements.(OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements <: subnet).prop_value +
        prop_external_gateway_info.(OS_Neutron_Router_external_gateway_info <: network).prop_value
}

// Composed Properties

sig OS_Neutron_Router_external_gateway_info extends HOT/MapOfProperties
{
    enable_snat: lone OS_Neutron_Router_external_gateway_info_enable_snat,
    external_fixed_ips: lone OS_Neutron_Router_external_gateway_info_external_fixed_ips,
    network: one OS_Neutron_Router_external_gateway_info_network

}{ // YAML OS_Neutron_Router_external_gateway_info: {'description': 'External network gateway configuration for a router.',
        //'immutable': False, 'required': False, 'schema': {'enable_snat': {'description': 'Enables Source
        //NAT on the router gateway. NOTE: The default policy setting in Neutron restricts usage of this property
        //to administrative users only.', 'immutable': False, 'required': False, 'type': 'boolean', 'update_allowed': True},
        //'external_fixed_ips': {'description': 'External fixed IP addresses for the gateway.', 'immutable': False,
        //'required': False, 'schema': {'*': {'immutable': False, 'required': False,
        //'schema': {'ip_address': {'constraints': [{'custom_constraint': 'ip_addr'}], 'description':
        //'External fixed IP address.', 'immutable': False, 'required': False, 'type': 'string',
        //'update_allowed': False}, 'subnet': {'constraints': [{'custom_constraint': 'neutron.subnet'}],
        //'description': 'Subnet of external fixed IP address.', 'immutable': False, 'required': False,
        //'type': 'string', 'update_allowed': False}}, 'type': 'map', 'update_allowed': False}},
        //'type': 'list', 'update_allowed': True}, 'network': {'constraints': [{'custom_constraint': 'neutron.providernet'}],
        //'description': 'ID or name of the external network for the gateway.', 'immutable': False,
        //'required': True, 'type': 'string', 'update_allowed': True}}, 'type': 'map', 'update_allowed': True}
    prop_name = ''OS_Neutron_Router_external_gateway_info''
    prop_required = false
    no prop_immutable
    no prop_update_allowed
    prop_description = ''External network gateway configuration for a router.''
    prop_type = ''map''
    no prop_default
}

sig OS_Neutron_Router_external_gateway_info_enable_snat extends HOT/FlatProperty
```

```
{
}{ // YAML enable_snat: {'description': 'Enables Source NAT on the router gateway.
        //NOTE: The default policy setting in Neutron restricts usage of this property
        //to administrative users only.', 'immutable': False, 'required': False, 'type': 'boolean', 'update_allowed': True}
    prop_name = ''enable_snat''
    prop_type = ''boolean''
    prop_description = ''Enables Source NAT on the router gateway.'' +
        ''NOTE: The default policy setting in Neutron restricts'' +
        ''usage of this property to administrative users only.''
    prop_required = false
    no prop_immutable
    no prop_update_allowed
    no prop_default
    #(prop_value) <= 1
    prop_value in Boolean
}

sig OS_Neutron_Router_external_gateway_info_external_fixed_ips extends HOT/ListOfProperties
{
}{ // YAML OS_Neutron_Router_external_gateway_info_external_fixed_ips: {'description':
        //'External fixed IP addresses for the gateway.', 'immutable': False, 'required': False,
        //'schema': {'*': {'immutable': False, 'required': False,
        //'schema': {'ip_address': {'constraints': [{'custom_constraint': 'ip_addr'}],
        //'description': 'External fixed IP address.', 'immutable': False, 'required': False,
        //'type': 'string', 'update_allowed': False},
        //'subnet': {'constraints': [{'custom_constraint': 'neutron.subnet'}],
        //'description': 'Subnet of external fixed IP address.', 'immutable': False,
        //'required': False, 'type': 'string', 'update_allowed': False}},
        //'type': 'map', 'update_allowed': False}}, 'type': 'list', 'update_allowed': True}
    prop_name = ''OS_Neutron_Router_external_gateway_info_external_fixed_ips''
    prop_required = false
    no prop_immutable
    no prop_update_allowed
    prop_description = ''External fixed IP addresses for the gateway.''
    prop_type = ''list''
    no prop_default
    elements in OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements
}

sig OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements extends HOT/MapOfProperties
{
    ip_address: lone OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements_ip_address,
    subnet: lone OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements_subnet

}{ // YAML OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements:
        //{'immutable': False, 'required': False, 'schema': {'ip_address': {'constraints': [{'custom_constraint': 'ip_addr'}],
        //'description': 'External fixed IP address.', 'immutable': False, 'required': False,
        //'type': 'string', 'update_allowed': False}, 'subnet': {'constraints': [{'custom_constraint': 'neutron.subnet'}],
        //'description': 'Subnet of external fixed IP address.', 'immutable': False, 'required': False,
        //'type': 'string', 'update_allowed': False}}, 'type': 'map', 'update_allowed': False}
    prop_name = ''OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements''
    prop_required = false
    no prop_immutable
    no prop_update_allowed
    no prop_description
    prop_type = ''map''
    no prop_default
}

sig OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements_ip_address extends HOT/FlatProperty
{
}{ // YAML ip_address: {'constraints': [{'custom_constraint': 'ip_addr'}],
```

```
        //'description': 'External fixed IP address.', 'immutable': False,
        //'required': False, 'type': 'string', 'update_allowed': False}
    prop_name = ''ip_address''
    prop_type = ''string''
    prop_description = ''External fixed IP address.''
    prop_required = false
    no prop_immutable
    no prop_update_allowed
    no prop_default
    #(prop_value) <= 1
    prop_value in String
}

sig OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements_subnet extends HOT/FlatProperty
{
}{ // YAML subnet: {'constraints': [{'custom_constraint': 'neutron.subnet'}],
        //'description': 'Subnet of external fixed IP address.', 'immutable': False,
        //'required': False, 'type': 'string', 'update_allowed': False}
    prop_name = ''subnet''
    prop_type = ''string''
    prop_description = ''Subnet of external fixed IP address.''
    prop_required = false
    no prop_immutable
    no prop_update_allowed
    no prop_default
    #(prop_value) <= 1
    prop_value in OS_Neutron_Subnet_Role
}

sig OS_Neutron_Router_external_gateway_info_network extends HOT/FlatProperty
{
}{ // YAML network: {'constraints': [{'custom_constraint': 'neutron.providernet'}],
        //'description': 'ID or name of the external network for the gateway.',
        //'immutable': False, 'required': True, 'type': 'string', 'update_allowed': True}
    prop_name = ''network''
    prop_type = ''string''
    prop_description = ''ID or name of the external network for the gateway.''
    prop_required = true
    no prop_immutable
    no prop_update_allowed
    no prop_default
    #(prop_value) <= 1
    prop_value in OS_Neutron_ProviderNet_Role
}

sig OS_Neutron_Router_l3_agent_ids extends HOT/ListOfProperties
{
}{ // YAML OS_Neutron_Router_l3_agent_ids: {'description': 'ID list of the L3 agent.
        //User can specify multi-agents for highly available router. NOTE:
        //The default policy setting in Neutron restricts usage of this property to
        //administrative users only.', 'immutable': False, 'required': False,
        //'schema': {'*': {'immutable': False, 'required': False, 'type': 'string',
        //'update_allowed': False}}, 'type': 'list', 'update_allowed': True}
    prop_name = ''OS_Neutron_Router_l3_agent_ids''
    prop_required = false
    no prop_immutable
    no prop_update_allowed
    prop_description = ''ID list of the L3 agent. User can specify multi-agents'' +
        ''for highly available router. NOTE: The default policy setting'' +
        ''in Neutron restricts usage of this property to administrative users only.''
    prop_type = ''list''
    no prop_default
```

```
    elements in OS_Neutron_Router_l3_agent_ids_elements
}

sig OS_Neutron_Router_l3_agent_ids_elements extends HOT/FlatProperty
{
}{ // YAML *: {'immutable': False, 'required': False, 'type': 'string', 'update_allowed': False}
    prop_name = ''*''
    prop_type = ''string''
    no prop_description
    prop_required = false
    no prop_immutable
    no prop_update_allowed
    no prop_default
    #(prop_value) <= 1
    prop_value in String
}

sig OS_Neutron_Router_tags extends HOT/ListOfProperties
{
}{ // YAML OS_Neutron_Router_tags: {'description': 'The tags to be added to the router.',
        //'immutable': False, 'required': False,
        //'schema': {'*': {'immutable': False, 'required': False,
        //'type': 'string', 'update_allowed': False}}, 'type': 'list', 'update_allowed': True}
    prop_name = ''OS_Neutron_Router_tags''
    prop_required = false
    no prop_immutable
    no prop_update_allowed
    prop_description = ''The tags to be added to the router.''
    prop_type = ''list''
    no prop_default
    elements in OS_Neutron_Router_tags_elements
}

sig OS_Neutron_Router_tags_elements extends HOT/FlatProperty
{
}{ // YAML *: {'immutable': False, 'required': False, 'type': 'string', 'update_allowed': False}
    prop_name = ''*''
    prop_type = ''string''
    no prop_description
    prop_required = false
    no prop_immutable
    no prop_update_allowed
    no prop_default
    #(prop_value) <= 1
    prop_value in String
}

// Predicates and Functions

pred OS_Neutron_Router.prop_external_gateway_info[enable_snat: lone Boolean, external_fixed_ips_elements:
set OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements, network: set OS_Neutron_ProviderNet_Role]
{ one p: OS_Neutron_Router_external_gateway_info {
    one enable_snat implies (p.@enable_snat.prop_value = enable_snat) else no p.@enable_snat
    (#external_fixed_ips_elements > 0) implies (one p.@external_fixed_ips and
    (p.@external_fixed_ips.elements = external_fixed_ips_elements)) else no p.@external_fixed_ips
    (#network > 0) implies (bind_one_role[p.@network.prop_value, network]) else no p.@network
  this.prop_external_gateway_info = p }
}

fun OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements[ip_address: lone String,
subnet: set OS_Neutron_Subnet_Role] : one OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements
{
```

```
  { p: OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements | (
    p.@ip_address.prop_value = ip_address and
    bind_one_role[p.@subnet.prop_value, subnet] )
  }
}

pred OS_Neutron_Router.prop_l3_agent_ids[elem: lone String]
{ one p: OS_Neutron_Router_l3_agent_ids_elements {
    one elem implies (p.@prop_value = elem) else no p.@prop_value
  p in this.prop_l3_agent_ids.elements }
}

pred OS_Neutron_Router.prop_tags[elem: lone String]
{ one p: OS_Neutron_Router_tags_elements {
    one elem implies (p.@prop_value = elem) else no p.@prop_value
  p in this.prop_tags.elements }
}

/** There exists some OS_Neutron_Router*/

run Show_OS_Neutron_Router{
} for 0 but
  8 Int,
  exactly 1 OS_Neutron_Router,
  exactly 1 HOT/Resource,
  exactly 0 HOT/Template,
  exactly 0 HOT/Parameter,
  exactly 0 HOT/ParameterGroup,
  exactly 0 HOT/ParameterConstraint,
  exactly 0 HOT/Output,
  exactly 0 HOT/Condition,
  exactly 0 HOT/Property,
  exactly 6 HOT/Attribute,
  exactly 1 HOT/Support_status,
  exactly 2 HOT/ResourceRole,
  exactly 0 HOT/map_resourceRole/Map,
  exactly 0 HOT/map_string/Map,
  exactly 0 HOT/map_int/Map,
  exactly 0 HOT/map_boolean/Map,
  exactly 1 LG/Location,
  exactly 0 LG/LocationGraph,
  exactly 1 LG/Sort,
  exactly 1 LG/Process,
  exactly 1 LG/Name,
  exactly 2 LG/Role,
  exactly 12 LG/Value
  expect 1

/*****************************************************************************
 * Facts for the invariants that are written in english in the spec (HOT_types.yaml)
*****************************************************************************/

/*************** facts for the invariants of OS_Neutron_Router *****************/

// distributed must not be used in conjuction with l3_agent_ids
fact not_distributed_with_l3_agent
{
 all router: OS_Neutron_Router | not (router.prop_distributed = true and #(router.prop_l3_agent_ids.elements.prop_value) > 0)
}

// ha and distributed cannot be specified at the same time
fact not_both_ha_and_distributed
```

```
{
 all router: OS_Neutron_Router | not (router.prop_distributed = true and router.prop_ha = true)
}

// subnet must belong to network
fact router_subnet_must_belong_to_network
{
 all router: OS_Neutron_Router |
 let router_subnet = router.prop_external_gateway_info.external_fixed_ips.elements.
 (OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements <: subnet).prop_value,
 router_net = router.prop_external_gateway_info.network.prop_value
  {
     #router_subnet > 0 implies router_subnet.~provided.(OS_Neutron_Subnet <: prop_network).~provided = router_net.~provided
  }
}

/********** facts for the invariants of OS_Neutron_RouterInterface **********/

// either port or subnet must be specified
fact routerInterface_either_port_or_subnet
{
 all routerInterface: OS_Neutron_RouterInterface|
 (one routerInterface.prop_port or one routerInterface.prop_subnet) and
 not (one routerInterface.prop_port and one routerInterface.prop_subnet)
}

/********* facts for the invariants of OS_Neutron_SecurityGroup *********/

// if the protocol is tcp or udp, port_range_min must be less than or equal to port_range_max
fact port_range_min_less_or_equal_to_port_range_max
{
 all security_group: OS_Neutron_SecurityGroup |
 let protocol = security_group.prop_rules.elements.protocol.prop_value,
 port_range_min = security_group.prop_rules.elements.port_range_min.prop_value,
 port_range_max = security_group.prop_rules.elements.port_range_max.prop_value
 {
   protocol in "tcp" + "udp" implies port_range_min <= port_range_max
 }
}

/********** facts for the invariants of OS_Nova_Flavor **********/

// flavorid must be unique
fact unique_flavor_id
{
 all disj f1, f2: OS_Nova_Flavor | (one f1.prop_flavorid and one f2.prop_flavorid) implies
                                   f1.prop_flavorid != f2.prop_flavorid
}

/********** facts for the invariants of OS_Cinder_Volume **********/

// size is required unless property backup_id or source_volid or snapshot_id is specified
fact size_is_required_if_no_backup_id_no_source_volid_and_no_snapshot_id
{
 all c: OS_Cinder_Volume |
   (no c.prop_backup_id and no c.prop_source_volid and no c.prop_snapshot_id) implies one c.prop_size
}

/********** facts for the invariants of OS_Neutron_ProviderNet **********/

// name must be unique
fact unique_ProviderNet_name
```

```
{
 all disj p1, p2: OS_Neutron_ProviderNet | (one p1.prop_name and one p2.prop_name) implies
                                           p1.prop_name != p2.prop_name
}

/********** facts for the invariants of OS_Keystone_User ****************************/

// not both domain and project
fact not_both_domain_and_project_for_user
{
 all user: OS_Keystone_User |
  not ( user.prop_roles.elements.(OS_Keystone_User_roles_elements <: domain).prop_value > 0 and
        user.prop_roles.elements.(OS_Keystone_User_roles_elements <: project).prop_value > 0
      )
}

/************ facts for the invariants of OS_Keystone_Group ***********************/

// not both domain and project
fact not_both_domain_and_project_for_group
{
 all group: OS_Keystone_Group |
   not ( group.prop_roles.elements.(OS_Keystone_Group_roles_elements <: domain).prop_value > 0 and
         group.prop_roles.elements.(OS_Keystone_Group_roles_elements <: project).prop_value > 0
       )
}

/********* facts for the invariants of OS_Keystone_Region **************************/

// id must be unique in the openstack
fact id_must_be_unique_in_a_template
{
 all disj region1, region2: OS_Keystone_Region | region1.prop_id != region2.prop_id
}

/***************** facts for the invariants of OS_Heat_ScalingPolicy ******************/

// min_adjustment_step can be used only when specifying percent_change_in_capacity for the adjustment_type property
fact min_adjustment_step_can_be_used_only_when_adjustment_type_is_equal_to_percent_change_in_capacity
{
  all scalingP: OS_Heat_ScalingPolicy |
       not ( one scalingP.prop_min_adjustment_step and
             scalingP.prop_adjustment_type != "percent_change_in_capacity"
           )
}

/*******************************************************************************
 * Facts for the ommitted invariants
*******************************************************************************/

/***************** facts for the invariants of OS_Neutron_Net ***********************/

// if port_security_enabled is false, the ports of this network must not
    // have security_groups. If not, deployment_error in openstack
fact no_port_with_security_groups_on_a_network_when_its_port_security_enabled_is_false
{
  all net: OS_Neutron_Net, port: OS_Neutron_Port |
  let subnet = port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet).prop_value.~provided,
  subnet_id = port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet_id).prop_value.~provided,
  network = port.prop_network.~provided,
  port_all_networks = network + subnet.(OS_Neutron_Subnet <: prop_network).~provided +
  subnet_id.(OS_Neutron_Subnet <: prop_network).~provided
```

```
  {
    (net.prop_port_security_enabled = false and net in port_all_networks) implies
     no port.prop_security_groups
  }
}

// if port_security_enabled is false, the ports of this network must not
    // have allowed_address_pairs. If not, deployment_error in openstack
fact no_port_with_allowed_address_pairs_on_a_network_when_its_port_security_enabled_is_false
{
  all net: OS_Neutron_Net, port: OS_Neutron_Port |
  let subnet = port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet).prop_value.~provided,
  subnet_id = port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet_id).prop_value.~provided,
  network = port.prop_network.~provided,
  port_all_networks = network + subnet.(OS_Neutron_Subnet <: prop_network).~provided +
  subnet_id.(OS_Neutron_Subnet <: prop_network).~provided
  {
    (net.prop_port_security_enabled = false and net in port_all_networks) implies
     no port.prop_allowed_address_pairs
  }
}

/***************** facts for the invariants of OS_Neutron_Subnet ***************************/

// allocation_pools: end must be greater or equal to start
   // if not: deployment error if not in openstack
fact allocation_pools_end_must_be_greater_or_equal_to_start
{
 all subnet: OS_Neutron_Subnet |
    (#subnet.prop_allocation_pools.elements.end.prop_value > 0 and
    subnet.prop_allocation_pools.elements.start.prop_value > 0 ) implies
    subnet.prop_allocation_pools.elements.end.prop_value >= subnet.prop_allocation_pools.elements.start.prop_value
}

// subnets of the same network must have disjointed allocation_pools
   // if not, deployment error in openstack
fact no_subnets_of_the_same_network_with_joined_allocation_pools
{
  all disj subnet1, subnet2: OS_Neutron_Subnet |
  subnet1.prop_network.~provided = subnet2.prop_network.~provided implies
  #(subnet1.prop_allocation_pools.elements & subnet2.prop_allocation_pools.elements) = 0
}

// gateway_ip must be different from allocation_pools_end and allocation_pools_start
   // if not, deployment error in openstack
fact gateway_ip_different_from_allocation_pools_end_and_start
{
 all subnet: OS_Neutron_Subnet | one subnet.prop_gateway_ip implies
  ( subnet.prop_gateway_ip != subnet.prop_allocation_pools.elements.end.prop_value and
    subnet.prop_gateway_ip != subnet.prop_allocation_pools.elements.start.prop_value
  )
}

// if ip_version is equal to 4 and prefixlen is lower than 8 min_prefixlen of the associated subnetpool
// must be defined and must be lower than (or equal to) prefixlen
   // if not, deployment error in openstack. The reason is that min_prefixlen defaults to 8 for ipv4
fact when_ip_version_is_4_and_prefixlen_is_lower_than_8_min_prefixlen_of_the_subnetpool_must_be_defined
{
 all subnet: OS_Neutron_Subnet | (subnet.prop_ip_version = 4 and one subnet.prop_prefixlen and subnet.prop_prefixlen < 8) implies
                                  (one subnet.prop_subnetpool and
                                   subnet.prop_subnetpool.~provided.prop_min_prefixlen <= subnet.prop_prefixlen)
}
```

```
// ipv6_ra_mode and ipv6_address_mode must be equal if they are both specified
    // if not, parsing error in openstack
fact ipv6_ra_mode_must_be_equal_to_ipv6_address_mode_if_both_are_specified
{
  all subnet: OS_Neutron_Subnet | (one subnet.prop_ipv6_ra_mode and one subnet.prop_ipv6_address_mode) implies
                                  subnet.prop_ipv6_ra_mode = subnet.prop_ipv6_address_mode
}

// ipv6_ra_mode and ip_v6_address_mode must not be defined when ip_version = 4
    // if not, parsing error in openstack
fact ipv6_ra_mode_and_ip_v6_address_mode_must_not_be_defined_when_ip_version_is_4
{
 all subnet: OS_Neutron_Subnet | subnet.prop_ip_version = 4 implies
 (no subnet.prop_ipv6_ra_mode and no subnet.prop_ipv6_address_mode)
}

// one property between cidr or subnetpool must be specified
    // if not, parsing error in openstack
fact one_property_between_cidr_or_subnetpool_must_be_specified
{
 all subnet: OS_Neutron_Subnet | one subnet.prop_cidr or one subnet.prop_subnetpool
}

// cidr and prefixlen cannot be defined in the same instant
   // if not, parsing error in openstack
fact not_both_cidr_and_prefixlen
{
 all subnet: OS_Neutron_Subnet | not (one subnet.prop_cidr and one subnet.prop_prefixlen)
}

// cidr or subnetpool must be defined
   // if not, parsing error in openstack
fact cird_or_subnet_pool_must_be_defined
{
 all subnet: OS_Neutron_Subnet | one subnet.prop_cidr or one subnet.prop_subnetpool
}

/************ facts for the invariants of OS_Neutron_Router *********************/

// different routers in the same subnet or net cannot have the same ip_address
   // if not, deployment error in openstack
fact routers_of_the_same_subnet_or_net_cannot_have_the_same_ip_address
{
  all disj router1, router2: OS_Neutron_Router |
  let elems1 = router1.prop_external_gateway_info.external_fixed_ips.elements,
  elems2 = router2.prop_external_gateway_info.external_fixed_ips.elements,
  network1 = router1.prop_external_gateway_info.network.prop_value.~provided,
  network2 = router2.prop_external_gateway_info.network.prop_value.~provided,
  ip_addr1 = elems1.(OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements <: ip_address).prop_value,
  ip_addr2 = elems2.(OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements <: ip_address).prop_value,
  subnet1 = elems1.(OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements <: subnet).prop_value.~provided,
  subnet2 = elems2.(OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements <: subnet).prop_value.~provided
  {
    (one ip_addr1 and one ip_addr2) implies not ( ip_addr1 = ip_addr2 and (subnet1 = subnet2 or network1 = network2) )
  }
}

// external_gateway_info schema: external_fixed_ips schema* schema subnet must be
 // a subnet of network, if not deployment error in openstack
fact subnet_must_belong_to_network_for_a_router
{
```

```
 all router: OS_Neutron_Router |
 let elems = router.prop_external_gateway_info.external_fixed_ips.elements,
 subnet = elems.(OS_Neutron_Router_external_gateway_info_external_fixed_ips_elements <: subnet).prop_value,
 network = router.prop_external_gateway_info.network.prop_value
 {
   (#subnet > 0 and #network > 0) implies subnet.~provided.(OS_Neutron_Subnet <: prop_network).~provided = network.~provided
 }
}

// l3_agents_id cannot be used when ha = false
  // if not, parsing error in openstack
fact l3_agent_ids_cannot_be_used_when_ha_is_false
{
  all router: OS_Neutron_Router |
  not (#router.prop_l3_agent_ids.elements > 0 and router.prop_ha = false)
}

/****************** facts for the invariants of OS_Neutron_Router_Interface ******************/

// different interfaces cannot have the same port
   // if not, deployment error in openstack
fact no_interfaces_with_the_same_port
{
 all disj interface1, interface2: OS_Neutron_RouterInterface |
 interface1.prop_port.~provided != interface2.prop_port.~provided
}

/*************** facts for the invariants of OS_Neutron_SecurityGroup ***********************/

// port_range_min, port_range_max can be used only when protocol is in [tcp, udp, udplite, sctp, dccp]
   // if not, deployment error in openstack
fact port_range_min_port_range_max_can_be_defined_only_for_some_protocols
{
 all security_group: OS_Neutron_SecurityGroup |
 let port_range_min = security_group.prop_rules.elements.port_range_min.prop_value,
 port_range_max = security_group.prop_rules.elements.port_range_max.prop_value,
 protocol = security_group.prop_rules.elements.protocol.prop_value
 {
   protocol not in "tcp" + "udp" + "udplite" + "sctp" + "dccp" implies
  ( no port_range_min and no port_range_max)
 }
}

// protocol must be not in [icmp−v6, ipv6−encap, ipv6−frag, ipv6−icmp, ipv6−nonxt, ipv6−opts, ipv6−route]
   // when ethertype = IPv4. If not, deployment error in openstack
fact protocol_must_not_be_an_ipv6_type_when_ethertype_is_ipv4
{
 all security_group: OS_Neutron_SecurityGroup |
 let protocol = security_group.prop_rules.elements.protocol.prop_value,
 ethertype = security_group.prop_rules.elements.ethertype.prop_value
 {
   ethertype = "IPv4" implies protocol not in "icmp−v6" + "ipv6−encap" + "ipv6−frag"
                                + "ipv6−icmp" + "ipv6−nonxt" + "ipv6−opts" + "ipv6−route"
 }
}

/*************** facts for the invariants of OS_Neutron_Port ************************/

// a port cannot have several ip_addresses on the same subnet
   // if not, deployment error in opensack
fact no_port_with_several_addresses_on_the_same_subnet
{
```

```
  all port: OS_Neutron_Port |
  all disj subnet1, subnet2: port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet).prop_value |
  subnet1.~provided != subnet2.~provided
}

// a port cannot have several ip_addresses on the same subnet_id
   // if not, deployment error in opensack
fact no_port_with_several_addresses_on_the_same_subnet_id
{
  all port: OS_Neutron_Port |
  all disj subnet_id1, subnet_id2: port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet_id).prop_value |
  subnet_id1.~provided != subnet_id2.~provided
}

// a port cannot have ip_adresses on a subnet and subnet_id that are the same
    //if not, deployment error in openstack
fact no_port_with_ip_adresses_on_subnet_and_subnet_id_that_are_the_same
{
 all port: OS_Neutron_Port |
 let subnets = port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet).prop_value,
 subnet_ids = port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet_id).prop_value
 {
   #(subnets.~provided & subnet_ids.~provided) = 0
 }
}

// different ports on the same subnet or net cannot have the same ip address
    // if not, deployment error in openstack
fact ports_on_the_same_net_or_subnet_must_have_distinct_ip_addresses
{
  all disj port1, port2: OS_Neutron_Port |
  let fixed_ips_elems1 = port1.prop_fixed_ips.elements,
  fixed_ips_elems2 = port2.prop_fixed_ips.elements,
  subnets1 = fixed_ips_elems1.(OS_Neutron_Port_fixed_ips_elements <: subnet).prop_value.~provided +
  fixed_ips_elems1.(OS_Neutron_Port_fixed_ips_elements <: subnet_id).prop_value.~provided,
  subnets2 = fixed_ips_elems2.(OS_Neutron_Port_fixed_ips_elements <: subnet).prop_value.~provided +
  fixed_ips_elems2.(OS_Neutron_Port_fixed_ips_elements <: subnet_id).prop_value.~provided,
  network1 = port1.prop_network.~provided,
  network2 = port2.prop_network.~provided,
  ip_addr1 = fixed_ips_elems1.(OS_Neutron_Port_fixed_ips_elements <: ip_address).prop_value,
  ip_addr2 = fixed_ips_elems2.(OS_Neutron_Port_fixed_ips_elements <: ip_address).prop_value
  {
   ( one ip_addr1 and one ip_addr2) implies not ( ip_addr1 = ip_addr2 and (subnets1 = subnets2 or network1 = network2) )
  }
}

// a port cannot have several ip addresses of the same version on the same network
   // if not, deployment error in openstack
fact port_fixed_ips_subnets_of_the_same_network_must_have_different_ip_versions
{
 all port: OS_Neutron_Port, subnet1_role, subnet2_role:
 port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet).prop_value |
 (subnet1_role.~provided.(OS_Neutron_Subnet <: prop_ip_version) = subnet2_role.~provided.(OS_Neutron_Subnet <: prop_ip_version) and
 subnet1_role.~provided.(OS_Neutron_Subnet <: prop_network).~provided =
 subnet2_role.~provided.(OS_Neutron_Subnet <:prop_network).~provided)
 implies subnet1_role = subnet2_role
}

// subnet and subnet_id cannot be defined at the same instant
   // if not, deployment error in openstack
fact not_both_subnet_and_subnet_id_for_a_port
{
```

```
  all port: OS_Neutron_Port |
  let subnet = port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet).prop_value,
  subnet_id = port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet_id).prop_value
  {
    not (#subnet > 0 and #subnet_id > 0)
  }
}*/

// if both are specified, subnet must belong to network
  //if not, deployment error in openstack

fact subnet_must_belong_to_network_for_a_port {
  all port: OS_Neutron_Port |
  let subnet = port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet).prop_value
  { (#subnet > 0 and one port.prop_network) implies
     subnet.~provided.(OS_Neutron_Subnet <: prop_network).~provided = port.prop_network.~provided} }

// if both are specified, subnet_id must belong to network
  // if not, deployment error in openstack
fact subnet_id_must_belong_to_network_for_a_port
{
  all port: OS_Neutron_Port |
  let subnet_id = port.prop_fixed_ips.elements.(OS_Neutron_Port_fixed_ips_elements <: subnet_id).prop_value
  {
    (#subnet_id > 0 and one port.prop_network) implies
     subnet_id.~provided.(OS_Neutron_Subnet <: prop_network).~provided = port.prop_network.~provided
  }
}

// different ports cannot have the same mac address
  // if not, deployment error in openstack
fact no_distinct_ports_with_the_same_mac_address
{
 all disj port1, port2: OS_Neutron_Port |
    (one port1.prop_mac_address and one port2.prop_mac_address) implies
     port1.prop_mac_address != port2.prop_mac_address
}

// security_groups, allowed_address_pairs must not be defined when port_security_enabled is false
    // if not, deployment error in openstack
fact no_security_groups_and_allowed_address_pairs_when_port_security_enabled_is_false
{
 all port: OS_Neutron_Port | port.prop_port_security_enabled = false implies
      (#port.prop_security_groups = 0 and no port.prop_allowed_address_pairs)
}

/********************** facts for the invariants of OS_Nova_Server ********************************/

// a server cannot be attached more than once to a port
  // if not, deployment error in openstack
fact no_server_attached_more_than_once_to_the_same_port
{
 all server: OS_Nova_Server, port1, port2: server.prop_networks.elements.port.prop_value |
  (port1.~provided = port2.~provided) implies port1 = port2
}

// no disctinct servers with the same port
    // if not, deployment error in openstack
fact no_servers_with_the_same_port
{
 all disj server1, server2: OS_Nova_Server |
   server1.prop_networks.elements.port.prop_value.~provided !=
```

```
    server2.prop_networks.elements.port.prop_value.~provided
}

// block_device_mapping and block_device_mapping_v2 cannot be defined at the same instant
   // if not, parsing error in openstack
fact not_block_device_mapping_and_block_device_mapping_v2
{
  all server: OS_Nova_Server |
    not ( one server.prop_block_device_mapping and
          one server.prop_block_device_mapping_v2
        )
}

// a server cannot have a volume more than once in block_device_mapping + block_device_mapping_v2
    // if not, deployment error in openstack
fact server_cannot_have_a_volume_id_more_than_once
{
  all server: OS_Nova_Server,
  disj vol_bm1, vol_bm2: server.prop_block_device_mapping.elements.(OS_Nova_Server_block_device_mapping_elements <: volume_id).prop_value,
  disj vol_bm1_v2, vol_bm2_v2: server.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: volume_id).p
  {
        vol_bm1.~provided != vol_bm2.~provided and
        vol_bm1_v2.~provided != vol_bm2_v2.~provided
  }
}

// different servers cannot have the same volume_id in block_device_mapping + block_device_mapping_v2
    // if not, deployment error in openstack
fact different_servers_must_have_different_volume_id_in_block_device_mapping_union_block_device_mapping_v2
{
  all disj server1, server2: OS_Nova_Server |
  let vols_id1 = server1.prop_block_device_mapping.elements.(OS_Nova_Server_block_device_mapping_elements <: volume_id).prop_value.~provided +
                 server1.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: volume_id).prop_value.~provi
  vols_id2 = server2.prop_block_device_mapping.elements.(OS_Nova_Server_block_device_mapping_elements <: volume_id).prop_value.~provided +
             server2.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: volume_id).prop_value.~provided
  {
    #(vols_id1 & vols_id2) = 0
  }
}

// different servers of the same subnet or net must have different fixed_ip_addresses
   // if not, deployment error in openstack
fact servers_of_the_same_subnet_or_net_must_have_different_fixed_ip_addresses
{
 all disj server1, server2: OS_Nova_Server |
 let fixed_ip1 = server1.prop_networks.elements.fixed_ip,
 fixed_ip2 = server2.prop_networks.elements.fixed_ip,
 subnet1 = server1.prop_networks.elements.(OS_Nova_Server_networks_elements <: subnet).prop_value.~provided,
 subnet2 = server2.prop_networks.elements.(OS_Nova_Server_networks_elements <: subnet).prop_value.~provided,
 net1 = server1.prop_networks.elements.(OS_Nova_Server_networks_elements <: network).prop_value.~provided +
        server1.prop_networks.elements.(OS_Nova_Server_networks_elements <: uuid).prop_value.~provided,
 net2 = server2.prop_networks.elements.(OS_Nova_Server_networks_elements <: network).prop_value.~provided +
        server2.prop_networks.elements.(OS_Nova_Server_networks_elements <: uuid).prop_value.~provided
 {
   not ( fixed_ip1 = fixed_ip2 and (subnet1 = subnet2 or net1 = net2) )
 }
}

// a server cannot have a fixed_ip_address, on the same subnet, more than once
    // if not, deployment error
fact no_server_with_several_times_the_same_fixed_ip_address_on_the_same_subnet
{
```

```
 all server: OS_Nova_Server, disj elem1, elem2: server.prop_networks.elements |
 let fixed_ip1 = elem1.fixed_ip.prop_value,
 fixed_ip2 = elem2.fixed_ip.prop_value,
 subnet1 = elem1.(OS_Nova_Server_networks_elements <: subnet).prop_value.~provided,
 subnet2 = elem2.(OS_Nova_Server_networks_elements <: subnet).prop_value.~provided,
 net1 = elem1.(OS_Nova_Server_networks_elements <: network).prop_value.~provided +
        elem1.(OS_Nova_Server_networks_elements <: uuid).prop_value.~provided,
 net2 = elem2.(OS_Nova_Server_networks_elements <: network).prop_value.~provided +
         elem2.(OS_Nova_Server_networks_elements <: uuid).prop_value.~provided
  {
    not ( fixed_ip1 = fixed_ip2 and (subnet1 = subnet2 or net1 = net2) )
  }
}

// network schema* schema: uuid and network cannot be specified at the same instant
   //if not, deployment error in openstack
fact not_uuid_and_network_for_a_server
{
  all server: OS_Nova_Server |
  let network = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: network).prop_value,
  uuid = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: uuid).prop_value
  {
    not (#network > 0 and #uuid > 0)
  }
}

// networks schema * schema: subnet must belong to network
   // if not, deployment error in openstack
fact subnet_must_belong_to_network_for_a_server
{
  all server: OS_Nova_Server |
  let subnet = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: subnet).prop_value,
  network = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: network).prop_value
  {
    (#subnet > 0 and #network > 0) implies subnet.~provided.(OS_Neutron_Subnet <: prop_network).~provided = network.~provided
  }
}

// networks schema* schema: floating_ip and port.floating_ip must not be on the same network
    // if not, deployment error
// 1. through floatingIP
fact floating_ip_and_port_floating_ip_must_not_have_the_same_network_fact1
{
  all server: OS_Nova_Server, a_floating_ip: OS_Neutron_FloatingIP |
  let server_floating_ips = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: floating_ip).prop_value,
  server_ports = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: port).prop_value
  {
    not ( a_floating_ip not in server_floating_ips.~provided and
          a_floating_ip.prop_port_id.~provided in server_ports.~provided and
          server_floating_ips.~provided.prop_floating_network.~provided =
          a_floating_ip.prop_floating_network.~provided
        )
  }
}

//2. through floating ip association
fact floating_ip_and_port_floating_ip_must_not_have_the_same_network_fact2
{
  all server: OS_Nova_Server, ip_assos: OS_Neutron_FloatingIPAssociation |
  let server_floating_ips = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: floating_ip).prop_value,
  server_ports = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: port).prop_value
  {
```

```
    not ( ip_assos.prop_floatingip_id.~provided not in server_floating_ips.~provided and
          ip_assos.prop_port_id.~provided in server_ports.~provided and
          ip_assos.prop_floatingip_id.~provided.prop_floating_network.~provided =
          server_floating_ips.~provided.prop_floating_network.~provided
        )
  }
}

// networks schema * schema: fixed_ip and port cannot be specified at the same instant
   // if not, parsing error in openstack
fact not_both_fixed_ip_and_port_for_a_server
{
  all server: OS_Nova_Server |
  let fixed_ip = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: fixed_ip).prop_value,
  port = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: port).prop_value
  {
    not (#fixed_ip > 0 and #port > 0)
  }
}

// block_device_mapping_v2: image and image_id cannot be defined at the same instant
   // if not, parsing error in openstack
fact not_image_and_image_id_for_block_device_mapping_v2
{
 all server: OS_Nova_Server |
 let image = server.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: image).prop_value,
 image_id = server.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: image_id).prop_value
 {
   not (#image > 0 and #image_id > 0)
 }
}

// block_device_mapping_v2: volume_id, snapshot_id, image, image_id, swap_size, ephemeral_size,
  // ephemeral_format cannot be defined, two by two or more, at the same instant.
  // if not, parsing error in openstack
fact not_volume_id_snapshot_id_image_image_id_swap_size_ephemeral_size_and_ephemeral_format
{
  all server: OS_Nova_Server |
  let volume_id = server.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: volume_id).prop_value,
  snapshot_id = server.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: snapshot_id).prop_value,
  image = server.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: image).prop_value,
  image_id = server.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: image_id).prop_value,
  swap_size = server.prop_block_device_mapping_v2.elements.swap_size.prop_value,
  ephemeral_size = server.prop_block_device_mapping_v2.elements.ephemeral_size.prop_value,
  ephemeral_format = server.prop_block_device_mapping_v2.elements.ephemeral_format.prop_value
  {
    #volume_id > 0 implies #snapshot_id + #image + #image_id + #swap_size + #ephemeral_size + #ephemeral_format = 0
    #image > 0 implies #volume_id + #snapshot_id + #image_id + #swap_size + #ephemeral_size + #ephemeral_format = 0
    #image_id > 0 implies #volume_id + #snapshot_id + #image + #swap_size + #ephemeral_size + #ephemeral_format = 0
    #swap_size > 0 implies #volume_id + #snapshot_id + #image + #image_id + #ephemeral_size + #ephemeral_format = 0
    #ephemeral_size > 0 implies #volume_id + #snapshot_id + #image + #image_id + #swap_size + #ephemeral_format = 0
    #ephemeral_format > 0 implies #volume_id + #snapshot_id + #image + #image_id + #swap_size + #ephemeral_size = 0
  }
}

// networks: fixed_ips and port cannot be defined in the same instant
  // if not, parsing error in openstack
fact no_networks_fixed_ip_and_port_for_a_server
{
  all server: OS_Nova_Server |
  let fixed_ip = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: fixed_ip).prop_value,
  port = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: port).prop_value
```

```
  {
    not (#fixed_ip > 0 and #port > 0)
  }
}

// if the server is associated to a software deployment,
  // user_data_format must be SOFTWARE_CONFIG
  // if not, parsing error
fact used_data_format_must_be_software_config_when_software_deployment_is_used_for_a_server
{
 all server: OS_Nova_Server, sfw_dep: OS_Heat_SoftwareDeployment |
                                sfw_dep.prop_server in server.provided implies
                                server.prop_user_data_format = "SOFTWARE_CONFIG"
}

// either an image or one bootable device must be specified
  // if not, parsing error in openstack
fact either_an_image_or_one_bootable_device_must_be_specified
{
 all server: OS_Nova_Server,
 non_bootable_devices: server.prop_block_device_mapping_v2.elements.boot_index.prop_value - (-1 + -2 + -3) |
 let server_image = server.prop_image,
 num_bootable_devices_block_device_mapping =
 #server.prop_block_device_mapping.elements.(OS_Nova_Server_block_device_mapping_elements <: snapshot_id).prop_value +
 #server.prop_block_device_mapping.elements.(OS_Nova_Server_block_device_mapping_elements <: volume_id).prop_value,
 num_devices_block_device_mapping_v2 =
 #server.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: image).prop_value +
 #server.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: volume_id).prop_value +
 #server.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: image_id).prop_value +
 #server.prop_block_device_mapping_v2.elements.(OS_Nova_Server_block_device_mapping_v2_elements <: snapshot_id).prop_value,
 number_bootable_devices = num_bootable_devices_block_device_mapping + num_devices_block_device_mapping_v2 - #non_bootable_devic
 {
  (one server_image or number_bootable_devices = 1) and not (one server_image and number_bootable_devices = 1)
 }

}

// networks schema* schema: at least, one of "network", "port", "allocate_network" or "subnet" must be defined
  // if not, parsing error in openstack
fact at_least_one_of_network_port_allocate_network_or_subnet_must_be_specified_for_a_server
{
 all server: OS_Nova_Server |
 let network = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: network).prop_value,
 port = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: port).prop_value,
 allocate_network = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: allocate_network).prop_value,
 subnet = server.prop_networks.elements.(OS_Nova_Server_networks_elements <: subnet).prop_value
 {
   #network > 0 or #port > 0 or #allocate_network > 0 or #subnet > 0
 }
}*/

/************* facts for the invariants of OS_Neutron_FloatingIP ************************/

// floating_subnet must belong to floating_network
  // if not, deployment error in openstack
fact floating_subnet_must_belong_to_floating_network
{
 all floating_ip: OS_Neutron_FloatingIP |
   (one floating_ip.prop_floating_subnet and one floating_ip.prop_floating_network) implies
   floating_ip.prop_floating_subnet.~provided.(OS_Neutron_Subnet <: prop_network).~provided =
   floating_ip.prop_floating_network.~provided
}
```

```
// a floating_ip_address cannot be used more than once
 // if not, deployment error
fact a_floating_ip_address_cannot_be_used_more_than_once
{
  all disj floating_ip1, floating_ip2: OS_Neutron_FloatingIP |
      (one floating_ip1.prop_port_id and one floating_ip2.prop_port_id) implies
      floating_ip1.prop_floating_ip_address != floating_ip2.prop_floating_ip_address
}

// different floating ips on the same subnet and net cannot be attached to the same port
 // if not, deployment error in openstack
fact no_floating_ips_on_the_same_subnet_and_net_attached_to_the_same_port
{
  all disj floating_ip1, floating_ip2: OS_Neutron_FloatingIP |
  let subnet1 = floating_ip1.prop_floating_subnet.~provided,
  subnet2 = floating_ip2.prop_floating_subnet.~provided,
  net1 = floating_ip1.prop_floating_network.~provided,
  net2 = floating_ip2.prop_floating_network.~provided,
  port1 = floating_ip1.prop_port_id.~provided,
  port2 = floating_ip2.prop_port_id.~provided
  {
    not ( subnet1 = subnet2 and net1 = net2 and port1 = port2 )
  }
}

/************* facts for the invariants of OS_keystone_Project ************************/

// projects of the same domain must have distinct name
    // if not, deployment error in openstack
fact distinct_name_for_projects_of_the_same_domain
{
  all disj project1, project2: OS_Keystone_Project |
       project1.prop_domain = project2.prop_domain implies
       project1.prop_name != project2.prop_name
}

/************* facts for the invariants of OS_keystone_User ************************/

// users of the same domain must have distinct name
    // if not, deployment error in openstack
fact distinct_name_for_users_of_the_same_domain
{
  all disj user1, user2: OS_Keystone_User |
       user1.prop_domain = user2.prop_domain implies
       user1.prop_name != user2.prop_name
}

/************* facts for the invariants of OS_keystone_Role ************************/

// roles of the same domain must have distinct name
    // if not, deployment error in openstack
fact distinct_name_for_roles_of_the_same_domain
{
  all disj role1, role2: OS_Keystone_Role |
       role1.prop_domain = role2.prop_domain implies
       role1.prop_name != role2.prop_name
}

/************* facts for the invariants of OS_keystone_Domain ************************/

// two domains (or more) cannot have the same name
```

```
  // if not, deployment error
fact distinct_name_for_domains
{
 all disj domain1, domain2: OS_Keystone_Domain | domain1.prop_name != domain2.prop_name
}


/************* facts for the invariants of OS_Keystone_Group ***********************/

// groups of the same domain cannot have the same name
    // if not, deployment error in openstack
fact distinct_name_for_groups
{
 all disj group1, group2: OS_Keystone_Group |
  group1.prop_domain = group2.prop_domain implies
  group1.prop_name != group2.prop_name
}

/************* facts for the invariants of OS_Neutron_SubnetPool ***********************/

// if address_scope is not specified or addess_scope.ip_version = 4
  // min_prefixlen must be > 0. If not, deployment error in openstack
fact min_prefixlen_for_ipv4_addresscope
{
 all subnetpool: OS_Neutron_SubnetPool |
       (no subnetpool.prop_address_scope or
       subnetpool.prop_address_scope.~provided.(OS_Neutron_AddressScope <: prop_ip_version) = 4 and
       one subnetpool.prop_min_prefixlen) implies subnetpool.prop_min_prefixlen > 0
}

// distinct subnetpools cannot have the same prefixes
    // if not, deployment error in openstack
fact disctinct_prefixes_for_subnetpools
{
 all disj subnetpool1, subnetpool2 : OS_Neutron_SubnetPool |
  subnetpool1.prop_prefixes.elements.prop_value !=
  subnetpool2.prop_prefixes.elements.prop_value
}

// max_prefixlen must be greater or_equal to min_prefixlen
 // if not, parsing error in openstack
fact max_prefixlen_must_be_greater_or_equal_to_min_prefixlen
{
  all subnetpool: OS_Neutron_SubnetPool |
   (one subnetpool.prop_max_prefixlen and one subnetpool.prop_min_prefixlen) implies
   subnetpool.prop_max_prefixlen >= subnetpool.prop_min_prefixlen
}

/****************** facts for the invariants of OS_Neutron_ProviderNet ***********************/

// physical_network is required when network_type is equal to flat
    // if not, parsing error in openstack
fact physical_network_is_required_when_network_type_is_flat
{
  all p_net: OS_Neutron_ProviderNet | p_net.prop_network_type = "flat" implies
                                        one p_net.prop_physical_network
}

/****************** facts for the invariants of OS_Cinder_volume ***************************/

// a volume with multiattach is false cannot be attached more than once
  // if not, deployment error in openstack
```

```
fact a_volume_with_multiattach_false_cannot_be_attached_more_than_once
{
 all vol: OS_Cinder_Volume, vol_att1, vol_att2: OS_Cinder_VolumeAttachment |
      (vol.prop_multiattach = false and
      vol_att1.prop_volume_id.~provided =
      vol_att2.prop_volume_id.~provided and
      one vol_att1.prop_instance_uuid and
      one vol_att2.prop_instance_uuid) implies vol_att1 = vol_att2
}

// when size is provided, only one of image, source_volid, snapshot_id can be specified
   // if not, parsing error in openstack
fact when_size_is_provided_only_of_image_source_volid_snapshot_id_must_be_specified
{
 all vol: OS_Cinder_Volume |
 let image = vol.prop_image,
 source_volid = vol.prop_source_volid,
 snapshot_id = vol.prop_snapshot_id,
 size = vol.prop_size
 {
  one size implies #(image + source_volid + snapshot_id) = 1
 }
}*/

/******************** facts for the invariants of OS_Heat_AutoScalingGroup ******************/

// desired_capacity must be between min_size and max_size
   // if not, parsing error in openstack
fact desired_capacity_must_be_between_min_size_and_max_size
{
 all scaling_group: OS_Heat_AutoScalingGroup |
    (one scaling_group.prop_desired_capacity and one scaling_group.prop_min_size) implies
    scaling_group.prop_desired_capacity >= scaling_group.prop_min_size and
    (one scaling_group.prop_desired_capacity and one scaling_group.prop_max_size) implies
    scaling_group.prop_desired_capacity <= scaling_group.prop_max_size
}

// min_size must be lower than (or equal to) max_size
fact min_size_must_be_lower_or_equal_to_max_size
{
 all scaling_group: OS_Heat_AutoScalingGroup |
      (one scaling_group.prop_min_size and
       one scaling_group.prop_max_size) implies
       scaling_group.prop_min_size <= scaling_group.prop_max_size
}
```

7.2 Limitations of the HOT specification

Producing an interpretation of HOT in our reference computational model requires to (1) read the HOT specification, (2) deploy HOT templates in order to understand the behavior of Heat, and (3) formalize the HOT specification with Location Graphs in Alloy. This allows us to identify 3 categories of errors in the HOT specification and correct them.

7.2.1 Errors identified by reading the HOT specification

We read the HOT specification for the structure of a template, and the resources types for 120 types of 17 services (e.g. Nova, Neutron). This allows us for identifying a first category of errors, in the specification, related to `missing required and/or type` fields in the parameters, parameter_groups,

resources, outputs and conditions sections.

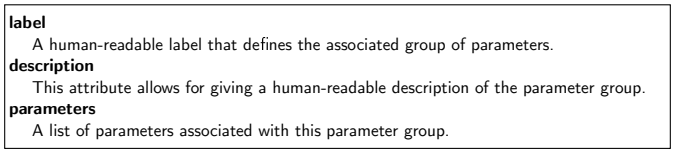


Figure 5: Parameter_groups specification

Example of error: for a parameter_group it is not specified whether the $label$ field is $required$ or not and its $type$ is not set (cf. Fig. 5). The same holds for the $name$ of a parameter. Therefore, when designing a template, one has to guess if these fields are $required$ and/or their $type$ (e.g. string). This must be avoided by explicitly specifying them.

Overview of the first category of errors: this category contains 11 errors. They were corrected in our formal interpretation of HOT by defining the missing type and/or required fields in the sections. For instance, for a parameter_group, we give the type string to the $label$ field and set it as required because it is used for identification purpose.

7.2.2 Errors identified by deploying HOT templates

We write a set of templates that conform the HOT specification and deploy them using Heat. We notice that some of these templates, even compliant with the specification, lead to a failed or partial deployment and Heat does not give an alert. One has to check the state of the deployment to know that it actually does not succeed. Other templates lead to a successful deployment but the deployed system is useless. This allows us to identify a second category of errors mainly related to `inconsistencies between the HOT specification and the behavior of Heat`.

First example of error: let us consider the resource type *OS::Neutron::FloatingIP*. Its YAML specification is the example given in Fig. 4. This specification states that a resource of this type has a property called floating_network. The value expected for this property is a string but must be the id of an *OS::Neutron::Net* resource (cf. the custom_constraint of this property). Based on this specification, we first write a HOT template that defines 1 network and 1 floating_ip with a floating_network value equal to the network. Note that this template conforms the HOT specification.

```
heat_template_version: stein
resources:
  network:
    type: OS::Neutron::Net

  floating_ip:
    type: OS::Neutron::FloatingIP
    properties:
      floating_network: { get_resource: network }
```

Then, we provide this template to Heat. After a while, when we check the deployment state, we notice that it is failed due to an error, as shown in Fig. 6. This error says that network is not a valid external network. Therefore, Heat is not waiting for the id of an *OS::Neutron::Net* but for the id of an *OS::Neutron::ProviderNet* which is the resource type corresponding to external network. The HOT specification should explicitly state this to prevent from such a failure.

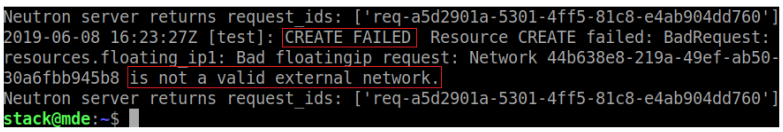


Figure 6: Result of the floatingIP template deployment

Second example of error: let us consider a small system that consists of a connected virtual machine. With HOT, this system can be designed using a set of resource types, provided by Nova and Neutron, in four different ways:

1. an *OS::Nova::Server* connected to an external network;

```
VM1:
  type: OS::Nova::Server
  properties:
    flavor: { get_param: flavor }
    image: { get_param: image }
    networks:
      - network: { get_param: public_network }
```

2. an *OS::Nova::Server* connected to a private network;

```
VM2:
  type: OS::Nova::Server
  properties:
    flavor: { get_param: flavor }
    image: { get_param: image }
    networks:
      - network: { get_resource: network2 }

network2:
  type: OS::Neutron::Net

subnet2:
  type: OS::Neutron::Subnet
  properties:
    network: { get_resource: network2 }
    cidr: 10.2.0.0/24
    ip_version: 4
```

3. an *OS::Nova::Server* connected to a private network that is attached to a router;

```
VM3:
  type: OS::Nova::Server
  properties:
    flavor: { get_param: flavor }
    image: { get_param: image }
    networks:
      - network: { get_resource: network3 }

network3:
  type: OS::Neutron::Net

subnet3:
  type: OS::Neutron::Subnet
  properties:
    network: { get_resource: network3 }
    cidr: 10.3.0.0/24
```

```
      ip_version: 4

  router3:
    type: OS::Neutron::Router

  router_interface3:
    type: OS::Neutron::RouterInterface
    properties:
      router: { get_resource: router3 }
      subnet: { get_resource: subnet3 }
```

4. an *OS::Nova::Server* connected to a private network that is connected to an external network attached to a router;

```
VM4:
  type: OS::Nova::Server
  properties:
    flavor: { get_param: flavor }
    image: { get_param: image }
    networks:
      - network: { get_resource: network4 }

network4:
  type: OS::Neutron::Net

subnet4:
  type: OS::Neutron::Subnet
  properties:
  network: { get_resource: network4 }
  cidr: 10.4.0.0/24
  ip_version: 4

router4:
  type: OS::Neutron::Router
  properties:
    external_gateway_info:
      network: { get_param: public_network }

router_interface4:
  type: OS::Neutron::RouterInterface
  properties:
    router: { get_resource: router4 }
    subnet: { get_resource: subnet4 }
```

In all the four cases, the deployment done by Heat succeeded. However, for the first and the second cases, the deployed virtual machine cannot be accessed (e.g. through its console on the OpenStack dashboard). When, we check the detailed logs of the virtual machine (in the dashboard), we notice that there is a configuration error as shown in Fig. 7. Indeed, the cloud-init fails to correctly configure the virtual machine which is then non accessible and useless.

This error does not occur in the third and the fourth cases (i.e. when the virtual machine is connected to a network that is attached to a router). In this case, the configuration is correctly done by the cloud-init and the virtual machine is accessible. We deploy all the four cases in different versions of OpenStack and we obtain the same results. Based on these results, we draw the following conclusions: (1) the configuration of a Virtual Machine (VM) by the cloud-init requires a router, (2) this configuration is correctly done if the VM is connected to a network that is attached to a router. Therefore, to always ensure the correct configuration of virtual machines, we define an invariant that states that: a virtual machine must be connected to network that is, directly or indirectly, attached to a router. The HOT types

```
info: initramfs: up at 0.60
GROWROOT: CHANGED: partition=1 start=16065 old: size=64260 end=80325 new: size=41913585,end=41929650
info: initramfs loading root from /dev/vda1
info: /etc/init.d/rc.sysinit: up at 8.87
info: container: none
Starting logging: OK
modprobe: module virtio_blk not found in modules.dep
modprobe: module virtio_net not found in modules.dep
WARN: /etc/rc3.d/S10-load-modules failed
Initializing random number generator... done.
Starting acpid: OK
cirros-ds 'local' up at 20.84
no results found for mode=local. up 29.48. searched: nocloud configdrive ec2
Starting network...
udhcpc (v1.20.1) started
Sending discover...
Sending select for 10.2.0.6...
Lease of 10.2.0.6 obtained, lease time 86400
cirros-ds 'net' up at 32.29
checking http://169.254.169.254/2009-04-04/instance-id
failed 1/20: up 32.44. request failed
failed 2/20: up 41.10. request failed
failed 3/20: up 44.10. request failed
failed 4/20: up 49.12. request failed
failed 5/20: up 52.12. request failed
failed 6/20: up 57.13. request failed
failed 7/20: up 60.13. request failed
failed 8/20: up 65.14. request failed
failed 9/20: up 68.14. request failed
failed 10/20: up 73.15. request failed
failed 11/20: up 76.15. request failed
failed 12/20: up 81.16. request failed
failed 13/20: up 84.16. request failed
failed 14/20: up 89.18. request failed
failed 15/20: up 92.18. request failed
failed 16/20: up 97.18. request failed
failed 17/20: up 100.18. request failed
failed 18/20: up 105.19. request failed
failed 19/20: up 108.19. request failed
failed 20/20: up 113.20. request failed
failed to read iid from metadata. tried 20
no results found for mode=net. up 116.21. searched: nocloud configdrive ec2
failed to get instance-id of datasource
Top of dropbear init script
Starting dropbear sshd: failed to get instance-id of datasource
WARN: generating key of type ecdsa failed!
OK
```

Figure 7: Configuration logs of VM1 and VM2

specification should explicitly defines this invariant in order to prevent from configuration errors of virtual machines.

Overview of the second category of errors: the second category has 84 errors that include 1 configuration error. The configuration error is more subtitle than the others. Indeed, the deployment succeed and nothing show that an error has occurred. Hence, one may think that the system is correctly deployed but this is not the case as it is useless. There is a risk that the final user of the system is the one who detect the error by noticing that the service is not delivered. The consequence is a bad user experience and must be avoided. We correct the errors of this category by modifying the YAML specification of HOT types and by writing an Alloy fact for invariant of the configuration error.

7.2.3 Errors identified by formally analyzing the HOT specification

We formally model the HOT specification with Location Graph in Alloy. Then, we analyze the obtained model with the Alloy Analyzer to find instances. This allows us to detect a third and last category of errors in the HOT specification. This category is related to `missing invariants that prevent from unrealistic behaviors`.

First example of error: let us consider, in Fig. 8, an instance of HOT template found by the Alloy Analyzer. This instance consists of 2 virtual machines and 1 port attached to both virtual machines (the port has 2 provided roles and each of them is attached to one of the virtual machines). This actually cannot happen in a realistic scenario. However, this instance is found by the analyzer. The reason is that there is no invariant to prevent from this in the specification.

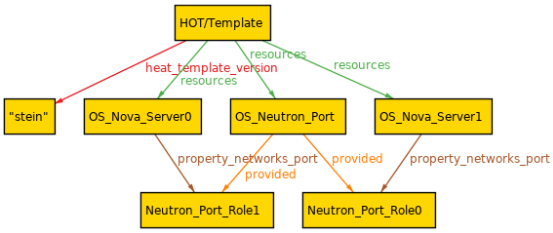


Figure 8: Two virtual machines with the same port

Second example of error: Fig. 9 shows another instance found by the Alloy Analyzer. In this instance, a network has two subnets with the same address (10.0.0.0/24). This is unrealistic behavior and, therefore, it must be avoided.

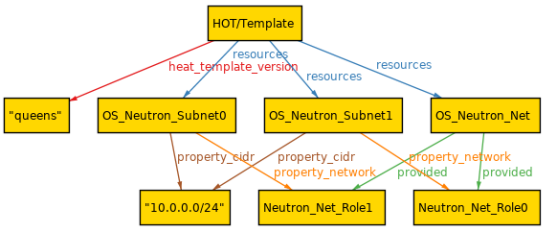


Figure 9: Two subnets with the same address in the same network

Third example of error: let us consider another instance shown in Fig. 10. This instance consists of a port attached to the subnet of a network. In this instance, the port has an IP address (172.24.4.2) that does not belong to the subnet cidr (10.0.0.0/24). This is actually an inconsistent behavior as the port address cannot be allocated from the subnet.

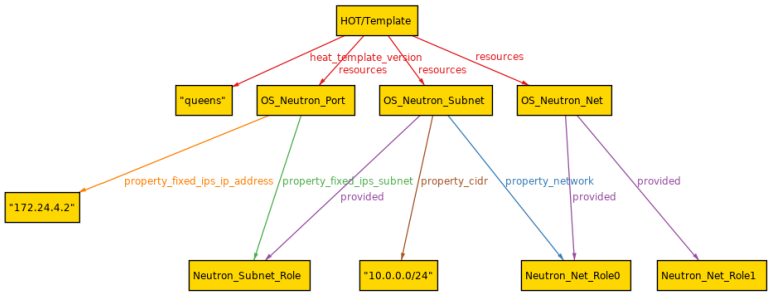


Figure 10: A port with a bad IP address on a subnet

Fourth example of error: Fig. 11 presents yet another instance that is one OS::Neutron::SecurityGroup. For this resource, the Ethernet traffic is IPv4 but the protocol is icmpv6 (i.e. icmp for IPV6). This behavior must be avoided.

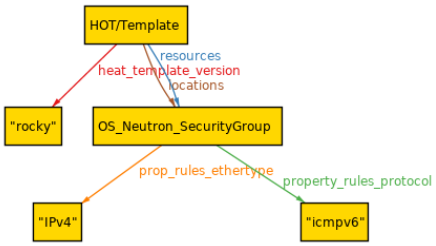


Figure 11: A security group with invalid ethertype and protocol

Overview of the third category of errors: this category is the one that has the highest number of errors. For the HOT types specification, considering 36 resource types among the most used ones, 52 missing invariants are identified. The last three examples given in section 7.1.2 are among the 52 invariants. Similarly, 8 missing invariants are detected in the HOT core specification. Examples of these invariants are the three last one in section 7.1.4. In overall, the second category of errors, in the current status of this work, identifies 60 missing invariants in the HOT specification.

These errors are corrected by (1) implementing a set of functions, and (2) writing a set of facts. The functions are written in Python for the invariants that contain IP addresses in order to prevent from the related unrealistic behaviors (cf. the examples given in Fig 9, Fig. 10). The motivation is that Python provides a library, named *ipaddress* [17], dedicated to the processing of IPv4, IPv6 addresses and cidr. This library allows for instance to verify if two cidr overlap, if an IP address is syntactically valid or if it belongs to a given network. This prevents us from re-implementing such a library. These functions are used in the implemented verification tool (cf. Section 7.3).The facts are written, in Alloy, for all the other invariants and are added in the files that result from our interpretation of the HOT language.

Discussion about the missing invariants: in the YAML version of the resource types specification, there are some invariants. However, they are written in English and hidden in the $description$ field of some resource types properties. This prevents a tool that processes the YAML version from taking these invariants into account. Moreover, these invariants are only related to the properties of the same resource type. For instance, an OS::NeutronRouterInterface has properties including port and subnet. It is specified in the description of these properties that "either port or subnet must be specified". More generally, there is a lack of other invariants that constrain the interactions between resources of different types (e.g. Fig. 8). In our formal interpretation of HOT, these missing invariants are identified and added.

7.3 The implemented verification tool

We implemented a tool called verify_HOT to enable the automatic and formal verification of HOT templates. This tool generates for a HOT template, a formal interpretation that is then verified by the Alloy analyzer. In the followings, the architecture of the implemented tool is first presented. Then, an example of HOT template is given and verified.

7.3.1 Architecture of the implemented tool

Fig.12 presents the implemented tool. It takes as input a HOT template with its environment file, if there is one. This file defines the external resources that are re-used in the template and gives values to the parameters. Then, the tool specifies if the template is deployable (i.e. it does not contain errors). If not, the errors it contains are detected. This verification tool, based on our formal interpretation of the HOT language and on the Alloy Analyzer, consists of

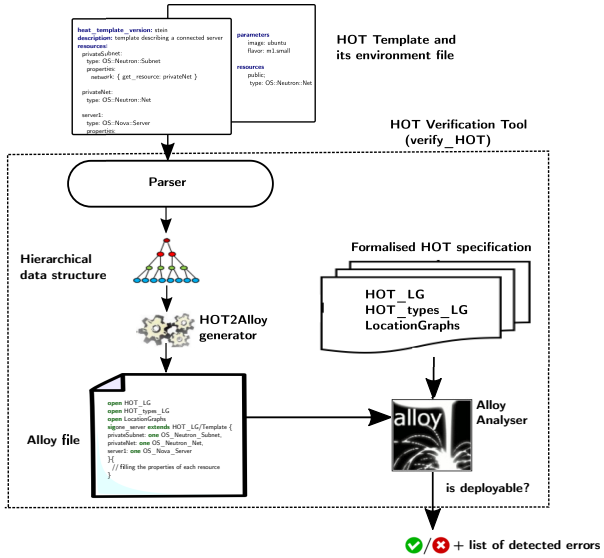


Figure 12: Architecture of the implemented verification tool

- **a parser:** is implemented in Python. It takes as input the HOT template with its environment file and parses them against the YAML syntax to generate a hierarchical data structure. This is done with the PyYAML module of Python. Then, it performs HOT related verifications on the data structure. These verifications are related to syntax, type, and invariants checking. The latter prevents from unrealistic behaviors related to IP addresses using the functions presented in Section 7.2.3. At this stage, when an error is detected, it is returned to the user and the process stops. The user can, thus, correct the error and use the verification tool on the modified template;

- **a HOT2Alloy generator:** it takes as input the hierarchical data structure and generates an Alloy file that is the formal interpretation of the template. This file consists of a signature, a fact and a run command. The signature declares the parameters, parameter_groups, resources, outputs and conditions of the template and the fact affects values to their fields. The run command is executed by the Alloy analyzer, based on the formal interpretation of the HOT specification. If the template is deployable, an instance of it is found. Otherwise, the errors it contain are given to the user for their correction. This generator was implemented in Python and has 1810 lines of code.

7.3.2 Illustration of the implemented tool

Let us consider a small system that consists of two connected virtual machines. These machines are connected to a private network which is connected to a public network through a router. The virtual machines have floating IP addresses on the public network in order to be accessible for specific protocols (i.e. ssh, and icmp for pinging).

The HOT template of this system defines 10 resource types. These types are: 2 OS::Nova::Server, 1 OS::Neutron::Net, 1 OS::Neutron::Subnet, 1 OS::Neutron::Router, 1 OS::Neutron::RouterInterface, 2 OS::Neutron::Floating IP, 2 OS::Neutron::Port.

heat_template_version: stein

description: HOT template to deploy two connected Virtual Machines (VM).

```
parameters:
  image:
    type: string
    description: Name of image used to boot the VMs
    default: ubuntu

  flavor:
    type: string
    description: size of the VMs
    default: m1.small

  public_net:
    type: string
    description: ID of public network to allocate floating IP addresses
    default: public

resources:
  private_net:
    type: OS::Neutron::Net


  private_subnet:
    type: OS::Neutron::Subnet
    properties:
      network: { get_resource: private_net }
      cidr: 10.0.0.0/24
      dns_nameservers: [10.194.51.39, 10.194.51.29]

  router:
    type: OS::Neutron::Router
    properties:
      external_gateway_info:
        network: {get_param: public_net}

  router_interface:
    type: OS::Neutron::RouterInterface
    properties:
      router: { get_resource: router }
      subnet: { get_resource: private_subnet }

  VM1:
    type: OS::Nova::Server
    properties:
      image: { get_param: image }
      flavor: { get_param: flavor }
      networks:
        - port: { get_resource: VM1_port }

  VM1_port:
    type: OS::Neutron::Port
    properties:
      network: { get_resource: private_net }

  VM1_floating_ip:
    type: OS::Neutron::FloatingIP
    depends_on: router_interface
    properties:
      floating_network: { get_param: public_net }
      port_id: { get_resource: VM1_port }

  VM2:
    type: OS::Nova::Server
```

```
    properties:
      image: { get_param: image }
      flavor: { get_param: flavor }
      networks:
        - port: { get_resource: VM2_port }

  VM2_port:
    type: OS::Neutron::Port
    properties:
      network: { get_resource: private_net }

  VM2_floating_ip:
    type: OS::Neutron::FloatingIP
    depends_on: router_interface
    properties:
      floating_network: { get_param: public_net }
      port_id: { get_resource: VM2_port }

outputs:
  VM1_private_ip:
    description: IP address of VM1 on the private network
    value: { get_attr: [ VM1, first_address ] }

  VM1_public_ip:
    description: Floating IP address of VM1 on the public network
    value: { get_attr: [ VM1_floating_ip, floating_ip_address ] }

  VM2_private_ip:
    description: IP address of VM2 on the private network
    value: { get_attr: [ VM2, first_address ] }

  VM2_public_ip:
      description: Floating IP address of VM2 on the public network
      value: { get_attr: [ VM2_floating_ip, floating_ip_address ] }
```

This template is given to the verify_HOT tool. The generated interpretation consists of a signature that declares the resource types, a fact that fills their fields, and a run command. This command is executed by the Alloy analyzer and the result is shown in Fig. 13. An instance of is system is found meaning that the template does not contain errors.

```
//-----------------------------------------------------------------
// Imports
//-----------------------------------------------------------------

open HOT
open HOT_types
open LocationGraphs as LG

//-----------------------------------------------------------------
// Template
//-----------------------------------------------------------------

sig TwoVMs_template extends HOT/Template
{
/** Parameters */

// YAML image: {'type': 'string', 'description': 'Name of image to use for VMs', 'default': 'ubuntu'}
image: one Parameter,
// YAML flavor: {'type': 'string', 'description': 'flavor to use as size of the VMs', 'default': 'm1.small'}
flavor: one Parameter,
// YAML public_net: {'type': 'string', 'description': 'ID of public network for floating IP addresses', 'default': 'public'}
public_net: one Parameter,
```

```
/** Resources */

// YAML private_net: {'type': 'OS::Neutron::Net'}
private_net: one OS_Neutron_Net,
// YAML private_subnet: {'type': 'OS::Neutron::Subnet',
//'properties': {'network': {'get_resource': 'private_net'}, 'cidr': '10.0.0.0/24', 'dns_nameservers': ['10.194.51.39', '10.194.51.29']}}
private_subnet: one OS_Neutron_Subnet,
// YAML router: {'type': 'OS::Neutron::Router',
//'properties': {'external_gateway_info': {'network': {'get_param': 'public_net'}, 'enable_snat': None, 'external_fixed_ips': None}}}
router: one OS_Neutron_Router,
// YAML router_interface: {'type': 'OS::Neutron::RouterInterface',
//'properties': {'router': {'get_resource': 'router'}, 'subnet': {'get_resource': 'private_subnet'}}}
router_interface: one OS_Neutron_RouterInterface,
// YAML VM1: {'type': 'OS::Nova::Server', 'properties': {'image': {'get_param': 'image'},
//'flavor': {'get_param': 'flavor'}, 'networks': [{'port': {'get_resource': 'VM1_port'}, 'allocate_network': None,
//'fixed_ip': None, 'floating_ip': None, 'network': None, 'port_extra_properties': None, 'subnet': None, 'tag': None, 'uuid': None}]}}
VM1: one OS_Nova_Server,
// YAML VM1_port: {'type': 'OS::Neutron::Port', 'properties': {'network': {'get_resource': 'private_net'}}}
VM1_port: one OS_Neutron_Port,
// YAML VM1_floating_ip: {'type': 'OS::Neutron::FloatingIP', 'depends_on': 'router_interface',
//'properties': {'floating_network': {'get_param': 'public_net'}, 'port_id': {'get_resource': 'VM1_port'}}}
VM1_floating_ip: one OS_Neutron_FloatingIP,
// YAML VM2: {'type': 'OS::Nova::Server', 'properties': {'image': {'get_param': 'image'},
//'flavor': {'get_param': 'flavor'}, 'networks': [{'port': {'get_resource': 'VM2_port'}, 'allocate_network': None,
//'fixed_ip': None, 'floating_ip': None, 'network': None, 'port_extra_properties': None, 'subnet': None, 'tag': None, 'uuid': None}]}}
VM2: one OS_Nova_Server,
// YAML VM2_port: {'type': 'OS::Neutron::Port', 'properties': {'network': {'get_resource': 'private_net'}}}
VM2_port: one OS_Neutron_Port,
// YAML VM2_floating_ip: {'type': 'OS::Neutron::FloatingIP', 'depends_on': 'router_interface',
//'properties': {'floating_network': {'get_param': 'public_net'}, 'port_id': {'get_resource': 'VM2_port'}}}
VM2_floating_ip: one OS_Neutron_FloatingIP,
// YAML public: {'type': 'OS::Neutron::ProviderNet', 'properties': {'network_type': 'flat'}}
public: one OS_Neutron_ProviderNet,
// YAML ubuntu: {'type': 'OS::Glance::Image', 'properties': {'disk_format': 'ami',
//'container_format': 'bare', 'location': 'https://ubuntu-images.fr'}}
ubuntu: one OS_Glance_Image,
// YAML m1.small: {'type': 'OS::Nova::Flavor', 'properties': {'vcpus': 1, 'ram': 1}}
m1_small: one OS_Nova_Flavor,

/** Outputs */

// YAML VM1_private_ip: {'description': 'IP address of VM1 on the private network',
//'value': {'get_attr': ['VM1', 'first_address']}}
VM1_private_ip: one Output,
// YAML VM1_public_ip: {'description': 'Floating IP address of VM1 on the public network',
//'value': {'get_attr': ['VM1_floating_ip', 'floating_ip_address']}}
VM1_public_ip: one Output,
// YAML VM2_private_ip: {'description': 'IP address of VM2 on the private network',
//'value': {'get_attr': ['VM2', 'first_address']}}
VM2_private_ip: one Output,
// YAML VM2_public_ip: {'description': 'Floating IP address of VM2 on the public network',
//'value': {'get_attr': ['VM2_floating_ip', 'floating_ip_address']}}
VM2_public_ip: one Output
} {

/** Heat_template_version */

heat_template_version = ''stein''

/** Description */
```

```
description = ''HOT template to deploy two connected Virtual Machines (VM).''

/** Parameter Groups */

// no parameter groups
no parameter_groups

/** Parameters */

// YAML image: {'type': 'string', 'description': 'Name of image to use for VMs', 'default': 'ubuntu'}
parameter[image]
image.name = ''image''
image.type = ''string''
image.description = ''Name of image to use to boot the VMs''
image.default = ''ubuntu''
image.value = ''ubuntu''
no image.label
image.hidden = false
no image.constraints
image.immutable = false
no image.tags

// YAML flavor: {'type': 'string', 'description': 'flavor to use as size of the VMs', 'default': 'm1.small'}
parameter[flavor]
flavor.name = ''flavor''
flavor.type = ''string''
flavor.description = ''size of the VMs''
flavor.default = ''m1.small''
flavor.value = ''m1_small''
no flavor.label
flavor.hidden = false
no flavor.constraints
flavor.immutable = false
no flavor.tags

// YAML public_net: {'type': 'string', 'description': 'ID of public network for floating IP addresses', 'default': 'public'}
parameter[public_net]
public_net.name = ''public_net''
public_net.type = ''string''
public_net.description = ''ID of public network to allocate floating IP addresses''
public_net.default = ''public''
public_net.value = ''public''
no public_net.label
public_net.hidden = false
no public_net.constraints
public_net.immutable = false
no public_net.tags

/** Resources */

// YAML private_net: {'type': 'OS::Neutron::Net'}
resource[private_net]
private_net.id = ''private_net''
private_net.deletion_policy = ''delete''
no private_net.condition
no private_net.external_id
private_net.prop_admin_state_up = true
no private_net.prop_dhcp_agent_ids
no private_net.prop_dns_domain
no private_net.prop_name
no private_net.prop_port_security_enabled
no private_net.prop_qos_policy
```

private_net.prop_shared = false
no private_net.prop_tags
no private_net.prop_tenant_id
no private_net.prop_value_specs
no private_net.depends_on
no private_net.metadata

// YAML private_subnet: {'type': 'OS::Neutron::Subnet',
//'properties': {'network': {'get_resource': 'private_net'}, 'cidr': '10.0.0.0/24', 'dns_nameservers': ['10.194.51.39', '10.194.51.29']}}
resource[private_subnet]
private_subnet.id = ''private_subnet''
private_subnet.prop_network[get_resource[private_net]]
private_subnet.prop_cidr = ''10.0.0.0/24''
private_subnet.prop_dns_nameservers[''10.194.51.39'']
private_subnet.prop_dns_nameservers[''10.194.51.29'']
no private_subnet.prop_allocation_pools
private_subnet.prop_enable_dhcp = true
no private_subnet.prop_gateway_ip
no private_subnet.prop_host_routes
private_subnet.prop_ip_version = 4
no private_subnet.prop_ipv6_address_mode
no private_subnet.prop_ipv6_ra_mode
no private_subnet.prop_name
no private_subnet.prop_prefixlen
no private_subnet.prop_segment
no private_subnet.prop_subnetpool
no private_subnet.prop_tags
no private_subnet.prop_tenant_id
no private_subnet.prop_value_specs
private_subnet.deletion_policy = ''delete''
no private_subnet.condition
no private_subnet.external_id
no private_subnet.depends_on
no private_subnet.metadata

// YAML router: {'type': 'OS::Neutron::Router',
//'properties': {'external_gateway_info': {'network': {'get_param': 'public_net'}, 'enable_snat': None, 'external_fixed_ips': None}}}
resource[router]
router.id = ''router''
router.prop_external_gateway_info[**none**, **none**, get_param_role[public_net]]
router.prop_admin_state_up = true
no router.prop_distributed
no router.prop_ha
no router.prop_l3_agent_ids
no router.prop_name
no router.prop_tags
no router.prop_value_specs
router.deletion_policy = ''delete''
no router.condition
no router.external_id
no router.depends_on
no router.metadata

// YAML router_interface: {'type': 'OS::Neutron::RouterInterface',
//'properties': {'router': {'get_resource': 'router'}, 'subnet': {'get_resource': 'private_subnet'}}}
resource[router_interface]
router_interface.id = ''router_interface''
router_interface.prop_router[get_resource[router]]
router_interface.prop_subnet[get_resource[private_subnet]]
no router_interface.prop_port
router_interface.deletion_policy = ''delete''
no router_interface.condition

```
no router_interface.external_id
no router_interface.depends_on
no router_interface.metadata

// YAML VM1: {'type': 'OS::Nova::Server', 'properties': {'image': {'get_param': 'image'},
//'flavor': {'get_param': 'flavor'}, 'networks': [{'port': {'get_resource': 'VM1_port'},
//'allocate_network': None, 'fixed_ip': None, 'floating_ip': None, 'network': None,
//'port_extra_properties': None, 'subnet': None, 'tag': None, 'uuid': None}]}}
resource[VM1]
VM1.id = ''VM1''
VM1.prop_image[get_param_role[image]]
VM1.prop_flavor[get_param_role[flavor]]
VM1.prop_networks[none, none, none, none, get_resource[VM1_port], none, none, none, none]
no VM1.prop_admin_pass
no VM1.prop_availability_zone
no VM1.prop_block_device_mapping
no VM1.prop_block_device_mapping_v2
no VM1.prop_config_drive
no VM1.prop_deployment_swift_data
no VM1.prop_diskConfig
VM1.prop_flavor_update_policy = ''RESIZE''
VM1.prop_image_update_policy = ''REBUILD''
no VM1.prop_key_name
no VM1.prop_metadata
no VM1.prop_name
no VM1.prop_personality
no VM1.prop_reservation_id
no VM1.prop_scheduler_hints
no VM1.prop_security_groups
VM1.prop_software_config_transport = ''POLL_SERVER_CFN''
no VM1.prop_tags
no VM1.prop_user_data
VM1.prop_user_data_format = ''HEAT_CFNTOOLS''
VM1.prop_user_data_update_policy = ''REPLACE''
VM1.deletion_policy = ''delete''
no VM1.condition
no VM1.external_id
no VM1.depends_on
no VM1.metadata

// YAML VM1_port: {'type': 'OS::Neutron::Port', 'properties': {'network': {'get_resource': 'private_net'}}}
resource[VM1_port]
VM1_port.id = ''VM1_port''
VM1_port.prop_network[get_resource[private_net]]
VM1_port.prop_admin_state_up = true
no VM1_port.prop_allowed_address_pairs
VM1_port.prop_binding_vnic_type = ''normal''
no VM1_port.prop_device_id
no VM1_port.prop_device_owner
no VM1_port.prop_dns_name
no VM1_port.prop_fixed_ips
no VM1_port.prop_mac_address
no VM1_port.prop_name
no VM1_port.prop_port_security_enabled
no VM1_port.prop_qos_policy
no VM1_port.prop_security_groups
no VM1_port.prop_tags
no VM1_port.prop_value_specs
VM1_port.deletion_policy = ''delete''
no VM1_port.condition
no VM1_port.external_id
no VM1_port.depends_on
```

no VM1_port.metadata

// YAML VM1_floating_ip: {'type': 'OS::Neutron::FloatingIP', 'depends_on': 'router_interface',
//'properties': {'floating_network': {'get_param': 'public_net'}, 'port_id': {'get_resource': 'VM1_port'}}}
resource[VM1_floating_ip]
VM1_floating_ip.id = ''VM1_floating_ip''
VM1_floating_ip.depends_on[router_interface]
VM1_floating_ip.prop_floating_network[get_param_role[public_net]]
VM1_floating_ip.prop_port_id[get_resource[VM1_port]]
no VM1_floating_ip.prop_dns_domain
no VM1_floating_ip.prop_dns_name
no VM1_floating_ip.prop_fixed_ip_address
no VM1_floating_ip.prop_floating_ip_address
no VM1_floating_ip.prop_floating_subnet
no VM1_floating_ip.prop_value_specs
VM1_floating_ip.deletion_policy = ''delete''
no VM1_floating_ip.condition
no VM1_floating_ip.external_id
no VM1_floating_ip.metadata

// YAML VM2: {'type': 'OS::Nova::Server', 'properties': {'image': {'get_param': 'image'},
//'flavor': {'get_param': 'flavor'}, 'networks': [{'port': {'get_resource': 'VM2_port'},
//'allocate_network': None, 'fixed_ip': None, 'floating_ip': None, 'network': None,
//'port_extra_properties': None, 'subnet': None, 'tag': None, 'uuid': None}]}}
resource[VM2]
VM2.id = ''VM2''
VM2.prop_image[get_param_role[image]]
VM2.prop_flavor[get_param_role[flavor]]
VM2.prop_networks[**none**, **none**, **none**, **none**, get_resource[VM2_port], **none**, **none**, **none**, **none**]
no VM2.prop_admin_pass
no VM2.prop_availability_zone
no VM2.prop_block_device_mapping
no VM2.prop_block_device_mapping_v2
no VM2.prop_config_drive
no VM2.prop_deployment_swift_data
no VM2.prop_diskConfig
VM2.prop_flavor_update_policy = ''RESIZE''
VM2.prop_image_update_policy = ''REBUILD''
no VM2.prop_key_name
no VM2.prop_metadata
no VM2.prop_name
no VM2.prop_personality
no VM2.prop_reservation_id
no VM2.prop_scheduler_hints
no VM2.prop_security_groups
VM2.prop_software_config_transport = ''POLL_SERVER_CFN''
no VM2.prop_tags
no VM2.prop_user_data
VM2.prop_user_data_format = ''HEAT_CFNTOOLS''
VM2.prop_user_data_update_policy = ''REPLACE''
VM2.deletion_policy = ''delete''
no VM2.condition
no VM2.external_id
no VM2.depends_on
no VM2.metadata

// YAML VM2_port: {'type': 'OS::Neutron::Port', 'properties': {'network': {'get_resource': 'private_net'}}}
resource[VM2_port]
VM2_port.id = ''VM2_port''
VM2_port.prop_network[get_resource[private_net]]
VM2_port.prop_admin_state_up = true
no VM2_port.prop_allowed_address_pairs

VM2_port.prop_binding_vnic_type = ''normal''
no VM2_port.prop_device_id
no VM2_port.prop_device_owner
no VM2_port.prop_dns_name
no VM2_port.prop_fixed_ips
no VM2_port.prop_mac_address
no VM2_port.prop_name
no VM2_port.prop_port_security_enabled
no VM2_port.prop_qos_policy
no VM2_port.prop_security_groups
no VM2_port.prop_tags
no VM2_port.prop_value_specs
VM2_port.deletion_policy = ''delete''
no VM2_port.condition
no VM2_port.external_id
no VM2_port.depends_on
no VM2_port.metadata

// YAML VM2_floating_ip: {'type': 'OS::Neutron::FloatingIP', 'depends_on': 'router_interface',
//'properties': {'floating_network': {'get_param': 'public_net'}, 'port_id': {'get_resource': 'VM2_port'}}}
resource[VM2_floating_ip]
VM2_floating_ip.id = ''VM2_floating_ip''
VM2_floating_ip.depends_on[router_interface]
VM2_floating_ip.prop_floating_network[get_param_role[public_net]]
VM2_floating_ip.prop_port_id[get_resource[VM2_port]]
no VM2_floating_ip.prop_dns_domain
no VM2_floating_ip.prop_dns_name
no VM2_floating_ip.prop_fixed_ip_address
no VM2_floating_ip.prop_floating_ip_address
no VM2_floating_ip.prop_floating_subnet
no VM2_floating_ip.prop_value_specs
VM2_floating_ip.deletion_policy = ''delete''
no VM2_floating_ip.condition
no VM2_floating_ip.external_id
no VM2_floating_ip.metadata

// YAML public: {'type': 'OS::Neutron::ProviderNet', 'properties': {'network_type': 'flat'}}
resource[public]
public.id = ''public''
public.prop_network_type = ''flat''
public.prop_admin_state_up = true
no public.prop_name
no public.prop_physical_network
no public.prop_port_security_enabled
public.prop_router_external = false
no public.prop_segmentation_id
public.prop_shared = true
public.deletion_policy = ''delete''
no public.condition
no public.external_id
no public.depends_on
no public.metadata

// YAML ubuntu: {'type': 'OS::Glance::Image', 'properties': {'disk_format': 'ami',
//'container_format': 'bare', 'location': 'https://ubuntu−images.fr'}}
resource[ubuntu]
ubuntu.id = ''ubuntu''
ubuntu.prop_disk_format = ''ami''
ubuntu.prop_container_format = ''bare''
ubuntu.prop_location = ''https:*//ubuntu−images.fr''*
no ubuntu.prop_architecture
no ubuntu.prop_extra_properties

```
no ubuntu.prop_id
ubuntu.prop_is_public = false
no ubuntu.prop_kernel_id
ubuntu.prop_min_disk = 0
ubuntu.prop_min_ram = 0
no ubuntu.prop_name
no ubuntu.prop_os_distro
no ubuntu.prop_owner
ubuntu.prop_protected = false
no ubuntu.prop_ramdisk_id
no ubuntu.prop_tags
ubuntu.deletion_policy = ''delete''
no ubuntu.condition
no ubuntu.external_id
no ubuntu.depends_on
no ubuntu.metadata

// YAML m1.small: {'type': 'OS::Nova::Flavor', 'properties': {'vcpus': 1, 'ram': 1}}
resource[m1_small]
m1_small.id = ''m1_small''
m1_small.prop_vcpus = 1
m1_small.prop_ram = 1
m1_small.prop_disk = 0
m1_small.prop_ephemeral = 0
no m1_small.prop_extra_specs
no m1_small.prop_flavorid
m1_small.prop_is_public = true
no m1_small.prop_name
m1_small.prop_rxtx_factor = 1
m1_small.prop_swap = 0
no m1_small.prop_tenants
m1_small.deletion_policy = ''delete''
no m1_small.condition
no m1_small.external_id
no m1_small.depends_on
no m1_small.metadata

/** Outputs */

// YAML VM1_private_ip: {'description': 'IP address of VM1 on the private network',
//'value': {'get_attr': ['VM1', 'first_address']}}
output[VM1_private_ip]
VM1_private_ip.name = ''VM1_private_ip''
VM1_private_ip.description = ''IP address of VM1 on the private network''
VM1_private_ip.value = ''get_attr: ['VM1', 'first_address']''
no VM1_private_ip.condition

// YAML VM1_public_ip: {'description': 'Floating IP address of VM1 on the public network',
//'value': {'get_attr': ['VM1_floating_ip', 'floating_ip_address']}}
output[VM1_public_ip]
VM1_public_ip.name = ''VM1_public_ip''
VM1_public_ip.description = ''Floating IP address of VM1 on the public network''
VM1_public_ip.value = ''get_attr: ['VM1_floating_ip', 'floating_ip_address']''
no VM1_public_ip.condition

// YAML VM2_private_ip: {'description': 'IP address of VM2 on the private network',
//'value': {'get_attr': ['VM2', 'first_address']}}
output[VM2_private_ip]
VM2_private_ip.name = ''VM2_private_ip''
VM2_private_ip.description = ''IP address of VM2 on the private network''
VM2_private_ip.value = ''get_attr: ['VM2', 'first_address']''
no VM2_private_ip.condition
```

```
// YAML VM2_public_ip: {'description': 'Floating IP address of VM2 on the public network',
//'value': {'get_attr': ['VM2_floating_ip', 'floating_ip_address']}}
output[VM2_public_ip]
VM2_public_ip.name = ''VM2_public_ip''
VM2_public_ip.description = ''Floating IP address of VM2 on the public network''
VM2_public_ip.value = ''get_attr: ['VM2_floating_ip', 'floating_ip_address']''
no VM2_public_ip.condition

/** Conditions */

// no conditions
no conditions
}

/** There exists some Ugw_template */

run Show_TwoVMs{
} for 0 but
  4 Int,
  0 seq,
  exactly 1 TwoVMs_template,
  exactly 1 HOT/Template,
  exactly 1 LG/LocationGraph,
  exactly 0 HOT/ParameterGroup,
  exactly 3 HOT/Parameter,
  exactly 0 HOT/ParameterConstraint,
  exactly 0 HOT/Length_interval,
  exactly 0 HOT/Range_interval,
  exactly 0 HOT/Interval,
  exactly 0 HOT/StepOffset,
  exactly 13 HOT/Resource,
  exactly 13 LG/Location,
  exactly 1 OS_Glance_Image,
  exactly 2 OS_Glance_Image_Role,
  exactly 2 OS_Neutron_FloatingIP,
  exactly 2 OS_Neutron_FloatingIP_Role,
  exactly 1 OS_Neutron_Net,
  exactly 3 OS_Neutron_Net_Role,
  exactly 2 OS_Neutron_Port,
  exactly 4 OS_Neutron_Port_Role,
  exactly 1 OS_Neutron_ProviderNet,
  exactly 3 OS_Neutron_ProviderNet_Role,
  exactly 1 OS_Neutron_Router,
  exactly 1 OS_Neutron_Router_Role,
  exactly 1 OS_Neutron_RouterInterface,
  exactly 2 OS_Neutron_RouterInterface_Role,
  exactly 1 OS_Neutron_Subnet,
  exactly 1 OS_Neutron_Subnet_Role,
  exactly 1 OS_Nova_Flavor,
  exactly 2 OS_Nova_Flavor_Role,
  exactly 2 OS_Nova_Server,
  exactly 2 OS_Nova_Server_Role,
  exactly 1 OS_Neutron_Router_external_gateway_info,
  exactly 2 OS_Nova_Server_networks,
  exactly 2 OS_Nova_Server_networks_elements,
  exactly 3 HOT/FlatProperty,
  exactly 22 HOT/ResourceRole,
  exactly 22 LG/Role,
  exactly 1 LG/Sort, // all locations share the same sort
  exactly 13 LG/Name,
  exactly 1 LG/Process, // all locations share the same process
```

```
exactly 37 HOT/HOTValue, // 8 Property, 22 Role, 3 Parameter, 4 Output, 0 Condition, 0 Attribute, 0 Support_status
exactly 52 LG/Value, // 8 Property, 22 Role, 1 Sort, 1 Process, 13 Name, 3 Parameter, 4 Output, 0 Condition, 0 Attribute, 0 Support_status
exactly 0 HOT/Attribute,
exactly 0 HOT/Support_status,
exactly 4 HOT/Output,
exactly 0 HOT/Condition,
exactly 0 HOT/map_string/Map
expect 1
```

```
Executing "Run Show_TwoVMs for 0 but 4 int, 0 seq, exactly 1 TwoVMs_template, exactly 1 Template, exactly 1 LocationGraph, ...
   Solver=minisatprover(jni) Bitwidth=4 MaxSeq=0 SkolemDepth=1 Symmetry=20
   1609793 vars. 269108 primary vars. 11792008 clauses. 22083ms.
   Instance found. Predicate is consistent, as expected. 3860ms.
```

Figure 13: Verification of the two virtual machines template

7.3.3 Comparison with other verification tools

Heat tools: Heat provides a set of tools for the verification of templates before deployment. These tools consists of the Heat parser, the *heat template-validate* command, and the *dry-run* option. These tools perform syntax and type checking. They also verify if an external resource used in a template exists in the target OpenStack platform. They also detect the errors that are presented in the first category (presented in Section 7.2.1). However, using the Heat tools require from users to connect to a running OpenStack platform even if their aim is to just verify the template without performing the deployment of the related system. Moreover, these tools do not detect the errors of the second and most errors of the third categories. In the third category, the only error that are detected by these tools is the loop dependency between resources as it is not possible to define the order in which the resources have to be created. Therefore, using these tools does not prevent from failed or partial deployment as they do not detect a set of errors (those in the second category and almost all of the third one). In the opposite, these errors are detected by our tool.

ONAP tools: ONAP provides a tool, called vvp [18], for the verification of HOT templates before deployment. This tool verifies if a template conforms the ONAP naming rules. It performs type checking to verify if a resource type (e.g. OS::Neutron::FloatingIP) not allowed by ONAP is used in the template. The motivation is that ONAP defines a set of naming rules, best practices and constraints that must be respected when designing a template. However, vvp does not perform formal verification in order to detect complex errors such as those we present in the third category.

8 The Aeolus Model

The Aeolus component model [31] was designed for configuring and deploying distributed applications in cloud environments. The Aeolus model supports the description of different application component characteristics, including dependencies between components, conflicts between components, and non-functional requirements in the form of replication and load constraints. The key insight of the Aeolus model is that each component can go through a specific lifecycle for their configuration and deployment, and that the interfaces of a component may vary during their configuration and deployment lifecycle. Aeolus models configuration and deployment lifecycles as finite state machines, with dependencies indicating which component interfaces are activated in a given state. The Aeolus models is supported by a set of tools called Aeolus *Blender* [30], containing a dependency manager called *Zephyrus*, a deployment planner called *Metis* and a deployment orchestrator called *Armonic*.

In this section, we present a location graph interpretation of the Aeolus model together with its operational semantics. We then prove that our interpretation is faithful to the Aeolus semantics. Finally, we show in Section 8.4 how to extend the TOSCA core model with Aeolus concepts. We begin with a brief presentation of the Aeolus model and of its operational semantics.

8.1 An overview on the Aeolus component model

We present in this section the main concepts of the Aeolus component model. We consider here the full Aeolus model, including component creation and deletion, and conflicts between components ([31] describes different sub-models of the full Aeolus model for which the planning problem becomes tractable).

Aeolus components are essentially finite state automata equipped with ports. Ports represent interfaces, through which a component can provide services to, or require services from, its environment. A given state in a component automaton may activate different ports, meaning that the interfaces the component provides to, or requires from, its environment may vary with its state. Each port comes equipped with cardinality constraints which indicate the maximum number of other components a given provide port may service (be *bound* to), and the minimum number of components servicing a given require port (a form of replication constraint for servicing components).

The following disjoint sets are given: $\mathcal{I}$ stands for the set of interfaces, Γ for the set of component types, $\mathcal{Q}$ for the set of states, and $\mathcal{Z}$ for the set of components. $\mathbb{N}$ denotes the set of strictly positive numbers, $\mathbb{N}_\infty$ denotes the set $\mathbb{N} \cup \{\infty\}$, and $\mathbb{N}_0$ stands for $\mathbb{N} \cup \{0\}$.

The Aeolus component model distinguishes between *component* and *component types*. A component type corresponds to a template, i.e. a description of the deployment lifecycle associated with components of the type. A component corresponds to a deployed software element.

Definition 1 Component Type. *A component type is defined as a* $5-$*tuple* $\langle Q, q_0, T, P, D\rangle$ *where:*

- $Q \subset \mathcal{Q}$ *is a finite set of states;*
- $q_0 \in Q$ *is the initial state;*
- $T \subset Q \times Q$ *is a finite set of transitions;*
- $P = \langle \mathbf{P}, \mathbf{R}\rangle$*, where* $\mathbf{P}, \mathbf{R} \subset \mathcal{I}$*, is a pair of provided and required port sets, respectively.*
- D *(the* activation *function) is a function which, for a given state* $q \in Q$ *returns two partial functions* $(\mathbf{P} \mapsto \mathbb{N}_\infty)$ *and* $(\mathbf{P} \mapsto \mathbb{N}_0)$ *which associate for each provide port the maximum number of bindings it can support, and respectively, for each require port the minimum number of bindings (to provide ports having the same interface and belonging to different components) that must support it.*

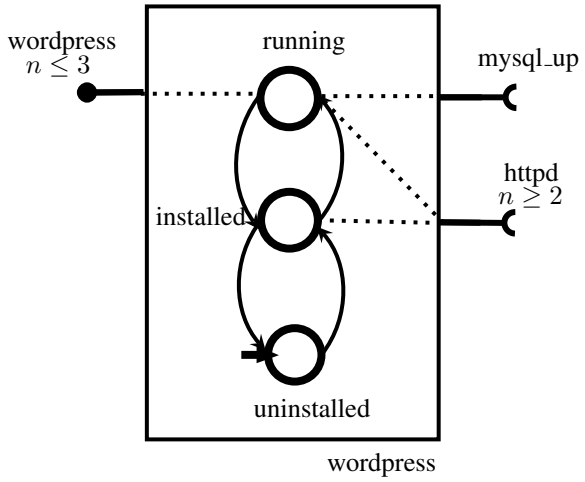


Figure 14: A component type

Notation. Given a component type $\mathcal{T} = \langle Q, q_0, T, \langle \mathbf{P}, \mathbf{R} \rangle, D \rangle$, we define $\mathcal{T}.\texttt{states} = Q$, $\mathcal{T}.\texttt{trans} = T$, $\mathcal{T}.\texttt{init} = q_0$, $\mathcal{T}.\texttt{prov} = \mathbf{P}$, and $\mathcal{T}.\texttt{req} = \mathbf{R}$.

Figure 14 depicts a component type for a Wordpress component. Provide ports are depicted by small black circles, whereas require ports are depicted by small half circles. Cardinality constraints adorn the ports in the form of inequations indicating the maximum or minimum number n of bindings that can attach to the port. Ports that are activated in a given state are indicated by dotted lines connecting a state and the ports it activates. Interface names (such as wordpress, httpd) also adorn the ports.

Note that, in an Aeolus component or component type, there is at most one port associated with a given interface name (or interface, for short).

In the Aeolus model, a configuration is defined as a set of component types (components types that can be deployed) together with a set of components and *bindings*. A binding is a connection between a required port, with a given interface n, in some component, and a provided port with the same interface n, in some other component.

Definition 2 Configuration. *A configuration $\mathcal{C}$ is a $4-$tuple $\langle U, Z, S, B \rangle$ where:*

- *$U \subset \mathcal{T}$ is the set of available component types;*
- *$Z \subset \mathcal{Z}$ is the set of currently deployed components;*
- *S is the component state description, i.e, a function that associates to each component z in Z a pair $\langle \mathcal{T}, q \rangle$, where $\mathcal{T} = \langle Q, q_0, T, P, D \rangle$ is a component type and $q \in Q$ is the current state of z;*
- *$B \in \mathcal{I} \times Z \times Z$ is the set of bindings, i.e. 3-tuples composed by an interface, a component with required port bearing that interface name and a component with a provied port bearing that interface name.*

The set of configurations is noted $\mathbb{C}$.

Figure 15 depicts an Aeolus configuration, where the components are z_1 of type wordpress, z_2 of type apache and z_3 of type mysql_up. The current state of each component is depicted by a filled circle. At the current state (running), the z_3 component provides the service mysql_up to z_1 component.

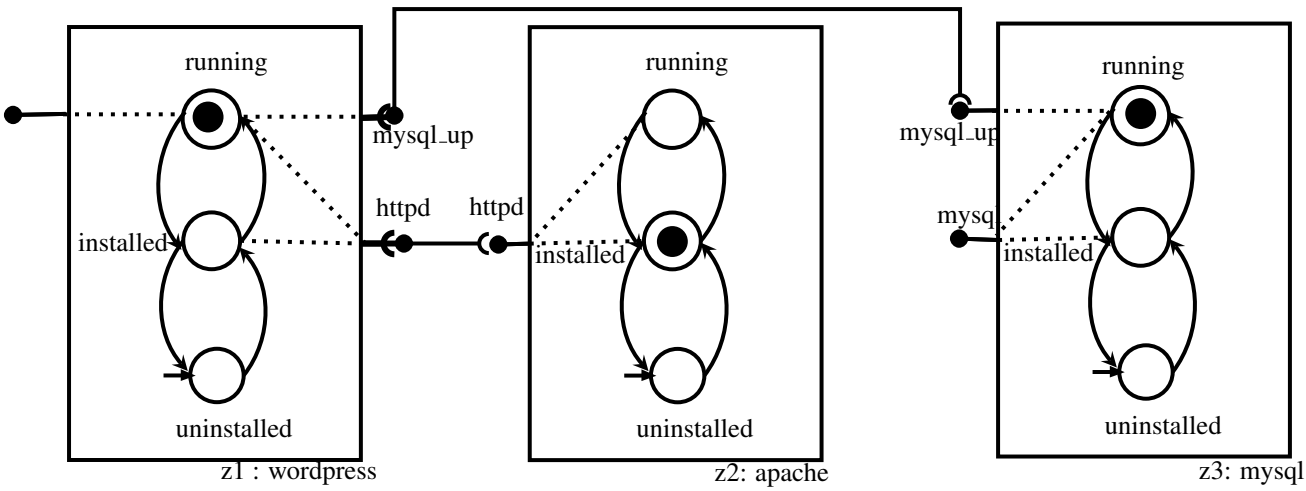


Figure 15: An Aeolus Configuration

Notation. When $\mathcal{C} = \langle U, Z, S, B\rangle$, $\mathcal{C}[z]$ stands for the pair $\langle \mathcal{T}, q\rangle = S(z)$, $\mathcal{T}.\mathbf{P}(q)$, resp. $\mathcal{T}.\mathbf{R}(q)$, stand for the set of pairs of activated provided ports, resp. required ports, with their associated cardinality constraint. We also write $\mathcal{C}[z].\texttt{prov}$ and $\mathcal{C}[z].\texttt{req}$ for $\mathcal{T}.\texttt{prov}$ and $\mathcal{T}.\texttt{req}$, respectively.

Definition 3 Configuration Correctness. *Let $\mathcal{C} = \langle U, Z, S, B\rangle$ be a configuration.*

The predicate $\mathcal{C} \models_{req} (z, t, n)$ indicates that the required port of component z with interface t and with associated number n is satisfied. Formally,

- *if $n = 0$, all components except from z cannot have a provided port with interface i;*
- *if $n > 0$, then the port there exists at least n bindings to active ports, i.e, there exists n distinct components $z_1, z_2, \ldots z_n \in Z \setminus \{z\}$ such that for all $1 \leq k \leq n$, we have that $\langle r, z, z_k\rangle \in B \wedge C[z_i] = \langle \mathcal{T}^k, q^k\rangle$ and the interface t is in the domain of $\mathcal{T}^k.\mathbf{P}(q^i)$.*

For provided ports, $\mathcal{C} \models_{prov} (z, t, n)$ likewise indicates that the provided port of component z with interface t and with associated number n is not bound to more than n components.

The configuration $\mathcal{C}$ is correct *if for each component $z \in Z$, with $S(z) = \langle \mathcal{T}, q\rangle$, for each $(p \mapsto n_p) \in \mathcal{T}.\mathbf{P}(q)$ we have $\mathcal{C} \models (z, p, n_p)$, and for each $(r \mapsto n_r) \in \mathcal{T}.\mathbf{R}(q)$ we have $\mathcal{C} \models (z, r, n_r)$.*

The operational semantics of the Aeolus model is given by a labelled transition relation $\rightarrow \subseteq \mathbb{C} \times A \times \mathbb{C}$. We write $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ for $\langle \mathcal{C}, \alpha, \mathcal{C}'\rangle \in \rightarrow$, where $\mathcal{C}, \mathcal{C}'$ are configurations, and $\alpha \in A$ is an action of the following form:

- $\{\texttt{stateChange}(z_j, q_j^1, q_j^2) \mid j \in J\}$, where $z_j \in \mathcal{Z}$, $q_j^1, q_j^2 \in \mathcal{Q}$, and $J \neq \emptyset$;
- $\texttt{bind}(r, z_1, z_2)$, where $r \in \mathcal{I}$, $z_1, z_2 \in \mathcal{Z}$;
- $\texttt{unbind}(r, z_1, z_2)$, where $r \in \mathcal{I}$, $z_1, z_2 \in \mathcal{Z}$;
- $\texttt{new}(z : \mathcal{T})$, where $z \in \mathcal{Z}, \mathcal{T} \in \Gamma$;
- $\texttt{del}(z)$, where $z \in \mathcal{Z}$.

$$[\textsc{Change}]\ \frac{\{z_j \mid j \in J\} \subseteq Z \qquad \begin{array}{c}\forall j \in J,\ \mathcal{C}(z_j) = \langle \mathcal{T}_j, q_j^1 \rangle \ \wedge\ \langle q_j^1, q_j^2 \rangle \in \mathcal{T}.\texttt{trans} \ \wedge\ S'[z_j] = \langle \mathcal{T}_j, q_j^2 \rangle \\ \forall z' \in Z \setminus \{z_j \mid j \in J\},\ S'[z'] = S[z']\end{array}}{\langle U, Z, S, B\rangle \xrightarrow{\{\texttt{stateChange}(z_j, q_j^1, q_j^2) \mid j \in J\}} \langle U, Z, S', B\rangle}$$

$$[\textsc{Bind}]\ \frac{\langle r, z_1, z_2\rangle \notin B \qquad r \in \mathcal{C}[z_1].\texttt{req} \cap \mathcal{C}[z_2].\texttt{prov}}{\langle U, Z, S, B\rangle \xrightarrow{\texttt{bind}(r, z_1, z_2)} \langle U, Z, S, B \cup \{\langle r, z_1, z_2\rangle\}\rangle}$$

$$[\textsc{Unbind}]\ \frac{\langle r, z_1, z_2\rangle \in B}{\langle U, Z, S, B\rangle \xrightarrow{\texttt{unbind}(r, z_1, z_2)} \langle U, Z, S, B \setminus \{\langle r, z_1, z_2\rangle\}\rangle}$$

$$[\textsc{New}]\ \frac{z \notin Z \qquad \mathcal{T} \in U \qquad S'[z] = \langle \mathcal{T}, \mathcal{T}.\texttt{init}\rangle \qquad \forall z' \in Z,\ S'[z'] = S[z']}{\langle U, Z, S, B\rangle \xrightarrow{\texttt{new}(z:\mathcal{T})} \langle U, Z \cup \{z\}, S', B\rangle}$$

$$[\textsc{Del}]\ \frac{z \in Z \qquad B' = \{\langle n, z_1, z_2\rangle \mid z \notin \{z_1, z_2\}\} \qquad \forall z' \in Z \setminus \{z\},\ S'[z'] = S[z']}{\langle U, Z, S, B\rangle \xrightarrow{\texttt{del}(z)} \langle U, Z \setminus \{z\}, S', B \setminus B'\rangle}$$

Figure 16: Operational semantics rules for the Aeolus model

Intuitively, the Aeolus transition relation defines all the possible actions possible for deploying a given set of component types U. A deployment is deemed successful when a target correct configuration $\mathcal{C}_{n+1}$ is reached from a correct starting configuration $\mathcal{C}_0$ by a sequence of transitions $\mathcal{C}_i \xrightarrow{\alpha_i} \mathcal{C}_{i+1}$, $i \in \{0, \ldots, n\}$, where $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ are correct configurations.

The transition relation $\rightarrow$ is formally defined as the least relation satisfying the inference rules in Figure 16. These rules correspond exactly to the so-called *reconfiguration* rules in [31], except for rule SC which takes directly into account the possibility of atomic multiple state changes in [31].

The intuitive meaning of the rules in Figure 16 is as follows. Rule Change corresponds to a transition where multiple components $\{z_j \mid j \in J\}$ spontaneously change their state (from q_j^1 to q_j^2 for component z_j). Rule Bind establishes a new binding between two components z_1 and z_2, connecting respectively their required and provided ports bearing the interface name r. Rule Unbind removes a binding between two components. Rule New adds a new component z with component type $\mathcal{T}$. Rule Del removes component z from the set of deployed components, together with all the bindings involving that component. Notice that in all the rules the set U of available component types remains unchanged.

8.2 Location-Graph interpretation of the Aeolus model concepts

An Aeolus configuration $\langle U, Z, S, B\rangle$ is interpreted as a location graph where the set of available templates U is held by an Administrator location, the set of deployed components Z is modeled as a set of Aeolus_component locations, and the set of bindings B is modeled as a set of Aeolus_binding locations. Function S in an Aeolus configuration, which maps a deployed component to its component type and its current state, is represented by attributes of deployed components.

The Administrator location can be interpreted as a reification of a deployment orchestrator for Aeolus configurations, i.e. the component responsible for deploying a given Aeolus configuration, conforming to a set of component types. The role of an Administrator in our interpretation is thus to hold the set of available component types to use in the deployment of the target configuration, and to drive the deployment of the target configuration by interacting with, and modifying, the set of deployed components. Note that the definition of the Administrator behaviour is highly non deterministic, reflecting the fact that it just corresponds to the possible Aeolus transitions, and does not embody any specific deployment strategy.

In a Configuration we require the Administrator to be bound to (in location graph parlance, i.e. to have a

shared role with) each deployed component and each binding by some anotif role. These anotif roles are used by the administrator to receive notifications from deployed components (hence the name), as well as to send command to deployed components and bindings, e.g. for binding and unbinding deployed components (facts AllComponentsHaveNotif and AllBindingsHaveNotif in the definition of Configuration below).

An Administrator maintains two pieces of information: its templates, i.e. the set of component types of its configuration, and its zconfig, i.e. a representation of the deployed components and established bindings of its configuration. For simplicity in our specification, a ZConfiguration contains *exactly* the deployed components and bindings found in the configuration (facts admin.process.zconfig.zcomponents = components and admin.process.zconfig.zbindings = bindings in the definition of Configuration). An Aeolus component type is represented by a DL_Machine ("DL" stands for deployment lifecycle). The definition of a DL_Machine closely follows that of an Aeolus component type. If $\langle Q, q_0, T, P, D\rangle$ is an Aeolus component type, the corresponding elements of the associated DL_Machine are as follows:

- The dstates attribute corresponds to the set of states Q.
- The dinit attribute corresponds to the initial state q_0.
- The dtransitions attribute, which is just a set of DL_Transitions, corresponds to the transition relation T.
- the pports attribute corresponds to the set of provided ports given by the first element of the pair P.
- the rports attribute corresponds to the set of required ports given by the first element of the pair P.
- the activates attribute, which is a set of Activations, corresponds to the activation function D.

We have added to the definition of a DL_Machine the attribute dfinal corresponding to a set of possible final states for a component type. Final states are useful to determine a notion of target configuration, which we define below.

Note that ports in an Aeolus component type (which are just interface names) are represented in a DL_Machine as Names.

An Activation records, for a given state astate, the set of provided ports ap, and the set of required ports ar, which are activated in this state, together with their cardinalities (attribute acard in PortCard). A PortCard just refers to a port by its interface name (attribute asort), modelled as a Sort, and records its cardinality (an integer) via attribute acard.

```
sig Configuration extends LocationGraph {
  admin: one Administrator,
  components: set Aeolus_component,
  bindings: set Aeolus_binding
}{
  locations = components + bindings + admin
  admin.process.zconfig.zcomponents = components
  admin.process.zconfig.zbindings = bindings
  AllComponentsHaveNotif[this]
  AllBindingsHaveNotif[this]
}

sig Administrator extends Location {}
{
  process in Admin_process
  process.anotif in required
}

sig Admin_process extends Process {
  templates: set DL_Machine,
  zconfig: one ZConfiguration,
```

```
  anotif: set Role
}
{
  zconfig.zcomponents.process.template in templates
}

sig ZConfiguration extends LocationGraph {
  zcomponents: set Aeolus_component,
  zbindings: set Aeolus_binding
}{
  locations=zbindings+zcomponents
}

sig DL_Machine{
  dinit: one DL_State, // the initial state of the automata
  dstates: set DL_State,
  dtransitions: set DL_Transition,
  pports: set Name,
  rports: set Name,
  activates: set Activation,
  dfinal: set DL_State
}{
  dinit in dstates
  dtransitions.initial + dtransitions.terminal in dstates
  activates.astate in dstates
  activates.ap.asort in pports
  activates.ar.asort in rports
  dfinal in dstates
}


sig DL_State extends Value{}

sig Activation {
  astate: DL_State,
  ap: set PortCard,
  ar: set PortCard,
}

sig PortCard{
  asort: one Name,
  acard: Int,
}

sig DL_Transition{
  initial: one DL_State,
  terminal: one DL_State
}
```

An Aeolus deployed component is modelled as an Aeolus_component, i.e. a Location whose process is an Aeolus_process. In an Aeolus_process, the current attribute corresponds to the current state of a deployed component (the state q, if the deployed component is z in some Aeolus configuration $\langle U, Z, S, B\rangle$, and $S(z) = \langle \mathcal{T}, q\rangle$). The template attribute corresponds to the component type of the deployed components. The cnotif attribute identifies the provided role through which an Aeolus_component can interact with the configuration Administrator in a given Configuration. The pprov attribute corresponds to the provided Ports of the deployed component, and the preq attribute corresponds to its required Ports. A Port in an Aeolus_component has an interface name (its psort attribute), but also a collection of roles (its roles attribute) which are used for establishing bindings between Aeolus_components. More precisely, a Port gets a new role when a binding is established with this component (corresponding to the Bind operation in the Aeolus

model), and loses a role when a binding with this component is removed (corresponding to the Unbind operation – see the next section on the interpretation of the Aeolus operational semantics for details). The facts associated with Aeolus_process register basic consistency requirements:

- The current state should be a state allowed by the component template (its DL_Machine).
- The interface names of provided and required ports must appear in the template provided and required interface names, respectively.
- Two provided Ports in a given Aeolus_component are uniquely identified by their interface name (their psort). Likewise for required Ports.
- The roles associated with a given Port are distinct from those of any other Port in the same Aeolus_component. In other terms, each role in a given Port identifies a unique binding to some other component.

An Aeolus binding is modelled as an Aeolus_binding, i.e. a location which connects tow Aeolus_components. An Aeolus_binding has a notification role (its bnotif attribute) to interact with its Administrator in a given Configuration, a role in required position (its prov attribute) to connect to a provided Port of an Aeolus_component, and a role in provided position (its req attribute) to connect to a required Port.

```
sig Aeolus_component extends Location {}
{
  process in Aeolus_process
  provided = process.pprov.roles + process.cnotif
  required = process.preq.roles
}


sig Aeolus_process extends Process{
  current: one DL_State,
  template: one DL_Machine,
  pprov: set Port,
  preq: set Port,
  cnotif: one Role
}{
  current in template.dstates
  pprov.psort = template.pports
  preq.psort = template.rports
  all p1,p2: pprov | p1.psort = p2.psort implies p1 = p2
  all p1,p2: preq | p1.psort = p2.psort implies p1 = p2
  all disj p1,p2: pprov | no p1.roles & p2.roles
  all disj r1,r2: preq | no r1.roles & r2.roles
}

abstract sig Port {
  psort: one Sort,
  roles: set Role
}

sig Aeolus_binding extends Location {
  bsort: one Sort,
  req: one Role,
  prov: one Role,
  bnotif: one Role
}{
  provided = req + bnotif
  required = prov
}
```

A few auxiliary functions are used in the specification for defining configuration correctness predicates. PPortCardForPort extracts the PortCard associated with a given interface name of a provided port. RPortCardForPort does the same for a required port interface name. ActivePPorts, resp. ActiveRPorts returns the set of provided, resp. required, ports currently actived in an Aeolus_component.

```
fun PPortCardForPort[m: DL_Machine, p: Sort]: one PortCard {
   { q: m.activates.ap | q.asort = p }
}

fun RPortCardForPort[m: DL_Machine, r: Sort]: one PortCard {
   { q: m.activates.ar | q.asort = r }
}

fun ActivePPorts[z: Aeolus_component] : set Port {
  {p : z.process.pprov | p.psort in z.process.template.activates.ap.asort}
}


fun ActiveRPorts[z: Aeolus_component] : set Port {
  {r : z.process.preq | r.psort in z.process.template.activates.ar.asort}
}
}
```

We now define a well-formedness predicate, WFConfiguration, for Configurations. It enforces consistency constraints related to the construction of a Configuration:

- The set of components and bindings maintained by an Administrator in its zconfig attribute reflects the actual state of deployed components and bindings (predicate ZconfigReflectsConfig).
- All Aeolus_bindings in a given Configuration connect two different Aeolus_components in the configuration (predicate BindingsConnectDifferentComponents).
- All Aeolus_bindings and Aeolus_components in a Configuration have a notification role which binds them to the Administrator of the Configuration (predicates AllBindingsHaveNotif and AllComponentsHaveNotif).
- Finally, all templates of Aeolus_components in a Configuration are templates that are taken from the set of templates maintained by the Administrator of the Configuration (predicate AllComponentsHaveTemplatesFromAdmin).

We can also define predicates characterizing a correct Configuration, namely, a Configuration that respects the cardinality constraints associated with its different ports, as in the definition of a correct Aeolus configuration (predicates AllComponentsRespectTheirProvidedPortsCardinalityConstraints, AllComponentsRespectTheirRequiredPortsCardinalityConstra and AllConflictsConstraintsAreMet). Note that cardinality constraints on ports are interpreted as follows:

- For a Provided port, a non 0 integer indicates a maximum number of roles this port can have when active (hence a maximum number of bindings with the port's sort the component can have). A Provided port with 0 in its cardinality means there are no constraints on the number of roles this port can have.
- For a Required port, a non 0 integer indicates the minimum number of roles this port can have when active (hence the minimum number of bindings with this sort). A Required port with cardinality 0 forbids the presence of other components in the current configuration with an active Provided port of the same sort as this port.

```
pred ZconfigReflectsConfig[c: Configuration] {
  c.admin.process.zconfig.locations = (c.locations − c.admin)
}
```

```
pred BindingsConnectDifferentComponents [c: Configuration] {
  all b : c.bindings | some disj z1,z2: c.components | some disj p1,p2: Port | some disj r1,r2: Role |
      p1 in z1.process.template.rports
      and p2 in z2.process.template.pports
      and r1 in p1.roles
      and r2 in p2.roles
      and p1.psort = b.bsort
      and p2.psort = b.bsort
      and b.req = r1
      and b.prov = r2
}

pred AllComponentsHaveNotif [c: Configuration] {
   all z: c.components | some r : Role | r in c.admin.process.anotif & z.process.cnotif
}

pred AllBindingsHaveNotif [c: Configuration] {
   all b: c.bindings | some r : Role | r in c.admin.process.anotif & b.bnotif
}

pred AllComponentsHaveTemplatesFromAdmin [c:Configuration] {
   all z:c.components | StripDLMachineFromRoles[z.process.template] in c.admin.process.templates
}

pred AllComponentsRespectTheirProvidedPortsCardinalityConstraints [c: Configuration] {
   all z: c.components | all p : ActivePPorts[z] | let n = PPortCardForPort[z.process.template,p].acard |
   n > 0 implies #(p.roles) <= n
}

pred AllComponentsRespectTheirRequiredPortsCardinalityConstraints [c: Configuration] {
   all z: c.components | all r : ActiveRPorts[z] | let n = RPortCardForPort[z.process.template,r].acard |
   n > 0 implies #(r.roles) >= n
}

pred AllConflictsConstraintsAreMet [c: Configuration] {
   all z: c.components | all r: ActiveRPorts[z] | RPortCardForPort[z.process.template,r].acard = 0 implies
                                  no p: ActivePPorts[c.components - z] | p.psort = r.psort
}

pred WFConfiguration [c: Configuration] {
  WellFormedLocationSet[c.locations]
  and ZconfigReflectsConfig[c]
  and BindingsConnectDifferentComponents[c]
  and AllComponentsHaveNotif[c]
  and AllBindingsHaveNotif[c]
  and AllComponentsHaveTemplatesFromAdmin[c]
}
```

We finally add a notion of target configuration, defined by the predicate TargetConfiguration. The Aeolus model defines a notion of final correct configuration, which is just given by a set component types and a deployed component in a given state. Our notion of possible final states in DL_Machines allows us to be more precise in defining what we expect in a target deployed configuration, namely that the configuration obtained be well-formed and that all deployed components be in a final state with respect to their deployment lifecycle.

```
pred TargetConfiguration [c : Configuration] {
  WFConfiguration[c]
  and ( all z: c.components | z.process.current in z.process.template.dfinal )
}
```

We can check that our specification makes sense and, in particular, that there exists a non-trivial

well-formed Configuration.

```
run Model{
   some c: Configuration| WFConfiguration[c] and #c.components>1 and #c.components.process.template.activates.ap > 1
} for 10 but exactly 1 Configuration, exactly 2 DL_Machine, exactly 4 Aeolus_component, exactly 2 PortCard expect 1
```

8.3 Location-Graph interpretation of the Aeolus model operational semantics

In this subsection, we give the location-graph interpretation of the different actions of Aeolus: new component creation, component deletion, binding and unbinding. For each one of them, we explicit the global transition of the system and we detail the local transitions that allow the execution of such a global transition. In fact, as explained in Section. 2.2, any global transition is either the result of an uncoupling where one location evolves to a new location graph and the rest of the initial graph is unchanged, or the result of the synchronization of a set a local transitions. When interpreting Aeolus actions, we find both of the cases.

8.3.1 State Change

First, we define the expected global transition for a component state change. Apart from the presence of the administrator, and the fact that new role are added to the target components to bind them to their binding location, this global transition mirrors the StateChange action of the original Aeolus operational semantics: a component z changes its state by following a transition of its deployment life cycle automaton. The initial configuration is transformed in a new configuration where: the z component has updated its current state with the terminal state of the DL automaton transition.

```
pred Global_StateChange_Transition[c1,c2: Configuration, zs: set Aeolus_component, t: Transition]
{ some a1, a2: Administrator, zps: set Aeolus_component |
        t.init = c1.locations
        and (t.label.signals = none)
        and t.term = c2.locations
        and c1.admin = a1
        and zs in c1.components & a1.process.zconfig.zcomponents
        and a2.name = a1.name
        and a2.process.anotif = a1.process.anotif
        and a2.provided = a1.provided
        and a2.required = a1.required
        and a2.process.templates = a1.process.templates
        and a2.process.zconfig.zcomponents = (a1.process.zconfig.zcomponents − zs) + zps
        and a2.process.zconfig.zbindings = a1.process.zconfig.zbindings
        and ( all zp : zps | one z: zs | some dt: z.process.template.dtransitions |
                                    ChangeState[zp, z, dt.terminal] and dt.initial = z.process.current )
        and c2.admin = a2
        and c2.components = (c1.components − zs) + zps
        and c2.bindings = c1.bindings
}


pred ChangeState[zp, z: Aeolus_component, q: DL_State] {
            zp.name = z.name
            and zp.provided = z.provided
          and zp.required = z.required
            and zp.process.cnotif = z.process.cnotif
            and zp.process.current = q
            and zp.process.template = z.process.template
            and zp.process.preq = z.process.preq
            and zp.process.pprov = z.process.pprov
}
```

We now define individual transitions of administrator and deployed component to effect a state change in the target deployed component:

- the chosen deployed component changes its state and notifies its administrator of the new state
- the administrator updates its knowledge of the current configuration with the chosen deployed component in the new state that it notifies

```
pred Admin_State_Change_Transition[a1: Administrator, zs: set Aeolus_component, t:Transition]{
    some a2: Administrator, zps: set Aeolus_component, is: set Interaction, v: ScOp |
        zs in a1.process.zconfig.zcomponents
        and t.init = a1
        and a2.name = a1.name
        and a2.process.anotif = a1.process.anotif
        and a2.provided = a1.provided
        and a2.required = a1.required
        and a2.process.templates = a1.process.templates
        and a2.process.zconfig.zcomponents = (a1.process.zconfig.zcomponents − zs) + zps
        and ( all zp: zps, i: is | one z: zs | some dt: z.process.template.dtransitions |
                                               ChangeState[zp, z, dt.terminal] and dt.initial = z.process.current
                                               and i.irole = z.process.cnotif and i.polarity in Minus and i.payload = v and v.scstate = dt.terminal
        and t.label.signals = is
        and t.term = a2
}


sig ScOp extends Value{
    scstate: one DL_State
}


pred Comp_State_Change_Transition[z: Aeolus_component, t: Transition]{
    some zp: Aeolus_component, i: Interaction, dt: z.process.template.dtransitions, v: ScOp |
      t.init = z
      and dt.initial = z.process.current
      and i.irole = z.process.cnotif and i.polarity in Plus and i.payload = v and v.scstate= dt.terminal
      and t.label.signals = i
      and ChangeState[zp, z, dt.terminal]
      and t.term = zp
}

pred State_Change_Transition[c1,c2:Configuration, zs: set Aeolus_component, t: Transition]{
     some ta, t1: Transition, tzs: set Transition | let e = t.env |
     zs in c1.components
     and ta.env = e and t1.env = e and ts.env = e
     and Admin_State_Change_Transition[c1.admin, z, ta]
     and ( all tz: tzs | one z: zs | Comp_State_Change_Transition[z,tz] )
     and Synchronizing_Transition[t1, ta + tzs]
     and Uncoupling_Transition[ t, t1, c1.locations − (c1.admin + z) ]
     and c2.locations = t.term
     and c2.admin =AdminFromLocationSet[t.term]
     and c2.components = ComponentsFromLocationSet[t.term]
     and c2.bindings = BindingsFromLocationSet[t.term]
}


fun AdminFromLocationSet[s: set Location] : one Administrator {
  { a: Administrator | a in s }
}

fun ComponentsFromLocationSet[s: set Location] : set Aeolus_component {
```

```
  { z: Aeolus_component | z in s }
}

fun BindingsFromLocationSet[s: Location] : set Aeolus_binding {
  { b : Aeolus_binding | b in s }
}
```

8.3.2 Bind

First, we define the expected global transition for creating a new binding. Apart from the presence of the administrator, and the fact that new role are added to the target components to bind them to their binding location, this global transition mirrors the Bind action of the original Aeolus operational semantics: bound components z1 and z2 are unchanged (except for their new role) and there should not exist a binding between z1 and z2 of the chosen interface (sort here our interpretation). The initial configuration is transformed in a new configuration where:

- a new binding location is added
- a new administrator component replaces the initial one, reflecting the new configuration
- each one of the two components linked by the new binding is replaced by a new component in the terminal configuration, differing by an additional role which is the point of attachement with the new binding location.

```
pred Global_Bind_Transition[c1, c2: Configuration, disj z1, z2: Aeolus_component, s: Sort, t: Transition]
{ some a1,a2: Administrator, b: Aeolus_binding, disj z1p, z2p: Aeolus_component, disj r, r1,r2: Role |
        c1.admin =a1
        and z1 + z2 in c1.components & a1.process.zconfig.zcomponents
        and s in z1.process.pprov.psort & z2.process.preq.psort
        and Unbound[c1,z1,z2,s]
        and r + r1 + r2 not in t.env.eatoms
        and t.init= c1.locations
        and a2.name = a1.name
        and a2.process.anotif = a1.process.anotif + r
        and a2.provided = a1.provided
        and a2.required = a1.required + r
        and a2.process.templates = a1.process.templates
        and a2.process.zconfig.zcomponents = (a1.process.zconfig.zcomponents − (z1 + z2)) + z1p + z2p
        and a2.process.zconfig.zbindings = a1.process.zconfig.zbindings + b
        and b.bsort = s
        and b.bnotif = r
        and b.prov=r1
        and b.req =r2
        and AddProvidedRole[z1p, z1,s,r1]
        and AddRequiredRole[z2p, z2,s,r2]
        and (no t.label.signals)
        and t.term = c2.locations
        and c2.admin = a2
        and c2.components = (c1.components − (z1 + z2)) + z1p + z2p
        and c2.bindings = c1.bindings + b
}

pred Unbound[c: Configuration, disj z1,z2: Aeolus_component, s: Sort]{
       z1 + z2 in c.components
      and no b: c.bindings | b.bsort = s
}

pred PortFromSort[p: Port, z: Aeolus_component, s:Sort]{
```

```
  (p in z.process.pprov + z.process.preq) and p.psort = s
}

pred AddRoleToPort[q, p: Port, r: Role] {
  q.psort = p.psort and q.roles = p.roles + r
}

pred AddProvidedRole[zp, z: Aeolus_component, s:Sort, r: Role] {
  some p,q: Port |
        PortFromSort[p, z, s]
        and zp.name = z.name
        and zp.process.cnotif = z.process.cnotif
        and p in z.process.pprov
        and zp.provided = z.provided + r
    and zp.required = z.required
        and zp.process.current = z.process.current
        and zp.process.template = z.process.template
        and zp.process.preq = z.process.preq
        and zp.process.pprov = (z.process.pprov - p) + q
        and AddRoleToPort[q, p, r]
}

pred AddRequiredRole[zp,z: Aeolus_component, s:Sort, r: Role] {
   some p,q: Port |
          PortFromSort[p, z,s]
          and zp.name = z.name
          and zp.process.cnotif = z.process.cnotif
          and p in z.process.preq
          and zp.provided = z.provided
      and zp.required = z.required + r
          and zp.process.current = z.process.current
          and zp.process.template = z.process.template
          and zp.process.pprov = z.process.pprov
          and zp.process.preq = (z.process.preq - p) + q
          and AddRoleToPort[q,p,r]
}
```

We now define individual transitions of administrator and deployed components to create a binding between the tarrget deployed components.

```
pred Admin_Binding_Transition[a1: Administrator, disj z1,z2: Aeolus_component, s:Sort, t:Transition]{
   some a2: Administrator, b: Aeolus_binding, disj z1p, z2p: Aeolus_component, disj r,r1,r2: Role, disj i1,i2: Interaction, v1: BdpOp, v2: B
      z1 + z2 in a1.process.zconfig.zcomponents
      and s in z1.process.pprov.psort & z2.process.preq.psort
      and UnboundZ[a1.process.zconfig,z1,z2,s]
      and r + r1 + r2 not in t.env.eatoms
      and t.init= a1
      and a2.name = a1.name
      and a2.process.anotif = a1.process.anotif + r
      and a2.provided = a1.provided
      and a2.required = a1.required + r
      and a2.process.templates = a1.process.templates
      and a2.process.zconfig.zcomponents = (a1.process.zconfig.zcomponents - (z1 + z2)) + z1p + z2p
      and a2.process.zconfig.zbindings = a1.process.zconfig.zbindings + b
      and b.bsort = s
      and b.bnotif = r
      and b.prov=r1
      and b.req =r2
      and AddProvidedRole[z1p,z1,s,r1]
      and AddRequiredRole[z2p,z2,s,r2]
      and i1.irole = z1.process.cnotif and i1.polarity in Minus and i1.payload = v1 and v1.bdprole = r1
      and i2.irole = z2.process.cnotif and i2.polarity in Minus and i2.payload = v2 and v2.bdrrole = r2
```

```
        and t.label.signals = i1 + i2
        and t.term = a2 + b
}

sig BdpOp extends Value {
  bdprole: one Role
}

sig BdrOp extends Value {
  bdrrole: one Role
}

pred Provided_Comp_Binding_Transition[z: Aeolus_component, s:Sort, t: Transition]{
   some zp: Aeolus_component, r: Role, i: Interaction, v: BdpOp |
      t.init = z
      and i.irole = z.process.cnotif and i.polarity in Plus and i.payload = v and v.bdprole = r
      and t.label.signals = i
      and AddProvidedRole[zp,z,s,r]
      and t.term = zp
}

pred Required_Comp_Binding_Transition[z: Aeolus_component, s:Sort, t: Transition]{
   some zp: Aeolus_component, r: Role, i: Interaction, v: BdrOp |
      t.init = z
      and i.irole = z.process.cnotif and i.polarity in Plus and i.payload = v and v.bdrrole = r
      and t.label.signals = i
      and AddRequiredRole[zp,z,s,r]
      and t.term = zp
}


pred Binding_Transition[c1,c2:Configuration, z1,z2: Aeolus_component, s: Sort, t: Transition]{
    some ta, tz1, tz2, t1: Transition | let e = t.env |
    z1 + z2 in c1.components
    and ta.env = e and tz1.env = e and tz2.env = e and t1.env = e
    and Admin_Binding_Transition[c1.admin,z1,z2,s,ta]
    and Provided_Comp_Binding_Transition[z1,s,tz1]
    and Required_Comp_Binding_Transition[z2,s,tz2]
    and Synchronizing_Transition[t1, ta + tz1 + tz2]
    and Uncoupling_Transition[t, t1, c1.locations − (c1.admin + z1 + z2) ]
    and t.init = c1.locations
    and t.term = c2.locations
    and c2.admin = AdminFromLocationSet[t.term]
    and c2.components = ComponentsFromLocationSet[t.term]
    and c2.bindings = BindingsFromLocationSet[t.term]
}

pred UnboundZ[c: ZConfiguration, disj z1,z2: Aeolus_component, s: Sort]{
       z1 + z2 in c.zcomponents
      and no b: c.zbindings | b.bsort = s
}
```

8.3.3 Unbind

First, we define the expected global transition for an unbind transition. Apart from the presence of the administrator, and the fact a binding is represented by a Binding location, this global transition mirrors the Unbind action of the original Aeolus operational semantics: a binding between two target components is removed. The initial configuration is transformed in a new configuration where the binding between the two target components has been removed.

```
pred Global_Unbind_Transition[c1, c2: Configuration, disj z1, z2: Aeolus_component, s: Sort, t: Transition]
{ some a1, a2: Administrator, disj z1p, z2p: Aeolus_component, b: Aeolus_binding |
        c1.admin = a1
        and z1 + z2 in c1.components & a1.process.zconfig.zcomponents
        and b = BindingFromConfiguration[c1, z1, z2, s]
        and t.init= c1.locations
        and a2.name = a1.name
        and a2.process.anotif = a1.process.anotif − b.bnotif
        and a2.provided = a1.provided
        and a2.required = a1.required − b.bnotif
        and a2.process.templates = a1.process.templates
        and a2.process.zconfig.zcomponents = (a1.process.zconfig.zcomponents − (z1 + z2)) + z1p + z2p
        and a2.process.zconfig.zbindings = a1.process.zconfig.zbindings − b
        and RemoveProvidedRole[z1p, z1, s, b.prov]
        and RemoveRequiredRole[z2p, z2, s, b.req]
        and (no t.label.signals)
        and t.term = c2.locations
        and c2.admin = a2
        and c2.components = (c1. components − (z1 + z2)) + z1p + z2p
        and c2.bindings = c1.bindings − b
}


fun BindingFromConfiguration [c: Configuration, z1,z2: c.components, s: Sort]: Aeolus_binding {
  { b : c.bindings | b.bsort = s and b.prov in z1.process.pprov.roles and b.req in z2.process.preq.roles }
}

pred RemoveProvidedRole[zp, z: Aeolus_component, s: Sort, r: Role] {
  some p, q: Port |
        PortFromSort[p, z,s]
        and zp.name = z.name
        and zp.provided = z.provided − r
        and zp.required = z.required
        and zp.process.current = z.process.current
        and p in z.process.pprov
        and zp.process.template = z.process.template
        and zp.process.pprov = (z.process.pprov − p) + q
        and RemoveRoleFromPort[q, p, r]
        and zp.process.preq = z.process.preq
        and zp.process.cnotif = z.process.cnotif
}

pred RemoveRoleFromPort [q, p: Port, r: Role] {
  q.psort = p.psort and q.roles = p.roles − r
}

pred RemoveRequiredRole[zp, z: Aeolus_component, s: Sort, r: Role] {
  some p,q: Port |
        PortFromSort[p, z,s]
        and zp.name = z.name
        and zp.provided = z.provided − r
        and zp.required = z.required
        and zp.process.current = z.process.current
        and p in z.process.preq
        and zp.process.template = z.process.template
        and zp.process.preq = (z.process.preq − p) + q
        and RemoveRoleFromPort[q,p,r]
        and zp.process.pprov = z.process.pprov
        and zp.process.cnotif = z.process.cnotif
}
```

We now define individual transitions of administrator, binding and deployed components to effect an

unbind: - the chosen deployed component changes its state and notifies its administrator of the new state - the administrator updates its knowledge of the current configuration with the chosen deployed component in the new state that it notifies.

```
pred Admin_Unbinding_Transition[a1: Administrator, z1,z2: Aeolus_component, s: Sort, t:Transition]{
    some a2: Administrator, z1p, z2p: Aeolus_component, b: Aeolus_binding, i, i1, i2: Interaction, v : UbOp, v1: UbpOp, v2: UbrOp |
      z1 + z2 in a1.process.zconfig.zcomponents
      and s in z1.process.pprov.psort & z2.process.preq.psort
      and b = BindingFromZConfiguration[a1.process.zconfig, z1, z2, s]
      and a2.name = a1.name
      and a2.provided = a1.provided
      and a2.required = a1.required − b.bnotif
      and a2.process.template = a1.process.template
      and a2.process.anotif = a1.process.anotif − b.bnotif
      and a2.process.zconfig.zcomponents = (a1.process.zconfig.zcomponents − (z1 + z2)) + z1p + z2p
      and a2.process.zconfig.zbindings = a1.process.zconfig.zbindings − b
      and RemoveProvidedRole[z1p, z1, s, b.prov]
      and RemoveRequiredRole[z2p, z2, s, b.req]
      and t.init = a1
      and t.term = a2
      and t.label.signals = i + i1 + i2
      and i.irole = b.bnotif and i.polarity in Minus and i.payload = v
      and i1.irole = z1.process.cnotif and i1.polarity in Minus and i1.payload = v1 and v1.ubprole = b.prov
      and i2.irole = z2.process.cnotif and i2.polarity in Minus and i2.payload = v2 and v2.ubrrole = b.req
}

fun BindingFromZConfiguration [c: ZConfiguration, z1,z2: c.zcomponents, s: Sort]: Aeolus_binding {
  { b : c.zbindings | b.bsort = s and b.prov in z1.process.pprov.roles and b.req in z2.process.preq.roles }
}

one sig UbOp extends Value{}

sig UbpOp extends Value{
    ubprole: one Role
}

sig UbrOp extends Value {
    ubrrole: one Role
}


pred Provided_Comp_Unbinding_Transition[z: Aeolus_component, s:Sort, t: Transition]{
    some zp: Aeolus_component, r: Role, i: Interaction, v: UbpOp |
      t.init = z
      and i.irole = z.process.cnotif and i.polarity in Plus and i.payload = v and v.ubprole = r
      and t.label.signals = i
      and RemoveProvidedRole[zp,z,s,r]
      and t.term = zp
}


pred Required_Comp_Unbinding_Transition[z: Aeolus_component, s:Sort, t: Transition]{
    some zp: Aeolus_component, r: Role, i: Interaction, v: UbrOp |
      t.init = z
      and i.irole = z.process.cnotif and i.polarity in Plus and i.payload = v and v.ubrrole = r
      and t.label.signals = i
      and RemoveRequiredRole[zp,z,s,r]
      and t.term = zp
}
```

```
pred Remove_Binding_Unbinding_Transition[b: Aeolus_binding, t: Transition] {
    some i: Interaction, v: UbOp |
        t.init = b
        and t.label.signals = i
        and i.irole = b.bnotif and i.polarity in Plus and i.payload = v
        and t.term = none
}


pred Unbinding_Transition[c1,c2: Configuration, disj z1, z2: Aeolus_component, s: Sort, t: Transition]{
    some ta, tz1, tz2, tb, t1: Transition, b: Aeolus_binding | let e = t.env |
        z1 + z2 in c1.components
        and b in c1.bindings
        and ta.env = e and tz1.env = e and tz2.env = e and t1.env = e
        and Admin_Unbinding_Transition[c1.admin, z1, z2, s, ta]
        and Provided_Comp_Unbinding_Transition[z1, s, tz1]
        and Required_Comp_Unbinding_Transition[z2, s, tz2]
        and Remove_Binding_Unbinding_Transition[b, tb]
        and Synchronizing_Transition[t1, ta + tb + tz1 + tz2]
        and Uncoupling_Transition[t, t1, c1.locations − (c1.admin + z1 + z2 + b)]
        and t.init = c1.locations
        and c2.admin = AdminFromLocationSet[t.term]
        and c2.components = ComponentsFromLocationSet[t.term]
        and c2.bindings = BindingsFromLocationSet[t.term]
        and t.term = c2.locations
}
```

8.3.4 New

First, we define the expected global transition for the creation of a new component. Apart from the presence of the administrator, and the need to create a new notification role r, and a new name n for the new components, as well as assign a sort s to the new component, this global transition mirrors the New action of the original Aeolus operational semantics: a new component is created from a component template (a DL_Machine) and added to the set of deployed components.

```
pred Global_New_Component_Transition[c1,c2: Configuration, u: DL_Machine, t: Transition]
{ some a1, a2: Administrator, z: Aeolus_component, r: Role, n: Name |
        t.init = c1.locations
        and (t.label.signals = none)
        and t.term = c2.locations
        and c1.admin = a1
        and u in a1.process.templates
        and no (r + n) & t.env.eatoms
        and NewComponentFromTemplate[z, u, r, n]
        and a2.name = a1.name
        and a2.process.anotif = a1.process.anotif + r
        and a2.provided = a1.provided
        and a2.required = a1.required + r
        and a2.process.templates = a1.process.templates
        and a2.process.zconfig.zcomponents = a1.process.zconfig.zcomponents + z
        and a2.process.zconfig.zbindings = a1.process.zconfig.zbindings
        and c2.admin = a2
        and c2.components = c1.components + z
        and c2.bindings = c1.bindings
}


pred NewComponentFromTemplate[z: Aeolus_component, u: DL_Machine, r: Role, n: Name]{
        some s : Sort |
```

```
        z.name = n
      and z.sort = s
      and z.provided = r
      and z.required = none
      and z.process.current = u.dinit
      and z.process.template = u
      and z.process.pprov = u.pports
      and z.process.preq = u.rports
      and z.process.cnotif = r
}
```

We now define the individual transition of an administrator in a configuration to effect a new component creation.

```
pred Admin_New_Transition[a1: Administrator, u: DL_Machine, t:Transition]{
   some a2: Administrator, z: Aeolus_component, r: Role, n: Name |
      not (z in a1.process.zconfig.zcomponents)
      and no (r + n) & t.env.eatoms
      and t.init = a1
      and a2.name = a1.name
      and a2.process.anotif = a1.process.anotif + r
      and a2.provided = a1.provided
      and a2.required = a1.required + r
      and a2.process.templates = a1.process.templates
      and a2.process.zconfig.zcomponents = a1.process.zconfig.zcomponents + z
      and a2.process.zconfig.zbindings = a1.process.zconfig.zbindings
      and NewComponentFromTemplate[z, u, r, n]
      and (t.label.signals = none)
      and t.term = a2 + z
}


pred New_Transition [c1, c2: Configuration, u: DL_Machine, t: Transition]{
    some ta: Transition |
      t.init = c1.locations
      and Admin_New_Transition[c1.admin, u, ta]
      and Uncoupling_Transition[t, ta, c1.locations − c1.admin]
      and c2.admin = AdminFromLocationSet[t.term]
      and c2.components = ComponentsFromLocationSet[t.term]
      and c2.bindings = BindingsFromLocationSet[t.term]
      and t.term = c2.locations
}
```

8.3.5 Delete

First, we define the expected global transition for the deletion of a deployed component. Apart from the presence of the administrator, this global transition mirrors the Delete action of the original Aeolus operational semantics: a deployed component is deleted from a configuration, together with all the bindings that bind its interfaces.

```
pred Global_Delete_Component_Transition[c1,c2: Configuration, z: Aeolus_component, t: Transition]
{ some a1, a2: Administrator | let bs = BindingsToComponent[c1.bindings, z] |
      t.init = c1.locations
      and (t.label.signals = none)
      and t.term = c2.locations
      and c1.admin = a1
      and z in c1.components
      and a2.name = a1.name
      and a2.process.anotif = a1.process.anotif − (z.process.cnotif + bs.bnotif)
      and a2.provided = a1.provided
```

```
        and a2.required = a1.required − (z.process.cnotif + bs.bnotif)
        and a2.process.templates = a1.process.templates
        and a2.process.zconfig.zcomponents = a1.process.zconfig.zcomponents − z
        and a2.process.zconfig.zbindings = a1.process.zconfig.zbindings − bs
        and c2.admin = a2
        and c2.components = c1.components − z
        and c2.bindings = c1.bindings − bs
}


fun BindingsToComponent[ bs: set Aeolus_binding, z: Aeolus_component ] : set Aeolus_binding {
  { b : Aeolus_binding | b.prov in z.process.pprov.roles or b.req in z.process.preq.roles }
}
```

We now define the individual transition of an administrator in a configuration to effect a new component creation.

```
pred Admin_Delete_Transition[a1: Administrator, z: Aeolus_component, t:Transition]{
   some a2: Administrator, i: Interaction, is: set Interaction | let bs = BindingsToComponent[a1.process.zconfig.zbindings, z] |
        z in a1.process.zconfig.zcomponents
        and t.init = a1
        and a2.name = a1.name
        and a2.provided = a1.provided
        and a2.required = a1.required − (z.process.cnotif + bs.bnotif)
        and a2.process.templates = a1.process.templates
        and a2.process.anotif = a1.process.anotif − (z.process.cnotif + bs.bnotif)
        and a2.process.zconfig.zcomponents = a1.process.zconfig.zcomponents − z
        and a2.process.zconfig.zbindings = a1.process.zconfig.zbindings − bs
        and t.label.signals = i + is
        and i.irole = z.process.cnotif and i.polarity in Minus and i.payload in DelOp
        and is = { j : Interaction | one b : bs | j.irole = b.bnotif and j.polarity in Plus and j.payload in DelOp }
        and t.term = a2
}

one sig DelOp extends Value{}

pred Component_Delete_Transition[z: Aeolus_component, t: Transition]{
  some i : Interaction |
      t.init = z
      and t.label.signals = i
      and i.irole = z.process.cnotif and i.polarity in Plus and i.payload in DelOp
      and t.term = none
}

pred Binding_Delete_Transition[b: Aeolus_binding, t: Transition]{
    some i: Interaction |
        t.init = b
        and t.label.signals = i
        and i.irole = b.bnotif and i.polarity in Plus and i.payload in DelOp
        and t.term = none
}

pred Delete_Transition [c1, c2: Configuration, z: Aeolus_component, t: Transition]{
     some ta, tz, t1: Transition, tbs : set Transition |
          t.init = c1.locations
          and Admin_Delete_Transition[c1.admin, z, ta]
          and Component_Delete_Transition[z, tz]
          and (all tb : tbs | Binding_Delete_Transition[tb.init, tb] )
          and Synchronizing_Transition[t1, ta + tz + tbs]
          and Uncoupling_Transition[t, t1, c1.locations − (ta.init + tz.init + tbs.init)]
          and c2.admin = AdminFromLocationSet[t.term]
          and c2.components = ComponentsFromLocationSet[t.term]
```

```
        and c2.bindings = BindingsFromLocationSet[t.term]
        and t.term = c2.locations
}
```

8.3.6 Correctness of the Location Graph interpretation of the Aeolus model

We can run a number of checks on this interpretation of the Aeolus model operational semantics. We first verify that our different transition predicates are consistent. Note that we have not been able to verify mechanically the consistency of the predicate Multi_State_Change_Transition with the Alloy model checker because of the state explosion it induces. However, the consistency of the predicate is not in question.

```
/* Consistency of transition predicates */

run ModelGlobalSC {
   some c1,c2: Configuration, z: Aeolus_component, t: Transition | Global_StateChange_Transition[c1,c2, z, t]
} for 10

run ModelSC {
   some c1,c2: Configuration, z: Aeolus_component, t: Transition | State_Change_Transition[c1,c2, z, t]
} for 10

run ModelGlobalBind {
   some c1,c2: Configuration, z1,z2: Aeolus_component, s: Name, t: Transition | Global_Bind_Transition[c1, c2, z1, z2, s, t]
} for 20

run ModelBind {
   some c1,c2: Configuration, z1,z2: Aeolus_component, s: Name, t: Transition | Binding_Transition[c1, c2, z1, z2, s, t]
} for 20

run ModelGlobalUnbind {
  some c1,c2: Configuration, z1,z2: Aeolus_component, s: Name, t: Transition | Global_Unbind_Transition[c1, c2, z1, z2, s, t]
} for 20

run ModelUnbind {
   some c1,c2: Configuration, z1,z2: Aeolus_component, s: Name, t: Transition | Unbinding_Transition[c1, c2, z1, z2, s, t]
} for 25


run ModelGlobalNew {
   some c1, c2: Configuration, u: DL_Machine, t: Transition | Global_New_Component_Transition[c1, c2, u, t]
} for 10

run ModelNew {
   some c1, c2: Configuration, u: DL_Machine, t: Transition | New_Transition[c1, c2, u, t]
} for 10

run ModelGlobalDelete {
   some c1, c2: Configuration, z: Aeolus_component, t : Transition | Global_Delete_Component_Transition[c1, c2, z, t]
} for 10


run ModelDelete {
   some c1, c2: Configuration, z: Aeolus_component, t : Transition | Delete_Transition[c1, c2, z, t]
} for 10

run ModelGlobalMultiSC {
  some c1,c2: Configuration, zs, zps: set Aeolus_component, t: Transition | Global_Multi_StateChange_Transition[c1, c2, zs, zps,t] and #zs > 1
} for 12
```

```
/* Checks on State Change */

assert Global_StateChange_Respects_WF {
  all c1,c2: Configuration, z: Aeolus_component, t: Transition |
    WFConfiguration[c1] and Global_StateChange_Transition[c1,c2, z, t] implies WFConfiguration[c2]
}

check Global_StateChange_Respects_WF for 10 expect 0



assert Global_Multi_StateChange_Respects_WF {
  all c1,c2: Configuration, z,zp: Aeolus_component, t: Transition |
    WFConfiguration[c1] and Global_Multi_StateChange_Transition[c1,c2, z, zp, t] implies WFConfiguration[c2]
}

check Global_Multi_StateChange_Respects_WF for 10 expect 0

assert Global_StateChange_Respects_Template {
  all c1,c2: Configuration, z: Aeolus_component, t: Transition |
    ( WFConfiguration[c1] and Global_StateChange_Transition[c1,c2, z, t] )
    implies
    some zp: Aeolus_component, dt: z.process.template.dtransitions |
              zp.name = z.name and dt.initial = z.process.current and dt.terminal =zp.process.current
}

check Global_StateChange_Respects_Template for 10 expect 0


assert State_Change_Transition_Satisfies_Global_StateChange_Transition {
  all c1,c2:Configuration, z: Aeolus_component, t: Transition |
     (WFConfiguration[c1] and State_Change_Transition[c1,c2, z, t])
        implies
       Global_StateChange_Transition[c1,c2, z, t]
}

check State_Change_Transition_Satisfies_Global_StateChange_Transition for 10 expect 0


assert Multi_State_Change_Transition_Satisfies_Global_Multi_StateChange_Transition {
  all c1,c2:Configuration, z1,z2,zp1,zp2: Aeolus_component, tz1, tz2: Transition, t: Transition |
     (WFConfiguration[c1] and Multi_State_Change_Transition[c1,c2, z1 + z2,zp1 + zp2, tz1 + tz2, t])
        implies
       Global_Multi_StateChange_Transition[c1,c2, z1 + z2, zp1 + zp2, t]
}

 check Multi_State_Change_Transition_Satisfies_Global_Multi_StateChange_Transition for 10 expect 0


/* Checks on Bind */

assert Global_Bind_Respects_WF {
  all c1,c2: Configuration, z1, z2: Aeolus_component, t: Transition, s: Name |
    WFConfiguration[c1] and Global_Bind_Transition[c1,c2, z1, z2, s, t] implies WFConfiguration[c2]
}

check Global_Bind_Respects_WF for 10 expect 0


assert In_Glocal_Bind_Transition_Binding_Not_In_Initial_Configuration {
```

```
  all c1, c2: Configuration, z1, z2: Aeolus_component, s: Name, t: Transition, b: Aeolus_binding |
    Global_Bind_Transition[c1, c2, z1,z2,s,t] and b.bsort = s
    implies
    b not in c1.bindings
}

check In_Glocal_Bind_Transition_Binding_Not_In_Initial_Configuration for 10 expect 0

assert Binding_Transition_Satisfies_Global_Bind_Transition {
  all c1,c2: Configuration, z1,z2: Aeolus_component, s: Name, t: Transition |
      (WFConfiguration[c1] and Binding_Transition[c1,c2, z1, z2, s, t])
      implies Global_Bind_Transition[c1, c2, z1, z2, s, t]
}

check Binding_Transition_Satisfies_Global_Bind_Transition for 7 expect 0


/* Checks on Unbind */

assert Global_Unbind_Respects_WF {
  all c1,c2: Configuration, z1, z2: Aeolus_component, t: Transition, s: Name |
    WFConfiguration[c1] and Global_Unbind_Transition[c1,c2, z1, z2, s, t] implies WFConfiguration[c2]
}

check Global_Unbind_Respects_WF for 10 expect 0


assert Unbinding_Transition_Satisfies_Global_Unbind_Transition {
  all c1, c2: Configuration, z1,z2: Aeolus_component, s: Name, t: Transition |
    ( WFConfiguration[c1] and Unbinding_Transition[c1, c2, z1, z2, s, t] )
    implies Global_Unbind_Transition[c1, c2, z1, z2, s, t]
}

check Unbinding_Transition_Satisfies_Global_Unbind_Transition for 8 expect 0


/* Checks on New */

assert Global_New_Respects_WF {
  all c1,c2: Configuration, u: DL_Machine, t: Transition |
    WFConfiguration[c1] and Global_New_Component_Transition[c1,c2, u, t] implies WFConfiguration[c2]
}

check Global_New_Respects_WF for 10 expect 0

assert New_Transition_Satisfies_Global_New_Component_Transition {
  all c1, c2: Configuration, u: DL_Machine, t: Transition |
      (WFConfiguration[c1] and New_Transition[c1, c2, u, t])
      implies Global_New_Component_Transition[c1, c2, u, t]
}

check New_Transition_Satisfies_Global_New_Component_Transition for 10 expect 0


/* Checks on Delete */

assert Global_Delete_Respects_WF {
  all c1,c2: Configuration, z: Aeolus_component, t: Transition |
    WFConfiguration[c1] and Global_Delete_Component_Transition[c1,c2, z, t] implies WFConfiguration[c2]
}

check Global_Delete_Respects_WF for 10 expect 0
```

```
assert Delete_Transition_Satisfies_Global_Delete_Component_Transition {
  all c1, c2: Configuration, z: Aeolus_component, t : Transition |
    (WFConfiguration[c1] and Delete_Transition[c1, c2, z, t])
    implies Global_Delete_Component_Transition[c1, c2, z, t]
}

check Delete_Transition_Satisfies_Global_Delete_Component_Transition for 10 expect 0
```

Of course these checks cannot be exhaustive. We can do better by proving a strong correspondence property between the Aeolus operational semantics and our Location Graph interpretation. We first define a few auxiliary predicates and a few notations:

```
pred SCTransition[t: Transition] {
  some c1,c2:Configuration, z: Aeolus_component | State_Change_Transition[c1,c2, z, t]
}

pred BindTransition[t: Transition] {
  some c1,c2:Configuration, z1,z2: Aeolus_component, s: Sort | State_Change_Transition[c1, c2, z1, z2, s, t]
}

pred UnbindTransition[t: Transition] {
  some c1,c2:Configuration, z1,z2: Aeolus_component, s: Sort | State_Change_Transition[c1,c2, z1, z2, s, t]
}

pred NewTransition[t: Transition] {
  some c1,c2:Configuration, u: DL_Machine | State_Change_Transition[c1,c2, u, t]
}

pred DelTransition[t: Transition] {
  some c1,c2:Configuration, z: Aeolus_component | State_Change_Transition[c1,c2, z, t]
}

pred AeolusLGTransition [t: Transition] {
   SCTransition[t] or BindTransition[t] or UnbindTransition[t] or NewTransition[t] or DelTransition[t]
}
```

In the following, we make liberal use of relations and predicates defined in the Alloy specifications in this report (from the LocationGraphs, LG_Semantics, Aeolus, Aeolus_actions modules). We also borrow constructions from the Alloy language that we freely mix with semi formal logical notations, as in $\forall x$: `set Location` P, which means for any set of Locations x, statement P holds.

We begin with a first result: the transitions of well-formed Configurations lead to well-formed Configurations.

Lemma 1 *For any* c : `Configuration`*, if* `WFConfiguration`$[c]$ *and* $\Delta \vdash c \xrightarrow{\epsilon} c'$ *then* `WFConfiguration`$[c']$.

Proof. By case analysis for the five different kinds of Configuration transitions. We consider only the case of a transition verifying SCTransition, other cases are handled similarly. By assumption, we have `SCTransition`$[t]$, which means we have transitions t_a, t_z, t_1, Configurations c_1, c_2, and an Aeolus_component z such that

$$\begin{aligned}
&\texttt{Uncoupling_Transition}[t, t_1, ls_1 - (a + z)]\\
&\texttt{Synchronizing_transition}[t_1, t_a + t_z]\\
&\texttt{Comp_State_Change_Transition}[z, tz]\\
&\texttt{Admin_State_Change_Transition}[a, z, t_a]
\end{aligned}$$

where $ls_1 = c_1.\texttt{locations}$ and $a = c_1.\texttt{admin}$.

By definition of Admin_State_Change_Transition, we must have $t_a = \Delta \vdash a \xrightarrow{r:\texttt{Sc0p}(q)}_{\texttt{lg}} a[z \rightarrow z_p]$, where:

- $a[z \to z_p]$ stands for the Administrator which differs from a only by having z replaced by z_p in its zconfig (clause a2.process.zconfig.zcomponents = (a1.process.zconfig.zcomponents − z) + zp in the definition of Admin_State_Change_Transition).
- $z_p = z[p \to q]$, i.e. z_p is the Aeolus_component that differs from z only by having its current state p replaced by q (clause **some** dt: z.process.template.dtransitions | ChangeState[zp, z, dt.terminal] **and** dt.initial = z.process.current ... in the definition of predicate Admin_State_Change_Transition, and clause zp.process.current = q in the definition of predicate ChangeState).
- r : $\texttt{ScOp}(q)$ is the Interaction i such that r is the notification role of component z, and the value $\texttt{ScOp}(q)$ is a state change operation with parameter q (in the definition of Admin_State_Change_Transition, see the clause
  some dt: ... | ... i.irole = z.process.cnotif **and** i.polarity **in** Minus **and** i.payload = v **and** v.scstate = dt.terminal).

Likewise, by definition of Comp_State_Change_Transition, we must have $t_z = \Delta \vdash z \xrightarrow{\overline{r}:\texttt{ScOp}(q)}_{\texttt{lg}} z_p$, and by definition of Synchronizing_Transition, $t_1 = \Delta \vdash a + z \xrightarrow{\epsilon}_{\texttt{lg}} a[z \to z_p] + z_p$. Hence, by defintion of Uncoupling_Transition we must have $t = ls + a + z \xrightarrow{\epsilon}_{\texttt{lg}} ls + a[z \to z_p] + z_p$, where $ls = ls_1 - (a + z)$. Now if $\texttt{WFConfiguration}[c_1]$, then we must have $\texttt{WellFormedLocationSet}[ls + a + z]$, and thus, because none of the location names or location roles have been changed, $\texttt{WellFormedLocationSet}[ls + a[z \to z_p] + z_p]$. Likewise, all the other clauses in the definition of WFConfiguration remain the same, so we can conclude $\texttt{WFConfiguration}[c_2]$, where c_2 is the Configuration such that $c_2.\texttt{locations} = ls + a[z \to z_p] + z_p$. □

We now define a correspondence function $[\![\cdot]\!] : \mathbb{C} \to \texttt{set Configuration}$, which associates with an Aeolus configuration a set of Configurations. Intuitively, Configurations in $[\![\mathcal{C}]\!]$ represent the same Aeolus configuration $\mathcal{C}$, but differ in location names, notification roles and Port roles which have been chosen for Aeolus_components and Aeolus_bindings in these Configurations.

We first define the correspondence of an Aeolus component type with a set of DL_Machines. Let $\mathcal{T} = \langle Q, q_0, T, \mathbf{P}, \mathbf{R}, D\rangle$ be an Aeolus component type. Abusing notation, we define $[\![\mathcal{T}]\!]$ as the set of DL_Machine, ranged by M, verifying:

$$M.\texttt{dstates} = Q$$
$$M.\texttt{dinit} = q_0$$
$$M.\texttt{dtransitions} = \{t \in \texttt{DL_Transition} \mid \exists \langle p, q\rangle \in T,\ t.\texttt{initial} = p \wedge t.\texttt{terminal} = q\}$$
$$M.\texttt{pports} = \mathbf{P}$$
$$M.\texttt{rports} = \mathbf{R}$$
$$M.\texttt{activates} = \{a \in \texttt{Activation} \mid \exists q \in Q,\ a.\texttt{atate} = q \wedge a.\texttt{ap} = [\![\mathcal{T}.\mathbf{P}(q)]\!], a.\texttt{ar} = [\![\mathcal{T}.\mathbf{R}(q)]\!]\}$$

where we set:

$$[\![\mathcal{T}.\mathbf{P}(q)]\!] = \{p \in \texttt{PortCard} \mid \exists \langle z, n\rangle \in \mathcal{T}.\mathbf{P}(q),\ p.\texttt{asort} = z \wedge p.\texttt{acard} = [\![n]\!]\}$$
$$[\![\mathcal{T}.\mathbf{P}(q)]\!] = \{p \in \texttt{PortCard} \mid \exists \langle z, n\rangle \in \mathcal{T}.\mathbf{R}(q),\ p.\texttt{asort} = z \wedge p.\texttt{acard} = [\![n]\!]\}$$
$$[\![n]\!] = \texttt{if } n \in \mathbb{N} \texttt{ then } n \texttt{ else } 0$$
$$\texttt{DL_State} = \mathcal{Q}$$

Let $\mathcal{T}$ be an Aelus component type, and q be some state. We define

$$[\![\langle \mathcal{T}, q\rangle]\!] = \{M \in \texttt{Aeolus_component} \mid M.\texttt{process.templates} \in [\![\mathcal{T}]\!] \wedge M.\texttt{process.current} = q\}$$

Let $\langle r, z_1, z_2\rangle$ be an Aeolus binding. We define

$$[\![\langle r, z_1, z_2\rangle]\!] = \{b \in \texttt{Aeolus_binding} \mid b.\texttt{bsort} = r\}$$

Let $\mathcal{C} = \langle U, Z, S, B\rangle$ be an Aeolus configuration. We define $[\![\mathcal{C}]\!]$ as the set of Configurations, ranged by C, verifying:

$$
\begin{array}{l}
C.\texttt{admin.templates} = [\![U]\!] \\
|C.\texttt{components}| = |Z| \\
\forall z \in Z,\ \exists c \in C.\texttt{components},\ c \in [\![S(z)]\!] \wedge c.\texttt{name} = z \\
|C.\texttt{bindings}| = |B| \\
\forall b \in B,\ \exists c \in C.\texttt{bindings},\ c \in [\![b]\!] \\
\texttt{WFConfiguration}[C]
\end{array}
$$

Note that in the above we have taken $\mathcal{Z}$ to be such that $\mathcal{Z} \subset \texttt{Name}$. Given $C \in [\![\mathcal{C}]\!]$, we write $[\![z]\!]$ for the unique $c \in C.\texttt{components}$ such that $c.\texttt{name} = z$. The following result is an easy consequence of our definitions:

Lemma 2 *For any $M \in$ DL_Machine, there exists a unique Aeolus component type $\mathcal{T}$ such that $M = [\![\mathcal{T}]\!]$. We write $[\![M]\!]^{-1}$ to denote $\mathcal{T}$.*

For any $C \in$ Configuration such that $\texttt{WFConfiguration}[C]$, there exists a unique Aeolus configuration $\mathcal{C}$ such that $C = [\![\mathcal{C}]\!]$. We write $[\![C]\!]^{-1}$ to denote $\mathcal{C}$.

We write $\Delta \vdash c \xrightarrow{\Lambda} c'$ for a location graph transition t such that $t.\texttt{env} = \Delta$ $t.\texttt{init} = c$, $t.\texttt{term} = c'$ and $t.\texttt{label} = \Lambda$. We write ϵ for a label Λ such that $\Lambda.\texttt{signals} = \emptyset$. We define the labelled transition relation $\rightarrow_{\texttt{lg}} \subseteq$ set Location $\times A \times$ set Location, where A is the set of Aeolus actions defined in Section 8.1, as follows:

- $C \xrightarrow{\{\texttt{stateChange}(z_j, q_j^1, q_j^2) \mid j \in J\}}_{\texttt{lg}} C'$ if and only if there exists a location graph transition $t = \Delta \vdash C \xrightarrow{\epsilon} C'$ such that $\texttt{Global_Multi_StateChange_Transition}(C_1, C_2, cs, t)$, $C = C_1.\texttt{locations}$, $C' = C_2\texttt{locations}$, for some set $cs = \{c_j : C \mid j \in j \in J\}$ of Aeolus_component in C, with $\{z_j \mid j \in J\} = cs.\texttt{name}$, $\{q_j^1 \mid j \in J\} = cs.\texttt{process.current}$, and $\{q_j^2 \mid j \in J\} = \{c'_j \in C' \mid c'_j.\texttt{name} = z_j \wedge j \in J\}.\texttt{process.current}$.

- $C \xrightarrow{\texttt{bind}(r, z_1, z_2)}_{\texttt{lg}} C'$, if and only if there exists a location graph transition $t = \Delta \vdash C \xrightarrow{\epsilon} C'$ such that $\texttt{Binding_Transition}(C_1, C_2, c_1, c_2, r, t)$, where $C_1.\texttt{locations} = C$, $C_2.\texttt{locations} = C'$, $c_1, c_2 \in C$, $c_1.\texttt{name} = z_1$, $c_2.\texttt{name} = z_2$, $c_1.\texttt{process.pprov.psort} = c_2.\texttt{process.preq.psort} = r$.

- $C \xrightarrow{\texttt{unbind}(r, z_1, z_2)}_{\texttt{lg}} C'$, if and only if there exists a location graph transition $t = \Delta \vdash C \xrightarrow{\epsilon} C'$ such that $\texttt{Unbinding_Transition}(C_1, C_2, c_1, c_2, r, t)$ where $C_1.\texttt{locations} = C$, $C_2.\texttt{locations} = C'$, $c_1, c_2 \in C$, $c_1.\texttt{name} = z_1$, $c_2.\texttt{name} = z_2$, $c_1.\texttt{process.pprov.psort} = c_2.\texttt{process.preq.psort} = r$.

- $C \xrightarrow{\texttt{new}(z:\mathcal{T})}_{\texttt{lg}} C'$, if and only if there exists a location graph transition $t = \Delta \vdash C \xrightarrow{\epsilon} C'$ such that $\texttt{New_Transition}(C_1, C_2, u, t)$ where $C_1.\texttt{locations} = C$, $C_2.\texttt{locations} = C'$, $u \in [\![\mathcal{T}]\!]$, and $z = C'.\texttt{name} - C.\texttt{name}$.

- $C \xrightarrow{\texttt{del}(z)}_{\texttt{lg}} C'$, if and only if there exists a location graph transition $t = \Delta \vdash C \xrightarrow{\epsilon} C'$ such that $\texttt{Delete_Transition}(C_1, C_2, c, t)$, where $C_1.\texttt{locations} = C$, $C_2.\texttt{locations} = C'$, and $c.\texttt{name} = z$.

We can now state the correspondence result as follows:

Theorem 1 *The relation* $\mathcal{S} = \{\langle \mathcal{C}, C\rangle \mid \mathcal{C} \in \mathbb{C} \wedge C \in [\![\mathcal{C}]\!]\}$ *is a strong bisimulation between the transition systems* $\langle \mathbb{C}, \rightarrow\rangle$ *and* $\langle \texttt{Configuration}, \rightarrow_{\texttt{lg}}\rangle$.

Proof. We first prove that $\mathcal{S}$ is a a strong simulation, namely that for any $\langle \mathcal{C}, C\rangle \in \mathcal{S}, \mathcal{C}' \in \mathbb{C}$, we have

$$\mathcal{C} \xrightarrow{\alpha} \mathcal{C}' \implies \exists C' \in \texttt{Configuration},\ C \xrightarrow{\alpha}_{\texttt{lg}} C' \wedge \langle \mathcal{C}', C'\rangle \in \mathcal{S}$$

This is done by a case analysis depending on the form of action α obtained to derive $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$. We consider only the case $\alpha = \texttt{bind}(s, z_1, z_2)$, the other cases are handled similarly. Assume $\mathcal{C} \xrightarrow{\texttt{bind}(s,z_1,z_2)} \mathcal{C}'$, and let $C \in [\![\mathcal{C}]\!]$. We show that there exist $C' \in [\![\mathcal{C}']\!]$ and a transition t such that $\texttt{Binding_Transition}[C, C', [\![z_1]\!], [\![z_2]\!], s, t]$. Because $\texttt{WFConfiguration}[C]$ there exists a transition t_a such that $\texttt{Admin_Binding_Transition}[a, [\![z_1]\!], [\![z_2]\!], s, t_a]$, where $a = C.\texttt{admin}$. Indeed, all the precondition clauses in predicate `Admin_Binding_Transition` are met:

- $[\![z1]\!]+[\![z2]\!] \in a1.\texttt{process.zconfig.zcomponents}$ is met because $\texttt{ZconfigReflectsConfig}[C]$.
- $s \in [\![z_1]\!].\texttt{process.pprov.psort} \cap [\![z_2]\!].\texttt{process.preq.psort}$ is met because of the premises in rule Bind.
- $\texttt{UnboundZ}[a.\texttt{process.zconfig}, [\![z_1]\!], [\![z_2]\!], s]$ is met because $\texttt{ZconfigReflectsConfig}[C]$ and of the premises in rule BIND.

We thus have $t_a = \Delta \vdash a \xrightarrow{u_1:\texttt{BdpOp}(r_1), u_2:\texttt{BdrOp}(r_2)} a' + b$, where $b \in \texttt{Aeolus_binding}$, $b.\texttt{bsort} = s$, $b.\texttt{bnotif} = r$, $b.\texttt{prov} = r_1$, $b.\texttt{breq} = r_2$, for some $r, r_1, r_2 \notin \Delta$, $u_1 = [\![z_1]\!].\texttt{process.cnotif}$, $u_2 = [\![z_2]\!].\texttt{process.cnotif}$, and a' is the same as a with role r added to $a.\texttt{process.anotif}$. By definition, we also have transitions $t_{z_1} = [\![z_1]\!] \xrightarrow{\overline{u_1}:\texttt{BdpOp}(r_1)} z_1'$, where z_1' is the same as $[\![z_1]\!]$ with role r_1 added to its provided port with sort s, and $t_{z_2} = [\![z_2]\!] \xrightarrow{\overline{u_2}:\texttt{BdrOp}(r_2)} z_2'$, where z_2' is the same as $[\![z_2]\!]$ with role r_2 added to its provided port with sort s. Thus there exists a transition t_1 such that $\texttt{Synchronizing_Transition}[t_1, t_a{+}t_{z_1}{+}t_{z_2}]$, and there exists a transition t such that $\texttt{Binding_Transition}[C, C', [\![z_1]\!], [\![z_2]\!], s, t]$ where C' is the same as C with the following changes:

- b is added to $C.\texttt{process.zconfig.zbindings}$.
- $[\![z_1]\!]$ is changed into $z'1$, $[\![z_2]\!]$ is changed into z_2', and a is changed into a'.

Thanks to Lemma 1, we have $\texttt{WFConfiguration}[C']$. All the other clauses in the definition of $[\![\mathcal{C}']\!]$ are clearly met, and thus we can conclude that $C' \in [\![\mathcal{C}']\!]$, as required.

We then prove that $\mathcal{S}^{-1}$ is a strong simulation, namely that for any $\langle \mathcal{C}, C\rangle \in \mathcal{S}, C' \in \texttt{Configuration}$, we have

$$C \xrightarrow{\alpha}_{\texttt{lg}} C' \implies \exists \mathcal{C}' \in \mathbb{C},\ \mathcal{C} \xrightarrow{\alpha} \mathcal{C}' \wedge \langle \mathcal{C}', C'\rangle \in \mathcal{S}$$

This is done by case analysis on the derivation of $t = C \xrightarrow{\alpha}_{\texttt{lg}} C'$. We consider only the case $\alpha = \texttt{bind}(s, z_1, z_2)$, the other cases are handled similarly. Thus, we must have $\texttt{Binding_Transition}[C, C', c_1, c_2, s, t]$ with $c_1.\texttt{name} = z_1$ and $c_2.\texttt{name} = z_2$. The clause $s \in [\![c_1]\!].\texttt{process.pprov.psort} \cap [\![c_2]\!].\texttt{process.preq.psort}$ and the clause $\texttt{UnboundZ}[a.\texttt{process.zconfig}, c_1, c_2, s]$ in the definition of predicate Admin_Binding_Transition ensure the premises in rule BIND are met. Furthermore, we are guaranteed that z_1 and z_2 are in $\mathcal{C}$ by the clause

$$c_1 + c_2 \in C.\texttt{admin.process.zconfig.zcomponents}$$

in the definition of predicate Admin_Binding_Transition, and the fact that $\texttt{ZconfigReflectsConfig}[C]$. We can thus apply rule BIND to obtain: $\mathcal{C} \xrightarrow{\texttt{bind}(s,z_1,z_2)} \mathcal{C}'$. Now, it is easy to see that $C' \in [\![\mathcal{C}']\!]$, as required. □

8.4 Extending the TOSCA model with Aeolus concepts

In the previous Sections, we showed how the TOSCA model and the Aeolus model can be interpreted in terms of Location Graphs. In this subsection, we show how this allows us to extend TOSCA in a straightforward way with Aeolus concepts. The idea is to introduce a hybrid Aeolus_Tosca_Configuration signature which reflects the encoding of the structure of an Aeolus configuration as well as the structure of a Tosca Topology Template.

One needs however to decide how the TOSCA and Aeolus models are to be considered in relation with one another. The Aeolus model centers around concepts dealing with component deployment operations, which are not explicit in the TOSCA standard. We have opted for an hybridization of TOSCA with Aeolus concepts which minimizes the interference between the two models and the duplication of ceoncepts in our specification. More precisely, a TOSCA topology is interpreted as an abstract description of a target configuration to deploy, TOSCA nodes extended with Aeolus component types (DL_Machines in our interpretation) that specify quantitative and lifecycle constraints for the deployment. A

Thus, an hybrid Tosca_Aeolus_Configuration is an instance of an Aeolus configuration, where the configuration to be deployed is defined by a set of Aeolus component types and a TOSCA topology. The TOSCA topology to deploy is specified as an additional attribute of the administrator of the configuration (an instance of Tosca_Administrator). All the nodes in the TOSCA topology in a Tosca_Administrator) must be nodes extended with a DL_Machine. A deployed node in a hybrid TOSCA/Aeolus configuration is an Aeolus_component whose ports comprise an additional pid attribute that refers to the corresponding node ports in the TOSCA topology (recall that a capability or a requirement of a TOSCA node is interpreted as a role in the interpretation of the TOSCA model in Location Graphs).

```
module TOSCA_Aeolus
open LocationGraphs
open TOSCA
open Aeolus

sig Tosca_Aeolus_Configuration extends Configuration{
}{ admin in Tosca_Administrator }

sig Tosca_Port extends Port {
  pid: one Role
}

sig Node_DL extends Node {
  template: one DL_Machine
}

sig Tosca_Administrator extends Administrator{
   toscatopology: one TopologyTemplate
}{ toscatopology.nodes in Node_DL }
```

A target hybrid TOSCA/Aeolus configuration is thus an instance of Tosca_Aeolus_Configuration that satisfies the TopologyToTargetConfigurationMapped predicate, namely: it is a well-formed target Aeolus Configuration; any node appearing in the TOSCA topology maintained by the configuration administrator must give rise to one or more deployed Aeolus_components; conversely, any deployed Aeolus_component must correspond to some node in the configuration administrator topology ; any relationship in the TOSCA topology maintained by the configuration administrator must give rise to one or more Aeolus_bindings; conversely, any deployed Aeolus_binding must correspond to some relationship in the configuration administrator topology. Note that this allows: different Aeolus component types not mentioned in the TOSCA topology playing a role for the actual deployment of a hybrid TOSCA/Aeolus target configuration; the deployment of TOSCA topologies with Aeolus-like cardinality constraints.

```
pred PortsAreToscaPorts [c: Configuration] {
```

```
  c.components.process.(pprov + preq) in Tosca_Port
}

pred NodeToComponentTypeMapped[c: Tosca_Aeolus_Configuration] {
 ( all n : c.admin.toscatopology.nodes | one z : c.components | NodeToComponent[n,z] )
 and ( all z: c.components | one n: c.admin.toscatopology.nodes | NodeToComponent[n,z] )
}

pred NodeToComponent[n: Node_DL, z: Aeolus_component] {
  (n.provided = z.process.pprov.pid)
  and (n.required =z.process.preq.pid)
  and (n.template = z.process.template)
}

pred RelationshipToBindingMapped[c: Tosca_Aeolus_Configuration]{
  all r : c.admin.toscatopology.relationships
  { some b : c.bindings | b.bsort = r.name }

  all b :c.bindings
  { some r : c.admin.toscatopology.relationships | b.bsort = r.name }

}

pred TopologyToTargetConfigurationMapped [ c: Tosca_Aeolus_Configuration ] {
    TargetConfiguration[c] and RelationshipToBindingMapped[c] and NodeToComponentTypeMapped[c]
}
```

We can check that our hybridization of the TOSCA model with Aeolus concepts is consistent by looking for the existence of some small hybrid target configuration. To simplify the analysis, we add some facts before executing the run command to find a model.

```
/* Facts to minimize the size of generated models */

fact All_nodes_have_DL {
  Node in Node_DL
}

fact restrict_LG_types{
    LocationGraph in Configuration+ZConfiguration+TopologyTemplate+Null
}

fact all_configurations_are_aeolus_tosca_configurations{
  Configuration=Tosca_Aeolus_Configuration
}

fact all_acomponent_in_some_configuration{
   all ac: Aeolus_component| some c:Tosca_Aeolus_Configuration | ac in c.admin.process.zconfig.zcomponents}

fact template_belongs_to_acomponent_or_NodeLF{
   DL_Machine in Aeolus_component.process.template + Node_DL.template }

fact no_capability_nor_requirement_is_notif_role{
        no Capability & Aeolus_component.process.cnotif
   and no Requirement & Aeolus_component.process.cnotif }

fact all_topology_templates_belong_to_config_admin{
   TopologyTemplate in Tosca_Administrator.toscatopology
}

/* Checking there exists some hybrid TOSCA/Aeolus configuration */
```

```
run Model{
  "(anonymous)" in String
  some c: Configuration| #c.bindings>0 and #c.components>1
                         and some Admin_process.zconfig.zcomponents.process.pprov.roles
                         and some Admin_process.zconfig.zcomponents.process.preq.roles
                         and TopologyToTargetConfigurationMapped[c]
} for 20 but exactly 1 Tosca_Aeolus_Configuration, exactly 3 Aeolus_component , exactly 2 Aeolus_binding
```

9 Conclusion

We have presented in this report the first elements of a computational model for cloudnet systems, based on a subset of the hypercell / location graph framework. We have illustrated how it can play the role of a pivot model for configuration management tools and languages by encoding several models that have been and are being used for configuration management and orchestration in cluster and cloud computing systems, namely the Fractal component model, the OCCI model, the TOSCA model, the Docker Compose model, the OpenStack Heat Orchestration Template (HOT) model, and the Aeolus model. We have illustrated, by defining a structure-preserving mapping between the TOSCA and OCCI core models, how to make use of this pivot model to formally relate different models for configuration management. We have shown with the OpenStack HOT model that the formalization yields immediate benefits in terms of several classes of errors which have been remedied by introducing new invariants in the HOT specification, and we have implemented a verification tool based on the Alloy analyzer that detects errors in HOT configuration descriptions. We have shown with the Aeolus mapping that it was possible to capture not only structural aspects of configuration description languages but also operational semantics aspects related to deployment plans and processes. We have also shown how this can be exploited to extend one model (TOSCA) with features of another (Aeolus).

Acknowledgements

INRIA authors of this report were supported in part by funding from Orange Labs Research and by the european Celtic-Plus project SENDATE.

References

[1] Alloy Analyzer Web Site. Accessible at: `http://alloy.mit.edu/`.

[2] Amazon API. `https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html`.

[3] Ansible. `https://www.ansible.com/overview/how-ansible-works`.

[4] AWS CloudFormation. `https://docs.aws.amazon.com/cloudformation/?id=docs_gateway`.

[5] Chef and Puppet. `https://www.chef.io/puppet/`.

[6] CloudStack. `https://cloudstack.apache.org/`.

[7] Docker. `https://www.docker.com/`.

[8] Docker Compose Source Code. `https://github.com/docker/compose`.

[9] HOT Template Structure Specification. `https://docs.openstack.org/heat/stein/template_guide/hot_spec.html`.

[10] HOT Types Specification. `https://docs.openstack.org/heat/stein/template_guide/openstack.html`.

[11] ISO/IEC 19831:2015 - Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol. Accessible at: `https://www.iso.org/standard/66296.html`.

[12] Juju. https://docs.jujucharms.com/.

[13] Microsoft Azure. https://docs.microsoft.com/en-gb/azure/.

[14] OpenStack. https://www.openstack.org/.

[15] OpenStack Heat. https://wiki.openstack.org/wiki/Heat.

[16] Overview of Docker Compose. https://docs.docker.com/compose/overview/.

[17] Python ipaddress Library. https://docs.python.org/3/library/ipaddress.html#module-ipaddress.

[18] vvp Validation Tool. https://wiki.onap.org/pages/viewpage.action?pageId=3247218.

[19] OASIS Standard – Topology and Orchestration Specification for Cloud Applications – Version 1.0. Accessible at: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf, 2013.

[20] Oasis open – tosca test assertions – normative types. Accessible at: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf, 2015.

[21] TOSCA Simple Profile in YAML Version 1.0. Accessible at: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csprd01/TOSCA-Simple-Profile-YAML-v1.0-csprd01.pdf, 2015.

[22] M. Ahmed-Nacer, S. Tata, W. Gaaloul, P. Merle, J. Parpaillon, N. Plouzeau, and S. Challita. OCCI Behavioural Model. Technical Report D2.2.2, OCCIware Project, 2016.

[23] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 3rd edition, 2013.

[24] J. Bellendorf and Z.A. Mann. Cloud topology and orchestration using TOSCA: A systematic literature review. In *Service-Oriented and Cloud Computing - 7th IFIP WG 2.14 European Conference, ESOCC 2018*, volume 11116 of *Lecture Notes in Computer Science*. Springer, 2018.

[25] S. Bliudze and J. Sifakis. A notion of glue expressiveness for component-based systems. In *CONCUR*, volume 5201 of *LNCS*. Springer, 2008.

[26] A. Brogi, J. Soldani, and P. Wang. TOSCA in a nutshell: Promises and perspectives. In *3rd European Conference on Service-Oriented and Cloud Computing, ESOCC 2014*, volume 8745 of *Lecture Notes in Computer Science*. Springer, 2014.

[27] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.B. Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12), 2006.

[28] M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: the calculus of boxed ambients. *ACM. Trans. Prog. Languages and Systems*, 26(1), 2004.

[29] L. Cardelli and A. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1), 2000.

[30] R. Di Cosmo, A. Eiche, J. Mauro, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Automatic deployment of services in the cloud with aeolus blender. In *13th International Conference Service-Oriented Computing (ICSOC)*, volume 9435 of *Lecture Notes in Computer Science*. Springer, 2015.

[31] R. Di Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro. Aeolus: A component model for the cloud. *Information and Computation*, 239, 2014.

[32] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron. A classification framework for software component models. *IEEE Trans. Software Eng.*, 37(5), 2011.

[33] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: a deployment management system. In *ACM SIGPLAN Notices*, volume 47, pages 263–274. ACM, 2012.

[34] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural Description of Component-Based Systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[35] G. Goessler and J. Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3), 2005.

[36] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, Andrew A. Farrell, A. Lain, P. Murray, and P. Toft. The smartfrog configuration management framework. *SIGOPS Oper. Syst. Rev.*, 43(1), 2009.

[37] M. Hennessy, J. Rathke, and N. Yoshida. Safedpi: a language for controlling mobile code. *Acta Inf.*, 42(4-5), 2005.

[38] J. A. Hewson and P. Anderson. Modelling system administration problems with csps. *Constraint Modelling and Reformulation (ModRef'11)*, 2011.

[39] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition, 2012.

[40] Daniel Jackson. Alloy: a language and tool for exploring software designs. *Commun. ACM*, 62(9), 2019.

[41] S. S. T. Q. Jongmans and F. Arbab. Overview of thirty semantic formalisms for Reo. *Sci. Ann. Comp. Sci.*, 22(1), 2012.

[42] P. Merle, O. Barais, J. Parpaillon, N. Plouzeau, and S. Tata. A precise metamodel for open cloud computing interface. In *8th IEEE International Conference on Cloud Computing, CLOUD 2015*. IEEE, 2015.

[43] P. Merle and J.B. Stefani. A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA, France, 2008.

[44] T. Metsch, A. Edmonds, and B. Parak. Open Grid Forum – Open Cloud Computing Interface – Infrastructure. Accessible at: `https://www.ogf.org/documents/GFD.224.pdf`, 2016.

[45] T. Metsch and M. Mohamed. Open Grid Forum – Open Cloud Computing Interface – Platform. Accessible at: `https://www.ogf.org/documents/GFD.227.pdf`, 2016.

[46] M. M. Moscato, C. G. Lopez Pombo, and M. F. Frias. Dynamite: A tool for the verification of alloy models based on pvs. *ACM Trans. on Software Engineering and Methodology*, 23(2), 2014.

[47] R. Nyren, A. Edmonds, A. Papspyrou, T. Metsch, and B. Parak. Open Grid Forum – Open Cloud Computing Interface – Core. Accessible at: `https://www.ogf.org/documents/GFD.221.pdf`, 2016.

[48] A. Schmitt and J.B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*. Springer, 2005.

[49] S. Sicard, F. Boyer, and N. De Palma. Using Components for Architecture-Based Management: The Self-Repair Case. In *30th International Conference on Software Engineering (ICSE 2008)*. ACM, 2008.

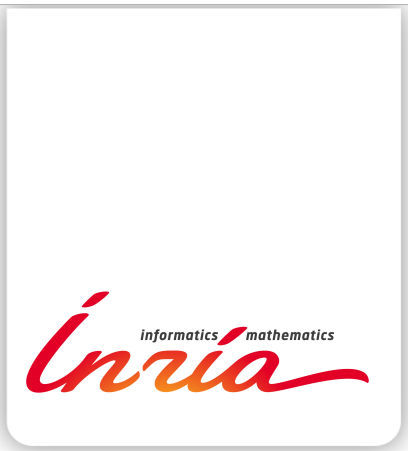
[50] A. Souri, N. J. Navimipour, and A. M. Rahmani. Formal verification approaches and standards in the cloud computing: A comprehensive and systematic review. *Computer Standards & Interfaces*, 58, 2018.

[51] J.B. Stefani. Components as location graphs. In *11th Int. Symposium Formal Aspects of Component Software FACS 2014, Revised Selected Papers*, volume 8997 of *Lecture Notes in Computer Science*. Springer, 2015.

[52] J.B. Stefani and M. Vassor. Encapsulation and sharing in dynamic software architectures: The Hypercell framework. In *39th Int. Conf. Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, Lecture Notes in Computer Science. Springer, 2019.

[53] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, and J. Soldani. The essential deployment metamodel: A systematic review of deployment automation technologies. *Software Intensive Cyber-Physical Systems*, to appear, 2019.

[54] Karn Yongsiriwit, Mohamed Sellami, and Walid Gaaloul. A semantic framework supporting cloud resource descriptions interoperability. In *9th IEEE International Conference on Cloud Computing, CLOUD 2016*, IEEE Computer Society, 2016.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399