A Formal and Tooled Framework for Managing Everything as a Service

www.occiware.org

Deliverable D2.2.2

OCCI Behavioural Model

OCCIware project is supported by Fonds national pour la Société Numérique (FSN) and the following competitive clusters Systematic, Minalogic, PICOM, Images & Réseaux and Solutions Communicantes Sécurisées.
Abstract
Open Cloud Computing Interface (OCCI) is an Open Grid Forum (OGF) community-based effort to create one of the first open extensible standards for managing any kind of cloud computing resources. The kernel of OCCI is a generic resource-oriented model called OCCI Core Model. The specification of this model as well as its extensions are written in English prose and lacks of precise semantics and are provided in an informal. This informal definition can lead to various different interpretations and interoperability issues between OCCI implementations.
In this deliverable presents a comprehensive formal specification of OCCI behavior, that covers all the elements of the original informal reference specifications.
### Document versioning

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Initial version</td>
<td>Mehdi Ahmed-Nacer</td>
</tr>
<tr>
<td>0.2</td>
<td>Updated document template</td>
<td>Mehdi Ahmed-Nacer</td>
</tr>
<tr>
<td>0.3</td>
<td>State of the Art</td>
<td>Mehdi Ahmed-Nacer</td>
</tr>
<tr>
<td>0.4</td>
<td>Alloy</td>
<td>Stéphanie Challita</td>
</tr>
<tr>
<td>1.0</td>
<td>Final version</td>
<td>Mehdi Ahmed-Nacer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stéphanie Challita</td>
</tr>
</tbody>
</table>

### Document reviews

<table>
<thead>
<tr>
<th>Review</th>
<th>Date</th>
<th>Ver.</th>
<th>Reviewer(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan</td>
<td>December 14, 2016</td>
<td>0.1</td>
<td>Philippe Merle and Walid Gaaloul</td>
<td>the deliverable structure</td>
</tr>
<tr>
<td>draft</td>
<td>June 21</td>
<td>0.4</td>
<td>Philippe Merle and Walid Gaaloul</td>
<td></td>
</tr>
<tr>
<td>Validation</td>
<td>November 15</td>
<td>0.4</td>
<td>Philippe Merle, Walid Gaaloul and Christophe Dorothe</td>
<td>abstract</td>
</tr>
<tr>
<td>BAT</td>
<td>November 30</td>
<td>1.0</td>
<td>Mehdi Ahmed-Nacer and Stéphanie Challita</td>
<td></td>
</tr>
</tbody>
</table>
# Contents

1. **Introduction**  
   2. **State of the Art and Motivations**  
      2.1 State machines  
      2.1.1 Basic concepts  
      2.1.2 Example  
      2.1.3 Hierarchical state machine  
      2.1.4 State machine of REST resources  
      2.2 Behavior description in OCCI specifications  
      2.2.1 Behavior description in OCCI Infrastructure  
      2.2.2 Behavior description in OCCI Platform  
      2.2.3 OCCI behavior description in autonomic computing  
      2.3 Analysis and Motivations  
3. **OCCI Behavioral Model**  
   3.1 Introduction & General Motivation  
   3.2 Related Work  
   3.3 Alloy  
      3.3.1 Motivation  
      3.3.2 Preliminary concepts  
   3.4 OCCI Foundations  
   3.5 CREATE  
      3.5.1 Create Resource  
      3.5.2 Create Link  
      3.5.3 Add UserMixin  
   3.6 RETRIEVE  
      3.6.1 Retrieve Resource  
      3.6.2 Retrieve Link  
      3.6.3 Retrieve Collection  
      3.6.4 Get All Categories  
   3.7 UPDATE  
      3.7.1 Update Resource  
      3.7.2 Update Link  
      3.7.3 Associate Mixin To Resource  
      3.7.4 Dissociate Mixin From Resource  
   3.8 DELETE  
      3.8.1 Delete Resource  
      3.8.2 Delete Link  
      3.8.3 Remove UserMixin
<table>
<thead>
<tr>
<th>Deliverable</th>
<th>D2.2.2</th>
<th>Status</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected</td>
<td>2016/11/30</td>
<td>Dissemination</td>
<td>public</td>
</tr>
<tr>
<td>Delivered</td>
<td>December 14, 2016</td>
<td>Version</td>
<td>1.0</td>
</tr>
</tbody>
</table>

3.9 EXECUTE ACTION ........................................... 30

3.10 Behavior properties definition and verification ........................................... 30

3.10.1 Consistency ........................................... 30

3.10.2 Desired properties ........................................... 32

3.10.3 Imposed properties ........................................... 33

3.11 Discussion ........................................... 34

3.11.1 Differences with the informal specification ........................................... 34

3.11.2 Using Alloy ........................................... 35

4 Conclusion and Perspectives ........................................... 37

A OCCI Specification in Alloy ........................................... 40
List of Figures

2.1 State Diagram Example. ................................................................. 7
2.2 Hierarchical State Diagram Example........................................... 7
2.3 The flow of processing Webmachine............................................ 8
2.4 UML State Diagram................................................................. 9
2.5 State Diagram for a Storage Instance........................................... 10
2.6 State Diagram for a NetworkInterface and StorageLink Instances... 10
2.7 UML State Diagram................................................................. 11
2.8 State Diagram of the Resources................................................... 12
2.9 State Diagram of the Links........................................................ 13

3.1 Alloy Model Finder................................................................. 16
3.2 OCCI Behavioral Metamodel...................................................... 18
3.3 OCCI HTTP Operations (Source: [1])........................................... 24
3.4 Example of Consistent Operation............................................... 31
Chapter 1

Introduction

Open Cloud Computing Interface (OCCI) is an Open Grid Forum (OGF) community-based effort to create one of the first open extensible standards for managing any kind of cloud computing resources [2]. The kernel of OCCI is a generic resource-oriented model called the OCCI Core Model and defined in [3]. The OCCI Core Model can be interacted with using renderings (including associated behaviours) and expanded through extensions.

Nevertheless, OCCI lacks a precise definition of its core concepts. Indeed, OCCI specifications are informal documents written in natural language and illustrated by UML diagrams. This informal definition of the OCCI Core Model can be interpreted in various different ways, which can lead to interoperability issues between OCCI implementations. Moreover, the OCCI Core Model has conceptual drawbacks and limitations: an imprecise semantics of its built-in type classification system, a nonextensible data type system for OCCI attributes, a vague and limited extension concept, and the absence of a configuration concept.

To tackle these issues, we have defined in a previous deliverable [4] a precise metamodel for OCCI. This metamodel defines rigorously the static semantics of the OCCI core concepts. Our metamodel proposes a precise type classification system, an extensible data type system, and both extension and configuration concepts.

Our objective in this deliverable is to go further in talking the lacks of OCCI by the definition of a formal model for the dynamic semantics of the OCCI core concepts. Indeed, when we visited specifications of OCCI core model as well as its extensions, we found out that sometimes the definition of OCCI kinds and mixins are provided in an informal way. This informal definition can lead to various different interpretations and interoperability issues between OCCI implementations.

This deliverable is organized as follows. First, Chapter 2 presents some basic concepts related to formal definition of behavioral semantics, the state of the art and motivations. Then, in Chapter 3 we present our formal model for the dynamic semantics of the OCCI core concepts. Finally, in Chapter 4 we conclude and provide perspectives that link this deliverable to the previous and future deliverables of the OCCIware project.
Chapter 2

State of the Art and Motivations

2.1 State machines

2.1.1 Basic concepts

State machines are used to describe system behaviors. They describe all possible states of systems as events occur. Their main elements are states and transitions. States store or represent the status of the system while transitions represent relations between states. State machine is modeled as a traversal of a graph of state nodes connected with transitions. Transitions are triggered by dispatching series of events. During the traversal, the state machine could also execute some activities.

States model a situation during which some invariant condition of the modeled system holds. The main properties of a state are:

- Name: The name of state,
- Entry: An optional behavior that is executed whenever this state is entered regardless of the transition taken to reach the state,
- Exit: An optional behavior that is executed whenever this state is exited regardless of which transition was taken out of the state,
- Do activity: An optional behavior that is executed while being in the state.

Transitions are the result of the invocation of an action that causes an important change in system state. The main properties of transition are:

- Event: an occurrence that may trigger a transition. Event types include an explicit signal from outside the system, an invocation from inside the system, the passage of a designated period of time, or a designated condition becoming true,
- Conditions: the conditions that must be fulfilled before the transition will occur. The conditions affect the behavior of a state machine by enabling actions or transitions only when they evaluate to true and disabling them when they evaluate to false.
- Action: when an event instance is dispatched, the state machine responds by performing actions.
2.1.2 Example

As an example, in Figure 2.1, the state machine diagram shows the states that a door goes through during its lifetime. The door can be in one of three states: "Opened", "Closed" or "Locked". It can respond to the events Open, Close, Lock and Unlock. All events are valid in all states. For example, if a door is opened, you cannot lock it until you close it. Also a state transition can have a guard condition attached: if the door is Opened, it can only respond to the Close event if the condition doorWay->isEmpty is fulfilled.

![State Diagram Example](image)

Figure 2.1 – State Diagram Example.

2.1.3 Hierarchical state machine

Hierarchical state machine design captures the commonality by organizing the states as a hierarchy. The states at the higher level in hierarchy perform the common message handling, while the lower level states inherit the commonality from higher level ones and perform the state specific functions. Figure 2.2 presents a hierarchical state machine example.

![Hierarchical State Diagram Example](image)

Figure 2.2 – Hierarchical State Diagram Example.

2.1.4 State machine of REST resources

A Webmachine application [5] is a set of resources, each one of them is a set of functions over the state of the resource. Unlike web frameworks used, Webmachine application is a REST toolkit. Alan Dean and Justin Sheehy illustrated by a state machine diagram the flow of processing that a Webmachine resource goes through from inception to response. This diagram is illustrated in 2.3.
Figure 2.3 – The flow of processing Webmachine.
2.2 Behavior description in OCCI specifications

2.2.1 Behavior description in OCCI Infrastructure

The OCCI Infrastructure document [6] details how an OCCI implementation can model and implement an Infrastructure as a Service API offering by using the OCCI Core Model [3]. In the following, we describe the main infrastructure types defined within OCCI Infrastructure, their actions and states defined for each of them. The main infrastructure types defined within OCCI Infrastructure are:

- **Compute** : represents a generic information processing resource. It exposes four actions: start, stop, restart and suspend. Figure 2.4a illustrates the state diagram for a Compute instance.

- **Network** : Interconnection resource and represents a L2 networking resource. It exposes two actions: up and down. Figure 2.4b illustrates the state diagram for a Network instance.

- **Storage** : The Storage type represents resources that record information to a data storage device. A Storage instance exposes five actions: online, offline, backup, snapshot and resize. Figure 2.5 illustrates the state diagram for a Storage instance.

- **NetworkInterface** : connects a Compute instance to a Network instance. This instance does not expose actions. Figure 2.6 illustrates the state diagram for a NetworkInterface instance.

- **StorageLink** : connects a Compute instance to a Storage instance. Like NetworkInterface instance, StorageLink instance also does not expose actions. Figure 2.6 illustrates the state diagram for a StorageLink instance.

For more details, the reader can refer to [6].

2.2.2 Behavior description in OCCI Platform

In [7], the authors defined an OCCI-based model for the description of the platform and application resources independently from the targeted PaaS. The proposed model extends the OCCI core model [3]. Platform resource types are derived from Resource type at OCCI core level whereas platform resource mixins are derived from the Mixin type and the interfaces, which links these resources between them, are derived from the Link type.

---

Figure 2.4 – UML State Diagram.
The main defined OCCI platform resources are:

**Database** resources which are data store resources for platform applications processing persistent data (e.g. MySQL, PostgreSQL, CouchDB, etc.). Actions applicable to DATABASE are: StartDB, StopDB, RestartDB and BackupDB. The execution of an action induces to the modification of its value according to the diagram presented in Figure 2.7a.

**Container** resources which are engines to host and run applications (e.g. Apache Axis container, Bonita, IBM WebSphere, etc.). In [8], the author proposed a set of action for CONTAINER instance: StartContainer, StopContainer, RestartContainer and SuspendContainer. The state value evolves after the execution of one of the actions according to the state diagram introduced in Figure 2.7b.

**Router** resources which are resources that provide protocols, message format transformations and routing (e.g. ESBPetals router, Apache Synapse, etc.). The actions applicable to a Router instance are: EnableRouter, ConfigureRouter and DisableRouter. The evolution of the state value in relation with the execution of these actions are schematized in Figure 2.7c.

**DatabaseLink** to connect a Container resource to a Database resource through the Bind action.

**ContainerLink** to enable connecting one or several homogeneous CONTAINER resources between them.

Figure 2.5 – State Diagram for a Storage Instance.

Figure 2.6 – State Diagram for a NetworkInterface and StorageLink Instances.
2.2.3 OCCI behavior description in autonomic computing

In [9], M. Mohamed defines a new functionality to Cloud resources in order to enable their monitoring, reconfiguration and autonomic management [10]. The definition was generic, that could be described using any description model. The author made the choice to define these functionalities as extensions of Open Cloud Computing Interface (OCCI). Indeed, OCCI provides an extensible model based on Mixin mechanism. A Mixin is an extension that could be added to any Cloud resource in order to define new functionalities or attributes. In the following we give an overview of the different Resource, Mixins and Links defined for monitoring and reconfiguration.

First, we describe the needed resources to instantiate an autonomic infrastructure using cloud resources. The defined resources inherit the Resource base type defined in OCCI. These resources could be started/stopped by invoking the start/stop action. They could be reconfigured by invoking the reconfigure action passing as parameter the new reconfiguration. The state diagram of the resources is presented in Figure 2.8.

**RouterLink** to enable connecting one or several heterogeneous CONTAINER resources to a ROUTER resource.
Autonomic Manager Resource: It analyzes a given SLA and carries out the list of actions to build the infrastructure. From a given SLA, this resource determines monitoring targets (i.e., the needed attributes to be monitored). It is also responsible of extracting the rules that will be used by the Analyzer resource and the reconfiguration actions to be used by the Reconfiguration Manager. After inspecting the contract (SLA), the Autonomic Manager starts the needed Resources and Links. Then, it instantiates the needed Mixins and eventually configure them with the needed parameters.

Analyzer: This resource allows to analyze monitoring data and eventually generate alerts.

To get monitoring data, the Analyzer resource may subscribe in a Notification service of a managed resource. Subscription is possible using an instance of the SubscriptionLink. The Analyzer resource uses RuleSet Mixin to specify rules to apply on monitoring data.

Planner: It receives alerts from the Analyzer resource through an Alert Link. The role of the Planner resource is to generate reconfiguration actions to apply on resources.

Figure 2.8 – State Diagram of the Resources.

To link the different defined resources, a new link inheriting the Link base type defined in OCCI Core was defined.

Agreement: The Agreement Link is a Link an Autonomic Manager Resource to a managed resource. It allows an Autonomic Manager to inspect the SLA of a managed resource.

Subscription: The Subscription Link is a Link between a managed resource and a consumer.

Notification: The Notification Link is a Link between a managed resource and a subscriber.

Alert: The Alert Link is a Link between the Analyzer resource and the Reconfiguration Manager resource.

Action: The Action Link is a Link between a Reconfiguration Manager Resource and a managed resource on which the latter applies reconfiguration actions. It models the transfer of actions from a Reconfiguration Manager Resource to be applied on a managed resource.

To customize the different resources and the links between them, a new Mixin was defined:
SpecificSLA: It models the needed tools that allows an Autonomic Manager can extract the needed information from a specific SLA.

Polling: It provides the needed functionalities to ensure monitoring.

Reconfiguration: It provides the needed functionalities to ensure reconfiguration.

Subscription: It allows other resources (clients) to subscribe on monitoring notifications related to this resource.

RuleSet: It represents a function applied by the Analyzer Resource to control monitoring information values against a defined threshold or conditions.

SubscriptionTool: It describes the different aspects of a subscription (i.e, its type, its duration, eventually its filters).

NotificationTool: It models the needed tools by which notifications are sent to subscribers.

AlertTool: models the needed tools by which alerts can reach a Planner resource.

StrategySet: It represents a function applied by the Planner resource to process incoming alerts.

ActionTool: It models the needed tools by which reconfiguration actions are applied on a specific Resource.

However, the author does not give the state diagram of the different defined Mixins.

2.3 Analysis and Motivations

Descriptions we have presented above are rather tentative for the modeling of OCCI behavior. These tentative are not based on a common model of behavior. They are informal and incomplete in the sense that they do not describe all the aspects of OCCI behavior. These insufficiencies can lead to various different interpretations and interoperability issues between OCCI implementations. To tackle this issue, we define a formal model for the dynamic semantics of the OCCI core concepts in the next chapter.
Chapter 3

OCCI Behavioral Model

As OCCI is a REST API, it gives access to cloud resources via CRUD operations (Create, Retrieve, Update, Delete), over the HTTP network protocol and via an Execute Action operation. Among the OCCI specifications, OCCI HTTP Rendering [1] provides a set of guidelines (i.e. recommended best practices) to create a unified API for managing cloud resources. However, this specification is written in English prose and lacks of a precise semantics.

This chapter defines the formal OCCI behavioral model. This model will be added upon OCCIware metamodel defined in [11] and [4]. Alloy, formal specification language and its analyzer, has been successfully applied to the definition and analysis of dynamic systems [12]. It is carried out in our work by adding the concept of Time to the structural specification of OCCI [11], in order to distinguish between mutable and immutable concepts. Then, we specify the semantics of CRUD operations, by explicitly defining pre- and postconditions.

This chapter is written in a literate programming style: the specification is presented in its entirety, the (informal) commentary on the formal specification being interspersed with excerpts of Alloy code. All assertions (Alloy facts) 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.

This chapter is organized as follows. Section 3.1 highlights the motivation behind the current formalization of OCCI specifications. Section 3.2 discusses related work. Section 3.3 introduces Alloy, its assets and preliminary concepts. Section 3.4 details the Alloy specification of OCCI core concepts. Sections 3.5 to 3.9 detail the Alloy behavioral specification of the different REST operations of the OCCI standard, including CREATE (Section 3.5), RETRIEVE (Sections 3.6), UPDATE (Section 3.7), DELETE (Section 3.8). Section 3.9 introduces EXECUTE ACTION operation, which aims at using state machine approach, in order to express dynamic semantics of this operation. Section 3.10 shows that the overall specification ensures general, imposed and desired properties. Section 3.11 contrasts the formal specification presented in this chapter with the informal one.

3.1 Introduction & General Motivation

The OCCI behavioral model is currently defined by an informal specification [1]. This specification briefly mentions the general operations that constitute the OCCI behavioral model. It has been implemented in several OCCI runtime frameworks such as erocci (Erlang), rOCCI (Ruby), pySSF (Python), and OCCI4Java (Java). However, there are aspects of the specification that remain decidedly insufficiently detailed, which lead to ambiguity and lack of interoperability between different implementations. The present chapter attempts to correct these deficiencies.
by developing a formal specification of the OCCI standard, which makes explicit the underlying constraints and proves properties that should be met in OCCI-based cloud systems.

Beyond ensuring the consistency of the OCCI standard, a formal specification for the OCCI standard can serve several purposes:

- to provide a more abstract, programming-language-independent specification of the OCCI standard;
- to allow a formal verification of OCCI structure and operations;
- to provide the basis of a formal architecture description language for OCCI;
- to allow a formal specification and verification of OCCI tools;
- to allow a rigorous comparison with other cloud computing solutions.

The latter is important because the OCCI specification aims to define a general cloud model from which more specialized cloud models can be derived and combined. A formal specification can thus help in assessing whether a cloud computing solution constitutes a proper refinement or specialization of the OCCI standard.

### 3.2 Related Work

In this section we present some work of OCCI formalization that are relevant to our work.

The most relevant work to ours is [11]. The authors built an Ecore metamodel for OCCI and added some Object Constraint Language (OCL) invariants. OCL is a constraint specification language applied on a metamodel for formalization purpose. They used this metamodel to describe cloud computing domains: inter-cloud networking, infrastructure, platform, application, service management, cloud monitoring, and autonomic computing. Then, EMF-VF\(^1\), the OCL interpreter, verified whether each instance of the metamodel is conform to its constraints or not. However, OCL presents some limitations that we present below:

- EMF-VF may proceed into possible erroneous validations and the OCL invariants may not be covering all unexpected OCCI Extension or Configuration instances. Some missing extensions can violate the existing OCL constraints. Thus, we can talk about incompleteness.

- Despite that OCL formalises OCCI metamodel by describing its static semantics, OCL syntax is not abstract enough\(^2\) and is not formal by itself. As for the dynamic semantics of OCCI, the authors in [11] are always using natural language to express it. Thus, we can still talk about lack of formalization.

- When OCL allows to make an analysis on the instances of OCCI metamodel, Alloy (cf. Section 3.3) allows for an analysis of the metamodel itself. In addition, unlike OCL, Alloy allows to check required properties of OCCI and not only to validate its constraints. Briefly, Alloy does what OCL does but even more.

For all the above reasons, we propose another specification for OCCI in Alloy and we believe it is necessary in order to solve OCL limitations, and especially to raise the abstraction level and formally encode OCCI behavior.

---

1EMF Validation Framework.
2OCL uses a syntax similar to programming languages and closely related to the syntax of UML.
3.3 Alloy

Formal methods are techniques, languages, tools that allow to reason rigorously, using mathematical logic on systems and to check/demonstrate their validity toward a certain specification. There are several schools for formal methods like Petri networks, languages based on logic, semantics programs, automata theory. In the past, the use of formal methods in practice seemed hopeless. The notation were too obscure, the techniques did not scale, and the tool support was inadequate or too hard to use. Few people had the training to use them effectively on the job. Recently, we have begun to see a more promising picture of formal methods. Examples of these lightweight formal methods include the Alloy object modeling notation [13] that we are interested in using in our current work. Alloy is a formal language based on the first order logic, so it is able to translate specifications into boolean expressions that can be automatically analyzed by SAT solvers. Alloy allows developers to describe models by writing concepts and constraints to specify their systems. Alloy is supported by a tool, the "Alloy analyzer", that is inspired by model checkers, but is implemented as a SAT solver to allow automatic verification, within a bounded scope, of a model against a metamodel. Alloy analyzer implements two commands for verification: i) Run to find instances conform to the specification and ii) Check to find a counter-example of a certain property.

One can say that Alloy is a model finder (cf. Figure 3.1).

![Alloy Model Finder](image)

**Figure 3.1 – Alloy Model Finder.**

3.3.1 Motivation

Specifying using Alloy is the process of describing the structure and behavior of a system and its desired properties. Alloy is a language with a mathematically defined syntax and semantics, and several other assets:

1. Our provided Alloy specification of OCCI is *precise*. It defines the *types of OCCI concepts* and describes *configurations of these types*.

2. The concepts of OCCI are defined with Alloy in a *concise way*, which is easy to learn and understand by cloud engineers.

3. The OCCI modeling language defined with Alloy has both a concrete *graphical* syntax and a concrete *textual* syntax. Actually, the graphical input is not very convenient; it is usually easier to enter a text than to draw a diagram [13]. Likewise, it is easier to exchange
models in a textual form. However, since the textual output can not be easily readable by a human, we think a graphical output remains simpler and usable in practice.

4. Thanks to Alloy analyzer which is based on SAT solver, the OCCI solution has now the possibility of solving constraints and completion in case of an incomplete specification (i.e., that lack of constraints). For example, choosing a suitable deployment environment or an optimal placement of a virtual machine, is reduced to finding the optimal answer to existing constraints.

5. After defining in Alloy the formal OCCI specification with its requirements and properties, we can verify using Alloy that the specification is instantiable. We can also analyze what could not be instantiated, thus deployed, and what may cause problems if instantiated. In these cases, our specification might be overconstraining and releasing some constraints seems to be necessary.

6. One of the great benefits of Alloy is the ability of incremental analysis. An engineer may explore design ideas starting from a tiny model which is then scaled up, with Alloy able to analyse it at every step.

### 3.3.2 Preliminary concepts

An Alloy module consists of a module header, a set of other module imports and zero or more paragraphs. The module header is a name of the module where paragraphs are defined. The import keyword specifies the inclusion of other modules. A paragraph can either be a signature declaration, a constraint, an assertion or a command.

Similarly to a class that represents a set of instances, a signature declaration denotes a set of atoms. An atom is a unity with three basic properties: it is indivisible, immutable and uninterpreted. Signature declarations can introduce fields. A field represents a relation among signatures.

**Facts** and **predicates** describe constraints. The difference between a fact and a predicate is that the first one always holds while the second one only holds when invoked.

**Assertions** allow the expression of properties that are expected to hold as consequence of specified facts.

Finally, **commands** instruct the Alloy analyzer to perform particular analysis using two possible instructions: run and check. The first one checks model consistency by requesting a valid instance, and the latter verifies an assertion by searching for a counter-example. Both commands define a scope, bounding the number of instances allowed for each signature.

### 3.4 OCCI Foundations

Although a detailed description of OCCI structural metamodel is presented in D 2.2.1, we briefly define in this section each of these metamodel concepts. This is important for a better understanding of the following sections. The entire set of constraints (facts) applied on OCCI structural metamodel is provided in Appendix A.

To describe the behavior of OCCI modeling language, we propose a strategy based on temporal dimension. That is why we consider time an important OCCI concept and we deem necessary to add the Time signature to OCCI structural metamodel.
• *Time* is an extra column that is added at the end of each mutable field to represent the state signature. In this idiom, OCCI operations are modeled as predicates that specify the relationship between pre- and post-states. To be more specific, an operation \( op \) will be specified using a predicate \( \text{pred } \) \( \text{op} \)[..., \( t \), \( t' \):Time] ... with two special parameters \( t \) and \( t' \) denoting, respectively, the pre- and post-states.

\[
\text{Time} \quad \{}
\]

In Figure 3.2, we present OCCI Ecore metamodel, with all its existing and added concepts. The latter are represented in blue color.

Taking into account the notion of time, we present in the following each of these OCCI concepts in Alloy. This allows us to distinguish between mutable fields, (i.e., those that are related to *Time*) and immutable ones (i.e., those that are not related to *Time*).

• *Entity* is the abstract class of all Resource and Link instances. The fields *id* and *kind* are immutable. As for *mixins* and *attributes*, they are mutable; they identify the association respectively between *mixins, attributes* and their *Time*.
In Alloy, a declaration of the form \( \text{id : one URI} \) can be read as declaring a feature, or instance variable, of the class Entity; formally, it declares a binary relation between the set of entities, Entity, and the set of URIs, URI.

- **Resource** represents any cloud computing resource, e.g., a virtual machine, a network, an application container, an application. Resource owns a set of mutable *links*.

\[
\text{sig Resource extends Entity (}
\text{  \hspace{0.5cm} links : set Link -> Time)}
\]

The keyword *extends* in Alloy indicates that a primitive set is declared as a subset of another one and that it will form, with other subsets similarly declared, a partition of the set it extends.

- **Link** is the relationship between Resource instances. It contains two mutable fields: source and target.

\[
\text{sig Link extends Entity (}
\text{  \hspace{0.5cm} source : Resource one -> Time,}
\text{  \hspace{1cm} target : Resource one -> Time)}
\]

- **Category** is the abstract class of all the Action, Kind and Mixin instances. All its fields are immutable.

\[
\text{abstract sig Category (}
\text{  \hspace{0.5cm} term : one String,}
\text{  \hspace{1cm} scheme : one URI,}
\text{  \hspace{2cm} title : lone String,}
\text{  \hspace{2cm} attributes : set Attribute)}
\]

The keyword *lone* is an example of a relation multiplicity. In our case, it signifies that a given title is to be associated to at most one String.

- **Kind** is the Entity type. For example, the kind of a Resource can be compute, network, application, etc. The fields parent and actions are immutable. The field entities is mutable.

\[
\text{sig Kind extends Category (}
\text{  \hspace{0.5cm} parent : lone Kind,}
\text{  \hspace{1cm} actions : set Action,}
\text{  \hspace{2cm} entities : set Entity -> Time)}
\]

- **Action** represents an operation that can be executed on an entity instance such as start virtual machine, stop virtual machine, restart an application, resize a storage, etc.

\[
\text{sig Action extends Category}
\]

- **Mixin** is to add additional features such as location and price. Only its field entities is mutable.

\[
\text{sig Mixin extends Category (}
\text{  \hspace{0.5cm} actions : set Action,}
\text{  \hspace{1cm} depends : set Mixin,}
\text{  \hspace{2cm} applies : set Kind,}
\text{  \hspace{2cm} entities : set Entity -> Time)}
\]

---

3Let URI=String

4In Alloy, primitive sets are just signatures or sets of atoms that have no internal structure e.g., sig Time.
• *Attribute* is the property of an entity like machine hostname, IP address of a network, parameter of an action, etc. All its fields are immutable.

```occi
sig Attribute {
    name : one String,
    mutable : lone Bool,
    required : lone Bool,
    default : lone String,
    description : lone String,
    type : lone DataType,
    multiple_values : lone Bool
}
```

In [11], four concepts were added to the initial OCCI specification [3]:

• *Configuration* is the abstraction of an OCCI-based running system. Modeling a configuration offline allows designers to think and analyse their cloud systems without having to deploy them concretely in the clouds. The field `use`, which is the extension set of a configuration, is immutable. The fields `resources` and `mixins` are mutable.

```occi
sig Configuration {
    use : set Extension,
    resources : set Resource -> Time,
    mixins : set Mixin -> Time
}
```

• *Extension* is a set of Resource and Mixin instances targeting a concrete cloud domain (IaaS, PaaS, SaaS, pricing, SLA, cloud monitoring, etc.). An Extension instance can use or extend other Extension instances. All its fields are immutable.

```occi
sig Extension {
    name : one String,
    scheme : one URI,
    import : set Extension,
    kinds : set Kind,
    mixins : set Mixin,
    types : set DataType
}
```

• *DataType* is used to extend the nonextensible data type system of the informal specification. Since *Extensions* use scalar data types (IP address, float, etc.) and enumeration, the classical data type defined as a string, boolean and integer is insufficient and requires to be extended. The field `name` of *DataType* is immutable.

```occi
abstract sig DataType {
    name : one String
}
```

• *AttributeState* is used to describe the value of an attribute.

```occi
sig AttributeState {
    name : one String,
    value : one String
}
```

In OCCI, Kinds and Mixins are provided with a behavior that can be captured by a state machine that is finite and deterministic, i.e., the set of machine states is finite. The machine is in only one state at a given time. It can change from one state to another over transitions when they are initiated by a triggering event or condition.

We define below progressively the semantics of our state machine by declaring three sets: *StateMachine*, *State* and *Transition*, and setting thirteen facts to be respected in our behavioral specification.

• *StateMachine*
In Alloy, + symbol denotes set union.

**Fact 1** Only one State Machine for one kind or mixin.

```
| no disj SM1, SM2 :StateMachine | SM1.@category = SM2.@category
```

The dot notation in 

```
SM1.category
```

is the standard notation for accessing a feature, or attribute, of an instance of a signature.

The symbol @ precedes a field name to prevent it from being expanded. Without the @ symbols, the constraint above would be short for this:

```
| no disj SM1, SM2 :StateMachine | SM1.this.category = SM2.this.category
```

which doesn’t even typecheck.

**Fact 2** No state shared between State Machines.

```
| no disj SM1, SM2 :StateMachine | some SM1.@states & SM2.@states
```

The Alloy keyword `some` denotes the existential qualifier symbol ∃. Thus, `some SM1`, where SM1 is a set, asserts that SM1 is not empty – i.e., that there is some element in SM1. & symbol denotes set intersection.

**Fact 3** The attribute in the State Machine is in the attributes of its category.

```
| attribute in category.attributes
```

In Alloy, `in` denotes the subset relation, or the set membership relation.

**Fact 4** There is only one initial state.

```
| one state : states | state.isInitial = True
```

**Fact 5** There is at least one final state.

```
| some state : states | state.isFinal = True
```

`some state : states | P`, where P is some fact, asserts the existence of some states verifying P. By extension, some state, where state is a set of states, asserts that state is not empty – i.e., that there is some element in state.

**Fact 6** All actions are present in the State Machine.

```
| all action : category.(Kind<:actions + Mixin<:actions) | action in transitions.event
```

In OCCIware
The keyword all denotes the universal quantifier symbol ∀, where a declaration such as
\texttt{action : category.(Kind<:actions + Mixin<:actions)} denotes an arbitrary element action
of set intersection between actions of category kinds and actions of category mixins.

\texttt{<:} symbol denotes domain restriction of Kind to actions and Mixin to actions.

\textbf{Fact 7} Transition target is a state in the State Machine.

\texttt{transition.target in states}

\textbf{Fact 8} Transition source is a state in the State Machine.

\texttt{transition.source in states}

\textbf{Fact 9} All states are present in the State Machine.

\texttt{all state : State | state in states}

- \textit{State} models a situation during which some invariant condition of the modeled system
  holds. The main fields of a state are:
  - Name: the name of state,
  - Description: the description of state,
  - Entities: the entity set which is contained in specific state set at a time t,
  - Exit: an optional state that is reached regardless of which transition was taken out
    of the state,
  - IsInitial: one state that denotes the initial state,
  - IsFinal: one or multiple states that denote the final state.

\texttt{sig State {
  name : one String ,
  entities : set Entity ,
  exit : one State ,
  description : one String ,
  isInitial : one Bool ,
  isFinal : one Bool
}}

\textbf{Fact 10} Each state is owned by one State Machine.

\texttt{one ~(StateMachine<:states)[this]}

- \textit{Transition} can be described by the triplet (Event, Condition, Action). To these fields, we
  add the Source and Target states.

  - Event: an occurrence that may trigger a transition. Event types include an explicit
    signal from outside the system, an invocation from inside the system, the passage of
    a designated period of time, or a designated condition becoming true,
  - Condition: the condition that must be fulfilled before the transition will occur. The
    condition affects the behavior of a state machine by enabling actions only when they
    are true and disabling them when they are false,
– Action: when an event instance is dispatched, the state machine responds by performing actions. In our OCCI specification, Action is a REST operation executed on the resource. The result of the invocation of an action causes an important change in system state.

– Source: the source state of a transition,

– Target: the target state of a transition.

```alloy
sig Transition {  
event: one Action,  
condition: one State,  
action: one Action,  
source: one State,  
target: one State
}
```

**Fact 11** No transition shared between State Machines.

```alloy
no disj SM1, SM2 : StateMachine | some SM1.transitions & SM2.transitions
```

**Fact 12** Each transition owned by one State Machine.

```alloy
one ~(StateMachine:<:transitions) [this]
```

**Fact 13** Events are actions of the State Machine category.

```alloy
all SM : StateMachine | event in SM.category.(Kind:<:actions + Mixin:<:actions)
```

After presenting the structural semantics of OCCI with Alloy, we present in the following sections (3.5 to 3.9) its behavioral aspect. Based on the informal OCCI behavioral specification detailed in [1] (cf. Figure 3.3), we add in Alloy fifteen operations to Configuration concept of OCCIware metamodel, in order to express the behavioral semantics of OCCI. These operations are defined as predicates with some arguments corresponding to the initial time, i.e., the time prior to the execution of the operation, and other arguments corresponding to the final time, i.e., the time resulting from the execution of the operation. The core of each predicate is carried out by defining explicitly the pre- and postconditions, which are constraints that must be always satisfied.

### 3.5 CREATE

*Create* refers to the first CRUD operation added to a configuration. It permits users to create configuration elements. It may be applied on an *Entity*, i.e., a *Resource* or a *Link*, and on a *UserMixin*.

#### 3.5.1 Create Resource

This operation is referred to as "Creation/Update of the resource instance, supplying the full representation of the resource instance" in Figure 3.3. We specify below the Create Resource operation in Alloy.

```alloy
pred CreateResource | config : Configuration, resourceId : URI, resourceKind : Kind, resourceMixins : set Mixin, resourceAttributes : set AttributeState, t, t' : Time ]
```

At time t, we specify that our configuration, passed as argument to the predicate, does not have a resource with the ID passed in the predicate arguments. As well, we specify that the kind and the mixins of the resource we want to create, are contained in our configuration extensions.
At time $t'$, we add to our configuration resources, one resource with the ID, kind, mixins and attributes, passed as argument to the predicate. We also include postconditions stating that mixins of our configuration must remain unchanged when executing this operation. It is not consensual whether such conditions must be specified, and one can assume an implicit invariability assumption, stating that what is not mentioned in a postcondition must remain unchanged. However, such assumption may lead to ambiguities in postcondition interpretation, and we require them to be explicitly specified.

```alloy
cone resource : Resource { resource.id = resourceId resource.kind = resourceKind resource.mixins.t' = resourceMixins resource.attributes.t' = resourceAttributes config.resources.t' = config.resources.t + resource }
}
cone config.mixins.t' = config.mixins.t
}
```

### 3.5.2 Create Link

"Link", similar to "Resource", is a concept that extends the abstract signature "Entity". So, beside Create Resource, we specify below the Create Link operation in Alloy.

```alloy
pred CreateLink[config : Configuration, linkId : URI, linkKind : Kind, linkSource : Resource, linkTarget : Resource, linkAttributes : set AttributeState, t, t' : Time] {
```
At time t, similarly to Create Resource preconditions, we specify that our configuration resources, passed as argument to the predicate, do not have a link with the ID, source and target passed in the predicate arguments. As well, we specify that the kind of the link we want to create is contained in our configuration extensions. In addition, we precise that our link source and target must be contained in our configuration resources.

no link : config/resources\(t\).links\(t\) {  
  link.id = linkId  
  link.source\(t\) = linkSource  
  link.target\(t\) = linkTarget  
  linkKind in config/use.kinds  
  linkSource in config/resources\(t\)  
  linkTarget in config/resources\(t\)  
}

At time \(t'\), we add to the links of our source resource, one link with the ID, kind, source, target and attributes, passed as argument to the predicate. We also include postconditions stating that resources and mixins of our configuration must remain unchanged when executing this operation.

one link : Link {  
  link.id = linkId  
  link.kind = linkKind  
  link.source\(t'\) = linkSource  
  link.target\(t'\) = linkTarget  
  link.attributes\(t'\) = linkAttributes  
  linkSource\(s\).links\(t'\) = linkSource\(s\).links\(t\) + link  
  config/resources\(t'\) = config/resources\(t\)  
  config.mixins\(t'\) = config.mixins\(t\)  
}

3.5.3 Add User Mixin

This operation is referred to as "Add a user-defined Mixin" in Figure 3.3. We specify below the Add User Mixin operation in Alloy.

\[
\text{pred AddUserMixin[config : Configuration, mixinScheme : String, mixinTerm : String, t, t' : Time]}\]

At time t, we specify that our configuration does not contain the mixin we want to add. The function \(\text{findUserMixin}\) can be interpreted as a Mixin finder that returns, if available, a mixin with the scheme and term that are passed as parameters.

\[
nocfg\cdot\text{findUserMixin[mixinScheme, mixinTerm, t]}\]

At time \(t'\), we add to our configuration mixins, one mixin with the scheme and term passed as argument to the predicate. We also state that our configuration resources must remain unchanged when executing this operation.

\[
one m : Mixin \{  
  m.scheme = mixinScheme  
  m.term = mixinTerm  
  config.mixins\(t'\) = config.mixins\(t\) + m  
  config/resources\(t'\) = config/resources\(t\)  
\}  
\]

3.6 RETRIEVE

Retrieve refers to the second CRUD operation added to a configuration. It permits users to get configuration elements. It works on entities, collections and categories.

3.6.1 Retrieve Resource

This operation is referred to as "Retrieval of the resource instance’s representation" in Figure 3.3. We specify below the Retrieve Resource operation in Alloy.
3.6.2 Retrieve Link

"Link", similar to "Resource", is a concept that extends the abstract signature "Entity". So, beside Retrieve Resource, we specify below the Retrieve Link operation in Alloy.

\[
\text{pred RetrieveLink} \left[ \text{config} : \text{Configuration}, \text{linkId} : \text{URI}, \text{t}, \text{t'} : \text{Time} \right]
\]

At time \( t \), we specify that our configuration, passed as argument to the predicate, has a link with the ID passed in the predicate arguments.

\[
\text{resource} = \text{config.resources.t} \left( \text{resource.id} = \text{resourceId} \right)
\]

At time \( t' \), we return source, target, mixins and attributes of the link and we explicitly specify that resources and mixins of our configuration remain unchanged.

\[
\begin{align*}
\text{one link} & : \text{config.resources.t}.\text{links.t} \left( \text{link.id} = \text{linkId} \right) \\
\text{one link} & : \text{config.resources.t}.\text{links.t'} \left( \text{link.id} = \text{linkId} \\
\text{link.source.t'} & = \text{link.source.t} \\
\text{link.target.t'} & = \text{link.target.t} \\
\text{link.mixins.t'} & = \text{link.mixins.t} \\
\text{link.attributes.t'} & = \text{link.attributes.t} \\
\text{config.resources.t'} & = \text{config.resources.t} \\
\text{config.mixins.t'} & = \text{config.mixins.t}
\end{align*}
\]

3.6.3 Retrieve Collection

This operation is referred to as "Retrieve a collection of resource instances" in Figure 3.3. We specify below the Retrieve Collection operation in Alloy.

\[
\text{pred RetrieveCollection} \left[ \text{config} : \text{Configuration}, \text{scheme1} : \text{URI}, \text{entities1} : \text{Entity}, \text{kind} : \text{Kind}, \text{t}, \text{t'} : \text{Time} \right]
\]

At time \( t \), we specify that the scheme passed as parameter is contained in our configuration extensions. Later on, we return the kind of the extension with this scheme, then we return the entities of this kind.

\[
\begin{align*}
\text{one extension} & : \text{config.use} \\
\{ \text{scheme1 in extension.scheme} \\
\text{kind in extension.kinds} \\
\text{entities1 in kind.entities.t} \}
\end{align*}
\]

At time \( t' \), we explicitly state that, as it is a retrieve then the configuration must not be changed.

\[
\begin{align*}
\text{config.resources.t'} & = \text{config.resources.t} \\
\text{config.mixins.t'} & = \text{config.mixins.t}
\end{align*}
\]

5To recall, one or more entity instances associated with the same Kind or Mixin instance, automatically form a collection.
3.6.4 Get All Categories

This operation is referred to as "Retrieve capabilities" in Figure 3.3. We specify below the Get All Categories operation in Alloy.

```alloy
pred GetAllCategories[config : Configuration, categories : set Category, t, t' : Time]
{
    At time t, we return the categories, i.e., the union of kind set and mixin set of our configuration extensions, as well as the mixin set of our configuration.
    categories = config.use.(Extension<:kinds + Extension<:mixins) + config.mixins.t

    At time t', we explicitly state that, as it is a retrieve then the configuration must not be changed.
    config.resources.t' = config.resources.t
    config.mixins.t' = config.mixins.t
}
```

3.7 UPDATE

Update refers to the third CRUD operation added to a configuration. It may alter the attribute of an Entity, i.e., a Resource or a Link, and associate/dissociate UserMixin to/from entities.

3.7.1 Update Resource

This operation is referred to as "Partial update of the resource instance" in Figure 3.3. We specify below, in Alloy, the Update Resource operation that aims to update the resource attributes.

```alloy
pred UpdateResource[config : Configuration, resourceId : URI, attribute1 : AttributeState, attribute2 : AttributeState, t, t' : Time]
{
    At time t, we specify that attribute1, which is the existing attribute state of our resource, is different than attribute2, which is the new attribute state of our resource. As well, we specify that our configuration, passed as argument to the predicate, has one resource with the ID and attribute1 passed in the predicate arguments.
    attribute1 ≠ attribute2
    one resource : config.resources.t | resource.id = resourceId
    and resource.attributes.t = attribute1

    At time t', we update the attribute set of our resource and we specify that mixins of our configuration must remain unchanged when executing this operation.
    one resource : config.resources.t[
        resource.attributes.t' = resource.attributes.t ++ attribute2
    ]
    config.mixins.t' = config.mixins.t
}
```

3.7.2 Update Link

"Link", similar to "Resource", is a concept that extends the abstract signature "Entity". So, beside Update Resource, we specify below the Update Link operation in Alloy.

```alloy
pred UpdateLink[config : Configuration, linkId : URI, attribute1, attribute2 : AttributeState, t, t' : Time]
{
    At time t, we specify that attribute1, which is the existing attribute state of our link, is different than attribute2, which is the new attribute state of our link. As well, we specify that our configuration resources have one link with the ID and attribute1 passed in the predicate arguments.
    one resource : config.resources.t[
        resource.attributes.t' = resource.attributes.t ++ attribute2
    ]
    config.mixins.t' = config.mixins.t
}
```
attribute1 ≠ attribute2

one link : config.resources.t.links.t | link.id = linkId
and link.attributes.t = attribute1

At time t’, we update the attribute set of our link and we specify that resources and mixins of our configuration must remain unchanged when executing this operation.

At time t’, we associate to our resource, the mixin with same term and scheme passed as arguments to the predicate. Later on, we precise that the resource attributes must remain unchanged.

3.7.3 Associate Mixin To Resource

This operation is referred to as "Adds a resource instance to this collection" in Figure 3.3. We specify below the Associate Mixin To Resource operation in Alloy.

pred AssociateMixinToResource [config : Configuration, resourceId : URI, mixinTerm : String, mixinScheme : URI, t, t’ : Time] {

At time t, we precise that our configuration contains one resource with the ID passed as argument to the predicate. As well, we precise that the mixin we want to add is contained in the extension mixins.

At time t’, we associate to our resource, the mixin with same term and scheme passed as arguments to the predicate. Later on, we precise that the resource attributes must remain unchanged.

3.7.4 Dissociate Mixin From Resource

This operation is referred to as "Removal of a single, a subset of or all the resource instances from the Mixin collection" in Figure 3.3. We specify below the Dissociate Mixin From Resource operation in Alloy.

At time t, we precise that our configuration contains one resource with the ID passed as argument to the predicate. As well, we precise that the mixin we want to dissociate from the resource is contained in the resource mixins.

At time t’, we remove from our resource, the mixin with same term and scheme passed as arguments to the predicate. Later on, we precise that the resource attributes must remain unchanged. Note that - symbol denotes set difference.
3.8 DELETE

Delete refers to the fourth CRUD operation added to a configuration. It permits users to delete configuration elements. It may be applied on an Entity, i.e., a Resource or a Link, and on a UserMixin.

3.8.1 Delete Resource

This operation is referred to as "Deletion of the resource instance" in Figure 3.3. We specify below the Delete Resource operation in Alloy.

```
pred DeleteResource[config: Configuration, resourceId: URI, t, t’: Time]
{
  At time t, we specify that our configuration, passed as argument to the predicate, has one resource with the ID passed in the predicate arguments.
  one resource: config.resources.t | resource.id = resourceId

  At time t’, we remove from our configuration resources, one resource with the ID passed as argument to the predicate. We also include postconditions stating that mixins of our configuration must remain unchanged when executing this operation.
  one resource: config.resources.t |
  resource.id = resourceId
  config.resources.t’ = config.resources.t - resource
  config.mixins.t’ = config.mixins.t
}
```

3.8.2 Delete Link

"Link", similar to "Resource", is a concept that extends the abstract signature "Entity". So, beside Delete Resource, we specify below the Delete Link operation in Alloy.

```
pred DeleteLink[config: Configuration, linkId: URI, t, t’: Time]
{
  At time t, we specify that our configuration, passed as argument to the predicate, has one link with the ID passed in the predicate arguments.
  one link: config.resources.t.links.t | link.id = linkId

  At time t’, we remove from the links of our configuration resources, one link with the ID passed as argument to the predicate. We also include postconditions stating that resources and mixins of our configuration must remain unchanged when executing this operation.
  one link: config.resources.t.links.t |
  link.id = linkId
  link.source.t.links.t’ = link.source.t.links.t - link
  no link.source.t’
  no link.target.t’
  config.resources.t’ = config.resources.t
  config.mixins.t’ = config.mixins.t
}
```

3.8.3 Remove User Mixin

This operation is referred to as "Removal of a user-defined Mixin" in Figure 3.3. We specify below the Remove User Mixin operation in Alloy.

```
pred RemoveUserMixin[config: Configuration, mixinScheme: String, mixinTerm: String, t, t’: Time]
{
  At time t, we specify that our configuration contains the mixin we want to remove. This is done by using the findUserMixin function.
}
```
At time $t'$, we remove our configuration mixins, one mixin with the scheme and term passed as argument to the predicate. We also state that our configuration resources must remain unchanged when executing this operation.

\[
\text{one config.findUserMixin} \left[ \text{mixinScheme, mixinTerm, t} \right]
\]

\[
\text{one m : Mixin} \left[ \\
\begin{array}{c}
\text{m.scheme = mixinScheme} \\
\text{m.term = mixinTerm} \\
\text{config.mixins.t' = config.mixins.t - m} \\
\end{array} \\
\text{config.resources.t' = config.resources.t}
\right]
\]

### 3.9 EXECUTE ACTION

This operation is referred to as "Performs actions on a collection of resource instances" in Figure 3.3.

The Execute Action operation is launched on entities and is added to the CRUD operations. In this case, the developer should manage and change the entity states according to each executed action.

For the CRUD operations, we captured the intended semantics of OCCI behavior using Alloy predicates. However, the specification of the intended semantics of the Execute Action, is better achieved by an explicit characterization of this operation in terms of a Finite State Machine.

\[
\text{pred ExecuteAction} \left[ \text{config : Configuration, action : Action, resource : Resource,} \\
\text{SM :StateMachine, t, t' : Time} \right]
\]

\[
\begin{align*}
\text{action in SM.transitions.event} \\
\text{SM.states.entities in resource.kind.entities.t} \\
\text{one exit : State | exit = SM.states<:exit} \\
\text{SM.transitions.source = SM.transitions.target}
\end{align*}
\]

As postconditions, we specify that the source state at time $t'$ was the target state at time $t$.

### 3.10 Behavior properties definition and verification

Once a formal specification has been developed and after specifying constraints (fact) and operations (pred), the specification is used as the basis of proving properties of the specification and reasoning on the developed system. There are several spots of verification and validation that can be performed using the Alloy analyzer to ensure that the model expresses the behavior desired by the developer. In order to ensure the validity of the assertion, a large scope of research is used within the command check.

#### 3.10.1 Consistency

At first, we verify whether some general properties hold in our OCCIware behavioral model. Among these properties, we are mainly interested in checking the consistency. It means that there are no contradictory constraints and each OCCI operation is executable, i.e., there exists a
model/execution trace for an OCCI system that combines all the pre- and postconditions defined in this operation.

For example, the CreateResource predicate (cf. Section 3.5.1), which asserts the existence of a configuration supporting all the different facts defined in the present operation, can be shown to have an instance (and hence to be consistent).

\[
\text{run CreateResource} \begin{cases}
\text{one config : Configuration, resourceId : URI, kind : Kind, mixins : Mixin,} \\
\text{t : Time | CreateResource[config, resourceId, kind, mixins, t, t.next]} \\
\end{cases} \text{for 3 but exactly 1 Configuration, exactly 2 Time}
\]

The keyword \textit{for} can be used to define a scope bounding the number of atoms allowed for each signature. The keyword \textit{but} establishes an exception for the boundary defined by \textit{for}. In this case, the number of Configuration atoms is limited to 1 and the number of Time atoms is limited to 2.

Note that to model finite execution traces, a total order is imposed on the Time signature using the predefined Alloy library \textit{util/ordering}. This library defines useful relations to manipulate the total order, namely \textit{first} to denote the first time, and \textit{next}, a binary relation that, given a time returns the following time in the order. Figure 3.4a and Figure 3.4b present two consecutive states of this trace moving the resource into the configuration.

(a) Create Resource Trace at Time 0.  
(b) Create Resource Trace at Time 1.

Figure 3.4 – Example of Consistent Operation.

Similarly, all the fourteen operations beside \textit{Create Resource} were proven to be consistent (cf. Appendix A). We list them below.

\[\begin{align*}
\text{CreateLink is consistent.} \\
\text{AddUserMixin is consistent.} \\
\text{RetrieveResource is consistent.} \\
\text{RetrieveLink is consistent.} \\
\text{RetrieveCollection is consistent.} \\
\text{GetAllCategories is consistent.} \\
\text{UpdateResource is consistent.} \\
\text{UpdateLink is consistent.} \\
\text{AssociateMixinToResource is consistent.}
\end{align*}\]
3.10.2 Desired properties

The notion of consistency is very basic and does not suffice in order to validate our model. Some more reliable notions consist in checking that every state of the `StateMachine` is reachable, some operations are sequential or reversible.

Reachability

**Definition 1** Every non-initial state can be reached from an initial state.

**Definition 2** From every non-final state, at least one final state can be reached.

For example, is it possible for a resource to have at least one exit? Again, using a `run` command, we can ask the analyzer to return a trace where a resource state reaches state exit. We didn’t succeed yet at proving this property using Alloy, but it is ineluctable to find ways to achieve this goal. We are open to exploit other solvers than Alloy (cf. Section 3.11.2).

Sequentiality

There are other examples of Verification & Validation tasks that can be performed using the Alloy Analyzer. For example, we can check that sometimes an operation cannot happen if another operation did not happen at the time before. In the informal specification, it is explicitly stated that `Update Resource` operation should be preceded by `Retrieve Resource` operation: "Before updating a resource instance it is RECOMMENDED that the client first retrieves the resource instance." [1]. We also add in our current specification that `Retrieve Resource` operation must be preceded by `Create Resource` operation.

```alloy
assert CreateResourceAndRetrieveResourceAreSequential {
  all config : Configuration, resourceId : URI, resourceKind : Kind, resourceMixins : set Mixin, t : Time, t' : t.next, t'' : t'.next
  CreateResource[config, resourceId, resourceKind, resourceMixins, t, t']
  and RetrieveResource[config, resourceId, t', t'']
  implies resource.id = resourceId
  and resource.kind = resourceKind
  and resource.mixins.t'' = resourceMixins
}
```

As previously mentioned, `check` verifies the assertion by searching for a counter-example. We specify a big scope in order to be confident that the assertion holds. In particular, we bound the number of atoms allowed for each signature to 10. Since no counter-examples are returned with such big scope, we can be more confident that this assertion holds. Note also the use of an implication of the form A1 implies B1, is equivalent to (A1 and B1) or ((not A1) and (not B1)).

The pairs of operations that are proven as sequential are listed below in an exhaustive way.

One can refer to Appendix A for more details.

```alloy
assert CreateLinkAndRetrieveLinkAreSequential
assert RetrieveResourceAndUpdateResourceAreSequential
assert RetrieveLinkAndUpdateLinkAreSequential
```
Deliverable D2.2.2  Status Final
Expected 2016/11/30  Dissemination public
Delivered December 14, 2016  Version 1.0

Table 3.1 – Imposed Properties Conforming to HTTP Protocol.

<table>
<thead>
<tr>
<th>Method</th>
<th>IsIdempotent</th>
<th>IsSafe</th>
<th>IsCacheable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Resource ~ PUT</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update Resource ~ POST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retrieve Resource ~ GET</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>Retrieve Link ~ GET</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
<tr>
<td>Delete Resource ~ DELETE</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create Link ~ PUT</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete Link ~ DELETE</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add UserMixin ~ PUT</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove UserMixin ~ DELETE</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Associate Mixin To Resource ~ POST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dissociate Mixin From Resource ~ POST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Get Collections ~ GET</td>
<td>✓</td>
<td>✓</td>
<td>?</td>
</tr>
</tbody>
</table>

Reversibility

We prove that some operations are reversible because they contain contradictory mathematical logic, such as Create Resource and Delete Resource or Create Link and Delete Link or Add User Mixin and Remove User Mixin.

```latex
\textbf{assert} CreateResourceAndDeleteResourceAreReversible
\{ all config: Configuration, resourceId: URI, resourceKind: Kind, resourceMixins: Mixin, t: Time, t':
  t.next
\} \implies CreateResource[config, resourceId, resourceKind, resourceMixins, t, t']
\textbf{check} CreateResourceAndDeleteResourceAreReversible for 10
```

We have also proven the following two assertions (cf. Appendix A):

```latex
\textbf{assert} CreateLinkAndDeleteLinkAreReversible
\textbf{assert} AddUserMixinAndRemoveUserMixinAreReversible
```

3.10.3 Imposed properties

As OCCI is a REST architecture that conforms to the HTTP protocol, it must conform to its specification too. Therefore, there are some imposed properties we want to have in our OCCI-based systems so they must be checked in our specification. According to the request for comments (RFC) HTTP 2 [14], we identified three properties of the HTTP methods and we proved in our formal specification that appropriate OCCI operations respect these properties, as shown in Table 3.1.

Idempotence

GET, PUT and DELETE methods are idempotent, i.e., they always produce the same server external state even if applied several times. Since Retrieve operation is associated to a GET HTTP method, Create operation is a PUT HTTP method and Delete operation is a DELETE HTTP method, we verify in our formal specification that Retrieve Resource, Retrieve Link, Retrieve Collection, Get All Categories, Create Resource, Create Link, Add User Mixin, Delete Resource, Delete Link and Remove User Mixin are idempotent. For example, as a result, the following assertion is valid:
In contrast, the following assertion is not valid, since an Update operation is not idempotent:

\[
\text{assert UpdateResourceIsIdempotent}
\\{ \\
\quad \text{all config : Configuration, resourceId : URI, attribute1 : AttributeState, attribute2 : AttributeState, t : Time} \\
\quad \text{UpdateResource [config, resourceId, attribute1, attribute2, t, t.next]} \\
\quad \text{implies UpdateResource [config, resourceId, attribute1, attribute2, t.next, t.next.next]} \\
\\}
\]

Safety

A GET method is safe, i.e., it does not change the external server status. It mainly concerns the retrieval of information. A safe method is necessarily an idempotent method, but not the reverse way. Therefore, we prove that Retrieve Resource, Retrieve Link, Retrieve Collection and Get All Categories respect this property, so they do not change the system configuration. An example of a safe operation is detailed below.

\[
\text{assert GetAllCategoriesIsSafe}
\\{ \\
\quad \text{all config : Configuration, categories : Category, t, t’ : Time} \\
\quad \text{GetAllCategories [config, categories, t, t’]} \\
\quad \text{implies config.resources.t = config.resources.t’} \\
\quad \text{and config.mixins.t = config.mixins.t’} \\
\\}
\]

Cacheability

GET and POST methods are cacheable, i.e., their answer may be stored in cache in order to answer other queries. Since Retrieve operation is associated to a GET HTTP method and Update operation is a POST HTTP method, we must verify in our formal specification that Retrieve Resource, Retrieve Collection, Get All Categories, Update Resource, Update Link, Associate Mixin To Resource and Dissociate Mixin From Resource are cacheable. However, according to our chosen level of abstraction in the current formal specification, the proof of the cacheability property is not feasible, at least not for the moment.

3.11 Discussion

We discuss in this section the main differences between the informal OCCI specification [1] and the present formal one. We also comment briefly on the use of Alloy language and of the Alloy analyzer.

3.11.1 Differences with the informal specification

- The first difference with the informal specification is the explicit definition of the types of OCCI core concepts found in Section 3.4. The informal specification only refers to them in passing, yet making them explicit is key to highlighting the efficiency of the OCCI modeling language. In particular, it enables us to define precisely configurations and to understand the behavior that can be expected from OCCI. In formalizing the OCCI concepts, we have also clarified features of the OCCI standard. We list them below.
• We allow configurations to be passed by value in operations. This feature is not discussed in the informal specification, obviously because the concept of Configuration did not even exist.

• We clearly list some behavioral properties to be respected in OCCI-based systems. The informal specification is not clear about this notion; especially that it does not consider at all the imposed properties that we depicted in our current specification. We believe that these properties are an added value that we brought to OCCI behavioral specification.

We list below enhancements which the formal specification brings to the informal one, and two differences between the two specifications:

• The concept of Time is added to the informal specification as a signature, in order to denote the overall state of the system, and model operations as predicates that specify the relationship between pre- and post-states.

• Pre- and postconditions of operations on the different CRUD operations defined in the specification are made explicit. For instance, the postconditions we defined on the Retrieve operations specify that the resources and the mixins in a configuration must remain unchanged. The informal specification is not explicit on these conditions.

• The present specification clarifies the admissible effects of operations not only through operations pre- and postconditions but also by highlighting valid and invalid assertions. The informal specification does not contain such a breadth of details on operations.

• The GetAllCategories operation in basic introspection,
  
  – does not return any Category if applied on a Configuration without any Extensions nor UserMixins,
  
  – returns only UserMixins if applied on a Configuration without Extensions,
  
  – returns only Extensions if applied on a Configuration without UserMixins.

The informal specification is evasive on that subject. We believe this form is preferable for introspection purposes.

• The informal specification distinguishes between two types of Create. A resource instance can be created using two ways - HTTP POST or PUT. We have left out this distinction in the formal specification, since we believe it has no effect on global and abstract behavior semantics, but it is a real source of confusion.

• We implement a Finite State Machine to model state changes when applying an Execute Action operation on a configuration.

### 3.11.2 Using Alloy

In this chapter, we present an OCCI-based formal cloud modeling language. We choose to use Alloy since it provides a fast specification development/specification debugging loop, thanks to the fully automatic character of the Alloy analyzer. Coupled with a fast learning curve due to its first-order character, a well-thought out syntax, and extremely useful model visualization features in the Alloy analyzer, developing specifications with Alloy is a very streamlined, and even enjoyable process. The surprising amount of bugs (trivial or not), that were weeded out from the specification thanks to this rapid feedback loop, especially during the early stages of
its development, is a testament to the efficacy and efficiency of the lightweight specification approach advocated by the Alloy designers [15].

The first limitation of Alloy for our specification exercise is in the specification of mathematically oriented structures such as our notion of transition for the specification of Execute Action.

The second limitation is regarding the conditions on character strings like the fact that the scheme of each Category instance must end with a sharp. This constraint cannot be translated in Alloy because it does not support string manipulation operations.

For these two limitations, SMT solvers are probably more suited. For example, Cubicle [16] is a model checker based on a SMT solver that is especially used for the problem of Reachability, i.e., there is always a transition from an initial state to a non-initial state, or from a non-final state to at least one final state (cf. Section 3.10.2). We intend to use Cubicle for this kind of properties. On the other hand, to solve the problem of limited search space in SMT and SAT solvers, and prove that the property is absolutely valid, a higher-order theorem prover is probably much more suited. Therefore, we propose to use mathematical theorem provers, such as the Why3 language [17], whose tool implements 22 demonstrators/solvers such as Coq [18], Isabelle/HOL [19], Z3 [20], etc.).
Chapter 4

Conclusion and Perspectives

We have presented in this deliverable a comprehensive formal specification of OCCI behavior, that covers all the elements of the original informal reference specifications. The formal specification is written in Alloy, a simple and natural (for object-oriented modellers) formal specification language. We have used the Alloy analyzer to check the consistency of the model and several properties.

Compared to the informal one, the formal specification described in this deliverable has several advantages:

- It provides a formal description of the foundations and the operations of the OCCI standard, which are explained in the informal specifications with too many details but ambiguous semantics. Our formal specification removes ambiguities from the informal specification, notably by the specification of the pre and postconditions of the REST operations. We have concentrated in this deliverable on providing a formal equivalent of the full informal OCCI specification.

- It is more abstract, thus more liberal than the informal one in a number of aspects (e.g., the location value in the HTTP request and the return code of the HTTP response). This has no effect except for Cacheability property. Indeed, having not specified the HTTP responses, it is impossible to prove that POST responses can be backed up in cache.

- It provides a truly language-independent specification of the OCCI standard.

We plan to extend this work in several directions, however:

- Extending this specification to describe, formally and abstractly, concepts, constraints and capabilities of OCCI inter-cloud networking, infrastructure, platform, application, service management, cloud monitoring, and autonomic computing, i.e., build a catalogue of OCCI extension models.

- Using this specification to verify OCCI specific properties such as checking the absence of inconsistent state situations between resource instances and corresponding link instances, e.g., Storage is in an offline state and Storagelink is in an active state.

- Using this specification as a basis for code generation and the development of new OCCI implementations (i.e., clients and servers), with formal semantics.

- Extending this specification to generate tests in existing OCCI implementations and check whether they are conform to the properties of our OCCI formal specification.
Bibliography


<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Status</th>
<th>Expected</th>
<th>Dissemination</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2.2.2</td>
<td>Final</td>
<td>2016/11/30</td>
<td>public</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Delivered December 14, 2016


Appendix A

OCCI Specification in Alloy

module OCCI

open util/boolean // for Bool, True and False signs.
open util/ordering [Time]
//open util/ordering [Configuration]
// Base sig for enumerations.
abstract sig EnumBase {}

// Signature for representing time instants.
sig Time ()

// Various OCCI string constants.

let OCCI_SCHEME="http://schemas.ogf.org/occi/core#"
//let OCCI_ENTITY_TERM="entity"
let OCCI_RESOURCE_TERM="resource"
let OCCI_LINK_TERM="link"

In OCCI metamodel:

datatype URI : 'java.lang.String' { serializable };

let URI=String

In OCCI metamodel:

abstract class Category
{
    attribute term : String[1];
    attribute scheme : URI[1];
    attribute title : String[1];
    property attributes : Attribute[*] { ordered composes };
    invariant IdentityUnique : Category.allInstances() ->\isUnique(scheme + term);
    invariant SchemeEndsWithSharp : scheme.substring(scheme.size(), scheme.size()) = '#';
    invariant AttributesNameUnique : attributes ->\isUnique(name);
}

abstract sig Category {
term : one String,
scheme : one URI,
title : lone String,
attributes : set Attribute
}

// OCL invariant IdentityUnique: Category.allInstances()->isUnique(scheme + term);
no disj cl, c2 : Category | sameIdentity[cl, c2]

// The scheme of each Category instance must end with a sharp.
// OCL invariant SchemeEndsWithSharp: scheme.substring(scheme.size(), scheme.size())='#';
// WARNING: No translation in Alloy because no string manipulation operations.

// OCL invariant AttributesNameUnique: attributes->isUnique(name);
no disj a1, a2 : attributes | a1.name = a2.name
// WARNING: Previous constraint has a strong impact on analysis execution time.

pred sameIdentity[cl : Category, c2 : Category]
{ cl.term = c2.term and cl.scheme = c2.scheme
}

// Not used currently.
// pred sameCategories[categories : set Category] {
// one categories.scheme and one categories.term
// }

// Equivalence predicate between two categories.
pred isoCategory[cl, c2 : Category]
{ cl.term = c2.term
cl.scheme = c2.scheme
cl.title = c2.title
cl.attributes = c2.attributes
}

In OCCI metamodel:

class Attribute {
    attribute name : String[1];
    attribute mutable : Boolean[?];
    attribute required : Boolean[?];
    attribute default : String[?];
    attribute description : String[?];
    property type :.ecore::EDataType[?];
    attribute multiple_values : Boolean[?] = 'false';
}

abstract sig DataType {
    name : one String
}

In OCCI metamodel:

class Kind extends Category {
    property parent : Kind[?];
    property actions : Action[*] { ordered composes };  
    property entities : Entity[*] { ordered derived readonly } 
}
// property depends
// property actions
// class Action extends Category {
//   actions : Action[*] { ordered composes } ;
//   property depends : Mixin[*] { ordered } ;
// }

In OCCI metamodel:

class Action extends Category {
   actions : Action[*] { ordered composes } ;
   property depends : Mixin[*] { ordered } ;
}

// WARNING: No translation in Alloy because no string manipulation operations.

// OCL invariant CorrectScheme: let category = oclContainer().oclAsType(Category) in scheme = category.scheme.substring(1, category.scheme.size()-1) + '/act' + category.term + '/action#';

In OCCI metamodel:

class Action extends Category {
   actions : Action[*] { ordered composes } ;
   property depends : Mixin[*] { ordered } ;
}

// The scheme of an Action instance must be the concatenation of both scheme and term of its
category plus the suffix /action#.

OCL invariant CorrectScheme: let category = oclContainer().oclAsType(Category) in scheme = category.scheme.substring(1, category.scheme.size()-1) + '/act' + category.term + '/action#';

// WARNING: No translation in Alloy because no string manipulation operations.

In OCCI metamodel:

class Action extends Category {
   actions : Action[*] { ordered composes } ;
   property depends : Mixin[*] { ordered } ;
}

// The scheme of an Action instance must be the concatenation of both scheme and term of its
category plus the suffix /action#.

OCL invariant CorrectScheme: let category = oclContainer().oclAsType(Category) in scheme = category.scheme.substring(1, category.scheme.size()-1) + '/act' + category.term + '/action#';

// WARNING: No translation in Alloy because no string manipulation operations.

In OCCI metamodel:

class Action extends Category {
   actions : Action[*] { ordered composes } ;
   property depends : Mixin[*] { ordered } ;
}
```ocd
// Equivalence predicate between two Entity atoms.

// invariant KindCompatibleWithOneAppliesOfEachMixin: mixins->forall(m | m.applies->forAll(m | m.applies->notEmpty()) implies m.applies->exists(k | kind->closure(parent)->includes(k))); //
```

```
// Inv.

// In OCCI metamodel:

abstract class Entity {
    attribute id : URI[1];
    property kind : Kind[1];
    property mixins :Mixin[*] { ordered, ordered_composes };
    attribute attributes : AttributeState[*] { ordered_composes };
} //
```

```
abstract sig Entity {
    id : one URI,
    kind : one Kind,
    mixins : set Mixin -> Time,
    attributes : set AttributeState -> Time
} //
```

```
// OCL invariant KindCompatibleWithOneAppliesOfEachMixin: mixins->forall(m | m.applies->forAll(m | m.applies->notEmpty()) implies m.applies->exists(k | kind->closure(parent)->includes(k))); //
```

```
// OCL invariant IdUnique: Entity.allInstances()->isUnique(id);

// WARNING: This constraint is too constraining for dynamic semantics where two entities can have the same id
// If we associate id to Time, we could add this constraint, since 2 entities cannot have the same id only at a time t

// OCL invariant AttributesNameUnique: attributes->isUnique(name);
```

```
// OCL invariant ActionTermUnicity: actions->isUnique(term);
```

```
// OCL invariant NoCyclicInheritance: depends->closure(depends)->excludes(self);
```

```
// invariant AttributesNameNotAlreadyDefinedInDepends: attributes.name->excludesAll({ depends->closure(depends).attributes.name });
```

```
// invariant ActionTermUnicity: actions->isUnique(term);
no disjoint a1, a2 : actions | a1.@term = a2.@term
```

```
// invariant CorrectScheme: let ownerScheme = self, ocIClaims(self) .oclAsType(Extension).
```

```
// invariant NoCyclicInheritance: depends->closure(depends)->excludes(self);
no disjoint a1, a2 : actions | a1.@term = a2.@term
```

```
// WARNING: No translation in Alloy because no string manipulation operations.
```

```
// A Mixin instance must not inherit from itself directly or transitively.
```

```
// invariant NoCyclicInheritance: depends->closure(depends)->excludes(self);
no disjoint a1, a2 : actions | a1.@term = a2.@term
```

```
// invariant CorrectScheme: let ownerScheme = self, ocIClaims(self) .oclAsType(Extension).
```

```
// invariant NoCyclicInheritance: depends->closure(depends)->excludes(self);
no disjoint a1, a2 : actions | a1.@term = a2.@term
```

```
// WARNING: This constraint is too constraining for dynamic semantics where two entities can have the same id
// If we associate id to Time, we could add this constraint, since 2 entities cannot have the same id only at a time t

// OCL invariant KindCompatibleWithOneAppliesOfEachMixin: mixins->forall(m | m.applies->notEmpty()) implies m.applies->exists(k | kind->closure(parent)->includes(k)); //
```

```
// invariant AttributesNameUnique: attributes->isUnique(name);
```
In OCCI metamodel:

class AttributeState {
    attribute name : String[1];
    attribute value : String[1];
}

sig AttributeState {
    name : one String,
    value : one String
}
    // Each attribute state is owned by only one entity, i.e. class Category { property attributes
    // AttributeState[*] { composes }; }
    // one "(Entity< attributes, Time)>[this]

In OCCI metamodel:

class Resource extends Entity {
    links : set Link -> Time
}
    // Each resource is owned by only one configuration, i.e. class Configuration { property
    // resources : Resource[*] { ordered composes }; }
    // one configuration
    // The kind of a Resource instance must inherit from the resource kind instance directly or
    // transitively.
    // OCL invariant ResourceKindIsInParent: kind->closure(parent)->exists(k | k.term = 'resource'
    // and k.scheme = 'http://schemas.ogf.org/occi/core#');
    // one k : kind.*parent | k.term = OCCiRESOURCE_TERM and k.(Category<scheme) = OCCi_SCHEME and
    // k.parent.isEntityKind
    // Make sure that links can not have same id.
    all t : Time | no disj l1, l2 : links.t | 11.@id = 12.@id

    // Equivalence predicate between two Resource atoms.
    pred isoResource[r1 : Resource, r2 : Resource] {
        isoEntity[r1, r2] // same entities
        r1.links = r2.links // same links
    }

In OCCI metamodel:

class Link extends Entity {
    // property source#links : Resource[1];
    // property target : Resource[1];
    // invariant LinkKindIsInParent: kind->closure(parent)->exists(k | k.term = 'link' and k.
    // scheme = 'http://schemas.ogf.org/occi/core#');
    //}

    sig Link extends Entity {
        source : Resource /+one*/ -> Time,
        target : Resource /+one*/ -> Time
    }
    // property links#source : Link[*] { composes }; // property source#links : Resource[1];
    // all t : Time | source.t = "(Resource<:links.t)[this]
    // Added to force sourced link to have a target
    // Avoid sourced link without target
    // one source implies one target
    // The kind of a Link instance must inherit from the link kind instance directly or
    // transitively.
    // OCL invariant LinkKindIsInParent: kind->closure(parent)->exists(k | k.term = 'link' and k.
    // scheme = 'http://schemas.ogf.org/occi/core#');
    // one k : kind.*parent | k.term = OCCi_LINK_TERM and k.(Category<scheme) = OCCi_SCHEME and
    // k.parent.isEntityKind

Deliverable D2.2.2
Expected 2016/11/30
Delivered December 14, 2016
Status Final
Dissemination public
Version 1.0

Delivered
December 14, 2016

Delivered
December 14, 2016

Delivered
December 14, 2016

Delivered
December 14, 2016
In OCCI metamodel:

class Extension {
    attribute name : String [1];
    attribute scheme : URI [1];
    property import : Extension [ ] { ordered composes };
    property kinds : Kind [ ] { ordered composes };
    property mixins : Mixin [ ] { ordered composes };
    attribute types : set EDataType [ ];
    attribute kinds : set Kind [ ];
    attribute mixins : set Mixin [ ];
    attribute name : String [1];
    attribute scheme : URI [1];
    property import : Extension [ ] { ordered composes };
    property kinds : Kind [ ] { ordered composes };
    property mixins : Mixin [ ] { ordered composes };
    attribute types : set EDataType [ ];
    attribute kinds : set Kind [ ];
    attribute mixins : set Mixin [ ];
}

sig Extension {
    l e t = kinds . extension | some e implies e = this
    // WARNING: is it really required?
    l e t = mixins . extension | some e implies e = this
    // WARNING: is it really required?
    l e t = types . extension | some e implies e = this
    // WARNING: is it really required?
}

OCL invariant SchemeUnique : Extension . a l l I n s t a n c e s () -> is Uni que ( scheme );
no disj e1 , e2 : Extension | e1 . @ s c h e m e = e2 . @ s c h e m e

// The scheme of all kinds must be equal to the scheme of the owning Extension instance.
// OCL invariant KindsSchemeValid : kinds -> f o r A l l ( k | k . s c h e m e = s e l f . s c h e m e );
all k : K i n d | k . @ s c h e m e = scheme

// The scheme of all mixins must start with the scheme of the owning Extension instance.
// OCL invariant MixinsSchemeValid : mixins -> f o r A l l ( m | m . s c h e m e . s u b s t r i n g ( 1 , s c h e m e . s i z e ( ) - 1 ) = scheme . s u b s t r i n g ( 1 , s c h e m e . s i z e ( ) - 1 ) );
// WARNING: No translation in Alloy because no string manipulation operations.

// The intersection of term in Kind and term in Mixin is empty.
// OCL invariant TermUnicity : kinds . t e r m . i n t e r s e c t i o n ( mixins . t e r m ) -> is E m p t y ( );
no k i n d s . t e r m & mixins . t e r m

// The parent of all the kinds of an extension must be defined or imported by this extension.
// OCL invariant KindParentLocalOrImported : kinds -> f o r A l l ( parent <> n u l l implies let
// parentExtension = parent . o c l C o n t a i n e r ( ) in parentExtension = self or import -> includes ( parentExtension ) );
all k : K i n d | k . @ s c h e m e = scheme

// All the depends of all the mixins of an extension must be defined or imported by this
// extension.
// OCL invariant MixinDependsLocalOrImported : mixins . depends -> f o r A l l ( let extension = o c l C o n t a i n e r ( ) in extension = self or import -> includes ( extension ) );
all m : M i x i n | m . @ s c h e m e = scheme
Deliverable | D2.2.2
--- | ---
Expected | 2016/11/30
Delivered | December 14, 2016

// WARNING: Not defined in OCCI ecore!!!
this not in import

pred Extension.localOrImported [extension : Extension] {
  extension in (this + this.import) }

fun Kind.extension [] : Extension {
  ~(Extension.<:kinds) [this]
}

fun Mixin.extension [] : Extension {
  ~(Extension.<:mixins) [this]
}

fun DataType.extension [] : Extension {
  ~(Extension.<:types) [this]
}

In OCCI metamodel:

```java
class Configuration {
  |  use : Extension[*] { ordered },
  |  resources : Resource[*] { ordered composes },
  |  invariant AllResourcesKindsInUse: use->includesAll(resources.kind.oclContainer());
  |  invariant AllResourcesMixinsInUse: use->includesAll(resources.mixins.oclContainer());
  |  oclContainer());
  |  invariant AllResourcesLinksKindsInUse: use->includesAll(resources.links.kind.oclContainer());
  |  invariant AllResourcesLinksMixinsInUse: use->includesAll(resources.links.mixins.oclContainer());
  |  invariant AllResourcesLinksTargetsInConfiguration: resources.links.target->forAll(r | r.oclContainer() = self);
  }
```

```java
sig Configuration {
  use : Extension[*] { ordered },
  resources : Resource[*] { ordered composes },
  invariant AllResourcesKindsInUse: use->includesAll(resources.kind.oclContainer());
  invariant AllResourcesMixinsInUse: use->includesAll(resources.mixins.oclContainer());
  oclContainer());
  invariant AllResourcesLinksKindsInUse: use->includesAll(resources.links.kind.oclContainer());
  invariant AllResourcesLinksMixinsInUse: use->includesAll(resources.links.mixins.oclContainer());
  invariant AllResourcesLinksTargetsInConfiguration: resources.links.target->forAll(r | r.oclContainer() = self);
  }
```

```java
fun Resource.configuration [] : Configuration {
  "(Configuration.<:resources.Time) [this]
}
```

```java
fun Configuration.findUserMixin [mixinScheme, mixinTerm : String , t : Time] : lone Mixin {
  { m : this.mixins.t | m.scheme = mixinScheme and m.term = mixinTerm }
}
```

State Machine adding
```ocaml
sig StateMachine {
    category: one (Kind + Mixin),
    attribute: one Attribute,
    states: set State,
    transitions: set Transition
}

// Only one State Machine for one kind or mixin.
no disj SM1, SM2 : StateMachine | SM1.@category = SM2.@category

// No state shared between State Machines.
no disj SM1, SM2 : StateMachine | some SM1.@states & SM2.@states

// The attribute in the State Machine is in the attributes of its category.
attribute in category.attributes

// There is only one initial state.
one state : states | state.isInitial = True

// There is at least one final state.
some state : states | state.isFinal = True

// All actions are present in the State Machine.
all action : category.(Kind < actions + Mixin < actions) | action in transitions.event

// Transition target is a state in the State Machine.
transitions.target in states

// Transition source is a state in the State Machine.
transitions.source in states

// All states are present in the State Machine.
all state : State | state in states
}

sig State {
    name: one String,
    entities: set Entity,
    exit: one State,
    description: one String,
    isInitial: one Bool,
    isFinal: one Bool
}

// Each state is owned by one State Machine.
one ~(StateMachine.<states).[this]

sig Transition {
    event: one Action,
    condition: one State,
    action: one Action,
    source: one State,
    target: one State
}

// No transition shared between State Machines.
no disj SM1, SM2 : StateMachine | some SM1.transitions & SM2.transitions

// Each transition owned by one State Machine.
one ~(StateMachine.<transitions).[this]

// Events are actions of the State Machine category
all SM : StateMachine | event in SM.category.(Kind < actions + Mixin < actions)

// Check consistency, i.e., there is at least one instance of this model.
run Consistency {} for 10

// Instances containing one mixin with some actions.
run OneMixinWithActions { some m : Mixin | some m.actions }

// Instances containing some configurations with extensions.
run SomeConfigurationWithUse { some c : Configuration | some c.use }

// End of the static specification.

// Start of the dynamic specification.

// Get all the categories, i.e., the kinds and the mixins in a configuration.
pred GetAllCategories[config : Configuration, categories : set Category, t, t' : Time] {
    // preconditions
```

categories = config.use.(Extension<:kinds + Extension<:mixins) + config.mixins.t
// postconditions
// As it is a get then the state must not be changed.
config.resources.t' = config.resources.t
config.mixins.t' = config.mixins.t
}
run GetAllCategories
{ one config : Configuration , categories : Category , t : Time |
GetAllCategories[config , categories , t , t . next] }
for 3 but exactly 1 Configuration , exactly 2 Time
assert GetAllCategoriesIsSafe
{ all config : Configuration , categories : Category , t , t' : Time |
GetAllCategories[config , categories , t , t . next] implies
config.resources.[t.next] = config.resources.[t.next.next] and config.mixins.[t.next] = config.mixins.[t.next.next] }
check GetAllCategoriesIsSafe for 10
assert GetAllCategoriesIsIdempotent
{ all config : Configuration , categories : Category , t , t' : Time |
GetAllCategories[config , categories , t , t . next] implies
categories = config.mixins.t }
check GetAllCategoriesIsIdempotent for 10
assert GetAllCategoriesOnConfigurationWithoutExtensionsAndUserMixinsReturnsNoCategory
{ all config : Configuration , categories : Category , t , t' : Time |
no config.use and no config.mixins.t and GetAllCategories[config , categories , t , t'] implies
no categories }
check GetAllCategoriesOnConfigurationWithoutExtensionsAndUserMixinsReturnsNoCategory for 10
assert GetAllCategoriesOnConfigurationWithoutExtensionsReturnsUserMixins
{ all config : Configuration , categories : Category , t , t' : Time |
no config.use and GetAllCategories[config , categories , t , t'] implies categories = config.mixins.t }
check GetAllCategoriesOnConfigurationWithoutExtensionsReturnsUserMixins for 10
assert GetAllCategoriesOnConfigurationWithoutUserMixinsReturnsAllExtensionsKindsAndMixins
{ all config : Configuration , categories : Category , t , t' : Time |
no config.mixins.t and GetAllCategories[config , categories , t , t'] implies categories = config.use.kinds + config.use.mixins }
check GetAllCategoriesOnConfigurationWithoutUserMixinsReturnsAllExtensionsKindsAndMixins for 10

---

Create resource.
// Ajouter liste des attributs (nom, valeur), liste de links
pred CreateResource[config : Configuration , resourceId : URI , resourceKind : Kind , resourceMixins :
set Mixin , resourceAttributes : set AttributeState , t , t' : Time] |
{} // preconditions at instant t
// no resource with the same id
no resource . config.resources . t [ resource . id = resourceId ]
// an extension contains kind
resourceKind in config.use.kinds
// each mixin is contained by an extension
resourceMixins in config.use.mixins
// postconditions at instant t'
// one resource with these id, kind, and mixins
one resource : Resource |
resource . id = resourceId
resource . kind = resourceKind
resource . mixins . t = resourceMixins
resource . attributes . t = resourceAttributes
config.resources . t' = config.resources . t + resource
// mixins unchanged
config.mixins . t' = config.mixins . t
}
run CreateResource
{ one config : Configuration | one resourceId : URI , kind : Kind , mixins : Mixin , attributes : AttributeState , t , t' : Time |
CreateResource[config , resourceId , kind , mixins , attributes , t , t . next] }
for 3 but exactly 1 Configuration , exactly 2 Time
assert CreateResourceIsIdempotent
{ all config : Configuration , resourceId : URI , kind : Kind , mixins : Mixin , attributes : AttributeState , t , t' : Time |
CreateResource[config , resourceId , kind , mixins , attributes , t , t . next] implies config.resources.[t.next].links.[t.next] =
check CreateResourceIsIdempotent for 10
assert CreateResourceImpliesResourceAddedToKind
{
  all config : Configuration, resourceId : URI, kind : Kind, mixins : Mixin, attributes : AttributeState, t : Time |
  CreateResource[config, resourceId, kind, mixins, attributes, t, t.next] implies resourceId in config.resources.(t.next).id and kind in config.resources.(t.next).#kind
}
}
check CreateResourceImpliesResourceAddedToKind for 10

// Retrieve resource.

pred RetrieveResource[config : Configuration, resourceId : URI, t, t' : Time]
{
  // preconditions at instant t
  // resource is returned
  one resource : config.resources.t { resource.id = resourceId }
  // postconditions at instant t'
  one resource : config.resources.t' { resource.id = resourceId |
    resource.mixins.t = resource.mixins.t |
    resource.attributes.t' = resource.attributes.t
  }
  // resources unchanged
  config.resources.t' = config.resources.t
  // mixins unchanged
  config.mixins.t' = config.mixins.t
}
run RetrieveResource
{
  one config : Configuration, resourceId : URI, t : Time |
  RetrieveResource[config, resourceId, t, t.next]
}
for 3 but exactly 1 Configuration, exactly 2 Time
assert RetrieveResourceIsSafe
{
  all config : Configuration, resourceId : URI, t : Time |
  RetrieveResource[config, resourceId, t, t.next] implies
  config.resources.(t.next) = config.resources.t and config.mixins.(t.next) = config.mixins.t
  and one resource : config.resources.(t.next) { resource.id = resourceId |
    resource.mixins.(t.next) = resource.mixins.t |
    resource.links.(t.next) = resource.links.(t.next)
  }
}
check RetrieveResourceIsSafe for 10
assert RetrieveResourceIsIdempotent
{
  all config : Configuration, resourceId : URI, t : Time |
  RetrieveResource[config, resourceId, t, t.next] implies
  config.resources.(t.next) = config.resources.(t.next) and config.mixins.(t.next) = config.mixins.(t.next)
  and one resource : config.resources.(t.next) { resource.id = resourceId |
    resource.mixins.(t.next) = resource.mixins.(t.next)
    resource.links.(t.next) = resource.links.(t.next)
  }
}
check RetrieveResourceIsIdempotent for 10
assert CreateResourceAndRetrieveResourceAreSequential
{
  all config : Configuration, resourceId : URI, resourceKind : Kind, resourceMixins : set Mixin, attributes : AttributeState, t : Time |
  CreateResource[config, resourceId, resourceKind, resourceMixins, attributes, t, t'] and
  RetrieveResource[config, resourceId, t, t.next]
  implies one resource : config.resources.(t.next) { resource.id = resourceId |
    resource.kind = resourceKind and resource.mixins.t' = resourceMixins
  }
}
check CreateResourceAndRetrieveResourceAreSequential for 10 //but exactly 1 Configuration, exactly 2 Resource, exactly 3 Time
// Update an attribute of a resource.

pred UpdateResource[config : Configuration, resourceId : URI, attribute1 : AttributeState, attribute2 : AttributeState, t, t' : Time]
{
  // preconditions at instant t
  // attribute1 is different from attribute2
  attribute1 ≠ attribute2 |
  // one resource with this id and this attribute
  one resource : config.resources.t { resource.id = resourceId ∧ resource.attributes.t = attribute1 }
  // postconditions at instant t'

// update resource with this id and this attribute
one resource : config.resources.t {
  resource.attributes.t' = resource.attributes.t ++ attribute2
}
// use unchanged
config.resources.t' = config.resources.t
// mixins unchanged
config.mixins.t' = config.mixins.t
}
run UpdateResource
{
  one config : Configuration, resourceId : URI, attribute1 : AttributeState, attribute2 : AttributeState, t : Time |
  UpdateResource[config, resourceId, attribute1, attribute2, t, t.next] implies
  UpdateResource[config, resourceId, attribute1, attribute2, t.next, t.next.next]
}
check UpdateResourceIsIdempotent for 10

// Delete resource.
pred DeleteResource:config : Configuration, resourceId : URI, t, t': Time |
{
  // preconditions at instant t
  one resource : config.resources.t
  // postconditions at instant t'
  remove resource with this id
  resource.id = resourceId
}
run DeleteResource
{
  one config : Configuration, resourceId : URI, t, t : Time |
  DeleteResource[config, resourceId, t, t.next] implies
  DeleteResource[config, resourceId, t.next, t.next.next]
}
check DeleteResourceIsIdempotent for 10

check DeleteResourceImpliesResourceRemovedFromConfiguration for 10

assett CreateResourceAndDeleteResourceAreReversible
{
  all config : Configuration, resourceId : URI, resourceKind : Kind, resourceMixins : Mixin, resourceAttributes : AttributeState, t, t': Time |
  CreateResource[config, resourceId, resourceKind, resourceMixins, resourceAttributes, t, t'] implies DeleteResource[config, resourceId, t, t', t.next]
}
check CreateResourceAndDeleteResourceAreReversible for 10

pred CreateLink:config : Configuration, linkId : URI, linkKind : Kind, linkSource : Resource, linkTarget : Resource, linkAttributes : set AttributeState, t, t': Time |
{
  // preconditions at instant t
  no link with the same id, source and target
  no link : config.resources.t.links.t {
    link.id = linkId
    link.source = linkSource
    link.target = linkTarget
  }
  // no extension contains kind
  linkKind in config.use.kinds
  // a source is contained in the resources
  linkSource in config.resources.t
  // a target is contained in the resources
}
linkTarget in config.resources.t
// link does have a source neither a target

// postconditions at instant t'
one link with these id, kind, source, target and attributes
one link : Link {
  link.id = linkId
  link.kind = linkKind
  link.source.t' = linkSource
  link.target.t' = linkTarget
  link.attributes.t' = linkAttributes
  linkSource.links.t' = linkSource.links.t + link
}
// resources unchanged
config.resources.t' = config.resources.t
// mixins unchanged
config.mixins.t' = config.mixins.t

run CreateLink
{
  one config : Configuration, linkId : URI, linkKind : Kind, source : Resource | one target : Resource, attributes : AttributeState, t : Time |
  CreateLink(config, linkId, linkKind, source, target, attributes, t, t.next)
} for 3 but exactly 1 Configuration, exactly 2 Time
assert CreateLinkIsIdempotent
{
  CreateLink(config, linkId, linkKind, source, target, attributes, t, t.next) implies config.resources.(t.next).links.(t.next) = config.resources.(t.next.next).links.(t.next.next)
}
check CreateLinkIsIdempotent for 10
assert CreateLinkImpliesLinkAddedToKind
{
  all config : Configuration, linkId : URI, linkKind : Kind, source : Resource, target : Resource, attributes : AttributeState, t, t' : Time |
  CreateLink(config, linkId, linkKind, source, target, attributes, t, t') implies linkId in config.resources.t'.links(t'.next) and linkKind in config.use.kinds
}
check CreateLinkImpliesLinkAddedToKind for 10

// Retrieve link.

pred RetrieveLink[config : Configuration, linkId : URI, t, t' : Time]
{
  // preconditions at instant t
  one link : config.resources.t.links.t (link.id = linkId)
  // postconditions at instant t'
  one link : config.resources.t'.links.t' (link.id = linkId
  link.source.t' = link.source.t
  link.target.t' = link.target.t
  link.mixins.t' = link.mixins.t
  link.attributes.t' = link.attributes.t)
  // resources unchanged
  config.resources.t' = config.resources.t
  // mixins unchanged
  config.mixins.t' = config.mixins.t
}
run RetrieveLink
{
  one config : Configuration, linkId : URI, t : Time |
  RetrieveLink(config, linkId, t, t.next)
} for 3 but exactly 1 Configuration, exactly 2 Time
assert RetrieveLinkIsSafe
{
  all config : Configuration, linkId : URI, t : Time |
  RetrieveLink(config, linkId, t, t.next) implies
  config.resources.(t.next) = config.resources.t
  and config.mixins.(t.next) = config.mixins.t
  and one link : config.resources.t.links.(t.next) (link.id = linkId
  link.source.(t.next) = link.source.t
  and link.target.(t.next) = link.target.t)
}
check RetrieveLinkIsSafe for 10
assert RetrieveLinkIsIdempotent
{
  all config : Configuration, linkId : URI, t : Time |
  RetrieveLink(config, linkId, t, t.next) implies
  config.resources.(t.next) = config.resources.(t.next.next)
  and config.mixins.(t.next) = config.mixins.(t.next.next)
  and one link : config.resources.t.links.(t.next) (link.id = linkId
  link.source.(t.next) = link.source.t
  and link.target.(t.next) = link.target.t
  }

link.id = linkId
link.source.(t.next.next) = link.source.(t.next)
and link.target.(t.next.next) = link.target.(t.next)
}
)
check RetrieveLinkIsIdempotent for 10
assert CreateLinkAndRetrieveLinkAreSequential
{
all config : Configuration, linkId : URI, linkKind : Kind, linkSource : Resource, linkTarget:
Resource, linkAttributes : set AttributeState, t : Time, t’, t.next, t’’ : t’.next | CreateLink[config, linkId, linkKind, linkSource, linkTarget, linkAttributes, t, t’] and
RetrieveLink[config, linkId, t’, t’’]
implies one link : config.resources.t.links.t’’ {
link.id = linkId
and link.kind = linkKind
and link.source.t’’ = linkSource
and link.target.t’’ = linkTarget
}
)
check CreateLinkAndRetrieveLinkAreSequential for 10
// Delete link.
pred DeleteLink[config : Configuration, linkId : URI, t, t’ : Time]
{
// preconditions at instant t
// one link with this id
one link : config.resources.t.links.t | link.id = linkId
// postconditions at instant t’
// remove link with this id
one link : config.resources.t.links.t {
link.id = linkId
link.source.t.links.t’ = link.source.t.links.t - link
no link.source.t’
no link.target.t’
}
// resource unchanged
config.resources.t’ = config.resources.t
// mixins unchanged
config.mixins.t’ = config.mixins.t
}
run DeleteLink
{
one config : Configuration, linkId : URI, t : Time |
DeleteLink[config, linkId, t, t.next] implies
}
for 3 but exactly 1 Configuration, exactly 2 Time, exactly 1 Link
assert DeleteLinkIsIdempotent
{
all config : Configuration, linkId : URI, t : Time |
DeleteLink[config, linkId, t, t.next] implies
config.resources.(t.next).links.(t.next) = config.resources.(t.next.next).links.(t.next.next)
}
check DeleteLinkIsIdempotent for 10
assert DeleteLinkImpliesLinkRemovedFromConfiguration
{
all config : Configuration, linkId : URI, t, t’ : Time {
DeleteLink[config, linkId, t, t’] implies linkId not in config.resources.t’.links.t’.id
}
}
check DeleteLinkImpliesLinkRemovedFromConfiguration for 10
assert CreateLinkAndDeleteLinkAreReversible
{
all config : Configuration, linkId : URI, linkKind : Kind, source : Resource, target : Resource,
attributes : AttributeState, t : Time, t : t.next {
CreateLink[config, linkId, linkKind, source, target, attributes, t, t’] implies
DeleteLink[config, linkId, t, t’]
}
)
check CreateLinkAndDeleteLinkAreReversible for 10
// Update an attribute of a link.
pred UpdateLink[config : Configuration, linkId : URI, attribute1 : AttributeState, attribute2 : AttributeState, t, t’ : Time]
{
// preconditions at instant t
attribute1 is different from attribute2
// one link with this id and this attribute
one link : config.resources.t.links.t | link.id = linkId ∧ link.attributes.t = attribute1
// postconditions at instant t’
// link with this id
one link : config.resources.t.links.t {
link.attributes.t’ = link.attributes.t ++ attribute2
}
// resource unchanged
config.resources.t’ = config.resources.t
// mixins unchanged
config.mixins.t’ = config.mixins.t
}
run UpdateLink
{ one config : Configuration , linkId : URI , attribute1 : AttributeState , attribute2 : AttributeState , t : Time |
    UpdateLink [ config , linkId , attribute1 , attribute2 , t , t . next ]
} for 3 but exactly 1 Configuration , exactly 2 Time
assert UpdateLinkIsIdempotent
{ all config : Configuration , linkId : URI , attribute1 : AttributeState , attribute2 : AttributeState , t : Time |
    UpdateLink [ config , linkId , attribute1 , attribute2 , t , t . next ] implies
    UpdateLink [ config , linkId , attribute1 , attribute2 , t . next , t . next . next ]
}
check UpdateLinkIsIdempotent for 10

run AddUserMixin
{ one config : Configuration , mixinScheme : String , mixinTerm : String , t , t ' : Time |
    // preconditions
    no config . findUserMixin [ mixinScheme , mixinTerm , t ]
    // postconditions
    one m : Mixin { m . scheme = mixinScheme 
        m . term = mixinTerm 
        config . mixins . t ' = config . mixins . t + m }
    config . resources . t ' = config . resources . t
} run AddUserMixin
{ one config : Configuration , mixinScheme , mixinTerm : String , t , t ' : Time |
    AddUserMixin [ config , mixinScheme , mixinTerm , t , t ' ]
} for 3 but exactly 1 Configuration , exactly 2 Time
assert AddUserMixinDoesNotModifyConfigurationExtensionsAndResources
{ all config : Configuration , mixinScheme , mixinTerm : String , t , t ' : Time |
    AddUserMixin [ config , mixinScheme , mixinTerm , t , t ' ] implies
    config . resources . t = config . resources . t ' 
}
check AddUserMixinDoesNotModifyConfigurationExtensionsAndResources for 10
assert AddUserMixinCanNotAddAlreadyExistingMixin
{ all config : Configuration , mixinScheme , mixinTerm : String , t , t ' : Time |
    one config . findUserMixin [ mixinScheme , mixinTerm , t ] implies not AddUserMixin [ config , mixinScheme , mixinTerm , t , t ' ]
}
check AddUserMixinCanNotAddAlreadyExistingMixin for 10
assert AddUserMixinDoesTheWork
{ all config : Configuration , mixinScheme , mixinTerm : String , t , t ' : Time |
    AddUserMixin [ config , mixinScheme , mixinTerm , t , t ' ] implies one config . findUserMixin [ mixinScheme , mixinTerm , t , t ' ]
}
check AddUserMixinDoesTheWork for 10

run RemoveUserMixin
{ one config : Configuration , mixinScheme , mixinTerm : String , t , t ' : Time |
    // preconditions
    one config . findUserMixin [ mixinScheme , mixinTerm , t ]
    // postconditions
    one m : Mixin { m . scheme = mixinScheme 
        m . term = mixinTerm 
        config . mixins . t ' = config . mixins . t + m }
    config . resources . t ' = config . resources . t
} run RemoveUserMixin
{ one config : Configuration , mixinScheme , mixinTerm : String , t , t ' : Time |
    RemoveUserMixin [ config , mixinScheme , mixinTerm , t , t ' ]
} for 3 but exactly 1 Configuration , exactly 2 Time
assert RemoveUserMixinDoesNotModifyConfigurationExtensionsAndResources
{ all config : Configuration , mixinScheme , mixinTerm : String , t , t ' : Time |
    RemoveUserMixin [ config , mixinScheme , mixinTerm , t , t ' ] implies
    config . resources . t = config . resources . t ' 
}
check RemoveUserMixinDoesNotModifyConfigurationExtensionsAndResources for 10
assert RemoveUserMixinCanNotRemoveNotExistingMixin
{ all config : Configuration , mixinScheme , mixinTerm : String , t , t ' : Time |
no config.findUserMixin[config, mixinScheme, mixinTerm, t] implies not RemoveUserMixin[config, mixinScheme, mixinTerm, t, t']
)
check RemoveUserMixinCanNotRemoveNotExistingMixin for 10
assert RemoveUserMixinDoesTheWork
{
  all config : Configuration, mixinScheme, mixinTerm : String, t, t' : Time |
  RemoveUserMixin[config, mixinScheme, mixinTerm, t, t'] implies no config.findUserMixin[config, mixinScheme, mixinTerm, t']
}
check RemoveUserMixinDoesTheWork for 10
check AddUserMixinAndRemoveUserMixinAreReversibleOperations_v1
{
  all config : Configuration, mixinScheme, mixinTerm : String, t, t' : Time |
  ( AddUserMixin[config, mixinScheme, mixinTerm, t, t'] and RemoveUserMixin[config, mixinScheme, mixinTerm, t', t''] ) implies config.mixins.t = config.mixins.t''
}
check AddUserMixinAndRemoveUserMixinAreReversibleOperations_v1 for 10
check AddUserMixinAndRemoveUserMixinAreReversibleOperations_v2
{
  all config : Configuration, mixinScheme, mixinTerm : String, t, t', t'' : Time |
  ( AddUserMixin[config, mixinScheme, mixinTerm, t', t''] and RemoveUserMixin[config, mixinScheme, mixinTerm, t, t'] ) implies config.mixins.t = config.mixins.t''
}
check AddUserMixinAndRemoveUserMixinAreReversibleOperations_v2 for 10

// Associate mixin To resource.
pred AssociateMixinToResource [config : Configuration, resourceId : URI, mixinTerm : String, mixinScheme : one URI, t, t' : Time]
{
  // preconditions at instant t
  // one resource with this id
  one resource : config.resources.t { resource_id = resourceId
  // one mixin in extension with these term and scheme
  one mixin : Extension.mixins { mixin.term = mixinTerm
  mixin.scheme = mixinScheme
  }
  // postconditions at instant t
  // associate mixin to resource
  resource.mixins.t'' = resource.mixins.t + mixin
  }
  // attributes unchanged
  config.resources.t'' . attributes.t'' = config.resources.t . attributes.t
}
run AssociateMixinToResource
{
  one config : Configuration, resourceId : URI, mixinTerm : String | one mixinScheme : URI, t : Time |
  AssociateMixinToResource [config, resourceId, mixinTerm, mixinScheme, t, t.next]
}
for 10 assert AssociateMixinToResourceIsIdempotent
{
  all config : Configuration, resourceId : URI, kind : Kind, mixins : Mixin, attributes : AttributeState, t : Time |
  all config : Configuration, resourceId : URI, mixinTerm : String, mixinScheme : URI, t : Time |
  AssociateMixinToResource [config, resourceId, mixinTerm, mixinScheme, t, t.next] implies config.resources.(t.next) = config.resources.(t.next.next)
  and config.mixins.(t.next) = config.mixins.(t.next.next)
}
check AssociateMixinToResourceIsIdempotent for 10

// Dissociate mixin From resource.
pred DissociateMixinFromResource [config : Configuration, resourceId : URI, mixinTerm : String, mixinScheme : URI, t, t' : Time]
{
  // preconditions at instant t
  // one resource with this id
  one resource : config.resources.t { resource_id = resourceId
  // one mixin with this term and scheme applied to a resource
  one mixin : resource.mixins.t { mixin.term = mixinTerm
  mixin.scheme = mixinScheme
  }
  // postconditions at instant t
  // dissociate mixin from resource
  resource.mixins.t'' = resource.mixins.t - mixin
  }
  // attributes unchanged
  config.resources.t'' . attributes.t'' = config.resources.t . attributes.t
}
run DissociateMixinFromResource
{ one config : Configuration, resourceId : URI, mixinTerm : String, mixinScheme : URI, t : Time | DissociateMixinFromResource [ config, resourceId, mixinTerm, mixinScheme, t, t.next ] }
} for 3 but exactly 1 Configuration, 1 Resource, exactly 2 Time

assert DissociateMixinFromResourceIsIdempotent
{ all config: Configuration, resourceId: URI, mixinTerm: String, mixinScheme: URI, t: Time | DissociateMixinFromResource [ config, resourceId, mixinTerm, mixinScheme, t, t.next ] implies
  config.resources.(t.next) = config.resources.(t.next.next)
  and config.mixins.(t.next) = config.mixins.(t.next.next) }
}

assert DissociateMixinFromResourceIsIdempotent for 10

check DissociateMixinFromResourceIsIdempotent for 10

assert AssociateMixinToResourceAndDissociateMixinFromResourceAreReversible
{ all config: Configuration, resourceId: URI, mixinTerm: String, mixinScheme: URI, t: Time, t': t.next
  } AssociateMixinToResource [ config, resourceId, mixinTerm, mixinScheme, t, t ] implies
DissociateMixinFromResource [ config, resourceId, mixinTerm, mixinScheme, t', t ]
}

assert AssociateMixinToResourceAndDissociateMixinFromResourceAreReversible for 10

check AssociateMixinToResourceAndDissociateMixinFromResourceAreReversible for 10

// Retrieve collection.

pred RetrieveCollection [ config : Configuration, scheme : URI, term : String, entities : Entity, t , t' : Time ]
{ // preconditions
  one Extension with this scheme
  one extension: config.use {
    scheme = extension.scheme
  }
  // one kind with this scheme
  one k : Kind {
    k = extension.kinds
  }
  // postconditions
  // entities with this kind
  entities = k.setEntities.t'
  }
} // As it is a retrieve then the state must not be changed.

config.resources.t' = config.resources.t
config.mixins.t' = config.mixins.t

run RetrieveCollection
{ one config: Configuration, scheme: URI, term: String, entities: Entity, t: Time | RetrieveCollection [ config, scheme, term, entities, t, t.next ]
  } for 3 but exactly 1 Configuration, exactly 2 Time

assert RetrieveCollectionIsSafe
{ all config: Configuration, scheme: URI, term: String, entities: Entity, t: Time | RetrieveCollection [ config, scheme, term, entities, t, t.next ] implies
  config.resources.t = config.resources.(t.next)
  and config.mixins.t = config.mixins.(t.next) }

check RetrieveCollectionIsSafe for 10

assert RetrieveCollectionOnConfigurationWithoutExtensionsAndUserMixinsReturnsNoCategory
{ all config: Configuration, categories: Category, t, t': Time | no config.use and no config.mixins.t and GetAllCategories [ config, categories, t, t' ] implies no categories }

check RetrieveCollectionOnConfigurationWithoutExtensionsAndUserMixinsReturnsNoCategory for 10

assert RetrieveCollectionOnConfigurationWithoutExtensionsReturnsUserMixins
{ all config: Configuration, categories: Category, t, t': Time | no config.use and GetAllCategories [ config, categories, t, t' ] implies categories = config.mixins.t }

check RetrieveCollectionOnConfigurationWithoutExtensionsReturnsUserMixins for 10

assert RetrieveCollectionOnConfigurationWithoutUserMixinsReturnsAllExtensionsKindsAndMixins
{ all config: Configuration, categories: Category, t, t': Time | no config.mixins.t and GetAllCategories [ config, categories, t, t' ] implies categories = config.use .kinds + config.use.mixins }

check RetrieveCollectionOnConfigurationWithoutUserMixinsReturnsAllExtensionsKindsAndMixins for 10

// Launch an action on a resource or link.

{ // preconditions
  action is an event of the SM transitions
  action in SM.transitions.event
  // the condition of a transition of a SM is a state of this SM
  SM.transitions.condition in SM.states
  }
// the entities of a state are entities of the resource kind
SM.states.entities in resource.kind.entities.t
// the exit state is contained in the states of the SM
one exit : State | exit = SM.states.<:exit
// postconditions
// the source a state at t was the target state at t
SM.transitions.source = SM.transitions.target
}

run ExecuteAction
{
  one config : Configuration, action : Action, resource : Resource,
  SM : StateMachine, t : Time |
  ExecuteAction[config, action, resource, SM, t, t.next]
} for 3 but exactly 1 Configuration, exactly 2 Time

// End of file.