A Formal and Tooled Framework for Managing Everything as a Service

www.occiware.org

Deliverable D2.1.1

OCCIware Initial Technical Architecture

OCCIware project is supported by Fond national pour la Société Numérique (FSN) and the following competitive clusters Systematic, Minalogic, PICOM, Images & Réseaux and Solutions Communicantes Sécurisées.
Abstract
The OCCLware project aims at building a comprehensive, yet modular software engineering toolchain dedicated to service-oriented applications. In this deliverable we introduce the global technical architecture of the project. In particular, we describe the role and interfaces of each technical task.

Tags
architecture, OCCI, OCCLware, meta-model, DSL, REST, model-driven runtime
### Document versioning

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Prepare deliv. plan</td>
<td>Jean Parpaillon</td>
</tr>
<tr>
<td>0.2</td>
<td>Add connectors section</td>
<td>Fawaz Paraiso</td>
</tr>
<tr>
<td>0.3</td>
<td>Add studio and decision support section</td>
<td>William Piers</td>
</tr>
<tr>
<td>0.4</td>
<td>Add models section</td>
<td>Jean Parpaillon</td>
</tr>
<tr>
<td>0.5</td>
<td>Introduction</td>
<td>Jean Parpaillon</td>
</tr>
<tr>
<td>0.6</td>
<td>DSL</td>
<td>Jean Parpaillon</td>
</tr>
<tr>
<td>0.7</td>
<td>Generators &amp; Connectors</td>
<td>Marc Dutoo</td>
</tr>
<tr>
<td>0.8</td>
<td>Corrections</td>
<td>Noël Plouzeau</td>
</tr>
<tr>
<td>0.9</td>
<td>Meta-model section</td>
<td>Olivier Barais</td>
</tr>
<tr>
<td>0.10</td>
<td>Typos / fixes</td>
<td>Codé Diop</td>
</tr>
<tr>
<td>1.0</td>
<td>Ready for delivery</td>
<td>Jean Parpaillon</td>
</tr>
</tbody>
</table>

### Document reviews

<table>
<thead>
<tr>
<th>Review</th>
<th>Date</th>
<th>Ver.</th>
<th>Auditor(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan</td>
<td>June 8th, 2015</td>
<td>0.1</td>
<td>Jean Parpaillon</td>
<td></td>
</tr>
<tr>
<td>Brouillon</td>
<td>July 22nd</td>
<td>0.9</td>
<td>Jean Parpaillon</td>
<td></td>
</tr>
<tr>
<td>Validation</td>
<td>July 23rd</td>
<td>0.10</td>
<td>Codé Diop</td>
<td></td>
</tr>
<tr>
<td>BAT</td>
<td>July 23rd</td>
<td>1.0</td>
<td>Jean Parpaillon</td>
<td></td>
</tr>
</tbody>
</table>
Contents

1 Introduction .......................................................... 4
   1.1 Objectives ..................................................... 4
   1.2 Challenges .................................................... 4
   1.3 Overall Architecture ......................................... 4

2 Conceptual Foundations ............................................ 7
   2.1 Meta-Model .................................................... 7
   2.2 Domain Specific Languages ................................... 8
      2.2.1 Rationale ................................................. 8
      2.2.2 OCCI DSL - Structural Part ................................ 8
      2.2.3 OCCI DSL - Behaviour Part ................................. 9
      2.2.4 A Particular Model Specific Language: QoS .................. 10
   2.3 Models ........................................................ 10

3 Eclipse Toolchain ................................................ 12
   3.1 OCCI Studio .................................................. 12
   3.2 Simulator ..................................................... 15
   3.3 Decision Support Tool for Cloud Services Transition ........... 16
   3.4 Generators and Connectors .................................... 16
      3.4.1 Generators ............................................. 17
      3.4.2 Connectors ............................................. 17

4 Runtime Support .................................................. 19

5 Conclusion ........................................................ 20
List of Figures

1.1 OCCIware Overall Technical Architecture ................................................. 6
2.1 OCCI Core Diagram .................................................................................. 9
2.2 OCCI Runtime Model ............................................................................... 11
3.1 The Extension Designer ......................................................................... 13
3.2 The OCCI Extension Textual Editor ......................................................... 13
3.3 Rendering on Github of an OCCI Extension generated documentation .... 14
3.4 The Docker modeler ................................................................................. 15
4.1 OCCI Runtime Architecture .................................................................... 19
Chapter 1

Introduction

1.1 Objectives

The objective of OCCIware is to build a formal, model-driven software engineering toolchain dedicated to service oriented software, addressing cross-domain and cross-vendor engineering concerns.

Neither model-driven toolchain, service oriented software development dedicated toolchain and interoperability frameworks are innovative on their own. Model-driven software engineering is extensively used in complex systems development or critical applications and the Eclipse Modeling Framework is dedicated to that. Regarding service oriented software development, it exists a large diversity of dedicated framework like Django REST Framework\textsuperscript{2} for Python, Jersey\textsuperscript{4} or RESTX\textsuperscript{5} for Java, webmachine\textsuperscript{12} or cowboy\textsuperscript{11} for erlang, just to cite a few. Finally, many cloud computing interoperability frameworks have been developed, addressing one or several domains (amongst IaaS, PaaS or SaaS), as libraries (jClouds\textsuperscript{14}) or service themselves (CompatibleOne\textsuperscript{14}).

The innovation targeted in OCCIware is the combination of these scientific and technical tools into an homogeneous toolchain.

1.2 Challenges

The integration of these different disciplines and tools, as well as their reuse by targeted audiences implies clear interfaces and pivot format to be defined.

For instance, while the Eclipse Modeling Framework can be considered as a \textit{de facto} standard technology in meta-modeling communities, existing OCCI implementations does not use this framework. Hence, in order to reuse these runtimes and their associate ecosystems (connectors, developers, etc.), a data format understandable by these runtimes as well as generated by the OCCIware Studio needs to be defined and documented.

Hence, this document aims at defining a first draft of the components and associated interfaces of the project. It is planned that these artefacts will evolve by the time use case developers will provide feedback regarding the architecture.

1.3 Overall Architecture

As illustrated on figure \ref{fig:architecture}, the different components of the project can be divided into three categories, related to the application development process steps.
THINK: these are conceptual tools allowing the description of applications independently from
the implementation; they include meta-model (see 2.1), non-functional models (see 2.3)
and domain specific languages (see 2.2);

DESIGN: design tools offers developers a way to manipulate conceptual tools graphically or
textually (see 3.1), then transforming them into target languages: runtime configurations,
documentation, simulations (see 3.2), decision support tools (see 3.3), etc.;

EXECUTE: finally, execution components implements the conceptual tools and are able to
execute the designed applications.
Figure 1.1: OCCIware Overall Technical Architecture
Chapter 2

Conceptual Foundations

2.1 Meta-Model

In software engineering, the use of models is more and more recommended in the trend of the Model Driven Engineering. This should be contrasted with the classical code-based development techniques. The MDE approach is meant to increase productivity by maximizing compatibility between systems (via reuse of standardized models), simplifying the process of design (via models of recurring design patterns in the application domain), and promoting communication between individuals and teams working on the system (via a standardization of the terminology and the best practices used in the application domain). Open Cloud Computing Interface as a standard for IaaS follows this direction. A modeling paradigm for MDE is considered effective if its models make sense from the point of view of a user that is familiar with the domain, and if they can serve as a basis for implementing systems. The models are developed through extensive communication among product managers, designers, developers and users of the application domain. As the models approach completion, they enable the development of software and systems.

In MDE world, a model always conforms to a unique metamodel. One of the currently most active branch of Model Driven Engineering is the approach named model-driven architecture proposed by OMG and technically supported within the Eclipse ecosystem of programming and modelling tools (Eclipse Modeling Framework). This approach is based on the utilization of a language to write metamodels called the Meta Object Facility or MOF. Typical metamodels proposed by OMG are UML, SysML, SPEM or CWM. All the languages presented below could be defined as MOF metamodels instance of a metamodel.

In the Eclipse ecosystem of programming and modelling tools (Eclipse Modeling Framework (EMF)), Ecore can be seen as a technical implementation of essential MOF. EMF is an eMOF implementation and code generation facility for building tools and other applications based on a structured data model. From a model specification, EMF provides tools and runtime support to create Domain Specific Language (DSL) on top of the Eclipse platform. Most important, EMF provides the foundation for interoperability with other EMF-based tools and applications using a default serialization strategy based on XMI. Consequently, EMF has been used to implement a large set of tools and thus, evolved into an efficient Java implementation of a core subset of the MOF API. As a first real benefit, EMF provides a transparent compatibility of the Models infrastructure with several design environments. All the tools built with frameworks such as Xtext, EMFText, GMF or Sirius can be directly plugged on the Models infrastructure to monitor the running system. The generated code is clean and provides an embedded visitor pattern and an observer pattern. EMF also provides an XMI marshaller and unmarshaller that can be used to easily share models. Finally EMF offers lazy loadings of resources allowing the
loading of single model elements on demand and caching them softly in an application.

OCCI defined a metamodel for all kinds of management tasks without a real metamodel implementation. On top of this modeling language, OCCI also defines Protocol and API for interacting with this model. Protocol and API follows a Representational State Transfer (REST) architecture. OCCI was originally initiated to create a remote management API for IaaS model based Services, allowing the development of interoperable tools for common tasks including deployment, autonomic scaling and monitoring. It has to evolve into a flexible modeling approach with a strong focus on integration, portability, interoperability and innovation while still offering a high degree of extensibility. In OCCIware, we provide an implementation of OCCI metamodel with the Eclipse Modeling Framework. We define a set of tools to design OCCI model and OCCI extension in preserving the API, protocol and the Rest style, well accepted in industry. Based on this metamodel, we also provide a clear static semantics and operational semantics using OCL and Alloy to precisely define the OCCI domain.

2.2 Domain Specific Languages

2.2.1 Rationale

The Open Cloud Computing Interface metamodel is resource-oriented: typing mechanism involves “Kind” and “Mixins” while instanciation mechanisms are called “Resource”, “Link” and are extensible. These are, amongst others, some specificities of this metamodel with regard to, for instance, object-oriented programming. This particular paradigm justifies the need for defining Domain Specific Language(s) (DSL). DSL allows the programmers to manipulate the real concepts, and only those ones, of the paradigm they are working with. Designing a DSL for each particular class of problem is called Language Oriented Programming [11].

Another approach consists in defining General Programming Language API for manipulating the Open Cloud Computing Interface metamodel.

The DSL approach is limited by the language learning curve. While some quantitative studies exist about the complexity of solving a particular use case with a DSL vs GPL API[13], a qualitative approach requires developers feedback on multiple use cases.

In OCCIware we intend to design three DSLs, allowing to validate their usability and complexity in use cases. We consider using the API approach for some of them in order to evaluate their respective usability.

2.2.2 OCCI DSL - Structural Part

The structural part of the Open Cloud Computing Interface DSL will allow to express types and instances of Open Cloud Computing Interface concepts. The Open Cloud Computing Interface Core Specification[10] defines 8 concepts, complemented with 4 additional concepts. They are illustrated in Figure 2.1, with coloured boxes indicating additional concepts.

In order to be easily implemented, this DSL will be described with 3 notations:

abstract syntax: using UML class diagrams;

Backus-Naur Form [?]: for direct implementation in parsers;

Graphical Syntax: for use in graphical development environments (e.g. OCCIware Studio, see 3.1).
2.2.3 OCCI DSL - Behaviour Part

The behavioural part of OCCI DSL must allow lifecycle management of OCCI types (kinds, mixins, actions, extensions, etc.), instances (resources, links, configurations, etc.) using CRUD\(^1\) operations. Contrary to structural language, the behavioural part can involve complex operations (e.g. arithmetic operations on attributes) and interactions between them (e.g. create or update \textit{Compute} resource on application load change).

These features can be implemented in different ways described in the following sections.

“From scratch” DSL

OCCI concepts, their lifecycle as well as operations - including arithmetic, string manipulation, etc - are described with a specific language.

Xtext Based DSL

Using the Xtext\(^2\) tool, one can easily describe specific parts of a DSL while reusing Xtext facilities for common features (arithmetic, string or collections manipulation, etc).

Specialized API

Without being a DSL \textit{stricto sensu}, an OCCI dedicated API described in a General Purpose Language (GPL) can lower the learning curve of developers.

Furthermore, several tools like GOObject Introspection\[^9\], or Xcore can drastically reduce the effort for synchronizing APIs in different GPL.

\(^1\)Create, Retrieve, Update, Delete
\(^2\)https://www.eclipse.org/Xtext/
2.2.4 A Particular Model Specific Language: QoS

While structural and behavioural parts of OCCI DSL will allow to manipulate OCCI metamodel concepts, the OCCIware project wants to evaluate an OCCI model-specific DSL.

As an example, a language dedicated to the manipulation of QoS resource types will be developed and will be available for use in “Datacenter as a Service” or “BigData / HPC as a Service” use cases.

2.3 Models

Models produced in OCCIware may be divided into three categories.

First, specific models will be designed to represent a particular use case, with no interoperability or standardization objective. Thanks to the Open Cloud Computing Interface metamodel and a standardized representation, these models can be interacted with from any Open Cloud Computing Interface-compliant tool, with limited semantics. These models may include:

- non-standard categories related to existing extensions, e.g. GPU-enabled compute (related task: datacenter as a service use case);
- new applications, e.g. public transportation schedule (related task: linked data as a service);
- etc.

The second category includes models aiming at being standardized, i.e. an abstraction of several tools providing a set of common features. For instance, a common model of configuration systems (Puppet, Chef, Docker, etc) should be proposed in the Deploy@OCCIware use case. Of course, specific models may be standardized, would a specific interest into that direction be demonstrated by partners and/or external contributors. But a particular effort will be put on the second category only for standardization.

Finally, wherever relevant, OCCIware components themselves will be described thanks to the Open Cloud Computing Interface formalism. It may include:

- runtime core and its components;
- extensions;
- studio and its components.

Runtime core and associated components models have already been defined as part of the task 4.1. They are illustrated in Figure [2.2] and fully described in OCCIWARE PROJECT DELIVERABLE 4.1.1 [8].
Figure 2.2: OCCI Runtime Model
Chapter 3

Eclipse Toolchain

3.1 OCCI Studio

The OCCI studio is a suite of tools designed to ease the development and the deployment of OCCI-based solutions. The studio implementation is based on Eclipse, which provides many frameworks for simplifying the development of OCCI studio’s tools:

- The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model.
- Sirius is an Eclipse project that eases the creation of graphical modeling workbenches by leveraging the Eclipse Modeling technologies, including EMF and GMF.
- (Eclipse) OCL is the implementation of the OMG standard language for describing constraints.
- XText is a framework for the development of DSLs with a textual syntax. It provides the parser/serializer for the syntax, along with a complete Eclipse integration (editor with syntax coloring, completion, outline).
- Acceleo is a pragmatic implementation of the OMG MOF Model to Text Language (MTL) standard. It provides a language (based on OCL) to easily describe generators based on EMF models.

The OCCI studio mainly consists in an OCCI Extension Designer (see Figure 3.1), which will help with OCCI Extensions edition. Then, for specific extensions, it will be possible to create easily specific designers.

The OCCI Extension Designer includes a graphical tool, based on Sirius, for designing the Extension using diagrams. Moreover, a textual editor allows to edit OCCI models (both extensions and configurations). Designed with XText, this utility will also provide a way to import/export textual OCCI definitions. A preliminary version of the user interface is illustrated in Figure 3.2.

An Acceleo generator can produce a documentation of the Extension encoded using the Textile markup language. Such a documentation can be rendered on a website such as github, as illustrated in Figure 3.3.

Once an Extension is described, it can be declined into a specific tooling stub that can be prepared by code generation. This tooling stub consists in:

1. an EMF metamodel project, inheriting the OCCI metamodel, and providing all the concepts of the extensions as EMF types: in this way the use of the extension with EMF is eased (modelers, generators...);
Figure 3.1: The Extension Designer

Figure 3.2: The OCCI Extension Textual Editor
Figure 3.3: Rendering on Github of an OCCI Extension generated documentation
2. A basic Sirius designer intended to be customized by the specifier, but providing a usable basic diagram editor.

A functional example of a fully developed extension is available: the Docker designer, generated from the OCCI Docker extension, which allows to create diagrams of Docker resources layout and execute them thanks to a connector integrated with the modeler. Several configurations generators are included in the Docker designer:

- docker configuration (yaml);
- curl file deploying the extension.

Figure 3.4 displays a screenshot of this application.

![Figure 3.4: The Docker modeler](image)

### 3.2 Simulator

The purpose of a simulation is to gain understanding on a system without manipulating the real system, either because it is not yet defined or available, or because it cannot be exercised directly due to cost, time, resource or risk constraints. Simulation is thus performed on a model of the system.

A simulation is generally defined through three steps. The first one consists in generating a representation of the workload, i.e. the set of inputs, to be applied to the studied system. This representation may be a trace (or scenario) depicting a real workload or a synthetic workload artificially generated by heuristics or stochastic functions. The second step concerns the simulation that is performed by applying a workload on a model of the system and producing results. Finally, the third step consists in analyzing simulation results in order to acquire a better understanding of the considered system.

These three steps may be separated and supported by different tools, like a scenario builder (as a workload generator), a model execution engine and result analysis tools. It is also possible to combine in the same tool two of these steps or even the three of them. For instance, it
is possible to interactively create a trace while the model is executed. It is also possible to
couple the execution engine and analysis tools to present, "on the fly" (i.e. during a simulation),
synthetic appropriate results.

In OCCIware, we mainly use an OcciModel to model the first step. We provide an interpreter
for OcciModel designed using the Gemoc approach\(^1\) that allows to bridge the graphical editors
on the simulation for implementing the second step. Finally, the constraint checker (alloy and
OCL) can be plugged on demand during the simulation to ensure that no constraint are violated.
A full specification of the simulator is provided in D3.4.2. The constraint checker is used for the
third step of the simulation.

3.3 Decision Support Tool for Cloud Services Transition

This tool helps to analyze the impact of the transition of an existing system to a cloud-based
solution, in terms of cost. To achieve this analysis, the process is the following:

1. reverse-engineering of an existing system, as much automatically as possible but also manu-
ally, depending on the system. For instance a system that already use cloud-based solutions
can be partially analyzed using OCCI connectors, that will produce an OCCI model of the
system. The goal of this step is to establish a structured model of the current system, the
"as is";

2. definition of a “to be”, a model at the same level of abstraction as the “as is” but specifying
the system as it should be after transition. As our main use case is based on cloud only,
it is entirely defined in OCCI;

3. computation of the transition cost, by adding several aspects: difference between the static
cost of the “as is” and the “to be”, for instance hosting a data center vs delegating to a cloud
company; effective transition cost, as in what should be updated, abandoned, created to
support this transition.

To achieve this goal the tool can be based on an Obéo product: SmartEA. Based on the
TOGAF framework and on open source technologies, Obéo SmartEA is designed to integrate
existing repositories and develop business transformation trajectories. The data is stored using
EMF CDO, which allows to use the SmartEA repository in a collaborative way. The use of a
viewpoint-based approach to build models of current and future enterprise architectures greatly
facilitates the execution of gap and impact analyses. These viewpoints are realized using Sirius.

SmartEA can be customized to embed the OCCI Studio tools and modelers, as they are
based on the same technologies. In this way they can be used to ease the description of “as is”
and “to be”.

3.4 Generators and Connectors

Generators and connectors enable OCCIWare models to reach out beyond their Eclipse-based
MDE platform and vice versa. Therefore, they are a critical part of making OCCIWare tooling
actually useful, both by integrating with other formats (generators) and runtime components
(connectors).

This section describes their basic principles and provides a few examples. It will be enriched
in future iterations, once use case requirements will have been gathered.

\(^1\)http://www.gemoc.org
3.4.1 Generators

Generators take advantage of the Acceleo templates component of the OCCIware tooling MDE chain to export OCCI models to other formats.

Here are the main kinds of such export targets:

1. **documentation**, be it office documents (PDF, Latex) or online (HTML) or both (ex. Markdown), to be published or used within development tools. This includes providing meta documentation such as a table of content, indexes, including referenced models that increase overall usability, and possibly even making examples executable;

2. **standard formats**, starting with the various OCCI representations, but also other cloud and architecture standards (such as UML, architecture languages),

3. and beyond that, **configuration formats** of various runtime components: erocci, Robo-conf and other deployment tools, Administration Console dashboard widgets, OASIS /Ozwillo's both Linked Data model definition and storage, etc.;

4. **code**, such as command line HTTP requests to REST APIs (curl), libraries (Java, but also scripting such as javascript - browser- or server-side, Python, Ruby, erlang) for either client (wrapper, test driver) or server (mocking and simulation, proxy), or even SNMP. Such code can constitute up to a full-fledged connector (see below), or be used in one.

Besides being used on their own to create generators, Acceleo templates are also used within connectors to generate parts of configuration applied or messages sent to achieve their purpose.

3.4.2 Connectors

Connectors provide the connection between models and running systems. Basically, connectors take a source model defined by a metamodel and project it into the running systems. Conversely, connectors monitor the running systems and transform them into the target models.

The connectors are based on the following principles:

1. **Transformation**: the connectors provide expressive model transformation techniques based on design patterns, which ease the specification of translations between the models and running systems.

2. **Introspection**: to introspect the running system, the connectors employ Model-Driven Engineering (MDE) techniques, which handle the introspection and analysis of the system at higher level of models. Using MDE techniques, different models describing certain constraints are derived and maintained at runtime.

3. **Synchronisation**: the connectors provide incremental synchronization between a running system and models. To detect model modifications efficiently, the connectors rely on a notification mechanism that reports when source model element has been changed. To synchronize the changes of model with the running system, the connectors check if model elements are still consistent by navigating efficiently between the source model and running system model using the correspondence model.

Regarding connectors targeted at the OCCI runtime, they should implement the backend interface defined in [8].

The main connectors are:
- *docker-connector*: this connector is specific to the docker model. The Graphical Modeler provides a tool named Docker Modeler that is used to represent the containers and the machines which host those containers graphically. This tool allows to perform some actions such as: Start, Stop, Restart, Import and Synchronize, which are reflected in the running system using the connector.

- *infrastructure-connector*: this connector is specific to the infrastructure model. This connector interacts with the most existing hypervisors and the cloud providers. Like the Docker Modeler, the Infrastructure Modeler is used to represent machines and the devices of those machines.

- *technical-architecture-analysis-connector*: to enable analysis of an Information System’s existing technical architecture as planned in task 3.5, connectors that extract models out of it are required, as defined in [6].
Chapter 4

Runtime Support

The runtime architecture is described in detail in OCCIware Project Deliverable 4.1.1 [8]. It is illustrated in Figure 4.1.

Figure 4.1: OCCI Runtime Architecture
Chapter 5

Conclusion

While the Open Cloud Computing Interface specification describes standard models and renderings for manipulating cloud computing resources, the reuse of Open Cloud Computing Interface-compliant tools is really limited by the absence of formal, designing and runtime tools.

In this document, we have described the architecture of a comprehensive toolchain made of several components with clear interfaces and role, so that every developer involved in the development of an Open Cloud Computing Interface-compliant application can use some or all of them, together or independently.

While the objectives of OCCIware project require a multi-disciplinary approach, they also bring the risk of building incoherent toolchain, with limited interaction between components. The designed architecture is dedicated to mitigating this risk.

The usability of the toolchain will be validated by the use cases and their feedback will be used to adjust the architecture wherever relevant.
Bibliography