Madeus: A formal deployment model

Maverick Chardet, Hélène Coullon, Dimitri Pertin, Christian Pérez

Madeus: A formal deployment model

Maverick Chardet, Hélène Coullon, Dimitri Pertin
IMT Atlantique, Inria, LS2N, UBL
F-44307 Nantes, France
maverick.chardet@inria.fr
helene.coullon@inria.fr
dimitri.pertin@inria.fr

Christian Perez
Univ. Lyon, Inria, CNRS, ENS de Lyon, UCBL, LIP
F-69342, Lyon Cedex 07, France
christian.perez@inria.fr

***Abstract*—Distributed software architecture is composed of multiple interacting modules, or components. Deploying such software consists in installing them on a given infrastructure and leading them to a functional state. However, since each module has its own life cycle and might have various dependencies with other modules, deploying such software is a very tedious task, particularly on massively distributed and heterogeneous infrastructures. To address this problem, many solutions have been designed to automate the deployment process. In this paper, we introduce Madeus, a component-based deployment model for complex distributed software. Madeus accurately describes the life cycle of each component by a Petri net structure, and is able to finely express the dependencies between components. The overall dependency graph it produces is then used to reduce deployment time by parallelizing deployment actions. While this increases the precision and performance of the model, it also increases its complexity. For this reason, the operational semantics needs to be clearly defined to prove results such as the termination of a deployment. In this paper, we formally describe the operational semantics of Madeus, and show how it can be used in a use-case: the deployment of a real and large distributed software (*i.e.*, OpenStack).**

***Keywords*—Automatic deployment; distributed software; component models; formal models**

I. Introduction

Distributed software architecture is composed of multiple interacting modules, or components. For this reason, component-based software engineering (CBSE) [1] is a domain well-suited for distributed software implementation [2], [3]. CBSE enhances code re-use, separation of concerns, and composability (thus maintainability) of software codes. A component-based application is made of a set of component instances connected together. A component is a black box that implements a functionality (or a service) of a piece of software that makes sense on its own, and which interacts with other components through well defined interfaces, called *ports*. For instance, to provide its service or functionnality, *Component A* might require a functionality provided by *Component B*. Such a composition of components is called an *assembly*.

Many component models focus on two aspects: first, the modeling of the component functionalities; second, the modeling of their connections. For instance, some component models are designed for distributed software [2]–[4], while others target High Performance Computing (HPC) and model their associated specific communication protocols (*e.g.*, MPI) [5]–[7]. However, only a few of them focus on modeling the deployment of components and assemblies (*i.e.*, their life cycle). Yet, when deploying component-based software to distributed infrastructures, being able to finely model and control the deployment life cycle of each component, as well as their coordination, is of major importance for safety and performance issues.

For this reason, this paper introduces Madeus, a formal component-based deployment model for distributed software. Madeus accurately describes the life cycle of each component by a structure close to a Petri net, and finely expresses the dependencies between components, which enables to orchestrate deployment actions in parallel. While this increases the precision and performance of the model, it also increases its complexity. Hence, the operational semantics of the model needs to be clearly defined to prove results such as the termination of a deployment. In this paper, we formally describe the operational semantics of Madeus, and we validate it through a real use-case: the deployment of OpenStack.

The rest of this paper is organized as follows. Section II describes related work. Section III details the Madeus model and its associated operational semantics. Section IV presents the OpenStack use-case and finally Section V concludes this work and opens to some perspectives.

II. Related work

Most component models handle a pre-defined API for their component life cycle management. This is the case, for instance, in CCM [2] or in L^2C [6] where an API can be used to configure, activate, deactivate and destroy components. Deployware [8] has been designed specifically to deploy distributed software. A component is also associated to a fixed set of deployment actions: install, configure, start, manage, stop, unconfigure and uninstall. Thus, while these component models offer a detailed API of the component life cycle, they do not enable to customize the deployment process of each component and their coordination.

A few component models have enhanced the flexibility of the deployment modeling. While in the Object Management Group's (OMG) specification [9] the deployment model is rigid and fixed by the model, in Fractal [3] and its evolutions GCM and GCM/ProActive [4], the control of a component

(*e.g.*, its deployment) is separated from its functionalities and grouped into a so-called *membrane* which is itself described as a component assembly. However, as a component is a black-box of code, the evolution of the deployment process cannot be controlled. The deployment has to be manually hard-coded which could be error-prone and difficult to verify.

The *Topology and Orchestration Specification for Cloud Applications* (TOSCA) is also a component model that addresses the deployment of its components. TOSCA [10], [11] is a standardization effort from the OASIS consortium to describe Cloud applications, their components and their deployment artifacts, using standard languages (*i.e.*, XML, YAML). A TOSCA description (or template) corresponds to a graph where nodes represent TOSCA resources (e.g. software components, virtual machines, physical servers), and where edges represent the relations between these nodes. Artifacts (of any type such as scripts, binaries, etc.) can be added to TOSCA descriptions in a CSAR (Cloud Service ARchive) to detail deployment steps. However, as for the components of the Fractal membrane, an artifact is also a black box making the coordination and control of the deployment process difficult.

Blender [12] is a complete deployment framework that has been maintained by the Mandriva linux distribution. The model behind Blender is Aeolus [13], a formal component model. In Aeolus the deployment process of a component is captured by an internal finite state machine. Each state can be connected to use, provide, or conflict ports to declare dependencies between multiple deployment processes of multiple components, thus enhancing the global coordination of the assembly. The Aeolus vision is static. Indeed, the list of states needed to reach a deployment is automatically and statically computed. Those states are then given to an external scheduler. As a result, the execution is left to the scheduler and there is no operational semantics directly associated to Aeolus. Moreover, only one deployment action can be executed at a time in a component.

III. The Madeus model

Madeus is a component-based deployment model inspired from Aeolus [13]. Like Aeolus, Madeus enables to define the deployment process of distributed software but it does not catch any information relative to the functional aspects of components. Intuitively, in Madeus, a component is modeled as a structure close to a Petri net (which we call *internal-net* in the rest of this paper) with *states*, *transitions* and *tokens*. States represent "milestones" in the deployment process and are passive entities, while transitions are attached to actions effectively performing the deployment. Tokens held by states or transitions mark the current status of the deployment. This *internal-net* is, constrained by *bindings* between its elements and the *ports* of the component. The reason for this new formalism is a higher abstraction level compared to Petri nets, more accessible to distributed software developers and distributed systems administrators. While Aeolus uses finite state machines and therefore is limited to one token per component, our *internal-nets* can hold multiple tokens.

From the software viewpoint, the overall deployment process is described by an *assembly* of components, each of which is in charge of describing the life cycle of a software module. Within an assembly, components are instantiated and compatible ports of these component instances are connected to form the complete software. As each component is defined by an *internal-net*, a Madeus component assembly is responsible for the coordination of multiple independent *internal-nets* according to the connections of component ports.

This section presents a detailed formalism of Madeus. Table I sums up the notations defined in this section.

A. Component

Formally in Madeus, a component is defined as a tuple that can be divided into four different parts: *places*, *transitions*, *ports* and *bindings*. Figure 1 depicts two components represented by black rectangles. In the following, we use this figure throughout the description of component's parts:

a) Places: A component in Madeus is first defined by a set of *places* denoted Π. A place is represented by a circle as illustrated in Figure 1. A component is also defined by two sets of *docks*. A dock is represented by a small square and is attached to a place. It is used to handle synchronization of parallel branches. The first set concerns input docks. It is denoted Δ_i and it is displayed under the place they are attached to. The second set is denoted Δ_o. It contains output docks and is displayed above the place they are attached to. The function $place : \Delta_o \cup \Delta_i \rightarrow \Pi$ returns the place a dock is attached to. Functions $dock_i : \Pi \rightarrow \mathcal{P}(\Delta_i)$ and $dock_o : \Pi \rightarrow \mathcal{P}(\Delta_o)$ respectively returns the subset of Δ_i and Δ_o attached to a place $\pi \in \Pi$. Places can be part of one or multiple groups which are subsets of Π. The set of groups is denoted G. A group of places, as illustrated in Figure 1, is represented by a red dashed box circling multiple places. Last, $I \subseteq \Pi$ is the non-empty set of initial places of the component.

b) Transitions: A component is also defined by a set of *transitions* denoted Θ. A transition $\theta \in \Theta$ is a pair containing one source dock and one destination dock: $\theta = (s, d) : s \in \Delta_o, d \in \Delta_i$. As illustrated in Figure 1, transitions are represented by arrows linking two docks. Each transition is associated to an *action*. The set of actions is denoted A and the function $action : \Theta \rightarrow A$ gives the action associated to a given transition. Finally, the function $end : A \rightarrow \mathbb{B}$, where $\mathbb{B} = \{\text{true}, \text{false}\}$ indicates whether the action of a transition has finished.

c) Ports: Places and transitions are internal elements of a Madeus component. The external interfaces of a component in Madeus is composed of *ports*. A component contains a set of *use-ports* denoted P_u, and a set of *provide-ports*, denoted P_p. Each port is associated to a type in T_{port}, and the function $type_p : P_u \cup P_p \rightarrow T_{port}$ maps the ports to their type. In Figure 1, provide ports are represented by small circles filled with black, while use ports are represented by semi-circles.

In addition to traditional use-provide ports of component models, Madeus handles a specific use-provide abstraction

	Places
Π	set of places of a component
Δ_i	set of input docks of a component
Δ_o	set of output docks of a component
$place$	function mapping a dock to its place
$dock_i$	function mapping a place to its input docks
$dock_o$	function mapping a place to its output docks
G	set of groups of places
I	subset of places holding a token at initialization
	Transitions
Θ	finite set of transitions
A	finite set of actions
$action$	function mapping a transition to its corresponding action
end	function indicating if the action of the transition has finished
	Ports
P_u	set of use ports of a component
P_p	set of provide ports
T_{port}	set of types of ports
$type_p$	function mapping a port to its type
D_u	set of data use ports of a component
D_p	set of data provide ports
T_{data}	set of types of data ports
$type_d$	function mapping a data port to its type
$\mathbb{D}$	set of possible data values
	Bindings
B_{P_u}	set of pairs mapping use ports to transitions
B_{P_p}	set of pairs mapping provide ports to groups of places
B_{D_u}	set of pairs mapping data use ports to transitions
B_{D_p}	set of pairs mapping data provide ports to places
	Assembly
C	set of component instances of an assembly
L_P	set of use-provide connections of an assembly
L_D	set of data-use-provide connections of an assembly
ebl	function indicating if a connection is enabled
	Semantics
mk	function indicating if an element holds a token
val_{A,D_p}	returns the value given by an action to a data-provide port
val	function mapping a data provide port to its current value

TABLE I: Notations used throughout this paper

for the transfer of data values. These ports are called *data-use-ports* and *data-provide-ports*. The set of data-use ports is denoted D_u, and the set of data-provide ports is denoted D_p. The set of possible data values is denoted by $\mathbb{D}$. Finally, the function $type_d : D_u \cup D_p \to T_{data}$ returns the data type of a given data port. In Figure 1, data-provide ports are represented by outgoing arrows of the component, while data-use ports are represented by incoming arrows in the component.

d) Bindings: In a Madeus component, places, groups of places and transitions can be bound to ports through *bindings*. There are four sets of bindings. First, we denote B_{P_u} the set of pairs that maps each use port to one or multiple internal transitions in Θ, indicating that these transitions use the service associated to this port: $(p, \theta) : p \in P_u, \theta \in \Theta$. Second, we denote B_{P_p} the set of pairs that maps each provide port to one or multiple groups of places, indicating that if at least one token exists in each group, the port is active: $(p, g) : p \in P_p, g \in G$. Third, we denote B_{D_u} the set of pairs that maps each data use port to one or multiple internal transitions in Θ, indicating that these transitions use the data associated to this port: $(d, \theta) : d \in D_u, \theta \in \Theta$. Finally, we denote B_{D_p} the set of pairs that maps each data provide port to one or multiple places, indicating that the data associated to this port

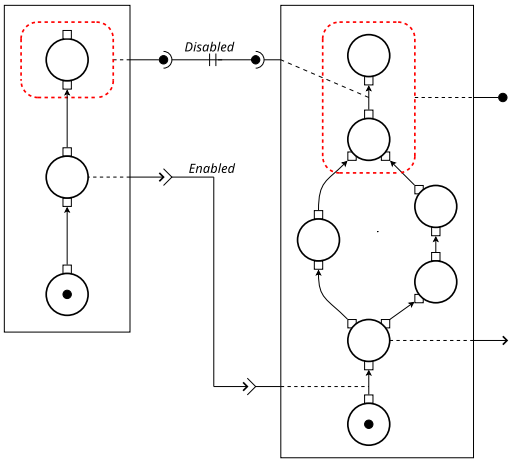

Fig. 1: Representation of an assembly based on two component instances and two connections, one in L_P and one in L_D.

is available if a token is or has been in one of these places: $(d, \pi) : d \in D_p, \pi \in \Pi$. Figure 1 displays bindings by dashed lines between ports, transitions, places and groups of places.

B. Assembly

An *assembly* of components represents the instantiation of components as defined in the previous section, and their connections through their ports. An assembly is similar to the main function of usual imperative programming languages. An example of component assembly is depicted in Figure 1. In Madeus an assembly is defined as a triplet (C, L_P, L_D), where C is a finite set of component instances, L_P is the set of connections (links) between use ports and provide ports of compatible types, and L_D is the set of connections between data-use ports and data-provide ports of compatible types. For all the components $c_1, \ldots, c_n \in C$, we denote with a star any union of the corresponding sets, for instance $\Pi^* = \Pi_1 \cup \cdots \cup \Pi_n$. We give a similar definition for functions, for instance $type_p^* : P_u^* \cup P_p^* \to T_{port}$. Connections are defined as follows:

- $(u, p) \in L_P, : u \in P_u^*, p \in P_p^*, type_p^*(u) = type_p^*(p)$,
- $(u, p) \in L_D, : u \in D_u^*, p \in D_p^*, type_d^*(u) = type_d^*(p)$.

C. Operational semantics

At each moment in the execution of a Madeus deployment assembly (C, L_P, L_D), we define three functions giving the current status of this assembly. First, we denote the function $ebl : L_P \cup L_D \to \mathbb{B}$, where $\mathbb{B} = \{\text{true}, \text{false}\}$, that indicates if a connection is enabled or not. On Figure 1, an enabled (resp. disabled) connection is representend by a continuous (resp. break) line. This *enabling* concept on connections will be used to coordinate component deployments. Second, the marking function is defined to evaluate if one element of any component holds a token: $mk : \Pi^* \cup \Delta_i^* \cup \Delta_o^* \cup \Theta^* \to \mathbb{B}$ where $\mathbb{B} = \{\text{true}, \text{false}\}$. By construction, an element can either hold one or zero tokens, the formal proof of this being left as future work.

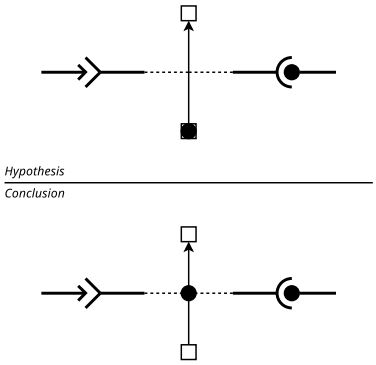

Fig. 2: Illustration of the rule of Equation (1) to fire transitions.

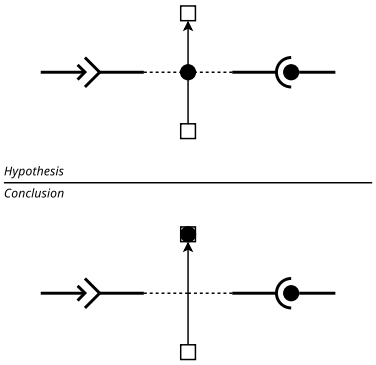

Fig. 3: Illustration of the rule of Equation (2) to end transitions.

Third, we define the function $val : D_P^* \to \mathbb{D} \cup \{\text{null}\}$ that returns the current value associated to a data provide port.

We call a *valuation* the tuple $\langle mk, ebl, val \rangle$, where mk indicates where tokens are located, ebl indicates whether connections are enabled or not, and val indicates the current value of any data provide port. At initialization the valuation is defined as follows:

- $mk(x) = \begin{cases} \text{true} & \text{if } x \in I^* \\ \text{false} & \text{if } x \in (\Pi^* \setminus I^*) \cup \Delta_i^* \cup \Delta_o^* \cup \Theta^* \end{cases}$
- $ebl(l) = \text{false} \quad \forall l \in L_P \cup L_D$
- $val(d) = \text{null} \quad \forall d \in D_p^*$

Notations.

- For a function $f : A \to B$, we denote $f' = f[a \coloneqq b] : A \to B$ such that:
$$f'(x) = \begin{cases} b & \text{if } x = a \\ f(x) & \text{if } x \neq a \end{cases}.$$
- We define the function $val_{A,D_p} : A \times D_P^* \to \mathbb{D}$ that returns the data value to assign to a given data-provide port after an action.

In this paper, we present seven rules to operate the Madeus model. We leave for future work the extension of these rules to support errors and reconfigurations. The rules are formally defined in Figure 9.

a) Firing transition: The first rule of Madeus is formally defined by Equation (1). The upper part of the rule indicates the hypotheses needed to fire a transition, *i.e.*, starting a transition, and the lower part indicates the conclusion of the rule, *i.e.*, the valuation changes. To fire a transition, the source dock of the transition θ needs a token, and for any port bound to the transition θ the connection of this port must be enabled. The conclusion of this rule is to move the token from the output dock to the transition. The upper part of Figure 2 illustrates the hypotheses and the lower part the conclusion of the rule.

b) Ending transition: The second rule of Madeus is formally defined by Equation (2). To end a transition θ, a token has to be present on the transition and the action performed by the transition must be terminated. When ending a transition, the token is moved from this transition to its destination dock. Note that the ports bound to the transition cannot be disconnected before applying this rule. Moreover, if the place to which the destination dock is attached is bound to data-provide ports, their values are replaced by those computed by the action. For this reason, the rule uses the function val_{A,D_p}. Figure 3 illustrates this rule.

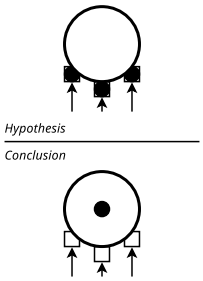

Fig. 4: Illustration of the rule of Equation (3) to move tokens from input docks to a place.

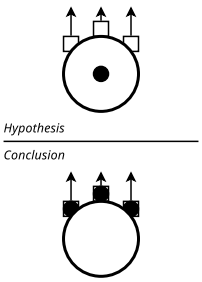

Fig. 5: Illustration of the rule of Equation (4) to move tokens from place to output docks.

c) Input docks to place: The third rule of Madeus is formally defined by Equation (3). To move tokens from input docks of a place to this place, all input docks must hold a token. The conclusion is to remove all the tokens within the docks and to add one token inside the place as illustrated in Figure 4.

d) Place to output docks: The fourth rule of Madeus is formally defined by Equation (4). To move a token from a place to its output docks, a token needs to be present onto the place. Also, if the place is part of a group which is itself bound to a used provide port, applying the rule must not make the last token of the group leave, otherwise this provide port becomes inactive. A provide port is said to be used if it is connected to a use port bound to a transition holding a token. If these conditions are met, the token can be removed from the place, and a token is added onto each output dock attached to this place. This rule is illustrated in Figure 5 in a simplified manner.

e) Enabling use-provide connections: The fifth rule of Madeus is formally defined by Equation (5). To enable a connection between use and provide ports, at least one token

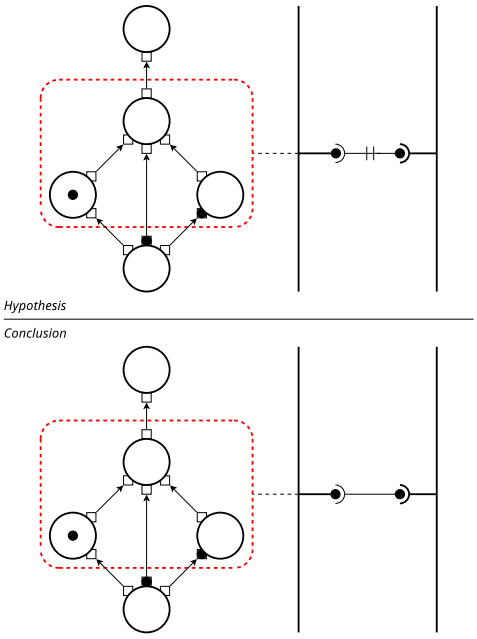

Fig. 6: Illustration of the rule of Equation (5) to enable a connection between use and provide ports of an assembly.

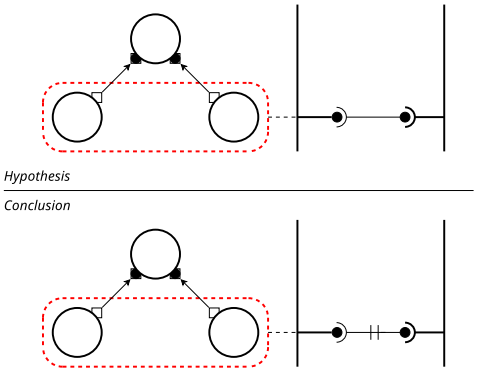

Fig. 7: Illustration of the rule of Equation (6) to disable a connection between use and provide ports of an assembly.

has to be present in each group of places bound to the provide port. A token is considered present in a group if it is placed on one of the places, or on a transition between docks, one of these docks being attached to one place of the group. The conclusion of the rule is the enabling of the connection as depicted in Figure 6.

f) Disabling use-provide connections: The sixth rule of Madeus is formally defined by Equation (6). Note that this rule has maximum priority and must be executed first if applicable. To disable a connection between use and provide ports, there must not be any token in any group of places bound to the provide port. The conclusion of the rule is to disable the connection as depicted in Figure 7.

g) Enabling data-use-provide connections: Finally, the seventh rule of Madeus is formally defined by Equation (7). To enable a connection between data-use and data-provide

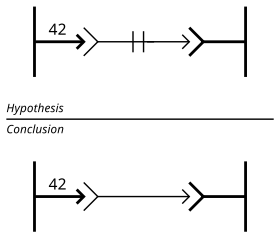

Fig. 8: Illustration of the rule of Equation (7) to enable a connection between data-use and data-provide ports of an assembly.

ports, a value has to be associated to the data-provide port. The conclusion of the rule is the activation of the connection as depicted in Figure 8. One can note that the position of tokens is not a hypothesis in this rule. Actually, we consider in the formal model that once a value has been attached to a data-provide port, by applying the rule that ends a transition (Equation (2)), the connection of this port can be enabled and will never be disabled. In pratice though, a disabling rule for data ports could be usefull to not consume extra resources.

Consistency.

In Madeus, the maximum number of tokens that can be used is the number of possible parallel branches. A token can only be created during a branching (rule *place to output docks*), where one token is created for each output dock of the place. For this reason, cycles are forbidden in Madeus, otherwise an infinity of tokens could be created which does not make sense within a deployment. We plan in future work on reconfiguration to allow cycles in very specific settings to keep control on the tokens and their creation.

IV. The OpenStack Case Study

In this section, we show how Madeus can model the deployment process of a real and large distributed software. To that end, we chose OpenStack[1], the de-facto open-source solution to address the IaaS level of the Cloud paradigm. Its community gathers more than 500 organisations (*e.g.*, Google, IBM, Intel) that have produced more than 20 million lines of code in six years. OpenStack is a large modular distributed system composed of more than 30 projects that manage the different aspects of an IaaS (*e.g.*, compute instances, storage and network resources). These projects are themselves composed of services, gathering more than 150 services in OpenStack that can be enabled or disabled depending on one's needs. Since the life cycle of each of these services can be modeled by a component, we aim at modeling the deployment of an OpenStack configuration by a Madeus assembly.

We have designed our use-case to fit a real production OpenStack configuration (*i.e.*, an assembly of OpenStack services). To that end, we rely on the basic deployment provided by Kolla[2], a popular tool to deploy OpenStack in production. To

[1]https://www.openstack.org/
[2]https://github.com/openstack/kolla

$$\frac{\theta = (s,d) \in \Theta^*, s \in \Delta_o^*, d \in \Delta_i^* \qquad mk\,(s) \qquad \forall p \in P_u \cup D_u, (p,\theta) \in B_{P_u}^* \cup B_{D_U}^* \,:\, \text{is\_ready}(p)}{\langle mk, ebl, val\rangle \rightarrow \langle mk\,[s \coloneqq \text{false}]\,[\theta \coloneqq \text{true}]\,, ebl, val\rangle} \tag{1}$$

$$\text{where: is\_ready}(p) = \exists\,(a,b) \in L_S \cup L_D,\, a = p \wedge ebl\,(a,b)\,.$$

$$\frac{\theta = (s,d) \in \Theta^* \qquad mk\,(\theta) \qquad end\,(action\,(\theta))}{\langle mk, ebl, val\rangle \rightarrow \left\langle mk\,[\theta \coloneqq \text{false}]\,[d \coloneqq \text{true}]\,, ebl, val\,\left[\forall p \in D_P^*, \text{dest}(p)\,:\, p \coloneqq val_{A,D_p}(action(\theta), p)\right]\right\rangle} \tag{2}$$

$$\text{where: dest}(p) = (p,g) \in B_{D_p}^*, place(d) \in g$$

$$\frac{\pi \in \Pi^* \qquad D_i = dock_i(\pi) \qquad \forall \delta \in D_i\, mk\,(\delta)}{\langle mk, ebl, val\rangle \rightarrow \langle mk\,[\forall \delta \in D_i\,:\,\delta \coloneqq \text{false}]\,[\pi \coloneqq \text{true}]\,, ebl, val\rangle} \tag{3}$$

$$\frac{\pi \in \Pi^* \qquad D_o = dock_o(\pi) \qquad mk\,(\pi) \qquad \forall g \in G, \pi \in g\,:\,\text{can\_leave}\,(\pi, D_o, g)}{\langle mk, ebl, val\rangle \rightarrow \langle mk\,[\forall \delta \in D_o\,:\,\delta \coloneqq \text{true}]\,[\pi \coloneqq \text{false}]\,, ebl, val\rangle} \tag{4}$$

$$\begin{aligned}
\text{where: can_leave}\,(\pi, D_o, g) &= \left(\exists p, (p,g) \in B_{P_p}\,:\,\text{is_used}(p)\right) \implies \neg\text{last_token_leaves}\,(\pi, D_o, g)\\
\text{is_used}\,(p) &= \exists u \in P_u^*, \theta \in \Theta^*:\ \ (p,u) \in L_P \wedge (u,\theta) \in B_{P_u}^* \wedge mk\,(\theta)\\
\text{last_token_leaves}\,(\pi, D_o, g) &= (\text{is_group_enabled}\,(g, mk) \wedge \neg\text{is_group_enabled}\,(g, mk\,[\pi \coloneqq \text{false}]\,[\forall d \in D_o\,:\,d \coloneqq \text{true}]))\\
\text{is_group_enabled}\,(g, mk) &= (\exists\,\pi \in g:\ mk\,(\pi))\\
&\quad \vee (\exists\,\delta \in \Delta_o^* \cup \Delta_i^*:\ mk\,(\delta) \wedge (\exists (s,d) \in \Theta^*:\ (\delta = s \vee \delta = d) \wedge \text{is_in_group}\,((s,d), g)))\\
&\quad \vee (\exists\,\theta \in \Theta^*:\ mk\,(\theta) \wedge \text{is_in_group}\,(\theta, g))\\
\text{is_in_group}\,((s,d), g) &= place\,(s) \in g \wedge place\,(d) \in g
\end{aligned}$$

$$\frac{l = (u,p) \in L_S \qquad \neg ebl\,(l) \qquad \forall g \in G, (p,g) \in B_{P_p}\,:\,\text{is\_group\_enabled}\,(g, mk)}{\langle mk, ebl, val\rangle \rightarrow \langle mk, ebl\,[l \coloneqq \text{true}]\,, val\rangle} \tag{5}$$

$$\frac{l = (u,p) \in L_S \qquad ebl\,(l) \qquad \exists g \in G, (p,g) \in B_{P_p}\,:\,\neg\text{is\_group\_enabled}\,(g, mk)}{\langle mk, ebl, val\rangle \rightarrow \langle mk, ebl\,[l \coloneqq \text{false}]\,, val\rangle} \tag{6}$$

$$\frac{l = (u,p) \in L_D \qquad \neg ebl\,(l) \qquad val\,(p) \neq \text{null}}{\langle mk, ebl, val\rangle \rightarrow \langle mk, ebl\,[l \coloneqq \text{true}]\,, val\rangle} \tag{7}$$

Fig. 9: The seven operational semantics rules of Madeus.

fit the roles defined in Kolla's blueprints, we have defined 11 Madeus components, associated to 11 OpenStack projects. The overall assembly deploys 36 OpenStack services providing the essential mechanisms to operate an infrastructure with OpenStack. For instance, while the *Nova* component deploys the Nova services, which are in charge of provisioning compute instances (*e.g.*, virtual machines), the *MariaDB* component deploys a SQL server used by most projects to store persistent data. Figure 10 depicts the Madeus assembly of our use-case, composed of 11 components with their connections through use-provide and data-use-provide ports. For the sake of simplicity and readability, we neither represent the internal details of components, nor the data-use-provide connections, nor all the use-provide connections in the figure. Still, one can note the complexity of a basic OpenStack deployment, due to the many dependencies between the different components.

Compared to Madeus, some production deployment tools are based on a retry mode, such as Kubernetes[3], which deploys simultaneously all the components without taking into account any dependency. Hence, errors occur when a dependency is not fulfilled, and the tool destroys and re-instantiates the related component. Well known low-level deployment tools, such as Ansible (used by Kolla), Puppet, or Chef[4], can apply a same instruction on multiple machines (Single Program, Multiple Data, or SPMD) but can not handle two different instructions in parallel. Some other production deployment tools (*e.g.*, Juju[5]), as well as academic models [8], [10], [11] can only express dependencies at the component level and thus, components with dependencies are deployed one by one. While in such models components without dependencies can be deployed simultaneously, they are not numerous in the OpenStack deployment, as shown in Figure 10.

Aeolus and Madeus both provide fine-grained definitions of the life cycle and dependencies of each component while automating the coordination process. The three red components shown in Figure 10 are detailed according to the Madeus model in Figure 11. One can note that when entering the

[3]http://kubernetes.io/

[4]https://www.ansible.com/, https://puppet.com/, https://www.chef.io/

[5]https://jujucharms.com/

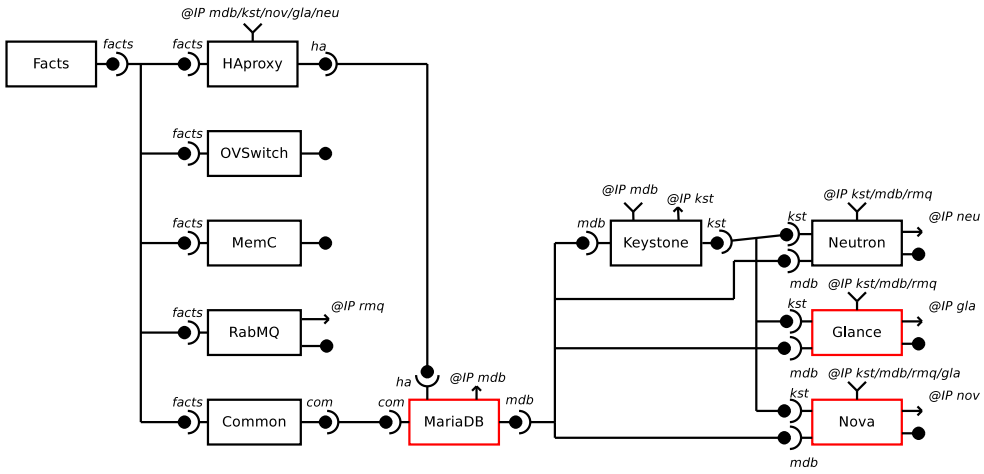

Fig. 10: Simplified representation of the Madeus assembly based on the 11 components involved in our use-case to deploy OpenStack. Those components fit a real production deployment as defined in Kolla. Red components are detailed in Figure 11.

detailed life cycle of each component the deployment process becomes even more complex. However, this level of details also offers more control and guarantees on the termination of a deployment without errors. As described in Section III, in Madeus, parallel branches are possible. This is illustrated for instance in the component *MariaDB*, where transitions *pull* and *boot* can be performed simultaneously. This is not possible in Aeolus. Moreover, we claim that the semantics is easier to understand for developers, for the following reason: since transitions are active and use external services or data, while places are passive and provide services or data, the temporal evolution of transitions and places is natural for the developer.

In Figure 11, the black tokens are set during initialization. One can note that semantic rule (4) can first be applied simultaneously to the three components to move the tokens to the output docks. Then semantic rule (1) that fires transitions can be applied to start the three transitions *provision*. The green tokens represent an example of evolution of the deployment process where *MariaDB* and *Glance* components have respectively reached the places that provide their IP addresses. As a result semantic rule (7) will enable the associated connections *@IP gla* and *@IP mdb*. In this scenario, the *Nova* component has reached a more complex state, where the transition *pull* is under execution and will soon end by applying semantic rule (2), the transition *register* is ended (*i.e.*, the *keystone* component, that is not represented in this figure, provides its services), and transitions *create-db* and *config* cannot be fired because their associated *use* and *data-use* connections are not enabled. In this situation, since the *Glance* component will enable its data-provide port, the transition *config* will be fired. However, the transition *create-db* will not be fired until the component *MariaDB* has reached its red token. When this red token is reached, semantic rule (5) will be applied and will enable the provide *mdb* connection. Finally, when both *Nova* and *Glance* components reach the blue tokens, semantic rule (3) will be applied in each of them to merge the tokens of the incoming docks to a single token in the associated place. Semantic rule (6) has not been used in this example.

To validate our use-case, we have developed an implementation of Madeus in Python and we successfully deployed OpenStack with it. We took the Ansible playbooks used by Kolla to deploy (sequentially) OpenStack services, and we split them into parts that could be executed in parallel. The transitions of our Madeus components call one of these smaller Ansible playbooks. Since Madeus can handle more parallelism than existing related work, we observed that Madeus can deploy our OpenStack configuration up to 58% faster than Kolla and 32% faster than a simulated Aeolus.

V. Conclusion

In this paper, we have introduced the Madeus model and its operational semantics. This model, inspired from Aeolus, offers a way to declare and automatically execute the deployment of an assembly of components. In Madeus, the life cycle of a component is declared as a structure close to a Petri net in which states and transitions are bound to the ports of the component. These ports are used to connect multiple components within an assembly to guarantee the coordination of the different life cycles at run-time. Moreover, we have applied the model to a real use-case study based on the deployment of OpenStack.

Because Madeus is a low-level model, it may be too technical for adoption by the industry. However we see it as a backend that can be used by higher level DSLs. In future works, we plan to prove some properties on Madeus assemblies, such as assembly consistency or the reachability of one or multiple valuations (*i.e.*, the deployment state of the software). To this end, we plan to transform an assembly into a global Petri net preserving its behavior, and we plan to use model checkers to produce proofs. Moreover we are currently working on an evolution of Madeus, which aims at handling reconfiguration of assemblies at run-time [14]–[16].

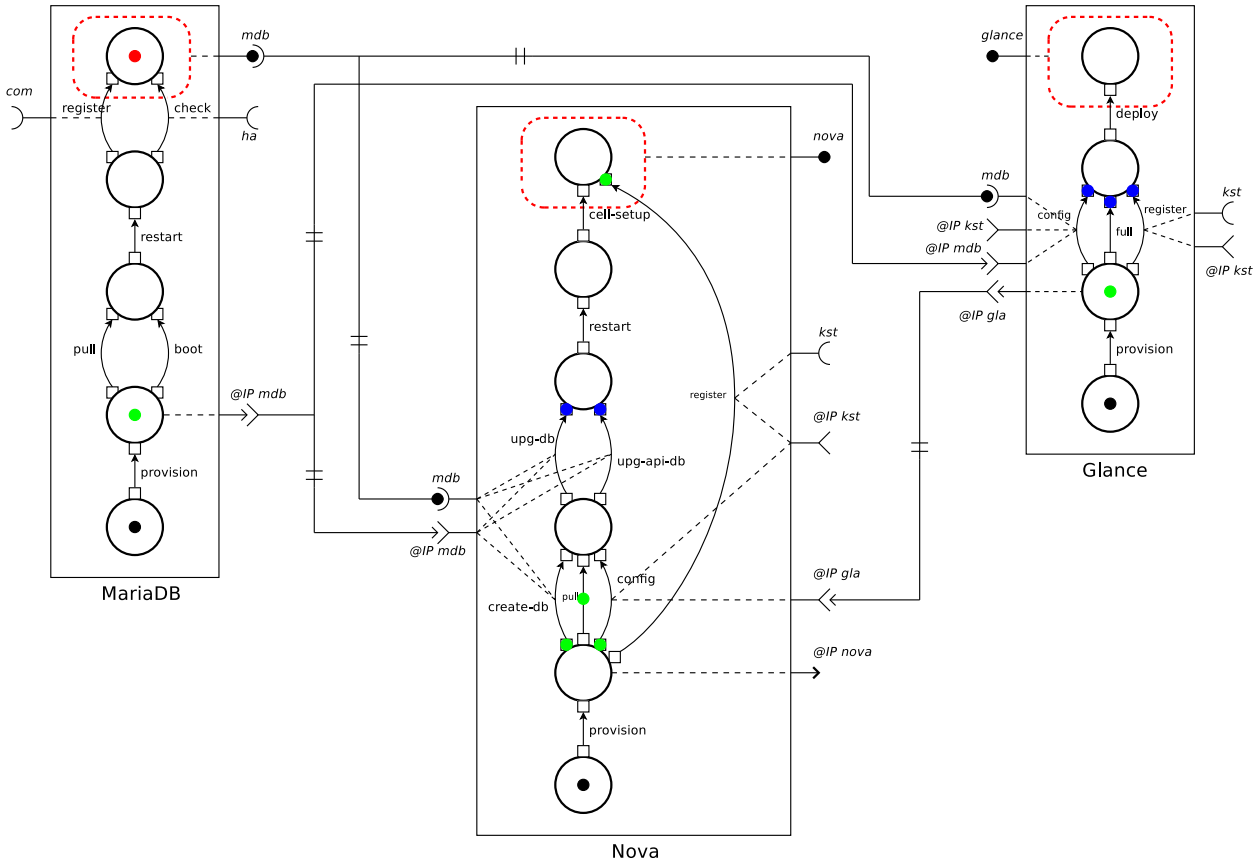

Fig. 11: A detailed sub-part of the full component assembly to deploy OpenStack. The three red components of Figure 10 are detailed by using Madeus. Black, green, blue and red tokens represent different scenarios during the deployment process.

ACKNOWLEDGMENT

This work was partially funded by the Discovery Inria Project Lab (see http://beyondtheclouds.github.io). The experiments presented in this paper used the Grid'5000 testbed, supported by a scientific interest group hosted by Inria, which includes CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

REFERENCES

[1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Pub. Co., Inc., 2002.

[2] Object Management Group, "CORBA Component Model," Apr. 2006. [Online]. Available: https://www.omg.org/spec/CCM/4.0/PDF

[3] G. Blair, T. Coupaye, and J.-B. Stefani, "Component-based architecture: the Fractal initiative," *Annals of telecommunications*, vol. 64, Feb 2009.

[4] F. Baude, L. Henrio, and C. Ruz, "Programming distributed and adaptable autonomous components – the GCM/ProActive framework," *Software: Practice and Experience*, May 2014.

[5] B. Allan, R. Armstrong, D. Bernholdt *et al.*, "A component architecture for high-performance scientific computing," *Intl J. of High Performance Computing Applications*, vol. 20, no. 2, pp. 163–202, 2006.

[6] J. Bigot and C. Pérez, "Increasing reuse in component models through genericity," Inria, Research Report RR-6941, 2009. [Online]. Available: https://hal.inria.fr/inria-00388508

[7] H. Coullon, J. Bigot, and C. Perez, "Extensibility and composability of a multi-stencil domain specific framework," *Intl J. of Parallel Programming*, Nov 2017.

[8] A. Flissi, J. Dubus, N. Dolet, and P. Merle, "Deploying on the Grid with Deployware," in *The Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, May 2008, pp. 177–184.

[9] Object Management Group, "Deployment and configuration of component-based distributed applications," Apr. 2006. [Online]. Available: https://www.omg.org/spec/DEPL/4.0/PDF

[10] "Topology and Orchestration Specification for Cloud Applications V1.0," http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html, 2013.

[11] A. Brogi, D. Neri, L. Rinaldi, and J. Soldani, "From (incomplete) TOSCA specifications to running applications, with Docker," in *Software Engineering and Formal Methods*, A. Cerone and M. Roveri, Eds. Springer Intl Pub., 2018, pp. 491–506.

[12] R. Di Cosmo, A. Eiche, J. Mauro *et al.*, "Automatic deployment of services in the Cloud with Aeolus Blender," in *13th Intl Conf. on Service-Oriented Computing*, A. Barros, D. Grigori, N. C. Narendra, and H. K. Dam, Eds., vol. 9435. Goa, India: Springer, Nov. 2015, pp. 397–411.

[13] R. Di Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro, "Aeolus: a component model for the Cloud," *Information and Computation*, pp. 100–121, Jan. 2014.

[14] J. Buisson, F. Dagnat, E. Leroux, and S. Martinez, "Safe reconfiguration of Coqcots and Pycots components," *Journal of Systems and Software*, vol. 122, pp. 430–444, dec 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215002630

[15] V. Lanore and C. Pérez, "A reconfigurable component model for HPC," in *CBSE 2015*. Montréal, Canada: ACM, May 2015, p. 10. [Online]. Available: https://hal.inria.fr/hal-01142606

[16] N. Gaspar, L. Henrio, and E. Madelaine, "Formally reasoning on a reconfigurable component-based system — a case study for the industrial world," in *The 10th International Symposium on Formal Aspects of Component Software*, Nanchang, China, Oct. 2013. [Online]. Available: https://hal.inria.fr/hal-00916115