HAL
open science

Enhancing Separation of Concerns, Parallelism, and Formalism in Distributed Software Deployment with Madeus

Maverick Chardet, Hélène Coullon, Christian Pérez, Dimitri Pertin, Charlène Servantie, Simon Robillard

To cite this version:

Maverick Chardet, Hélène Coullon, Christian Pérez, Dimitri Pertin, Charlène Servantie, et al.. Enhancing Separation of Concerns, Parallelism, and Formalism in Distributed Software Deployment with Madeus. 2020. hal-02737859

HAL Id: hal-02737859

https://inria.hal.science/hal-02737859v1

Preprint submitted on 2 Jun 2020

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

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

Enhancing Separation of Concerns, Parallelism, and Formalism in Distributed Software Deployment with Madeus

Maverick Chardet[a], Hélène Coullon[a], Christian Perez[b], Dimitri Pertin[a], Charlène Servantie[a], Simon Robillard[a]

[a] *IMT Atlantique, Inria, LS2N, UBL, F-44307 Nantes, France*
[b] *Univ Lyon, Inria, EnsL, UCBL, CNRS, LIP, Lyon, France*

Abstract

Complex distributed software systems are built by connecting software components across a large number of physical or virtual machines. Deploying such systems reliably and efficiently is a difficult task that involves multiple actors. Coordination mechanisms and programming support are needed. In this paper, we introduce Madeus, a component-based model that allows efficient and highly parallel deployment procedures for distributed software through a declarative approach. We describe the formal model of Madeus, its performance prediction model, and its concrete language and prototype. We evaluated Madeus on a complex real-world use case, the deployment of OpenStack, and measured a deployment time reduced by up to 71% compared to existing solutions.

Keywords: Distributed software; deployment; component models; parallelism; separation of concerns; formal methods

1. Introduction

Distributed software is designed in a modular architectural style where each component (a term that, in this paper, refers indifferently to a module, a service, or a microservice) is responsible for a specific part of the overall objective, and collaborates at runtime with other components that are potentially hosted on distant machines. With the advents of IT hardware, distributed software has become commonplace, whether it be executed in the cloud, on personal computers, mobile phones or objects.

However, deploying a large distributed software system on distributed infrastructures poses multiple challenges. Firstly, deployment involves a variety of actors with different roles and areas of expertise, from *developers* who are responsible for designing and coding the components, to *sysadmins* and *sysops* who are responsible for maintaining, configuring, and testing multi-user computer systems, with *devops* engineers working between them to oversee code releases and deployments. Secondly, deployment is a complex task: components must be chosen, configured and mapped to infrastructure nodes, dependencies have to be solved, virtualization layers handled, etc. The process is error-prone, and difficult to test because faults can be caused by very specific hardware or software conditions in a heterogeneous environment. Lastly, deployment is an often time-consuming process, owing to the scale of the systems. This last point is usually neglected in the literature. As a result, complex systems such as OpenStack can require in excess of one hour to deploy with existing solutions that do not take full advantage of the distributed nature of the underlying hardware. This becomes especially problematic in the context of continuous integration, with systems being deployed up to several hundreds of times every day.

For these reasons, programming models and tools need to be developed to facilitate documented, verifiable and efficient deployment procedures. To this end, we propose Madeus, a model that relies on a declarative description of dependencies to automate the execution of deployments with a high level of parallelism, both between components and with them. Prior to execution, Madeus deployments can also be analyzed for the purpose of verification and performance estimation, making it a useful tool during the conception of the deployment.

In this paper, among the difficulties related to the deployment of distributed software systems, we leave aside the mapping between pieces of software and the nodes of infrastructure, an optimization problem that is not in the scope of this paper [1, 2, 3, 4, 5]. Indeed, we restrict the scope of this paper to the *software commissioning* part of the deployment, i.e., the procedure responsible for leading the set of components to a valid running state while guaranteeing correct configurations and interactions. In particular, three metrics are considered to measure the quality of automation of distributed software commissioning: (1) separation of concerns between the different actors of the commissioning procedure to enhance productivity, maintainability and reusability of deployment code; (2) efficiency of the commissioning procedure in terms of par-

Email addresses: maverick.chardet@inria.fr (Maverick Chardet), helene.coullon@imt-atlantique.fr (Hélène Coullon), christian.perez@inria.fr (Christian Perez), dimitri.pertin@imt-atlantique.fr (Dimitri Pertin), charlene.servantie@imt-atlantique.fr (Charlène Servantie), simon.robillard@imt-atlantique.fr (Simon Robillard)

allelism expressiveness; and (3) formalization of the solution, such that formal properties can be proven on a given commissioning procedure. The contribution of this paper improves upon the related work by succeeding in the combination of these three metrics.

We have previously presented MADEUS in [6]. This journal paper brings the following additional contributions compared to our previous publication:

- an extended study of the related work (Section 3);
- a concrete language and a prototype (presented in Section 4, along with the overview of MADEUS);
- a revised and streamlined formal model that offers stronger guarantees (Section 5);
- a theoretical performance prediction model (Section 6);
- an extended reproducible evaluation on both synthetic use cases (Section 7) and one large use case on real infrastructures (Section 8).

2. Motivations

Distributed software deployment occurs between the development of components (functional part) and the execution of the distributed software on the infrastructure (management part). As a result, commissioning relies on information about both the behavior of the components and the infrastructure on which they will be executed, a frontier that is often called the DevOps domain. Note that our definition of software commissioning does not include placement and scheduling optimization problems, nor reconfiguration aspects of the distributed software management.

Commissioning distributed software requires coordinated interactions with various interfaces. Firstly, components have *control interfaces* provided by their developers, e.g., `start`, `stop` or `update`. Secondly, configuration files are used as *configuration interfaces* to obtain, from system administrators or operators, information that is not known when writing the functional code of the component. Thirdly, components may require packages or libraries to be installed on the host in order to work properly. Those requirements are directly related to *infrastructure management* and may be handled through virtualization or on physical nodes by sysadmins. In the end, the commissioning procedure of a single component is a *coordination program* between these three kinds of interfaces. Commissioning procedures are often explained in `README` files, or in documentation[1]. The deployment coordination of a single component can be simplified, e.g., by using deployment scripts or ready-to-use virtual images. In these cases, the deployment is automated and written once. However, because of the specificities of each infrastructure and application, these tools need to be parametric. Finding suitable scripts and images is often a challenge. For this reason, deployment coordination procedures remain difficult even for a single component. When considering a complete distributed software composed of a set of components, the picture becomes even more complex, because the commissioning procedures of various components, designed by different developers and interacting with each other, must be coordinated. For instance, when installing a very basic Apache/MariaDB system, additional documentation is required[2] to combine the components. The commissioning of Spark on top of Yarn[3] is another example. This complex coordination is usually the work of a *devops* engineer. With the increasing complexity of distributed software deployment, the number of languages and tools for the devops community has grown considerably in recent years.

Efficiency is an often neglected aspect of commissioning, but let us show that its importance should not to be underestimated. First, the commissioning of complex distributed software is very time-consuming. For example OpenStack (Section 8) can require in excess of one hour to deploy, because existing solutions do not favor parallelism, and therefore exploit only a fraction of the capabilities of the infrastructures that they target. Second, although common sense would dictate that commissioning occurs only once, this is not the case. System administrators perform the commissioning process every time a new machine or new cluster is installed in their infrastructure, when errors occur, or when updates are needed. Furthermore, with Continuous Integration (CI), Continuous Delivery (CD), and Continuous Deployment (CDep) of companies, research or open source projects, software commissioning is executed repeatedly in order to test new features continuously. For instance, the traces of the OpenStack Continuous Integration platform[4] show that over a nine-day period, from Februrary 19 to February 27, 2020, the Kolla[5] deployment of OpenStack was executed almost 3000 times, an average of more than 300 time per day. This period did not precede a release, so even higher numbers could be expected.

Many different techniques can be studied to improve the efficiency of distributed software deployments, notably using optimized and adapted system commands in commissioning scripts and programs (e.g., `rsync` instead of many `scp`); working on the optimization of a given system command (e.g., NIX package manager instead of `apt-get`); if using virtual images or container images, improving the boot time of hypervisors [7]; if using DOCKER images, optimizing the placement of image layers on the network [8]; fa-

[1] Cf. an Apache example at https://ubuntu.com/server/docs/web-servers-apache

[2] Cf. a LAMP example at https://ubuntu.com/server/docs/lamp-applications

[3] Cf. an example at https://spark.apache.org/docs/latest/running-on-yarn.html

[4] http://logstash.openstack.org

[5] https://wiki.openstack.org/wiki/Kolla

cilitating parallelism through commissioning languages [9]. This paper focuses on the last option.

Parallelism generally adds to the complexity of execution models, because a parallel program can execute in many different ways based on non-deterministic interleavings of instructions. This can create difficulties in the development of parallel programs, including software commissioning procedures. Since added performance should not come at the cost of correctness, it is important to mitigate this issue. Various strategies can be used, notably good programming practices and software verification, but all require an unambiguous definition of the semantics of the programming model. Developers need to understand the various possible behaviors of the software, independenly of the order of execution, which can be affected by factors outside of their control, e.g., implementation, communications, execution speed, etc. Software verification techniques for their part operate entirely on abstract models that need to accurately reflect the ultimate implementation of the software. For these reasons, we argue the importance of providing formal semantics for a distributed software commissioning tool, as they provide better precision than natural language descriptions. Indeed, conceiving the formal semantics along with the model, as opposed to providing semantics for an existing solution, allows for better integration.

Ultimately, we pay particular attention to three factors to assess the quality of the automation of distributed software commissioning: (1) separation of concerns between the different actors; (2) efficiency, i.e., level of parallelism; and (3) formalization of the solution. We give an overview of the related work based on these aspects.

3. Related Work

In this section we give an overview of existing solutions for the automation of distributed software commissioning. Then, we select the most relevant solutions for deeper analysis, and we compare them according to a specific set of metrics.

3.1. Automation of distributed software commissioning

We consider five classes of tools and languages to automate commissioning procedures. In practice, these tools are often used in combination.

Languages such as BASH or RUBY are commonly used to automate commissioning procedures. They are very flexible, and well known among system administrators and operators. However, in terms of software engineering, they suffer from many limitations, including poor separation of concerns, limited compositionality and complexity of coordination mechanisms.

Software configuration tools. This class contains tools such as ANSIBLE [10], PUPPET [11], CHEF [12], SALT [13], and some academic contributions such as SMARTFROG [14], ENGAGE [15] DEPLOYWARE [16], AEOLUS [17, 9]. Their goal is to enhance productivity when defining the deployment of components or services, and the coordination between their tasks. They typically add an abstraction layer on top of scripting languages, thus hiding some technical details (e.g., SSH connections), but may use different methods to achieve their goals. For instance, Ansible adopts a procedural imperative style in YAML, with a series of tasks to execute, while Puppet adopts a declarative approach in Python, which combines different instances of resources (i.e., services, packages, etc.). In a declarative approach, the user describes what is needed rather than how to get it, which is instead determined by the tool. This approach increases productivity at the expense of flexibility.

Infrastructure definition tools. Also known as *provisioning tools*, these tools are specifically designed to handle complex distributed infrastructures composed of multiple clusters and machines shared between users. Managing such infrastructures is very difficult and error-prone, as each application and each user may have different requirements, operating systems, package versions, etc. Virtualization has been introduced to solve this issue (in addition to improving portability, isolation between users, etc.). Tools such as DOCKER [18], TERRAFORM [19], JUJU [20], CLOUDFORMATION [21] or HEAT [22] use virtualization mechanisms (e.g., virtual machines, containers) to reduce the commissioning process to a set of commands that deploy a virtual resource on a physical machine, by using an image. If a suitable image is available, this reduces the complexity of the commissioning procedure. Otherwise, the image must be created by executing the commissioning commands. Some of these tools target a given virtualization technique or cloud provider (e.g., DOCKER, CLOUDFORMATION and HEAT), while others offer *providers* for different cloud infrastructures (e.g., TERRAFORM, JUJU). The TOSCA standard [23], and its implementations [24, 25, 26, 27] can be classified as a descriptive provisioning tool.

Orchestration tools. Recent tools such as KUBERNETES [28] and DOCKER SWARM [29] go further, offering an *orchestration* level to handle shared clusters of machines running many services simultaneously. In this case, deploying or installing distributed software is only part of the problem, as services also need to be restarted after failures, scaled to avoid overload, etc. These tools are outside the scope of this paper, but a subdivision of their architecture handles commissioning. For instance, KUBERNETES relies on DOCKER for deployments.

The specific class of CBSE. Component-based software engineering (CBSE) focuses on (distributed) software implementation and improves code re-use, separation of concerns, and composability (thus maintainability) [30]. A component-based application, called an *assembly*, includes a set of component instances connected together. Each

component is a black box that implements a functionality of the application, and interacts with other components through *ports* that are used to decouple the component internals from its interface. For instance, a port can be used to declare that the component either provides a service — in this case the port is attached to an internal method— or uses a service from another component. Many component models focus on the implementation of functionalities and interfaces [31, 32, 33, 34, 35, 36] of components, rather than on their commissioning. In the Object Management Group's (OMG) specification [37], the commissioning model is rigid and fixed by the model. However, a few component models have considered commissioning issues [32, 33, 16, 6, 17]. These CBSE solutions to deployment automation can be categorized in some of the four previous classes.

3.2. Related work metrics

Our analysis of the related work is based on several metrics that measure: (i) software engineering (SE) aspects, (ii) level of parallelism, and (iii) formal aspects. For each metric, we define four levels, presented in Table 1: (1) supported, denoted ✓and counting for 3 points in the score; (2) partially supported, denoted (✓) and counting for 2 points; (3) manually supported, denoted *M*, indicating a feature that can be manually coded by the user, and counting for 1 point; and finally (4) not supported, denoted -.

Software engineering. Software engineering properties are of prime importance when automating distributed software commissioning. Indeed, when several actors are involved in one given goal, software engineering techniques can help improve separation of concerns, meaning that each actor is responsible for her own expertise domain. This is improved by modular or component-based approaches, as each actor can be responsible for one component. Furthermore, component-based architectures also feature composition mechanisms that facilitate interactions between entities implemented by different actors. Additional actors may also be responsible for the composition. We define three SE metrics:

- *component*: whether the solution (tool, framework or scientific contribution) offers a component-oriented (e.g., services, modules) structure;
- *tasks*: whether the solution offers a way to divide the commissioning of each component in sub-elements that we call *tasks*, for a better structured procedural design;
- *separation of concerns*: whether the solution offers a way to build distributed software commissioning by composing components without the need to know their internal behaviors.

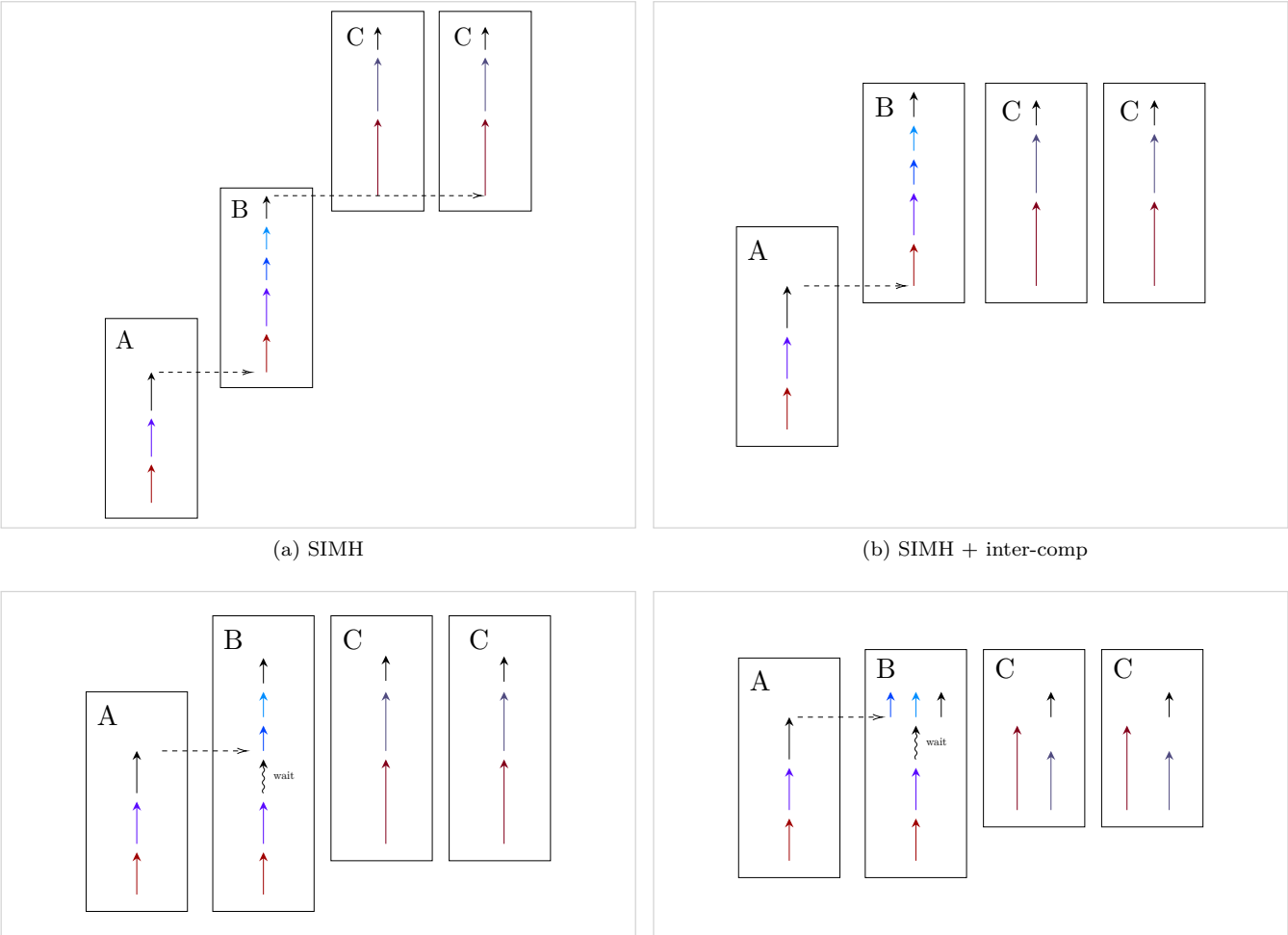
Parallelism level. A second important aspect of distributed software commissioning that has been introduced in the motivations is its efficiency and in particular (in this paper) the level of parallelism offered. We consider four metrics:

- *SIMH*: whether the solution offers a transparent way to perform the same instruction or set of instructions on multiple hosts simultaneously, when no dependencies exist between them;
- *inter-comp*: whether the solution offers a transparent way to simultaneously execute the commissioning of multiple components when no dependency exists between those;
- *inter-comp-tasks*: whether the solution offers a transparent way to simultaneously execute the commissioning tasks of multiple components until a dependency between tasks is reached;
- *intra-comp-tasks*: whether the solution offers a way to simultaneously execute two commissioning tasks of a given component, in other words whether a partial order on tasks can be given for a component.

The level of parallelism offered by a given tool is correlated to the type of dependencies that can be declared. For instance, without any dependency mechanism, only the SIMH level is achievable.

Figure 1 illustrates the four parallelism levels listed above through an example with three components or modules: component C is deployed on two hosts while components A and B are deployed on a single host. Moreover, components A and B need to be coordinated while component C is independent from both A and B. Sub-figure 1a illustrates the *SIMH* level, where parallelism can be introduced for component C. Sub-figure 1b illustrates the *inter-comp* parallelism level: as C is independent from B, both commissionings can be performed simultaneously. However, B depends on A, and because the dependencies are only available at the component level, their commissionings have to be performed sequentially. Sub-figure 1c shows the *inter-comp-tasks* parallelism level, where dependencies can be defined at the finer level of tasks. In this case, the commissionings of A and B can be started in parallel until they reach their dependencies. Thus, B has to wait until A performs needed tasks before continuing its deployment. Finally, sub-figure 1d illustrates the *intra-comp-tasks* parallelism level, where some tasks inside a component can be performed in parallel. Increased support in parallelism leads to a shorter expected commissioning time, as represented by the height of the figures.

Metrics for SE and parallelism are suitable for procedural approaches, as they consider *tasks* and not *resources*. This excludes Puppet or Salt, for instance. While declarative approaches have great advantages, they are not considered in this paper. Indeed we put a strong emphasis on



(a) SIMH

(b) SIMH + inter-comp

(c) SIMH + inter-comp + inter-comp-tasks

(d) SIMH + inter-comp + inter-comp-tasks + intra-comp-tasks

Figure 1: Examples to illustrate the four parallelism levels considered in this paper

parallelism in this paper. This requires a fine-grain definition of deployments, at the level of tasks. Declarative approaches on the contrary raise the abstraction level at the price of flexibility.

Formalism. The metric considered here is the existence of a formal model for each commissioning solution. Indeed, we claim that formal study of commissioning models and of their semantics is required for verification and safety in deployments. For instance, the formal model of MADEUS has been successfully used to verify safety properties on distributed software commissioning by model checking, as presented in [38].

3.3. Description and comparison of the related work

We have selected 8 distributed software commissioning solutions for a deeper comparison, including production tools (in particular those with a significant open source community) and academic solutions.

Shell scripts. Traditionally, operators automate software commissioning by transcribing actions and configurations from README files and tutorials into a sequence of commands in shell scripts. On the one hand, those scripts are written with low-level imperative languages, and with good programming skills, it is possible to express complex workflows (e.g., idempotency, parallelism, remote actions using SSH). For instance, parallelism can be managed by combining command execution in the background (e.g., using the control operator & in BASH) and synchronization commands such as `wait`. On the other hand, as the system grows, any custom script becomes error-prone, unpredictable, hard to understand and to maintain. Shell scripts lack software engineering aspects and offer no framework to express modules or tasks and their dependencies, thus hindering separation of concerns. Table 1 indicates that the features corresponding to our metrics can be implemented manually with shell scripts. Of course, this implementation is difficult, error-prone, and time-consuming.

Fractal. In the Object Management Group's (OMG) specification [37], the commissioning model is rigid and fixed by the model. In FRACTAL [32] and its evolutions GCM and GCM/ProActive [33], the control of a component (e.g., its commissioning) is decoupled from its functionalities into a so called *membrane* which is itself described as a component assembly written in Java. The membrane is handled by the FRACTAL runtime but the sub-assembly and its associated codes are entirely left to the user. That is why in Table 1, feature not natively supported by FRACTAL are shown manually implementable, using Java. Both *separation of concerns* and *inter-comp* are well handled by FRACTAL thanks to the notion of port (dependencies within the component or with other components) adapted to the membrane. Note that only FRACTAL components are supported, but the commissioning of existing modules can be encapsulated in a FRACTAL component.

Deployware. Flissi et al. proposed DEPLOYWARE (DW), a research effort targetting distributed software commissioning in the context of Grid computing [16]. Its implementation is based on the FRACTAL model. A component is called a *personality* and is associated with a fixed set of commissioning actions: install, configure, start, manage, stop, unconfigure and uninstall. Each action describes a sequence of tasks, written in a specific high-level language that uses pre-defined instructions (e.g., execute command, copy a file). While there is no notion of component ports, it is possible to express dependencies between components to initiate automatic coordination. For instance, when the operator triggers the action "install" on a component, the same action is triggered recursively to its dependencies. Because some features are not entirely controlled by the user, metrics *tasks*, *SIMH* and *inter-comp* are considered partially supported by DEPLOYWARE. Finally, DEPLOYWARE is based on FRACTAL and a formal effort has been carried out on FRACTAL, therefore the formal aspect support of DEPLOYWARE is considered partial.

Ansible. For devops used to shell scripts, ANSIBLE has become a popular configuration management tool, since it relies on a simple syntax written in YAML and does not require agents on administrated servers. Tasks are managed using only SSH and Python, which are commonly installed on every Linux distribution. In comparison, similar tools such as CHEF, PUPPET or CFENGINE not only require some understanding of Ruby or a custom language, but they are built on an agent-based architecture and require prior agent commissioning on remote hosts. ANSIBLE improves separation of concerns by defining *roles*, which can be seen as software components. Each role contains files that describe a sequence of tasks. To define a composition, a specific file called an ANSIBLE *playbook* is used to map the desired roles to the groups of nodes they will be applied to. Those groups of nodes are defined in a separate file called the *inventory*. When ANSIBLE is triggered, roles and their related tasks are sequentially executed to the associated groups of nodes. While tasks declarations are indeed managed sequentially, each task is executed in parallel when mapped to multiple remote hosts, thus offering *SIMH* support. Typically, an operator who wants to commission an APACHE web server and a MYSQL database would download two roles from Ansible Galaxy and register them in a playbook. Since ANSIBLE triggers roles in a sequential manner, if the operator is not aware that the database must be commissioned before the web server, she could make a mistake in the order of the roles she declared. This makes the *separation of concerns* support only partial. Finally, as one of the possible types of tasks in ANSIBLE is the execution of a shell command, any script could be executed as a task, thus allowing manually support for *intra-comp-tasks* parallelism.

Aeolus. Di Cosmo et al. proposed AEOLUS, a formal component-based model for the cloud [17]. Their component model captures the internal states of a component commissioning process thanks to a finite state machine. Each state can be connected to use, provide, or conflict ports to declare dependencies between the commissioning steps of different components. Hence, ports enable correct coordination of the global deployment process. The deployment procedure should then be written by the user or by an external tool [9] as a sequence of actions leading to a different state. As a result, the internal of each component must be known. For this reason, *separation of concerns* support is only partial. Furthermore as the deployment procedure is a sequence and as parallel transitions cannot be defined the *intra-comp-tasks* parallelism level is only possible in a manual way (not automated).

Juju. Canonical has developed their own software commissioning solution, JUJU[6], which aims at commissioning any kind of application on various cloud providers (e.g., AWS, OpenStack) and types of resources (container, VM or bare-metal). Its concepts are close to those of component models. Software modules are packaged as JUJU *charms* that describe the software commissioning steps through a set of scripts called *hooks*. Operators define their composition in a specific file called *bundle* in which they declare the desired charms with their *relations*. A relation is declared between two charms and used for component synchronization (by triggering hooks) and data sharing at runtime, similarly to component ports. As the concepts behind JUJU resemble those of AEOLUS, the metrics are similar with the exception of the formal aspect.

TOSCA. The *Topology and Orchestration Specification for Cloud Applications* (TOSCA) is another component-oriented model that partially addresses the commissioning of its components. TOSCA [23, 41] is a standardization effort from OASIS to describe cloud applications, their components and their deployment artifacts, using standard

[6] https://jujucharms.com/

		Coding		*Software configuration*			*Infrastructure*		*Orchestration*
		Shell	Fractal [39, 32, 33]	DeployWare [16]	Ansible [10]	Aeolus [9, 17, 40]	Juju [20]	Tosca [23, 41, 42, 27]	Kubernetes [28, 43]
SE	components	M	✓	✓	✓	✓	✓	✓	✓
SE	tasks	M	M	(✓)	✓	✓	(✓)	M	-
SE	sep. of con.	M	✓	✓	(✓)	(✓)	(✓)	✓	✓
Parallel	SIMH	M	✓	(✓)	✓	✓	✓	(✓)	✓
Parallel	inter-comp	M	✓	(✓)	-	✓	✓	(✓)	(✓)
Parallel	inter-comp-tasks	M	M	-	-	✓	(✓)	-	-
Parallel	intra-comp-tasks	M	M	M	M	M	M	M	-
	formal	-	✓	(✓)	-	✓	-	✓	-
	score	7	18	15	12	21	16	15	11

Table 1: Comparison of commissioning solutions based on aspects regarding software engineering (SE), parallelism and formalism. A supported metric is denoted ✓and counts for 3 points in the score; a partially supported metric is denoted (✓) and counts for 2 points; a manually supported metric is denoted *M* and counts for 1 point; and finally a non-supported metric is denoted -.

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: scripts, executable etc.) can be added to TOSCA descriptions in a CSAR (Cloud Service ARchive) to detail commissioning steps. Those commissioning steps can thus be customized by the developer, but there is no model, nor any guarantees associated with them. Therefore, the feature associated with the *tasks* and *intra-comp-tasks* metrics can be handled manually by the user. As there is no way to declare dependencies between artifacts of components, *inter-task* parallelism is not supported, however, relations between components make both *SIMH* and *inter-comp* parallelism theoretically available in Tosca. No information has been found on the complete support of these features in Tosca implementations [25, 26]. Finally, the Tosca standard [42] is formally defined to an extent.

Kubernetes. Initiated by Google, Kubernetes (K8S) is a popular framework to commission distributed software in the form of microservices that are packaged as a hierarchy of Docker containers and *pods*. A software component in Kubernetes is defined as a Docker container. These components have no ports to manage coordination, and their internals are fixed, since containers can only be started and stopped. As a consequence, the commissioning process is error-prone. For instance, a web server can be started before the required database and thus fail. For this reason, *inter-comp* parellelism is only partially supported. Kubernetes requires container to be started in any order, therefore any container must embed a waiting procedure w.r.t. its dependencies. If the deployment of a container fails, Kubernetes automatically retries. For stateless microservice architectures, this deployment strategy is popular.

3.4. Discussions

We have compared eight solutions according to software engineering metrics, parallelism metrics and one metric regarding the formal definition of the considered solution. Table 1 summarizes this comparison and raises a few key points that we discuss in the following section.

As usual when working on programming languages, the existing tools illustrate the difficult trade-off between flexibility and automation. On the one hand, when the tool is highly programmable, developers have the ability to manage their own code organization and to handle any kind of software engineering or efficiency property. For instance, by using shell scripts, any feature that we took into account could be handled. However, each of them would have to be hand-coded, which is difficult and error-prone. On the other hand, some solutions such as DeployWare and Juju restrict the internal commissioning behavior of components to a fixed set of actions (e.g., install, configure, start). This guarantees full control of the automated parallelism level, but also prevents potential optimizations allowed by the specifics of each component.

Aeolus is the solution with the highest score according to our metrics. Indeed, Aeolus combines advantages of component models to structure the code of software commissioning and enhance its separation of concerns, while introducing an additional way to model the internal commissioning behavior of each component through tasks. It seems that Aeolus offers a good trade-off between flexibility and automation. However, it handles only partial separation of concerns.

Furthermore, no existing solution offers full support for *intra-comp-tasks* parallelism. Although a few solutions already offer a way to model the internal commissioning behavior of each component by using tasks, dependencies between those tasks are limited to a sequential order, thus making *intra-comp-tasks* parallelism impossible. This could be handled manually in some of the existing tools, however these parallel aspects are difficult to implement,

and should preferably be handled automatically.

Although introducing more parallelism opens up potential performance gains, it also introduces more complexity for the user. For this reason, formalizing the commissioning solution is of high importance to guarantee properties of the commissioning, such as attainability.

4. Overview of Madeus

4.1. Principles

Madeus is a procedural *configuration tool* relying on a component-based coordination model for commissioning procedures. Inspired by Aeolus, it enhances automation, separation of concerns and level of parallelism for distributed software commissioning. Usually, component models are used to write distributed software in a modular fashion with guarantees on composition, thus improving separation of concerns between developers, promoting reuse of existing pieces of code, and enhancing flexibility and maintainability of code. Madeus brings these properties to commissioning design instead of functional code design.

A Madeus component is called a *control* component, as it is only intended to model the commissioning procedure of an already existing piece of software code (functional component, module or service). A Madeus control component is a type containing an internal net inspired by Petri nets, where places represent milestones of the commissioning, and transitions between places represent actions to be performed (e.g., `apt-get install`, `docker pull`, etc.). If multiple transitions leave a place, their parallel execution is automatically handled and synchronized by Madeus. A component may have dependencies with other components. These dependencies are declared through ports, a well-known concept in component-based software engineering. Two types of ports, working in pairs, are available in Madeus: *provide* and *use* ports. However, both are *coordination* ports, i.e., they model dependencies or synchronization but do not implement them. By using coordination ports, each component type can be defined independently and component instances can be connected later by another developer, thus improving separation of concerns and reusability of commissioning procedures. A provide coordination port is bound to sets of places that represents the milestones that have to be reached for the service or data to be offered. Finally, use coordination ports are bound to one or multiple transitions where data and service is actually required.

The overall commissioning procedure of a distributed software system is built by composition in an *assembly*, where component types are instantiated and connected. All control components execute simultaneously, thus introducing inter-component parallelism, in addition to the intra-component parallelism offered by parallel transitions. Two components connected by their respective compatible ports will automatically be coordinated so that a component cannot use a service or data if the associated provide port is not enabled.

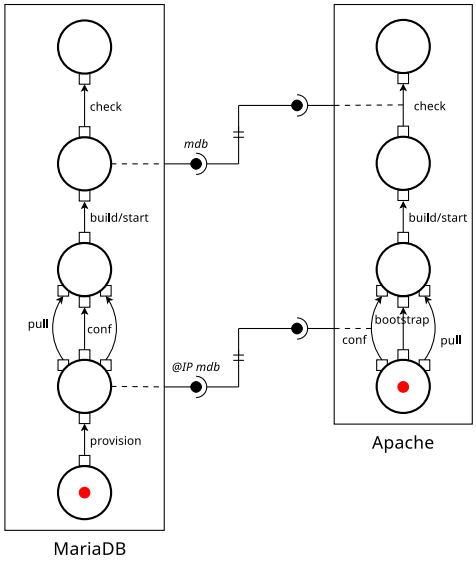


Figure 2: Example of a Madeus commissioning assembly with two components Apache and MariaDB. Places are represented by circles, with attached docks represented by small squares. Transitions are represented by arrows between the docks, and service ports by small black circles and semi-circles, data ports by outgoing or incoming arrows from components. Two initial red tokens are placed in each initial place of components in this example.

Madeus offers the expressiveness required to design composable and parallel commissioning procedures for complex distributed software systems.

Example. Figure 2 depicts a Madeus commissioning assembly of an Apache web server and a MariaDB database, using the graphical notation of Madeus. This example is based on a real container-based deployment described by RedHat[7][8]. Two Madeus control components are declared in this example: Apache and MariaDB. Apache contains four places (white circles), or milestones, while MariaDB contains five places. Some parallel transitions are declared for each of the component and can be observed in the figure (parallel arrows). Both components have two coordination ports. MariaDB provides both data and a service, while Apache uses a service and data. These instances are connected by their ports. Indeed, the Apache configuration depends on the IP address of the MariaDB component, and the testing phase for Apache, called `check`, uses the MariaDB service.

[7] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/getting_started_with_containers/install_and_deploy_an_apache_web_server_container

[8] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/getting_started_with_containers/install_and_deploy_a_mariadb_container

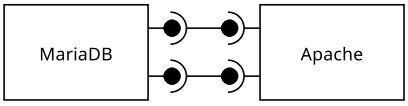


Figure 3: MADEUS assembly of an Apache component and a MariaDB component without knowing the details of each component.

MADEUS involves two kinds of actors. First is the *developer* of a control component, who may be the author of the associated existing piece of code, another developer, or even a system operator or administrator. Second is the *devops* engineer who designs an assembly, i.e., writes the overall commissioning procedure of a distributed software system to deploy on an infrastructure. MADEUS offers clear separation of concerns between the commissioning of a single component on the one hand, and the composition of an assembly on the other. The latter does not require detailed information about the commissioning of each component. This is a benefit compared to existing solution. For instance, even if Ansible offers properties close to composition (e.g., roles, playbooks, tasks, etc.), the devops still has to determine the correct order of composition. By contrast, the correct coordination, hence the correct order of execution, is automatically guaranteed by the MADEUS semantics and the composition of the component instances. Composition is illustrated in Figure 3 where the details of each component, since they are not needed, are omitted. This type of property is also offered by Aeolus [17], albeit without parallelism within components.

The formalization of MADEUS will be detailed in Section 5.

4.2. Concrete language

MADEUS also comes with a prototype and a concrete syntax that are implemented in PYTHON. MADEUS is a declarative language that follows the model previously presented to define control component types and assemblies of components. PYTHON has been chosen to prototype MADEUS because it is a widely known language within the DevOps community. Furthermore, with PYTHON, parametric components and assemblies can easily be defined. Note however that another implementation choice could have been to design a descriptive language closer to TOSCA or Ansible, for instance using YAML.

Listing 1 shows the declaration of the MariaDB component type of Figure 2. Lines 3 to 8 declare the places of the component type. A place is identified by a unique `string`. Lines 9 to 15 declare the transitions of the component type. A transition is identified by a unique key (a `string`), and is associated through a dictionary with a source and a destination place. Moreover, each transition is associated with a function to call to perform corresponding actions. For instance, the function `f_pull` of transition `pull` is declared on line 20, and provides the code to execute during this transition. Finally lines 16 to 19 declare the coordination ports of the control component type. A port is identified by a unique key (a `string`), and is associated with a type (use or provide) and the elements to which it is bound. As described previously, provide ports are bound to sets of places, and use ports to a set of transitions. In the case of MariaDB, one provide port, namely `serv`, is provided from the place `std`, and another (data) provide port, namely `ip` is provided from the first place `wtg`.

Listing 1: Madeus code of the MariaDB component type.

```
1  class MariaDB (MadeusComponent):
2    def create(self):
3      self.places = [
4          'wtg',
5          'prd',
6          'cfd',
7          'std',
8          'chd'
9      ]
10     self.initial_place = 'wtg'
11     self.transitions = {
12         'provision': ('wtg','prd', self.f_prov),
13         'pull': ('prd', 'cfd', self.f_pull),
14         'conf': ('prd', 'cfd', self.f_conf),
15         'bootstrap': ('prd', 'cfd', self.f_boots),
16         'start': ('cfd', 'std', self.f_start),
17         'check': ('std', 'chd', self.f_check)
18     }
19     self.dependencies = {
20         'ip': (DepType.PROVIDE, [['wtg']]),
21         'serv': (DepType.PROVIDE, [['std']])
22     }
23
24     def f_prov(self):
25         # execution of bash scripts
26         # execution of ansible playbooks
27         # etc.
28     def f_pull(self):
29         # ...
30     def f_conf(self):
31         # ...
32     def f_boots(self):
33         # ...
34     def f_start(self):
35         # ...
36     def f_check(self):
37         # ...
```

Similarly, Listing 2 shows the declaration of the Apache control component of Figure 2. The Apache component type contains two use coordination ports, one modeling data and one modeling a service (lines 18 to 21).

Listing 2: Madeus code of the Apache component type.

```
1  class Apache (MadeusComponent):
2    def create(self):
3      self.places = [
4          'wtg',
5          'cfd',
6          'std',
7          'chd'
8      ]
9      self.initial_place = 'wtg'
10     self.transitions = {
11         'pull': ('wtg', 'cfd', self.f_pull),
12         'conf': ('wtg', 'cfd', self.f_conf),
13         'bootstrap': ('wtg', 'cfd', self.f_boots),
14         'start': ('cfd', 'std', self.f_start),
15         'check': ('std', 'chd', self.f_check)
16     }
17     self.dependencies = {
18         'ipMDB': (DepType.USE, ['conf']),
19         'serviceMDB': (DepType.USE, ['check'])
20     }
21
```

```
    def f_pull(self):
        # execution of bash scripts
        # execution of ansible playbooks
        # etc.
    def f_conf(self):
        # ...
    def f_boots(self):
        # ...
    def f_start(self):
        # ...
    def f_check(self):
        # ...
```

Finally, Listing 3 shows the declaration of the assembly of components of Figure 2. Lines 6 and 7 respectively instantiate the component types MariaDB and Apache previously declared. Line 9 to 13 perform the creation of an assembly, the addition of component instances to the assembly, and the connections of the components. Finally, lines 15 and 16 run the assembly to perform the commissioning of MariaDB/Apache. An overview of this execution is given in the next section.

Listing 3: Madeus code of the assembly of Figure 2.

```
1  from components.mariadb import MariaDB
2  from components.apache import Apache
3
4  class ApacheWithDB (MadeusAssembly):
5    def create():
6      self.components = {
7        'apache': Apache(),
8        'mariadb': MariaDB()
9      }
10     self.dependencies = [
11       ('apache', 'ipMDB', 'mariadb', 'ip'),
12       ('apache', 'servMDB', 'mariadb', 'serv')
13     ]
14
15 if __name__ == '__main__':
16   assembly = ApacheWithDB()
17   assembly.run()
```

In this work, we assume that the placement optimization problem is solved. Thus, there is no particular need for a way to specify the infrastructure on which the deployment will be performed. In our use cases, the placement information is given exactly as in Ansible, i.e., using inventory files (Section 3).

4.3. Execution

In Madeus, executing a commissioning procedure requires the execution of an assembly. The Madeus execution model is governed by operational semantics rules to move from one configuration to another. The concept of configuration, which is introduced formally in Section 5, intuitively corresponds to a snapshot of the execution of an assembly. It is composed of the location of the tokens (modeling the evolution of the process), a history of places that have been reached, and the set of actions (transitions) under execution. In practice, semantics rules move tokens from places to transitions within components. Those rules are inspired by those of Petri nets, yet have a specific semantics for docks, transitions, bindings and ports. Details of the transformation from a Madeus assembly to a Petri net are given in [38].

In the formal model, transitions are composed of three elements: an output dock attached to the source place, an action, and an input dock attached to the destination place. Docks are used to model synchronization points in the execution: the output dock holds a token if the action is ready to be started, whereas the input dock holds a token after the action has ended.

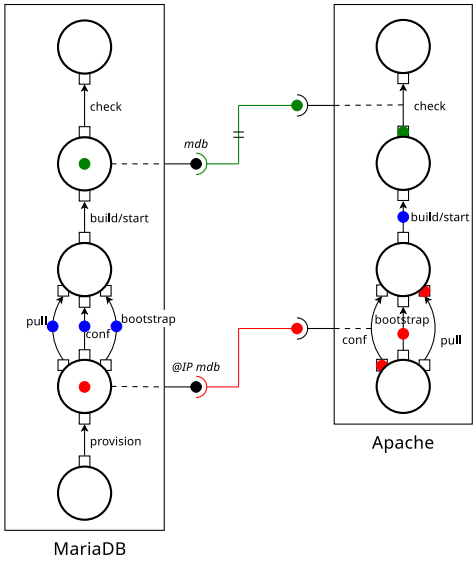


Figure 4: Three possible intermediate configurations during the commissioning of the assembly presented in Figure 2. Each configuration is represented by a different color.

In this paper, the execution model of Madeus has been streamlined compared to our previous work [6]. It is defined by five operational semantic rules, that are formally specified in Section 5. These rules correspond to the following behaviors:

1. *Reaching a place*: when all the actions required by a place have finished, the place can be reached. Note that the enabling of provide ports depends on place reached, so this event may enable additional ports.
2. *Leaving a place*: after a place of the deployment has been reached, new actions can be started. Thus tokens are moved to output docks (i.e., source docks of outgoing transitions).
3. *Firing a transition*: if the source dock of a transition contains a token, and all use ports bound to this transition are enabled, the action associated with the transition may be initiated.
4. *Terminating an action*: an ongoing action may terminate at any point during the execution. This captures the fact that, after an action has been initiated, its execution and termination are outside of the control of the Madeus model. However, our notion of execution excludes non-terminating actions.
5. *Ending a transition*: after an action terminates, a token is placed on the output dock of the corresponding transition.

Example. Let us detail the commissioning execution of the Apache/MariaDB example. The initial assembly, given in Figure 2, has already been described. Figure 4 depicts three examples of configurations that may occur at different steps of the commissioning. In chronological order, the first configuration is represented by red tokens, the second one by blue tokens, and the third one by green tokens.

In the first configuration, on the one hand, the red token of MariaDB is found on the second place, which can only occur after a token has been moved from the initial place of the component to the output dock of transition `provision` and the action associated with this transition has been executed. The provide port bound to this place models data provided by the component, in this case the IP adress resulting from the action of `provision`. As soon as this second place is reached, the provide port `@IP mdb` is enabled, and the use port connected to it is provided. On the other hand, the initial token of Apache has been replaced by three tokens, one for each output dock. Two of these tokens have been used to fire respectively the transitions `bootstrap` and `pull`, and `pull` has terminated. However, the third token has remained in its dock. The transition `conf` could not be fired until the bound use port became provided. In this configuration, this is the case and the transition may be fired.

The blue configuration illustrates a case where parallel transitions `pull`, `conf`, and `bootstrap` of MariaDB are executed simultaneously.

Finally, the green configuration illustrates an example where MariaDB reaches its penultimate place. The provide port `mdb` bound to it models the service offered by the component. As soon as the place holds a token, the provide port becomes enabled. As a result, the use port of Apache that is connected to it becomes provided, and the transition `check` of Apache can be fired.

5. The Madeus formal model

In this section, the Madeus formal model is presented, in a streamlined version compared to our previous publication [6]. First the concepts of control component and assembly are formally introduced, then the operational semantics is detailed, and finally a structural properties of the model are given.

5.1. Control component

Internally, a Madeus control component is comprised of a set of *places* linked by *transitions*. The outside interface of components is provided by their *ports*, which can be bound to the internal elements.

Places. A component in Madeus is first defined by a set of places, denoted Π. In order to define transitions and handle their synchronization, we also introduce the notion of *docks*, each dock being attached to one place. The docks are divided between *input* and *output* docks. The set of input (respectively output) docks is denoted Δ_i (respectively Δ_o), and we use the notations $\Delta_i(\pi)$ and $\Delta_o(\pi)$ to denote the set of input and output docks attached to a given place π. Conversely, we denote by $\pi(d)$ the single place to which an input or output dock d is attached. A place π such that $\Delta_i(\pi) = \emptyset$ is said to be *initial*, while if $\Delta_o(\pi) = \emptyset$, the place is called *final*.

	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
	Transitions
Θ	set of transitions
$\mathbb{A}$	set of actions
	Ports
S_u	set of use ports of a component
S_p	set of provide ports
$\mathbb{T}$	set of types of ports
	Bindings
B_{S_u}	binding relation between use ports and transitions
B_{S_p}	binding relation between provide ports and set of places
	Assembly
C	set of components of an assembly
L	set of use-provide connections of an assembly
	Semantics
$\mathcal{M}$	subset of elements holding a token
$\mathcal{R}$	subset of places that have been reached
$\mathcal{E}$	set of ongoing actions

Table 2: Defining element of the Madeus formal model

The set of transitions of a component is denoted Θ. A transition $\theta \in \Theta$ is a triple (s, α, d) with $s \in \Delta_o$ an output dock, $d \in \Delta_i$ an input dock, and $\alpha \in \mathbb{A}$ the action associated to the transition, taken from the set of actions $\mathbb{A}$.

Coordination ports and bindings. The coordination ports of a control component are divided between a set of *provide ports* denoted S_p and a set of *use ports* denoted S_u. Ports are given a type among a set of types $\mathbb{T}$. The type of a given port p is denoted $\mathbb{T}(p)$. This simple typing discipline is a way to distinguish categories of services or data that ports may provide or use.

A provide port may be bound to sets of places. This binding relation is denoted $B_{S_p} \subseteq S_p \times \mathcal{P}(G)$. The binding concept is closely related to the operational semantics of Madeus that will be detailed in Section 5.3. A provide port is *enabled* if all the sets of places to which it is bound include at least one place that is reached. A place is reached if it holds *or* has previously held a token. Intuitively, bindings allow the modeling of conjunctions (all sets of places must be reached) and disjunctions (one set of place is reached if at least one of its place is reached) of constraints to control the enabling of provide ports. Note that once a port is enabled, it remains in that state forever. Thus, bindings represent places from which provide ports are provided.

A use port may be bound to transitions, indicating that those transitions can be fired only if the port is *provided*, i.e., connected to an enabled provide port. This binding relation is denoted $B_{S_u} \subseteq S_u \times \Theta$.

5.2. Assembly

An *assembly* of components represents the instantiation of components as defined in Section 5.1, and their connections through their coordination ports. In MADEUS, an assembly is defined as a pair (C, L), where C is a set of components, and $L \subseteq S_u^* \times S_p^*$ is the set of connections (links) between use ports and provide ports. For all the components $c_1, \ldots, c_n \in C$, we denote with a star any union of the corresponding sets, for instance $\Pi^* = \bigcup_{i=1}^{n} \Pi_i$. Linked ports must be of compatible types, i.e., $(u, p) \in L$ only if $\mathbb{T}(u) = \mathbb{T}(p)$. This simple typing system could be extended in future work with heritage and connectors for instance [**?**].

5.3. Operational semantics

At each moment in the execution of a MADEUS deployment assembly (C, L), the *configuration* of this assembly is defined by a tuple $\langle \mathcal{M}, \mathcal{R}, \mathcal{E} \rangle$, where

- $\mathcal{M} \subseteq \Pi^* \cup \Delta_i^* \cup \Delta_o^* \cup \Theta^*$ denotes the elements that hold a token;
- $\mathcal{R} \subseteq \Pi^*$ denotes the places that have been reached (have held a token);
- $\mathcal{E} \subseteq A$ denotes the actions that are being executed.

The initial configuration of an assembly is given by $\langle I, I, \emptyset \rangle$, where $I = \{\pi \mid \Delta_i(\pi) = \emptyset\}$ (i.e., the set of initial places).

We now formally present the operational semantics of the MADEUS model, given as a binary relation $\rightsquigarrow$ over configurations. The rules describing this relation are given in Figure 9.

An execution of an assembly A is a sequence of configurations

$$\langle I, I, \emptyset \rangle \rightsquigarrow \langle \mathcal{M}_1, \mathcal{R}_1, \mathcal{E}_1 \rangle \rightsquigarrow\rightsquigarrow \langle \mathcal{M}_2, \mathcal{R}_2, \mathcal{E}_2 \rangle \rightsquigarrow \ldots$$

An execution may be finite only if the last configuration $\langle \mathcal{M}_n, \mathcal{R}_n, \mathcal{E}_n \rangle$ is such that no rule applies.

Reaching place. The rule Reach_π describes the activation of a place π. It requires that all the input docks connected to the place hold a token. In the conclusion, those tokens are removed from the docks, and one token is placed on π, as illustrated in Figure 5.

Leaving place. The rule Leave_π defines the transfer of a token from a place π. The token is be removed from π, and a token added to each output dock attached to π. This rule is illustrated in Figure 6.

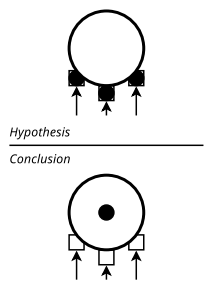


Figure 5: Rule Reach_π.

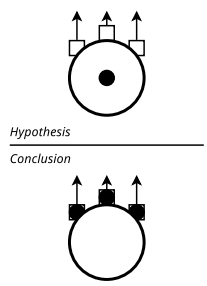


Figure 6: Rule Leave_π.

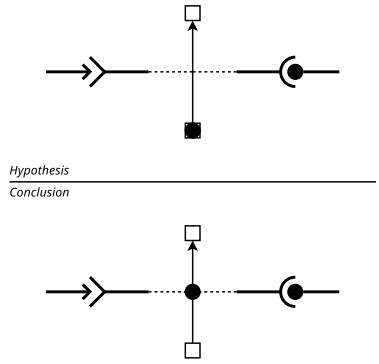


Figure 7: Rule Fire_θ.

Firing transition. The rule Fire_θ corresponds to the firing of a transition $\theta = (s, \alpha, d)$. The output dock s must hold a token, and any use port bound to θ must be provided. The token is transfered from the input dock to the transition itself, and the action α is executed, as illustrated in Figure 7

Terminating action. The rule Termin_α corresponds to the termination of an action α. Any such action in $\mathcal{E}$ can be removed by this rule.

Ending transition. The rule End_θ formally describes the end of a transition $\theta = (s, \alpha, d)$. It requires that θ holds a token and that the action α has terminated. When ending a transition, the token is moved from θ to the input dock d. Figure 8 illustrates this rule.

5.4. Structural constraints

A place π_1 is said to *precede* a place π_2 if there exists $(s, \alpha, d) \in \Theta$ such that $\pi(s) = \pi_1$ and $\pi(d) = \pi_2$. If there exists a sequence $(\pi_1, \pi_2, \ldots, \pi_n)$, such that π_i precedes π_{i+1} for any i where $1 \leq i < n$, then we say that π_n *depends* on π_1.

A MADEUS control component is said to be *well-formed* if there exists no place π that depends on itself. Intuitively, the component, viewed as a directed graph, must be acyclic.

The lemmas below all assume an assembly $A = (C, L)$ such that all components $c \in C$ are well-formed.

Lemma 5.4.1. *Any execution of A is finite.*

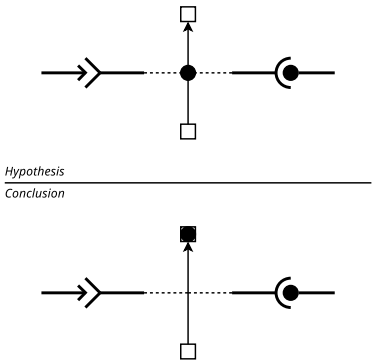


Figure 8: Rule End_θ.

Proof. By contradiction, assume that there exists an infinite execution of A. Since the sets of places, docks, transitions and actions are finite, the set of configurations is also finite. Therefore the execution must reach the same configuration more than once, i.e., it has a prefix

$$\langle I, I, \emptyset\rangle \rightsquigarrow \ldots \rightsquigarrow \langle \mathcal{M}, \mathcal{R}, \mathcal{E}\rangle_i \rightsquigarrow \ldots \rightsquigarrow \langle \mathcal{M}, \mathcal{R}, \mathcal{E}\rangle_j$$

such that $\langle \mathcal{M}, \mathcal{R}, \mathcal{E}\rangle_i = \langle \mathcal{M}, \mathcal{R}, \mathcal{E}\rangle_j$.
Let us denote $\mathcal{R}^+$ the set of places that are reached more than once. By case analysis over the relation $\langle \mathcal{M}, \mathcal{R}, \mathcal{E}\rangle_i \rightsquigarrow \langle \mathcal{M}, \mathcal{R}, \mathcal{E}\rangle_{i+1}$, we can check that $\mathcal{R}^+$ is non-empty:

- for the case of the rule Leave_π, a token is removed from the place π, therefore a token must be placed back on π to attain configuration $\langle \mathcal{M}, \mathcal{R}, \mathcal{E}\rangle_j$;
- for the case of the rules Reach_π, Fire_θ, Termin_α and End_θ, there must exist a transition $\theta = (s, \alpha, d)$ such that $\pi(s)$ is reached more than once.

Furthermore, because all components are well-formed, the precede-relation over places is well-founded, therefore there exists at least one place $\pi \in \mathcal{R}^+$ such that all places π' that precede π are in $\Pi \setminus \mathcal{R}^+$. This place is reached more than once, but all the places that precede it have been reached at most one time, leading to a contradiction. □

Lemma 5.4.2. *For any execution of A, if the conditions of a rule are eventually satisfied, this rule will eventually be applied.*

Proof. We can see that, for any execution and any condition of a rule, either the condition remains true forever (in particular, reached places and provided ports remain in that state) or the condition remains true until the rule is applied (e.g., a token is moved by the rule).

By Lemma 5.4.1, all executions must be finite, and by definition of an execution, no rule should be applicable in the final configuration. □

Lemma 5.4.3. *Let $c \in C$ be a control component, and π a place in it. In any execution of A, if all places π' that precede π are eventually reached, and all use ports of c are eventually provided, then π is eventually reached.*

Proof. Let us consider an execution such that (i) all places π' that precede π are eventually reached, and (ii) all use ports of c are eventually provided.

If $\Delta_i(\pi) = \emptyset$, then $\pi \in I$ and π is reached in the initial configuration.

Otherwise, for each place π' and each transition $\theta = (s, \alpha, d)$ where $\pi(s) = \pi'$ and $\pi(d) = \pi$, by (i), π' must eventually hold a token. By Lemma 5.4.2, θ must eventually be fired, α be terminated and θ be ended (rules Fire_θ, Termin_α and End_θ), thus any $d \in \Delta_i(\pi)$ will eventually hold a token. Furthermore, tokens are removed from $\Delta_i(\pi)$ only when π is reached (rule Reach_π), hence the conditions of Reach_π will eventually be satisfied, and π reached. □

Lemma 5.4.4. *Let c be a well-formed control component. For any execution where all use ports $u \in S_u$ are eventually provided, all places of c will eventually be reached.*

Proof. By contradiction, assume the existence of an execution such that (i) some place of c is never reached and (ii) all use ports are eventually provided.

Let $\mathcal{R}^\omega$ denote the set of places that are eventually reached in that execution. By (i), the set $\Pi \setminus \mathcal{R}^\omega$ is non-empty. Furthermore, because c is well-formed, the precede-relation is well-founded, therefore there exists at least one place $\pi \in \Pi \setminus \mathcal{R}^\omega$ such that all places π' that precede π are in $\mathcal{R}^\omega$. By (ii) and Lemma 5.4.3, $\pi \in \mathcal{R}^\omega$, leading to a contradiction. □

This lemma indicates that the MADEUS model provides, by construction, a guarantee of reachability at the component level, i.e., deadlocks cannot occur within a control component. At the level of an assembly, we cannot enforce similar structural constraints, because the different components are typically developed independently from each other, often by different actors. For this reason, generic reachability guarantees cannot be offered on assemblies. Instead we must analyze a given assembly. One way to perform this analysis is to use model-checking [38], but for most assemblies, a simpler analysis tool can be used. Such a solution is given in the next section.

6. Performance prediction model

In this section, we present a theoretical performance prediction model for MADEUS that estimates the total execution time of the commissioning of a given assembly, according to the execution time of each transition. Three goals are pursued with such a model. First, the theoretical execution time offers a way to evaluate the quality of the prototype of MADEUS. Second, for sysadmins, devops or developers, knowing the theoretical execution time of the commissioning procedure is useful. Third, in the case of

$$\frac{\pi \in \Pi^* \qquad \emptyset \subset \Delta_i(\pi) \subseteq \mathcal{M}}{\langle \mathcal{M}, \mathcal{R}, \mathcal{E} \rangle \rightsquigarrow \langle (\mathcal{M} \setminus \Delta_i(\pi)) \cup \{\pi\}, \mathcal{R} \cup \{\pi\}, \mathcal{E} \rangle} \ \mathsf{Reach}_\pi$$

$$\frac{\pi \in \Pi^* \qquad \pi \in \mathcal{M}}{\langle \mathcal{M}, \mathcal{R}, \mathcal{E} \rangle \rightsquigarrow \langle (\mathcal{M} \setminus \{\pi\}) \cup \Delta_o(\pi), \mathcal{R}, \mathcal{E} \rangle} \ \mathsf{Leave}_\pi$$

$$\frac{\theta = (s, \alpha, d) \in \Theta^* \qquad s \in \mathcal{M} \qquad \forall p.\ (p, \theta) \in B^*_{S_u} \rightarrow \mathit{provided}(p)}{\langle \mathcal{M}, \mathcal{R}, \mathcal{E} \rangle \rightsquigarrow \langle (\mathcal{M} \setminus \{s\}) \cup \{\theta\}, \mathcal{R}, \mathcal{E} \cup \{\alpha\} \rangle} \ \mathsf{Fire}_\theta$$

where $\mathit{provided}(p) \equiv \exists u.\ (u, p) \in L \wedge (\forall G.\ (p, G) \in B^*_{S_p} \rightarrow G \cap \mathcal{R} \neq \emptyset)$

$$\frac{\alpha \in \mathcal{E}}{\langle \mathcal{M}, \mathcal{R}, \mathcal{E} \rangle \rightsquigarrow \langle \mathcal{M}, \mathcal{R}, \mathcal{E} \setminus \{\alpha\} \rangle} \ \mathsf{Termin}_\alpha$$

$$\frac{\theta = (s, \alpha, d) \in \Theta^* \qquad \theta \in \mathcal{M} \qquad \alpha \notin \mathcal{E}}{\langle \mathcal{M}, \mathcal{R}, \mathcal{E} \rangle \rightsquigarrow \langle ((\mathcal{M} \setminus \{\theta\}) \cup \{d\}, \mathcal{R}, \mathcal{E} \rangle} \ \mathsf{End}_\theta$$

Figure 9: The operational semantics of Madeus.

a fully autonomic commissioning and reconfiguration process, which is left to future work, theoretical information on the execution time is needed to take adequate decisions, in particular for critical systems and services.

To build this performance prediction model, we need an estimation of the execution time of all the actions in the assembly, given as a function $t : \mathbb{A} \rightarrow \mathbb{R}^+$. Computing precise execution times for the kind of actions used in a deployment can be difficult and is itself the subject of many studies, e.g., using an analytical model or a statistical approach. However, we find that estimations are sufficient as long as they preserve the relative difference between transitions [38]. Intuitively, we automatically deduce the execution flow of a Madeus assembly based on Madeus' formal semantics. This is done by generating a dependency graph representing the execution flow of each Madeus component in the assembly and connecting them together according to their dependencies (the connections between their coordination ports). Then, a *source* vertex is connected to the vertices representing the beginning of the execution of each component, and a *sink* vertex is connected to the vertices representing the end of the execution of each component. Thus, a dependency graph representing the execution of the whole assembly is obtained. By weighting the arcs corresponding to the transitions with the individual execution times of their actions (and the other ones with 0), we can compute the expected total execution time of the assembly as the weighted longest path from the *source* vertex to the *sink* vertex.

6.1. Assumptions

The performance model given here is valid only for assemblies such that any set of places bound to a provide port contains exactly one place. Indeed, if a provide port is bound to a set containing multiple places, the requirement imposed by this binding is satisfied as soon as one of the places is reached. Thus, the total time then depends on the earliest time at which one of these places is reached, which cannot be modeled as a longest path. In practice this is not a severe restriction, as ports requirements very rarely include disjunctions. Indeed, none of our use cases rely on this feature.

The estimation provided by the model is a best-case scenario: it assumes that all actions are started as soon as possible, and that the hardware has the capacity to run all possible parallel actions without affecting their execution times.

6.2. Notations

Recall that an assembly is a tuple $A = (C, L)$. In the following, for any component $c_i \in C$, any of its associated set is denoted X_i, e.g., Π_i for the set of places of the component c_i. As previously, for each set X defining a component, we denote X^* the union (in the case of an assembly) or the extension (in the case of a function) for all components. For instance $\Pi^* = \bigcup_{i=1}^{n} \Pi_i$ denotes the set of all places in the assembly.

The execution flow graph is an oriented weighted graph (V, E) where V is the set of vertices and E is the set of weighted arcs with elements in $V \times \mathbb{R}^+ \times V$. We define V and E in the following.

6.3. Vertices

For each place, we introduce one vertex that represents the event of the place being reached.

$$V_\Pi = \bigcup_{\pi \in \Pi^*} \{v_\pi^{\text{reach}}\}$$

For each transition, we introduce a vertex representing its firing.

$$V_\Theta = \bigcup_{\theta \in \Theta^*} \{v_\theta^{\text{fire}}\}$$

For each provide port we introduce one vertex representing its enabling.

$$V_{S_p} = \bigcup_{p \in S_p^*} \left\{ v_p^{\text{enable}} \right\}$$

Finally, we define V as the union of all these, plus one source and one sink vertices.

$$V = V_\Pi \cup V_\Theta \cup V_{S_p} \cup \left\{ v^{\text{source}}, v^{\text{sink}} \right\}$$

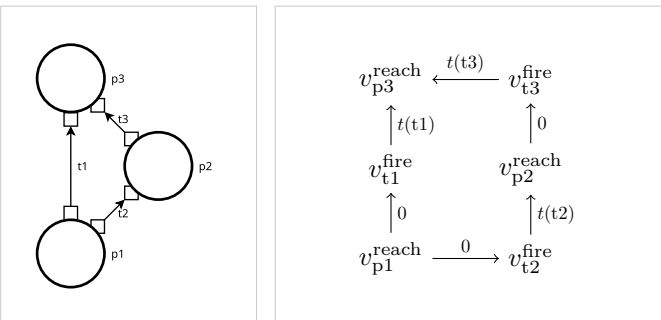
6.4. Arcs

In the dependency graph (V, E), arcs represent time constraints: the event represented by the destination vertex must happen after the one represented by the source vertex, at least w seconds apart where w is the weight of the arc. In practice, arcs corresponding to actions are weighed with the corresponding time, while other arcs have a weight 0 and merely represent a dependency.

For each transition $\theta = (s, \alpha, d)$, we introduce two arcs. The first, from $v_{\pi(s)}^{\text{reach}}$ to v_θ^{fire}, represents the fact that θ may only be fired after $\pi(s)$ has been reached. The second, from v_θ^{fire} to $v_{\pi(d)}^{\text{reach}}$ represents the fact that a token may enter $\pi(d)$ only after θ has ended, which requires a time $t(\alpha)$ corresponding to the action.

$$E_\Theta = \bigcup_{\theta = (s,\alpha,d) \in \Theta^*} \left\{ \left(v_{\pi(s)}^{\text{reach}}, 0, v_\theta^{\text{fire}} \right), \left(v_\theta^{\text{fire}}, t(\alpha), v_{\pi(d)}^{\text{reach}} \right) \right\}$$

Figure 10 depicts the dependency graph corresponding to an example made of three transitions *t1*, *t2* and *t3*.



(a) Madeus internal net example (b) Dependency graph

Figure 10: A set of Madeus transitions and their corresponding equivalent dependency graph

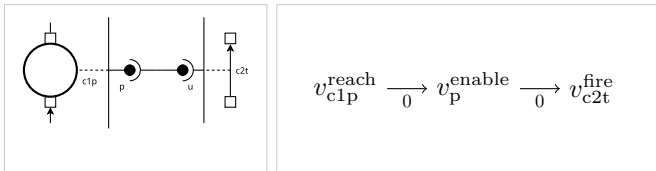
For each provide port p and each place π such that $\{\pi\}$ is bound to p, we introduce an arc from v_π^{reach} to v_p^{enable}. This represents the enabling of the port after all (sets of) places bound to the port have been reached.

$$E_{S_p} = \bigcup_{(p, \{\pi\}) \in B_{S_p}^*} \left\{ \left(v_\pi^{\text{reach}}, 0, v_p^{\text{enable}} \right) \right\}$$

Additionally, for each provide port p and each transition θ such that p is connected to a use port that is bound to θ, we introduce an arc from v_p^{enable} to v_θ^{fire}. This represents the activation of the transitions bound to a use port, after the port is provided.

$$E_{S_u} = \bigcup_{\substack{(u,p) \in L \\ (u,\theta) \in B_{S_u}^*}} \left\{ \left(v_p^{\text{enable}}, 0, v_\theta^{\text{fire}} \right) \right\}$$

Figure 11 depicts the dependency graph corresponding to a provide port p bound to a place *c1p* and connected to a use port u which is bound to a transition *c2t*.



(a) Madeus port connection (b) Dependency graph

Figure 11: A connection and its corresponding dependency graph

For each initial place π, we introduce an arc from v^{source} to v_π^{reach}, representing the fact that a token is placed in each initial place in the initial configuration.

$$E_I = \bigcup_{\pi, \Delta_i(\pi) = \emptyset} \left\{ \left(v^{\text{source}}, 0, v_\pi^{\text{reach}} \right) \right\}$$

For each final place π, we introduce an arc from v_π^{reach} to v^{sink}. Intuitively, this represents the fact that the commissionning is over only after all components have reached the places without outgoing transitions.

$$E_F = \bigcup_{\pi, \Delta_o(\pi) = \emptyset} \left\{ \left(v_\pi^{\text{reach}}, 0, v^{\text{sink}} \right) \right\}$$

Finally, we define E as

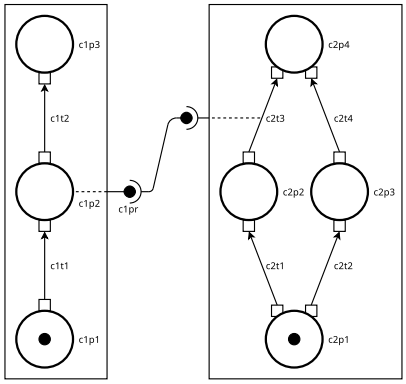
$$E = E_\Theta \cup E_{S_p} \cup E_{S_u} \cup E_I \cup E_F$$

Figure 12 depicts a complete example with a Madeus assembly and its associated dependency graph.

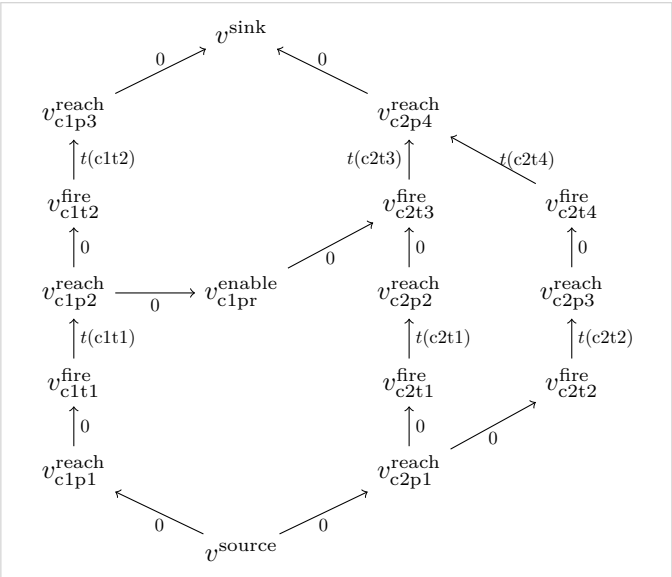
6.5. Time estimation

In the following, we denote $DG_A = (V, E)$ the dependency graph corresponding to an assembly $A = (C, L)$. We define the time estimation of the execution of the Madeus assembly A to be the length of the waighted longest path from v^{source} to v^{sink} in DG_A. Clearly, the size of DG_A is linear in the size of A. Furthermore, the graph is either a DAG or has a path of infinite length, so the longest path can be computed in polynomial time.

Since our execution model does not include a notion of time, proving the correctness of this performance model is outside the scope of this paper. However we show that this performance model can be used to assess reachability.



(a) MADEUS assembly



(b) Dependency graph

Figure 12: Two connected components and their equivalent dependency graph

Lemma 6.5.1. *Let A be an assembly of well-formed components. If the longest path from v^{source} to v^{sink} in DG_A has a finite length, then in any execution of A, all places in A are eventually reached.*

Proof. By construction of DG_A, there exists no cycle formed only of arcs of weight 0. Together with the assumption that there are no paths of infinite length from v^{source} to v^{sink}, this implies that no path from v^{source} to v^{sink} features a cycle. Since every vertex in DG_A is located on some path from v^{source} to v^{sink}, we have that DG_A is acyclic.

Then we can use well-founded induction on the structure of the graph to verify that, for any vertex v, if $v = v^{\text{reach}}_{\pi}$ for some place π, then π is eventually reached in any execution of A. The proof is similar to that of Lemma 5.4.3. □

To sum-up, from a given assembly without disjonctions on provide ports can be built an execution flow graph. If this graph does not contain infinite paths, which is easy to verify, and by Lemma 6.5.1, the reachability of the deployement modeled by the assembly is verified. The disjonction case is time dependent and creates non-determinism. The reachability of such complex cases should be verified by other techniques such as Model-checking [38].

7. Experimental synthetic evaluation

All results presented below can be reproduced by following a publicly available lab[9].

7.1. Prototype implementation details

MADEUS was implemented using PYTHON. As illustrated in Listings 1, 2 and 3, users of MADEUS declare a component by creating a class inheriting from the internal *Component* class, and containing a description of the internal net (places and transtions) as well as actions associated with transitions. These actions are PYTHON functions that can perform remote actions using SSH, Ansible or other tools. Users can then create an assembly by extending the appropriate classes and listing components and port connections. This assembly can then be executed according to the semantics of MADEUS. The commissioning is over if the only elements holding tokens are places without outgoing transitions. The semantics is executed by attempting to apply each rule on each component until this condition is met. When a transition is fired, the corresponding PYTHON function is executed in a thread, which does not block the execution of the semantics. Note that PYTHON threads do not take advantage of hardware parallelism capabilities, but because the functions usually run other (possibly remote) processes to launch configuration commands, this is not an issue. The Termin_α rule can be executed for the corresponding function when it has finished its execution.

7.2. Results

Madeus is a model that relies on the description of a control component for each software module to be deployed. It is a low-level model, therefore the developer is responsible for the choices of actions performed in the transitions as well as data and service exchanges (e.g., scp, VPN, REST API, etc.). This section evaluates the overhead introduced by the prototype of MADEUS as well as its scalability when increasing the number of components and transitions. These experiments are dry runs, meaning that transitions do not contain any code or command for which the execution time is unknown or variable, and instead use sleep commands to simulate time-consuming tasks. This allows us to measure the overhead introduced by the prototype. In addition to this, the experiments presented in this section are compared to the expected performance computed by the prediction model detailed

[9] https://gitlab.inria.fr/VeRDi-project/madeus-journal

in Section 6. This way, we validate both the theoretical performance model compared to reality, and the expected performance of the prototype on the set of use cases.

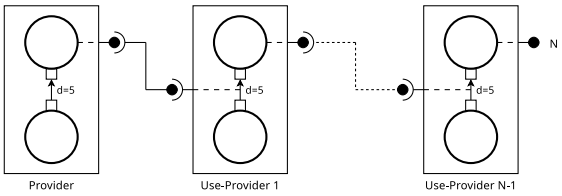


Figure 13: The MADEUS sequential assembly of Benchmark (A), with N components.

This evaluation is composed of three benchmarks illustrated in Figures 13 and 14. The first benchmark, denoted (A), models a sequential MADEUS assembly, depicted in Figure 13. It is composed of one *provider* component made of a transition and two places, and $N-1$ *user-provider* components that are also composed of a transition and two places, but where the transition uses the provide port of the preceeding component. The components are connected in a chain, resulting in sequential execution.

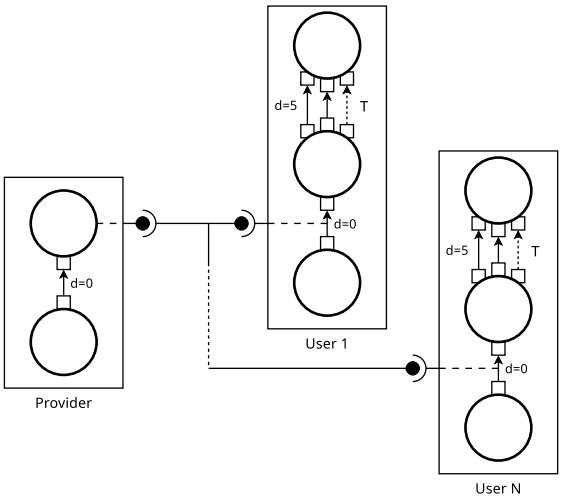


Figure 14: The MADEUS parallel assembly of Benchmark (B) and (C), with N parallel components, and T parallel transitions.

The two other benchmarks model MADEUS parallel assemblies and are depicted in Figure 14. The first assembly, denoted (B) evaluates parallelism at the component level, called *inter-comp* and *inter-comp-tasks*. The second one, denoted (C), evaluates parallelism at the transition level, i.e., *intra-comp-tasks*, when transitions are performed simultaneously. Both benchmarks use the same assembly that is composed of an initial *provider* component and N parallel *user* components connected to the provider. Each *user* component contains a first transition that uses the service provided by the provider component and T parallel transitions. For Benchmark (B), the number of components varies from 1 to 40 components with a single transition per component $T = 1$. For Benchmark (C), the number of components is fixed to $N - 1 = 1$ and the number of transitions varies from 1 to 40.

			theory(s)	measured(s)	diff
(A)	components	1	5	$5.015 \pm 0.021\%$	0.3%
		10	50	$50.11 \pm 0.0004\%$	0.22%
		20	100	$100.21 \pm 0.0004\%$	0.21%
		30	150	$150.316 \pm 0.0003\%$	0.21%
		40	200	$200.42 \pm 0.0009\%$	0.21%
(B)	components	1	5	$5.016 \pm 0.0049\%$	0.32%
		10	5	$5.021 \pm 0.1\%$	0.42%
		20	5	$5.026 \pm 0.12\%$	0.53%
		30	5	$5.028 \pm 0.068\%$	0.56%
		40	5	$5.03 \pm 0.073\%$	0.6%
(C)	transitions	1	5	$5.016 \pm 0.0057\%$	0.32%
		10	5	$5.024 \pm 0.094\%$	0.48%
		20	5	$5.025 \pm 0.06\%$	0.5%
		30	5	$5.029 \pm 0.098\%$	0.58%
		40	5	$5.033 \pm 0.136\%$	0.66%

Table 3: Theoretical and measured results of our three synthetic benchmarks on *nantes-ecotype*. The difference between the measured and expected theoretical time is represented as a percentage.

Experiments have been performed on a single node of the *nantes-ecotype* cluster of the experimental platform Grid'5000[10]. The detailed configuration of the node is given in Table 5. Each experiment presented in the results is an average of ten executions. The duration of some transitions of the benchmarks are set to $d = 0$ and the transitions that are interesting for the results are set to $d = 5$ (seconds). This execution time is guaranteed by a call to the sleep function of PYTHON. This execution time has been chosen to get a coherent and readable scale of results.

Table 3 presents the results of Benchmarks (A), (B) and (C) with the theoretical execution time, the measured execution time as well as its standard deviation, and the proportional difference between them. As (A) models a sequential assembly, the theoretical execution time is simply the sum of the execution time of each transition of each component (N components).

For Benchmark (B), the ideal theoretical performance is constant, and equal to the duration of the main transition of *user* components, i.e., 5 seconds. Indeed as components are independent from each other (no ports) they are deployed simultaneously.

Similarly, for Benchmark (C), because transitions are executed simultaneously, the theoretical expected time is constant, and equal to the duration of one of the T transitions, i.e., 5 seconds.

For all benchmarks, results on the MADEUS prototype

[10] www.grid5000.fr

are only slightly superior to the ideal theoretical result, showing that the prototype does not add significant overhead to the process even for a large number of sequential components, parallel components and parallel transitions on the same node. We may have increase the number of components and transitions but we claim that 40 parallel transitions and 40 components on a single node is already beyond normal usage when deploying distributed software systems.

These results allow us to point out that MADEUS does not by itself add a significant overhead to the deployment — at most 40 milliseconds in these synthetic experiments. Note however that according to the type of commands performed in the transitions, an additional overhead could be observed. For instance, a high number of simultaneous SSH connections could add an important overhead. In the experiments presented in this paper, this problem is handled by using Ansible within actions. Indeed, Ansible optimizes the number of SSH connections on a single host, limiting the impact of this issue.

8. Real use case evaluation

This section present an evaluation of MADEUS on a real case, the commissioning of OpenStack. As in the previous section, all results presented below can be reproduced by following a publicly available lab[11]. First, this lab offers the possibility to reproduce the figures presented in this section from the raw data obtained in our experiments on Grid'5000. Second, the lab outlines the complete process to reproduce the experiments on the Grid'5000 platform.

8.1. OpenStack

OpenStack is the de facto standard open-source solution to address the IaaS level of the cloud paradigm, in other words OpenStack can be seen as the open-source operating system of the cloud. Since 2010, its community has gathered nearly 700 organizations (such as Google, IBM or Intel) and has produced more than 20 million lines of code. Its adoption is still growing in various domains such as public administration, e-commerce and science[12].

OpenStack is a large distributed software system that brings together almost 100 software projects. Various projects are in charge of specific aspects of the infrastructure management (e.g., provisioning virtual machines, providing them with storage, interconnecting them through networks), and their cooperation is the key to providing the features required for cloud management. Those projects are themselves composed of several software modules that are responsible for very specific tasks (e.g., placement, hypervisors, etc.). Although not all are mandatory to deploy an operable IaaS, 250 software modules are available in those projects. An OpenStack instance is a composition of some of those modules by the operator. The chosen modules then cooperate to respond to the operator requirements. For instance, the operator may need services to manage virtual rather than bare-metal machines, object storage rather than file systems, while VLAN networks and billing services may not be desired in her use case. As defined in the large OpenStack documentation, each software module has its own commissioning process, and may depend on other modules commissionings. Thus, the deployment of a typical OpenStack instance involves many software modules whose commissioning process is characterized by a large amount of tasks and interplay. As a consequence, the commissioning process of OpenStack is complex to understand and can be very long when tasks are executed sequentially.

Kolla is one of the most popular tools for deploying OpenStack in production. It relies on ANSIBLE to deploy the modules of OpenStack as DOCKER containers, and will be our reference in the rest of this section. It allows operators to quickly deploy a basic OpenStack instance, but also offers complete customization for advanced administrators. The use case described in this section corresponds to the default Kolla deployment, which provides the essential mechanisms to operate an infrastructure with OpenStack.

In the following, we show how the commissioning process of an OpenStack project can be translated into a MADEUS component. Leveraging MADEUS enables us to express tasks and components coordination. As a consequence, the MADEUS modeling improves the clarity of the global commissioning process, and can be used to reduce commissioning time by exploiting *SIMH*, *inter-comp*, *inter-comp-tasks* and *intra-comp-tasks* parallelism levels.

To compare the performance of Kolla and MADEUS, we have defined 11 MADEUS components based on the ANSIBLE roles defined in Kolla's playbooks (i.e., ANSIBLE sequence of components to deploy). Their names usually indicate the OpenStack project they deploy. Table 4 lists these components and indicates which aspect of the cloud management they are responsible for.

Each of these control components has been designed such that ANSIBLE roles and their associated tasks are divided into MADEUS transitions. Table 4 displays some metrics for the MADEUS components designed from ANSIBLE roles of Kolla: the number of places, transitions and ports. This table also indicates whether parallel transitions exist, i.e., *intra-comp-tasks* parallelism, denoted `ICT`. Additionally, the higher the number of ports, the more coordination must be performed by MADEUS during the commissioning. As depicted in Table 4, Nova, Glance, Neutron and MariaDB are components of particular interest since they contain more transitions and ports than the others.

[11] https://gitlab.inria.fr/VeRDi-project/madeus-journal/lab

[12] See http://superuser.openstack.org/ for further information.

	Roles	Places	Transitions	Ports	ICT
Nova	Manages compute instances (e.g., Virtual machines)	5	8	8	✓
Glance	Compute image store	3	4	7	✓
Neutron	In charge of network resources	3	4	7	✓
MariaDB	An SQL server to store persistent information	4	5	4	✓
Keystone	Aauthentication, and service discovery	3	2	4	-
RabbitMQ	The message bus for inter-service communication	2	1	3	-
HAProxy	Load-balances the requests to OpenStack controllers	2	1	7	-
OpenVSwitch	Virtualizes network functions	3	1	2	-
MemCached	Caches ephemeral data for most OpenStack projects	2	1	2	-
Facts	Collects informations about every nodes	2	1	1	-
Common	Common utilities (e.g., cron, fluentd)	3	2	2	-
Total		32	30	47	

Table 4: Number of places, transitions and ports for each MADEUS component of the OpenStack assembly of Figure 15. *intra-comp-tasks* (`ICT`) indicates if parallel transitions exist in the component.

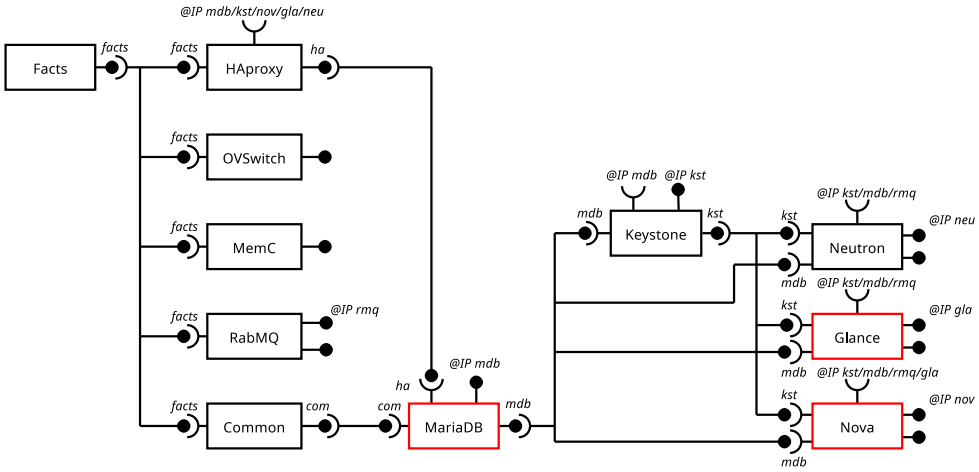


Figure 15: Simplified MADEUS assembly of the Kolla-based OpenStack deployment containing 11 components. Connections between data ports are not depicted. Red components are detailed in Fig. 16.

8.2. Expressivity and Separation of Concerns

Figure 15 depicts the use case from the perspective of the sysadmin or devops engineer (i.e., at the level of the MADEUS assembly). For the sake of simplicity and readability, the connections between ports representing data are not represented. This figure helps to understand the high level of interplay between components. For instance, Neutron, Glance and Nova require Keystone, while Keystone itself requires a database (i.e., MariaDB) for its commissioning process. Regarding separation of concerns, the devops engineer does not need to understand component internals. She just needs to compose the desired components by listing them and connecting their compatible ports. As a consequence, a component can be replaced by another one if they expose the same interfaces. For instance the operator could replace MariaDB with MySQL, another component that also implements a database and exposes the same ports.

We now study our use case from the perspective of the developer, focusing on the three colored control components from Figure 15: MariaDB, Nova and Glance. Figure 16 depicts the internals and interactions of these components.

The dependencies previously observed at the assembly level are more detailed at the developer level. For instance, if we isolate Nova and Glance, Figure 15 lets us think that Glance must be deployed before Nova, but it is clear here that once Nova obtains Glance's IP address (provided by the first place in Glance), both components can be deployed in parallel. This shows how MADEUS can be leveraged for *inter-comp* and *inter-comp-tasks* parallelism. In addition, as discussed previously, Figure 15 suggests that MariaDB must be deployed before Keystone, and Keystone before Neutron, Glance and Nova. However, the MADEUS representation depicted in Figure 16 shows that only the *register* transition of Glance and Nova requires the Keystone catalog service to be available (i.e., to register themselves in the catalog). We see on the figure that other tasks can be executed in parallel, while *register* waits for Keystone (e.g., for Nova, this transition is independent from all others). Similarly, Figure 16 depicts many parallel transitions for each component, showing how MADEUS can be leveraged for *intra-comp-tasks* parallelism in this use case.

8.3. Experimental Setup and Parameters

This section first defines the experimental setup: (i) how modules are distributed among nodes (i.e., servers or machines) and (ii) the testbed characteristics. Then we describe the parameters used during our experimentation: (iii) the assemblies we designed to compare our contribution with the related work and (iv) the way DOCKER images are fetched by nodes.

Node roles and module distribution. Each of the 11 components defined earlier in Kolla is in charge of an OpenStack project. As mentioned, each OpenStack project contains multiples software modules. Hence, each component actually deploys different modules (36 in total). A basic multi-node Kolla deployment targets three nodes. First, the *Control* node, which hosts control services, APIs and databases, deploying 16 services. The second one is the *Network* node that hosts network agents and HAProxy, and contains 11 services. Finally, the *Compute* node, in charge of compute services and VM placement, hosts 9 services.

Cluster	CPU	Memory	Network
Nantes Ecotype	2× Intel Xeon E5-2630L v4, 10 cores/CPU	128GB	2× 10Gbps
Lyon Nova	2× Intel Xeon E5-2620 v4, 8 cores/CPU	64GB	10Gbps

Table 5: Grid'5000 clusters configurations.

Testbed and resource provisioning. Our evaluations were conducted on two clusters of the experimental platform Grid'5000: *nantes-ecotype* and *lyon-nova*. Table 5 shows the hardware configuration for both clusters. The cluster *nantes-ecotype* has better hardware (CPU, memory, networkinterfaces) than *lyon-nova*, as described in the table. To design reproducible benchmarks, we used EnosLib[13], a library to build experimental frameworks on multiple testbeds (e.g., Grid5000, LibVirt), and Execo[14], another library for prototyping experiments. Since Kolla, our reference, does not manage resource provisioning, we do not include this phase in the use case, nor in our benchmark. Although resource provisioning could be managed by MADEUS, this step is left to EnosLib and not counted in execution times.

Assemblies. Our performance evaluation compares three assemblies that are designed to capture the behavior of ANSIBLE, AEOLUS and MADEUS. To that end, the component internals for each assembly vary with regards to the number of places, transitions and ports. Importantly, we re-used the ANSIBLE files provided by Kolla and split them into component transitions. By using MADEUS to coordinate ANSIBLE execution, it is possible for us to provide a way to fairly compare these solutions. Moreover, by using divided Kolla roles, the *SIMH* parallelism level is handled by ANSIBLE in the three assemblies.

The first assembly, called *m_ansible*, matches the Kolla-ANSIBLE commissioning process. Each component is triggered sequentially, in the same way and order as ANSIBLE triggers sequentially the roles defined in Kolla. Since the coordination between components is simply sequential, the

[13] https://gitlab.inria.fr/discovery/enoslib

[14] http://execo.gforge.inria.fr/doc/latest-stable/index.html

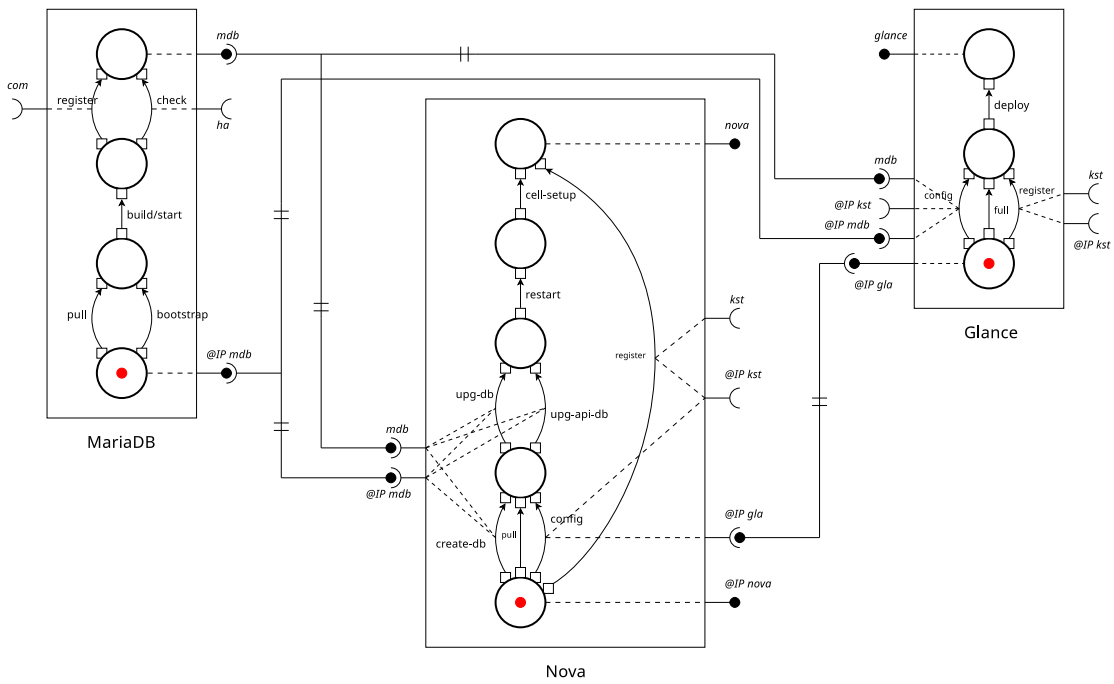


Figure 16: A detailed sub-part of the previous component assembly to deploy OpenStack.

components have two states connected by a single transition which performs all the commissioning tasks, such as in Kolla (i.e., no *intra-comp-tasks* parallelism). Each time a component is deployed, it activates the commissioning process of the next one (i.e., neither *inter-comp* nor *inter-comp-tasks* parallelism). This assembly features the first level of parallelism, which is managed by Ansible when tasks are mapped to multiple nodes, i.e., *SIMH* parallelism.

The second assembly, called *m_aeolus* is equivalent to an Aeolus commissioning of OpenStack. It provides parallelism at both the *inter-comp* and *inter-comp-tasks* levels in addition to *SIMH*, and no *intra-comp-tasks* parallelism. Coordination is performed through component ports. In this assembly, most components are built with two sequential transitions. When the assembly is initiated, the first transition of those components are triggered, while the second one depends on another component.

The third assembly, called *madeus*, leverages our contribution to commission OpenStack. It corresponds to the one we previously described when presenting the use case. Components are defined on a case-by-case basis, based on our understanding of the OpenStack commissioning process. As depicted previously in Table 4, most components include multiple places and transitions. This assembly makes use of all the parallelism expressiveness of Madeus.

Docker container registry. Finally, since Kolla relies on Docker containers, fetching Docker images has a significant impact on our results: images have to be downloaded, before being decompressed. To be as neutral as possible we have conducted experiments with three different modes for handling those images: (1) *cached* mode, where images are previously placed on OpenStack nodes, so fetching Docker images has very low impact on the results; (2) *local* mode, where images are previously downloaded on a new dedicated node of the cluster, from which images can be loaded (i.e., a local Docker registry); (3) *remote* mode, in which images are fetched from an Internet repository (i.e., the DockerHub registry). Table 6 gives for each OpenStack node (i.e., Compute, Network and Control) the number of Docker images to download and their compressed size. As depicted in this table, more than 10GB must be downloaded in our use case. Furthermore, the control node has to download almost twice as much data as the other nodes.

	Compute	Network	Control
Number of images	9	11	16
Total Size (MB)	2767	2705	4916

Table 6: Number of Docker images per node and their cumulated size in MB to download from the registry.

8.4. Results

In this section, we analyze the results of our benchmark through different aspects: (i) the performance of each assembly; (ii) the adequation between the theoretical predicted performance and the measured results and (iii) the influence of registry modes on our results.

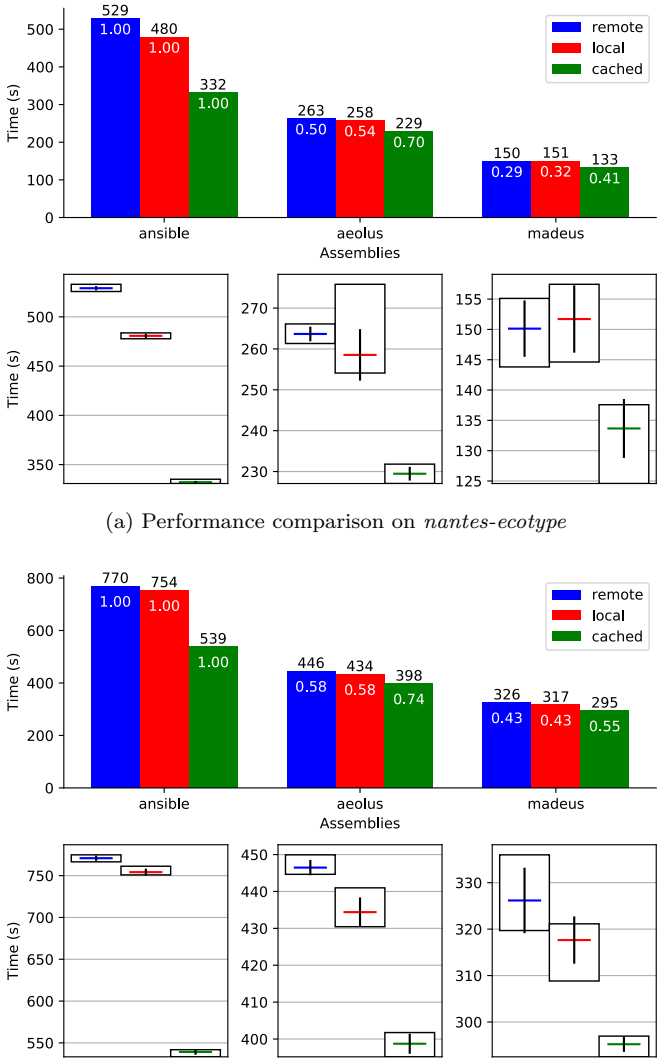
In these studies, we refer to Figures 17a and 17b which respectively show our results on *nantes-ecotype* and *lyon-*

			remote	local	cached
measured	mean(s)	ansible	529 ± 2	480 ± 2	332 ± 1
		aeolus	263 ± 1	258 ± 6	229 ± 1
		madeus	150 ± 4	151 ± 5	133 ± 4
	gain	ansible	0%	0%	0%
		aeolus	50%	46%	30%
		madeus	71%	68%	59%
theoretical(s)	max	ansible	533	483	335
		aeolus	266	275	231
		madeus	155	157	137
	min	ansible	525	477	330
		aeolus	261	254	227
		madeus	143	144	124

Table 7: Measured and theoretical results of our benchmark on *nantes-ecotype*.

			remote	local	cached
measured	mean(s)	ansible	770 ± 3	754 ± 4	539 ± 3
		aeolus	446 ± 2	434 ± 3	398 ± 2
		madeus	326 ± 7	317 ± 5	295 ± 1
	gain	ansible	0%	0%	0%
		aeolus	42%	42%	26%
		madeus	57%	57%	45%
theoretical(s)	max	ansible	774	761	541
		aeolus	449	440	401
		madeus	335	321	296
	min	ansible	766	750	533
		aeolus	444	430	395
		madeus	319	308	292

Table 8: Measured and theoretical results of our benchmark on *lyon-nova*.



(a) Performance comparison on *nantes-ecotype*

(b) Performance comparison on *lyon-nova*

Figure 17: Recorded time in seconds for OpenStack commissioning with different clusters, assemblies and registry modes. Upper parts represent mean values with relative ratios for each result compared to the reference *m_ansible*. Lower parts display means, standard deviations and the minimum and maximum values computed from the theoretical performance model, depicted as boxes.

nova clusters. The upper part displays the recorded times to commission OpenStack as a function of the three studied assemblies. For a better understanding of the comparison, the value of each result is written on top of the bars, while the ratio compared to *m_ansible*, our reference, is displayed below the bars' edges. Furthermore, for each assembly on the X-axis, the results for the three DOCKER registry settings are displayed with different colors: blue, red and green respectively for *remote*, *local* and *cached*. On the lower part of the figures, the means for each result are depicted as blue, red and green horizontal lines, the related standard deviations are represented by vertical lines, while boxes represent the minimum and maximum values computed with the theoretical performance model

of Section 6. For the sake of readability, the scale of the lower parts are differents for each assembly. Each result corresponds to the average computed across 10 iterations. The corresponding numerical values are also displayed in Table 7 and Table 8.

Impact of assemblies. We now compare the time measured to commission the three assemblies previously defined: *m_ansible*, *m_aeolus* and *madeus* (the lower, the better in Figure 17). As expected, the time required to commission these assemblies reflects the level of parallelism they implement. Since *m_ansible* is limited to the first parallelism level, i.e., *SIMH*, its commissioning time is longer than *m_aeolus*. By featuring *inter-comp* and *inter-comp-tasks* parallelism, the latter outperforms the former from 26% (*lyon-nova*, *cached*) to 50% (*nantes-ecotype*, *remote*). Leveraging *intra-comp-tasks* parallelism enables *madeus* to outperform *m_ansible* from 45% (*lyon-nova*, *cached*) to 71% (*nantes-ecotype*, *remote*), and *m_aeolus* from 16% (*lyon-nova*, *remote*/*local*) to 30% (*nantes-ecotype*, *cached*). The OpenStack commissioning on *nantes-ecotype* goes from almost 9 minutes with *m_ansible* to less than 3 minutes with *madeus*.

To go further, we propose to analyze the commissioning process at the level of transitions (i.e., tasks). To investigate this aspect, we implemented in MADEUS the ability to generate Gantt charts that display the execution time of the different transitions for each component. Figures 18a, 18b, and 19a respectively represent the Gantt charts of the commissioning execution of *m_ansible*, *m_aeolus* and *madeus*, when the registry is set to *cached* on *nantes-ecotype*. Each line of these figures represents a transition as a function of the elapsed-time displayed on the X-axis. First, as previously explained, Figure 18a shows that a single transition exists in each component of the *m_ansible* assembly. Thus, here, each line also corresponds to one component commissioning. As expected, the figure shows that each component is deployed in a sequential way. The first level of parallelism (i.e., *SIMH*) is not visible in these figures since it is handled internally by ANSIBLE playbooks executed in each transition of the assembly. One can note that Nova, MariaDB, Glance, Keystone and Neutron take particularly long to commission.

m_aeolus and *madeus* accelerate the process by (i) splitting the transition of components into smaller ones and (ii) managing dependencies between them more finely (depending on the ability to express *inter-comp*, *inter-comp-tasks* and *intra-comp-tasks* parallelism). Figure 18b illustrates that the components we highlighted previously (e.g., Nova in yellow, Neutron in gray) are based on two transitions in *m_aeolus*. This figure shows that this assembly can leverage both the *inter-comp* and *inter-comp-tasks* parallelism levels since multiple components and tasks (e.g., `glance.pull` and `haproxy.deploy`) are executed in parallel. As a consequence, the commissioning time drops from 5 minutes 31 seconds to 3 minutes 49 seconds.

Finally, Figure 19a clearly shows how MADEUS leverages the fourth level of parallelism (i.e., *intra-comp-tasks* parallelism) by displaying multiple transitions executed in parallel. For instance, `nova.pull` and `nova.config` (depicted in orange), are performed simultaneously which is not possible with ANSIBLE or AEOLUS. Consequently, the commissioning time drops from 5 minutes 31 seconds to 2 minutes 8 seconds.

	cached	local	remote
pull(s)	13	48	52
pull(%)	10%	32%	35%

Table 9: Time spent in the *pull* transition from Nova and percentage compared to the total time for *madeus* commissioning.

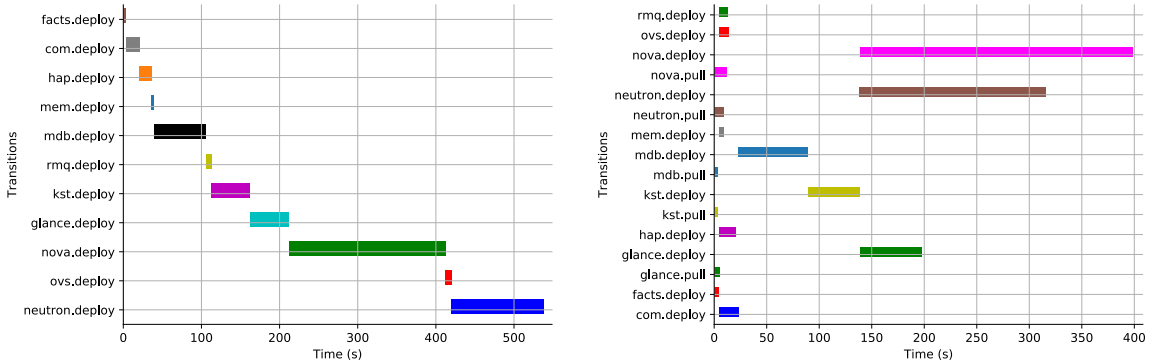
Precision of the performance model. The maximum and minimum values obtained by the performance model described previously is depicted in Table 7 and Table 8. These theoretical values are computed from the average minimum and maximum transitions execution times observed for the ten experiments of each benchmark. When analyzing these results, one can note that the measured mean is always between the expected maximum and minimum.

Influence of registry modes. Table 7 contains the gains relatively to *m_ansible*, associated to Figure 17a. This table shows that the gain obtained with *local* and *remote* registries are better than the one obtained with *cached* DOCKER images.

To better understand the origin of this difference, we can compare Figure 19a and Figure 19b. The former depicts the time spent by all transitions of *madeus*, on *nantes-ecotype*, when the DOCKER registry is set on *cached*, while it is set to *remote* for the latter. As we can see on the figures, the difference is mainly due to the parallel execution of `pull` transitions which are much longer in *remote* (and similarly in *local*) than in *cached* where images are already on nodes.

Table 9 represents the execution time of transition `pull` of the Nova component on *nantes-ecotype*, as well as the percentage compared to the total sequential execution time with *madeus*. Transition `nova.pull` takes 35% of the total commissioning time in *remote*, and only 10% in *cached*. This confirms that the time spent in transition `nova.pull` is much larger for *local* and *remote* than for *cached*. Thus, the gain when parallelizing theses transitions is proportionally higher for *local* and *remote*. This result illustrates the benefit of parallelizing data transfers in container-based commissionings when the network bandwidth is sufficient.

Finally, we observe that the global commissioning time for OpenStack is lower on *nantes-ecotype* than on *lyon-nova*. This is due to superior hardware capabilities for the former, as detailed in Table 5. Indeed as MADEUS introduces more parallelism in the commissioning procedure



(a) *m_ansible* with *cached* registry

(b) *m_aeolus* with *cached* registry

Figure 18: (a), (b) Gantt charts of the OpenStack commissioning for *m_ansible*, *m_aeolus* with the registry set in *cached*.

of OpenStack, the better the configuration of the cluster, the better the performance. We argue that the physical nodes on which complex distributed software systems are deployed often have very powerful hardware, and that exposing more parallelism in the commissioning process is an additional way to exploit their high performance level.

OpenStack Continuous Integration. To highlight the potential gain in real-world situations, we apply the above results to the traces of the OpenStack CI. Traces of the OpenStack Continuous Integration platform[15] have been recorded through an automated PYTHON script over nine days from Februrary 19 to February 27, 2020 with specific requests regarding the Kolla project. The script, as well as well as the raw data obtained from the requests, are available in the reproducible lab of the paper. Figure 20 shows the accumulated number of Kolla deployments for each day. Exactly 2963 deployments have been recorded in nine days, an average of 329 runs per day.

The Openstack CI log servers process a large amount of data coming from all the Openstack projects and only store up to 7 days of logs. The script makes requests for all the logs coming from the kolla-ansible project and aggregates them by build uuid, a unique identifier for each build on the Openstack CI, allowing us to count the number of CI operations related to the project. Because some CI operations do not generate a full Openstack deployment, we filter these results to gather the CI events from the kolla-ansible project that are longer than 15 minutes, which is a good indicator that an Openstack deployment happened.

Table 10 illustrates the projection of the gain based on these traces, when considering the deployment times of our experiments in *remote* mode. Of course the OpenStack CI traces probably also include additional tests, and deploy more complex versions of OpenStack, but we only illustrate the possible gain according to our results.

[15] http://logstash.openstack.org

	Kolla	MADEUS	gain
reference time(s)	529	150	71%
projection on 9 days(h)	435	123	71%
projection on av./day(h)	48	14	71%

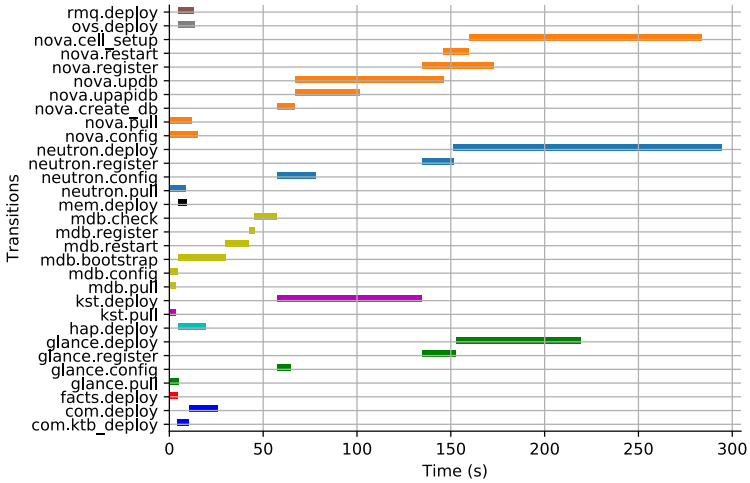
Table 10: Projection of the OpenStack CI traces with our reference experimental measurements with *remote* mode on *nantes-ecotype* (Table 7). Traces of Figure 20 on the deployment of Kolla over nine days in February 2020 are used with a total of 2963 Kolla run in 9 days and an average of 329 runs per day.

Over nine days a total of 312 hours of computations could have been saved on the CI platform, 34 hours per day on average. Finally, one should note that the recorded period was not particularly active as no OpenStack release was close. It is likely that a higher number of Kolla runs will be recorded on the CI platform in periods leading up to a release.

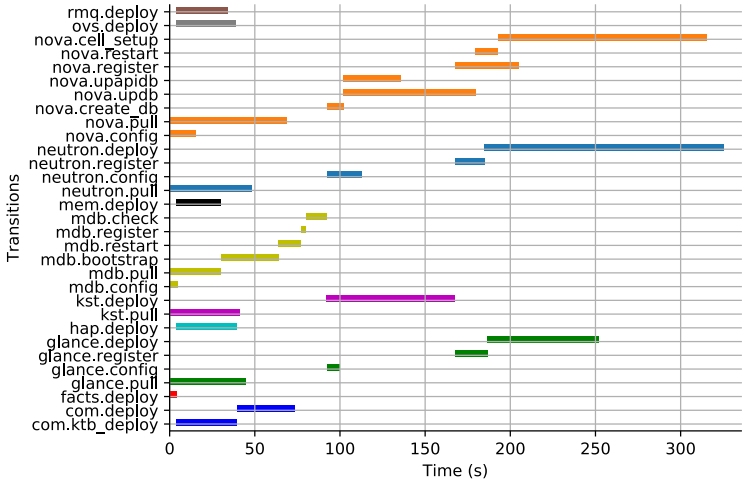
9. Conclusion

MADEUS is a new component-based model specifically designed for distributed software commissioning procedures. By adapting composition mechanisms and combining them with the notion of control components, MADEUS enhances both the separation of concerns and the efficiency of commissioning compared to previous solutions. In this paper, the MADEUS model has been extensively presented from both the theoretical and experimental perspectives.

First, after a detailed study of the related work, an overview of MADEUS was given. Second, the formalization of MADEUS was presented, with a streamlined theoretical model compared to our previous publications. Third, a performance prediction model of MADEUS was studied, based on the transformation of a MADEUS assembly into a directed graph that represents the execution flow of the assembly. Fourth, the prototype of MADEUS was evaluated on three synthetic benchmarks and compared to the



(a) *madeus* with *cached* registry



(b) *madeus* with *remote* registry

Figure 19: (a) Gantt charts of the OpenStack commissioning for *madeus* with the registry set in *cached*; (b) Gantt chart of the OpenStack commissioning for *madeus*, with the registry set in *remote*.

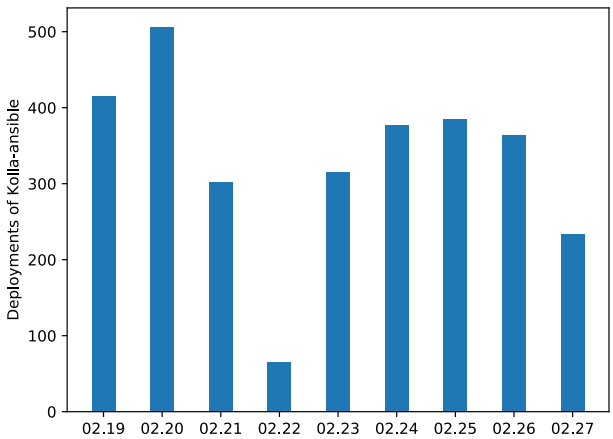


Figure 20: Traces recorded from the OpenStack CI platform over nine days on the deployment of the Kolla project.

expected performance predicted by the theoretical model. Results have shown that the overhead introduced by the prototype is very low, and that the prediction of the performance model is accurate. Finally, the benefits of MADEUS regarding separation of concerns and efficiency have been evaluated and discussed on a real complex use case: the commissioning procedure of OpenStack. Results have been extensively studied and have shown that MADEUS outperforms both ANSIBLE and AEOLUS in terms of efficiency, and that a projection on the traces of the OpenStack CI could save 34 hours of computation a day.

As future work, first, MADEUS is currently not equipped with mechanisms to handle faults during the commissioning procedure. Notably, the rule Termin_α does not consider the possibility of a failure during the action $\alpha \in \mathcal{E}$. We would like to equip MADEUS with if/else statements or switches as well as rollback mechanisms. Moreover, we

would like to provide formal guarantees in the presence of faults, for instance by using game theory or stochastic formal methods.

Second, we wish to study semi-automatic (e.g., using a light DSL) or possibly automatic inference of dependencies from Ansible playbooks, so that Madeus assemblies may be entirely or partially generated for the user.

Finally, we have already generalized Madeus to perform dynamic reconfiguration of distributed software [44]. Indeed, once commissioned, distributed software may need to adapt dynamically in order to respond to faults, to optimize some metrics (e.g., energy, efficiency), or to adapt the services to dynamic requirements (e.g., smart cities). When these reconfiguration decisions are taken, especially for critical systems, the duration of the reconfiguration should be taken into account. Thus, generalizing Madeus and the performance prediction model of Section 6 to dynamic reconfiguration is an important contribution.

[1] F. Hermenier, J. Lawall, G. Muller, BtrPlace: A Flexible Consolidation Manager for Highly Available Applications, IEEE Transactions on Dependable and Secure Computing 10 (5) (2013) 273–286.

[2] E. Ábrahám, F. Corzilius, E. B. Johnsen, G. Kremer, J. Mauro, Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies, in: M. Fränzle, D. Kapur, N. Zhan (Eds.), Dependable Software Engineering: Theories, Tools, and Applications, Springer International Publishing, Cham, ISBN 978-3-319-47677-3, 229–245, 2016.

[3] E. Cadorel, H. Coullon, J.-M. Menaud, A workflow scheduling deadline-based heuristic for energy optimization in Cloud, in: GreenCom 2019 - 15th IEEE International Conference on Green Computing and Communications, IEEE, Atlanta, United States, 1–10, URL https://hal.inria.fr/hal-02165835, 2019.

[4] E. Cadorel, H. Coullon, J.-M. Menaud, Online Multi-User Workflow Scheduling Algorithm for Fairness and Energy Optimization, in: Proceedings 20Th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, IEE, To appear.

[5] J. A. Hewson, P. Anderson, A. D. Gordon, A Declarative Approach to Automated Configuration, in: Proceedings of the 26th International Conference on Large Installation System Administration: Strategies, Tools, and Techniques, lisa'12, USENIX Association, USA, 51–66, 2012.

[6] M. Chardet, H. Coullon, D. Pertin, C. Pérez, Madeus: A formal deployment model, in: 4PAD 2018 - 5th International Symposium on Formal Approaches to Parallel and Distributed Systems (hosted at HPCS 2018), Orléans, France, 1–8, URL https://hal.inria.fr/hal-01858150, 2018.

[7] T. L. Nguyen, R. Nou, A. Lebre, YOLO: Speeding up VM and Docker Boot Time by reducing I/O operations, in: EURO-PAR 2019 - European Conference on Parallel Processing, Springer, Göttingen, Germany, 273–287, doi:10.1007/978-3-030-29400-7_20, URL https://hal.inria.fr/hal-02172288, 2019.

[8] J. Darrous, S. Ibrahim, A. C. Zhou, C. Pérez, Nitro: Network-Aware Virtual Machine Image Management in Geo-Distributed Clouds, in: CCGrid 2018 - 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE, Washington D.C., United States, 553–562, doi:10.1109/CCGRID.2018.00082, URL https://hal.inria.fr/hal-01745405, 2018.

[9] R. Di Cosmo, A. Eiche, J. Mauro, S. Zacchiroli, G. Zavattaro, J. Zwolakowski, Automatic Deployment of Services in the Cloud with Aeolus Blender, in: A. Barros, D. Grigori, N. C. Narendra, H. K. Dam (Eds.), 13th Intl Conf. on Service-Oriented Computing, vol. 9435, Springer, Goa, India, 397–411, doi:10.1007/978-3-662-48616-0_28, 2015.

[10] Ansible, https://www.ansible.com/, 2020.

[11] Puppet, https://puppet.com/, 2020.

[12] Chef, https://www.chef.io/, 2020.

[13] Saltstack, https://www.saltstack.com/, 2020.

[14] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, P. Toft, The SmartFrog Configuration Management Framework, SIGOPS Oper. Syst. Rev. 43 (1) (2009) 16–25, ISSN 0163-5980, doi:10.1145/1496909.1496915, URL https://doi.org/10.1145/1496909.1496915.

[15] J. Fischer, R. Majumdar, S. Esmaeilsabzali, Engage: a deployment management system, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012, 263–274, doi:10.1145/2254064.2254096, URL https://doi.org/10.1145/2254064.2254096, 2012.

[16] A. Flissi, J. Dubus, N. Dolet, P. Merle, Deploying on the Grid with DeployWare, in: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid, CCGRID '08, IEEE Computer Society, Washington, DC, USA, ISBN 978-0-7695-3156-4, 177–184, doi:10.1109/CCGRID.2008.59, URL https://doi.org/10.1109/CCGRID.2008.59, 2008.

[17] R. Di Cosmo, J. Mauro, S. Zacchiroli, G. Zavattaro, Aeolus: a Component Model for the Cloud, Information and Computation 239 (2014) 100–121, ISSN 0890-5401, doi:https://doi.org/10.1016/j.ic.2014.11.002, URL http://www.sciencedirect.com/science/article/pii/S0890540114001424.

[18] Docker, https://www.docker.com/, 2020.

[19] Terraform, https://www.terraform.io/, 2020.

[20] Juju, https://jaas.ai/, 2020.

[21] CloudFormation, https://aws.amazon.com/fr/cloudformation/, 2020.

[22] Heat, https://wiki.openstack.org/wiki/Heat, 2020.

[23] Topology and Orchestration Specification for Cloud Applications Version 1.3, http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.pdf, 2019.

[24] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner, OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-642-45005-1, 692–695, doi:10.1007/978-3-642-45005-1_62, URL http://dx.doi.org/10.1007/978-3-642-45005-1_62, 2013.

[25] Cloudify, http://getcloudify.org/whitepapers/intro-to-cloudify.html, 2020.

[26] OpenTosca, https://www.opentosca.org/sites/publications.html, 2020.

[27] M. Wurster, U. Breitenbücher, K. Képes, F. Leymann, V. Yussupov, Modeling and Automated Deployment of Serverless Applications Using TOSCA, in: 2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA), ISSN 2163-2871, 73–80, doi:10.1109/SOCA.2018.00017, 2018.

[28] Kubernetes, http://kubernetes.io/, 2020.

[29] DockerSwarm, https://docs.docker.com/engine/swarm/, 2020.

[30] C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley Longman Pub. Co., Inc., Boston, MA, USA, 2nd edn., ISBN 0201745720, 2002.

[31] Object Management Group, CORBA Component Model, URL http://www.omg.org/docs/formal/06-04-01.pdf, 2006.

[32] G. Blair, T. Coupaye, J.-B. Stefani, Component-based architecture: the Fractal initiative, Annals of telecommunications 64 (1) (2009) 1–4, ISSN 1958-9395, doi:10.1007/s12243-009-0086-1.

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

[34] B. A. Allan, et al., A Component Architecture for High-Performance Scientific Computing, Intl J. of High Performance Computing Applications 20 (2) (2006) 163–202, doi:10.1177/1094342006064488, URL http://hpc.sagepub.com/content/20/2/163.abstract.

[35] J. Bigot, C. Pérez, Increasing Reuse in Component Models

through Genericity, Research Report RR-6941, Inria, URL https://hal.inria.fr/inria-00388508, 2009.

[36] H. Coullon, J. Bigot, C. Perez, Extensibility and Composability of a Multi-Stencil Domain Specific Framework, Intl J. of Parallel Programming ISSN 1573-7640, doi:10.1007/s10766-017-0539-5.

[37] Object Management Group, Deployment and Configuration of component-based Distributed Applications, URL https://www.omg.org/spec/DEPL/4.0/PDF, 2006.

[38] H. Coullon, C. Jard, D. Lime, Integrated Model-checking for the Design of Safe and Efficient Distributed Software Commissioning, in: IFM 2019 - 15th International Conference on integrated Formal Methods, Bergen, Norway, 1–18, URL https://hal.archives-ouvertes.fr/hal-02323641, 2019.

[39] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, C. Pérez, GCM: a grid extension to Fractal for autonomous distributed components, Annals of telecommunications 64 (1-2) (2009) 5–24, ISSN 0003-4347, doi:10.1007/s12243-008-0068-8.

[40] J. Zwolakowski, A formal approach to distributed application synthesis and deployment automation, Theses, Univeristé Paris Diderot Paris 7, URL https://tel.archives-ouvertes.fr/tel-01172022, 2015.

[41] A. Brogi, D. Neri, L. Rinaldi, J. Soldani, From (Incomplete) TOSCA Specifications to Running Applications, with Docker, in: A. Cerone, M. Roveri (Eds.), Software Engineering and Formal Methods, Springer Intl Pub., ISBN 978-3-319-74781-1, 491–506, 2018.

[42] W. Chareonsuk, W. Vatanawood, Formal verification of cloud orchestration design with TOSCA and BPEL, in: 2016 13th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), ISSN null, 1–5, doi:10.1109/ECTICon.2016.7561358, 2016.

[43] D. K. Rensin, Kubernetes - Scheduling the Future at Cloud Scale, 1005 Gravenstein Highway North Sebastopol, CA 95472, URL http://www.oreilly.com/webops-perf/free/kubernetes.csp, 2015.

[44] M. Chardet, H. Coullon, C. Perez, Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto, in: Proceedings 20Th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, IEE, To appear.