Integrated Model-checking for the Design of Safe and Efficient Distributed Software Commissioning

Hélène Coullon, Claude Jard, Didier Lime

Integrated Model-checking for the Design of Safe and Efficient Distributed Software Commissioning

Helene Coullon[1,4], Claude Jard[3,4], and Didier Lime[2,4]

[1] IMT Atlantique, Inria helene.coullon@imt-atlantique.fr
[2] École Centrale de Nantes didier.lime@ec-nantes.fr
[3] Université de Nantes claude.jard@univ-nantes.fr
[4] LS2N, CNRS

Abstract. We present MADA, a deployment approach to facilitate the design of efficient and safe distributed software commissioning. MADA is built on top of the Madeus formal model that focuses on the efficient execution of installation procedures. Madeus puts forward more parallelism than other commissioning models, which implies a greater complexity and a greater propensity for errors. MADA provides a new specific language on top of Madeus that allows the developer to easily define the properties that should be ensured during the commissioning process. Then, MADA automatically translates the description to a time Petri net and a set of TCTL formulae. MADA is evaluated on the OpenStack commissioning.

Keywords: Distributed software commissioning · Deployment · Model checking · Safety · Liveness · Efficiency · Component models · Petri nets.

1 Introduction

This paper focuses on one specific challenge related to distributed software deployment: *distributed software commissioning*. By software commissioning we mean the complete installation, configuration and testing process when deploying distributed software on physical distributed resources, with or without a virtualization layer in between. This process is complex and error-prone because of the specificity of the installation process according to the operating system, the different kinds of virtualization layers used between the physical machines and the pieces of software, the amount of possible configuration options [23]. Recently, commissioning (or configuration) management tools such as Ansible[1], or Puppet[2], have been widely adopted by system operators. These tools commonly include good software-engineering practices such as code reuse and composition in management and configuration scripts. It is nowadays possible to

[1] https://www.ansible.com/

[2] https://puppet.com

build a new installation by assembling different pieces of existing installations[3,4] which improves the productivity of system operators and prevents many errors. Many distributed software commissionings are nowadays written with one the two above tools and by using containers between the host operating system and the pieces of software, such that portability of installations is improved. For instance, OpenStack, which is the de-facto open source operating system of Cloud infrastructures, can be automatically installed on clusters by using the *kolla-ansible* project, which uses both Docker containers and Ansible.

Yet, even for such well-established software, there is still much room for improving the efficiency of the commissioning process (i.e., reducing deployment times, minimizing services interruptions etc.). As manually coding parallelism into commissioning procedures is technically difficult and error prone, automated parallelism techniques should be introduced. To this purpose, not only dedicated tools, including Puppet, but also academic prototypes such as Aeolus [12] and Madeus [9], introduce parallelism capabilities within software commissioning, at different levels and by using different techniques more or less transparent for the user. For instance, we have observed a performance gain of up to 47% by using Madeus over the *kolla-ansible* reference approach (conducted on the Taurus cluster of the Grid'5000 experimental platform).

Madeus[5] is the commissioning model offering the highest parallelism level [9] in the literature, while offering a formal operational semantics. This makes Madeus the ideal candidate to further study challenges of efficient and safe distributed software commissioning. Madeus automatically handles the intricate details of parallelism coordination, by managing threads and their synchronizations for the user. However, users still have to design their parallel procedures, thus raising the following questions: (1) how to divide existing intricate commissioning scripts in interesting subtasks to introduce parallelism? (2) how to find correct dependencies between commissioning tasks? and (3) how to avoid safety issues such as deadlocks, wrong order of configurations etc.?

We study the feasibility of using model checking techniques to help in the three above challenges in the design of safe and efficient distributed software commissioning. To this purpose we present MADA, an extension of Madeus that brings the following contributions: (1) the automatic transformation process from a Madeus commissioning to an equivalent time Petri net; (2) a domain specific language on top of Madeus to easily express qualitative and quantitative properties related to both safety and efficiency; (3) the automatic translation of MADA properties to temporal logic properties, including liveness, observers and causality, which is uncommon for software commissioning; and (4) an evaluation of MADA on the real case study OpenStack[6] and compared to real experiments.

The rest of this paper is organized as follows. Section 2 introduces the related background. Section 3 presents MADA that is evaluated in Section 4 on the real

[3] `https://galaxy.ansible.com/`

[4] `https://forge.puppet.com/`

[5] `https://gitlab.inria.fr/Madeus/mad`

[6] `https://www.openstack.org/`

commissioning of OpenStack. Finally, Section 5 comments the related work, and Section 6 concludes this work and opens some perspectives.

2 Madeus and Petri Nets

Madeus [9] is a component-based model where a component represents the configuration and the installation of a software module of a distributed system. A component contains a set of places that represent milestones of the deployment, and a set of transitions that connect the places together and represent actions to be performed between milestones (e.g., `apt-get install`).

The internal commissioning behavior of a Madeus component is called in the rest of this paper an internal-net. In Madeus, a transition is attached to output and input docks of places, which are used to properly manage parallel actions in the operational semantics. Madeus components expose ports that represent connection points with other components. Four kinds of ports are available: *service-provide* and *service-use* ports to provide (resp. require) a service, and *data-provide* and *data-use* ports to provide (resp. require) a piece of data. Provide ports (service or data) are bound to one or more places (i.e., called *groups*), illustrating the set of milestones where the component is able to provide a service or data. Use ports (service or data) are bound to one or more transitions where services or data are actually used. A component *assembly* (also sometimes called a *configuration*), is made of component instances and connections between ports.

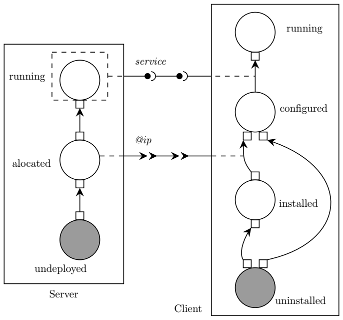


Fig. 1: A Madeus assembly of a server-client software commissioning. Each component (Server and Client) is composed of an internal-net associated to ports.

Figure 1 shows an assembly composed of two components. The internal-net of the *Server* component (on the left) is composed of three places (circles) and two transitions (arrows). The Server's *allocated* place is bound to the *ip* data-provide port (outgoing arrow), while the *running* place is within a group bound to the *service-provide* port (small black circle). The internal-net of the *Client* component (on the right) is composed of four places and four transitions, two of them being bound to (respectively) a *data-use* (incoming arrow) and a *service-use* (semi circle) ports. The ports of *Server* and *Client* are connected within the assembly forming data and service connections (i.e., dependencies).

The evolution of a Madeus deployment is modeled by a set of moving tokens within the internal-nets of components. The evolution of these tokens is handled by seven operational semantics rules which are specific to software commissioning procedures. Hereafter is an overview of these rules, which are formally detailed in [9]. Each initial place of a component (represented with a gray background in Figure 1) initially contains a single token. The token is allowed to leave a place belonging to a group only if this would not deactivate a service provide port with an active connection (i.e., under use). If so, and if the outgoing transition of a place is not bound to a use port, the token is directly able to move from the place to the transition. Otherwise, all the use ports have to be *ready* before the move can happen, which is the case when the associated use-provide connections are *activated*. When a token reaches a transition the associated deployment action is started. Several transitions can be fired simultaneously (e.g., *Client* in Figure 1). In this case, the token is duplicated on each transition. As soon as all the input transitions of a place have ended, their tokens are merged and go to the place. When a token moves to a place in a group that is bound to a provide port, the associated connection is activated, which eventually unlocks use ports of other components. Unlike the service-provide port, the data-provide port acts as a register and will never be disabled once being activated.

Petri nets [22] are a classic formalism for the modeling of concurrent systems. We recall here the basic definition. A *Petri net* (PN) is a 3-tuple (P, T, F), where P is finite set of *places*; T is finite set of *transitions*, with $P \cap T = \emptyset$; and $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the *flow function* that defines *weighted arcs* between places and transitions. A *marking* of a net (P, T, F) is a function $m : P \to \mathbb{N}$. For a given marking m, we say that place p contains $m(p)$ *tokens*. A pair $(\mathcal{N}, m_0)$, where $\mathcal{N}$ is a Petri net and m_0 is a marking, is called a *marked* Petri net, which we often call just Petri net, when clear from the context. Marking m_0 is called the *initial marking* of $\mathcal{N}$. A transition t is said to be *enabled* by marking m, when $\forall p \in P, m(p) \geq F(p, t)$. A transition t enabled at marking m can be *fired*, producing the new marking m' defined by: $\forall p \in P, m'(p) = m(p) - F(p, t) + F(t, p)$.

We also consider *time Petri nets* [21], in which a time interval is attached to each transition. A transition can then only be fired (and must be fired or be disabled) after being continuously enabled for a duration in that interval. See [5] for details.

At first sight a Madeus assembly looks fairly close to a Petri net. However, compared to Petri nets, Madeus assembly falls into the category of Domain Specific Languages (DSL) to model and execute distributed software commissioning. It therefore has a few high-level specific primitives, adapted to developers and system administrators, that do not have direct equivalent in Petri nets.

3 MADA

Madeus automatically handles the intricate details of parallelism coordination, by managing threads and their synchronizations for the user. However, users still have to design their parallel procedures. MADA (*MADeus Analyzer*) uses model-checking to help Madeus users design their safe and efficient commissioning. First, MADA is responsible for the automatic transformation from a Madeus assembly to a time Petri net. Second, MADA offers a set of high level qualitative and quantitative properties that are automatically transformed to Timed Computational Tree Logic (TCTL) formulae.

3.1 From Madeus to Petri Nets

Table 1 show the five rules that illustrate the complete translation from Madeus assemblies to (time) Petri nets. Since the terminologies are similar we distinguish between *m-place* and *m-transition* for Madeus assemblies and *pn-place* and *pn-transition* for Petri nets. As docks, ports and connections only concern Madeus, no special notations are introduced for them.

The first and second rule of Table 1 illustrate the basic constructions, when Madeus port are not involved in the component (i.e., no synchronization). One pn-place is created for each dock, one for each m-place, and one for each m-transition. Those are connected straightforwardly to ensure the Madeus sequence <source m-place, output dock, m-transition, input dock, target m-place >. When considering time, all pn-transitions are executed immediately (interval $[0, 0]$) except the *end* pn-transitions that bear the execution time interval of the associated m-transition, thus representing the time needed for the commissioning action.

Each provide port of Madeus is modeled by a dedicated pn-place. Thus, connections related to this provide port are available if and only if that pn-place is non-empty. Rule (3) models how the pn-place modeling m-transition t, that uses two provide ports $port_1$ and $port_2$, checks for the connections availability by testing the non-emptiness of pn-places $port_1$ and $port_2$. Note that for service-provide it is necessary to know whether some transition is currently using the port (as explained below). We therefore add two pn-places for each provide port, that are marked in exclusion, one to mark that a port is under use, the other one to mark the opposite: $port_1\_in\_use$ and $port_1\_not\_used$ in Table 1 (rule (3)), for instance. With this construction, the following property holds, thus respecting the Madeus semantics:

Property 1. For any provide port *port*, pn-place *port_in_use* contains as many tokens as there are marked pn-places representing an m-transition that currently uses *port*. The total number of tokens in port_in_use and port_not_used is always the total number of m-transitions that require *port*.

Rule (4) of Table 1 models the activation of a data-connection. In Madeus as soon as m-place P is reached, the connection associated to the provide port $port_2$ is activated. Since a data-connection once activated will always remain so, in the equivalent Petri net a token is added in the pn-place $port_2$ once and for all at the same time as the pn-place P is marked. With this construction the following property holds:

Property 2. The number of tokens in a pn-place modeling a data-provide port is equal to the number of pn-places modeling m-places bound to that port that has been marked.

Finally, rule (5) models the most complex part of the transformation: a group of m-places that provides a service. In this case, we must first ensure that the connection is activated if and only if a m-place in the group is active, or a dock or a m-transition between two such m-places is active. In the Petri net, this is ensured in the following way: (1) whenever a pn-transition entering the group is executed, a token is added to the pn-place modeling the Madeus provide port (e.g., pn-transitions $enter_{P_1}$ and $enter_{P_2}$); (2) whenever a pn-transition leaving the group is executed (i.e., from one of the last m-place of the group), a token is removed from the pn-place modeling the Madeus provide port (e.g., pn-transitions $exit_{P'_1}$ and $exit_{P'_2}$); (3) whenever a pn-transition within the group

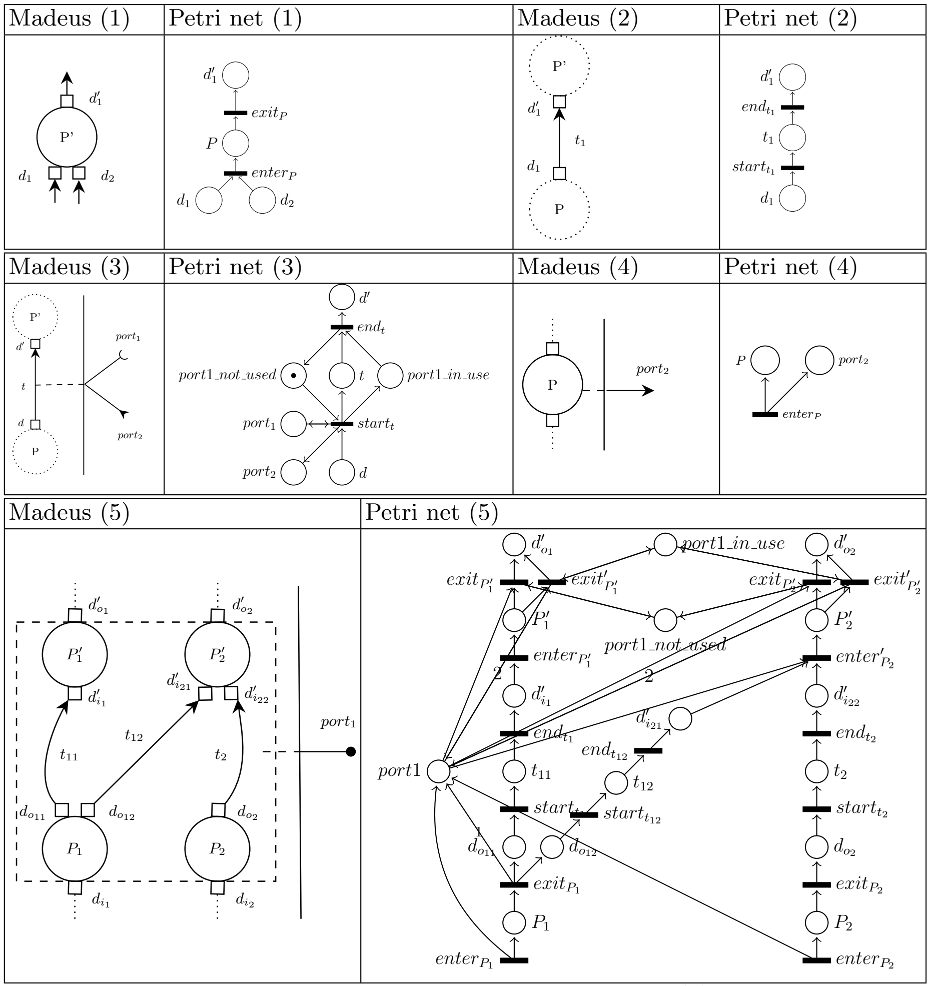


Table 1: Set of transformations from Madeus to Petri nets: (1) basic construction of a m-place and its associated docks, (2) basic construction of a m-transition and its source and destination docks, (3) *data-use* and *service-use* ports bound to a transition t, (4) *data-provide* port provided by a place P, (5) group of places bound to a *service-provide* port

(i.e., source and destination pn-places within the group) that goes to or comes from a pn-place modeling a m-place is executed, we accordingly remove in the provide port pn-place as many tokens as input docks, and add in the same provide port pn-place as many tokens as output docks. Note that, in order to minimize the number of arcs in the Petri net, we actually compute the net effect on the port pn-place so that most transitions (leaving one dock for a place, or one place for a dock) do not modify the number of tokens in the port pn-place. Thus, only forks and joins within the group add and remove tokens in the port pn-place (e.g., $exit_{P_1}$ and $enter_{P_2'}$ in Table 1, rule (5)). With this construction the following property holds:

Property 3. The number of tokens in a pn-place modeling a service-provide port is equal to the number of marked pn-places modeling either active m-places or m-transitions or docks in the group.

Once the port activated, we need to keep it active until all m-transitions using the port are completed. Thus there are two ways to leave a group from a pn-place, first if the provide port is not used, and second if it is used but leaving the pn-place does not remove the last token of the pn-place modeling the service-provide port. In the Petri net, this is ensured in the following way: (1) two outgoing pn-transitions are represented for each pn-place modeling a m-place that leaves a group (e.g., $exit_{P_1'}$ and $exit'_{P_1'}$ for the m-place P_1', $exit_{P_2'}$ and $exit'_{P_2'}$ for the m-place P_2', rule (5) of Table 1): (2) the first one tests if the pn-place *not_used* associated to the service-provide port is equal to the total number of m-transition that may require this port; (3) the second one tests if the pn-place *in_use* associated to the service-provide port is marked and checks that there are at least two tokens left in the pn-place of the provide port. With this construction the following property holds:

Property 4. (1) At any given time, at most one of the two pn-transitions that go outside a pn-place modeling a m-place that leaves is enabled. (2) The pn-place modeling the service-provide port cannot be emptied if the number of tokens in the corresponding *not_used* pn-place is not equal to the total number of m-transitions that can use the port.

We have focused on a high-level explanation of the translation but it would not be difficult, if a bit tedious, to formally prove a (weak) bisimulation between the formal semantics of Madeus given in [9] and the Petri net obtained by the process outlined above. We leave it out of the scope of the paper to improve its readability and due to space concerns.

3.2 Property Language and Temporal Logic

Using MADA, an equivalent Petri net of a given Madeus assembly, i.e., software commissioning, is automatically generated. This saves the user from the transformation burden and prevents possible errors in this process. Furthermore, Madeus users are not familiar with Petri nets. This also holds for the properties on the generated Petri net. For this reason, MADA extends Madeus with a set of property functions that abstract away from the user the details of temporal logic and that are easy to understand for systems operators.

```
def set_interval(self, component, transition, min, max)
def add_deployment(self, name, dict_components_places)
def deployability(self, deployment_name, with_intervals)
def sequentiality(self, ordered_list_components_transition)
def forbidden(self, list_marked, list_unmarked)
def parallelism(self, full_assembly, list_components)
def gantt_boundaries(self, deployment_name, mini, maxi, critical)
```

Fig. 2: Methods signatures for MADA's `Properties`.

Figure 2 shows the set of functions offered by MADA in Python. First, a time interval can be associated to each m-transition of the Madeus assembly. A default interval is set to $[1, 100]$ and can be updated by the user. The interval of each m-transition can be specified by the user with the function `set_interval`.

Once an interval is declared for each m-transition, a set of *deployments* can be defined by the user. A deployment is identified by a name and a set of places to reach. Then the library offers five properties divided in two categories: qualitative properties and quantitative properties. Quantitative properties are only available if intervals have been indicated in the time Petri net as they offer results related to time spent in m-transition. However, as it will be shown in Section 4 these intervals do not necessarily have to be precise.

Qualitative properties. Three such properties are available in MADA. First, for a given deployment D, its *deployability* is the property that all of the (modeled) paths eventually lead to D. The associated syntax is illustrated on the line 4 of Figure 2. In the Petri net, the deployability property can be verified as an inevitability property, that is an AF property in TCTL [1]. Secondly, if an assembly is well defined by the developer with the needed connections between components and the right synchronization of transitions, the sequential orders between transitions will be true by construction. However, designing a Madeus assembly could become tricky for large distributed software, and the resulting high concurrency level could lead to unwitting errors. For this reason, MADA offers a way to define with an easy syntax the sequential orders that must be ensured between transitions of the assembly. The function `sequentiality` takes an ordered list of tuples as input, where each tuple contains a component name and its transition name. The order in which are given the tuples is the sequence to check. Sequentiality is a safety property that can be checked in Petri nets using an observer subnet : add an error pn-place p_{err} to the Petri net. Add also, for each pn-transition pnt_t modeling the end of a m-transition t (i.e., timed pn-transition), a pn-place p_t, initially not marked and a pn-place $p_{\neg t}$, initially with one token. pn-transition pnt_t moves the token from $p_{\neg t}$ to p_t. Then for each ordered relation of m-transitions $t << t'$, a pn-transition is added from $p_{\neg t}$ and $p_{t'}$ to p_{err}. That transition will be firable only if t' has fired but t has not fired. Sequentiality is therefore equivalent to the impossibility of putting tokens in p_{err}, which is a basic safety property: AG $p_{err} = 0$ in TCTL. Finally, systems operators may want to ensure that a given configuration is not reachable during the software commissioning. To this purpose the method `forbidden` is available in MADA and checks that the following invariant holds: we always have at least one of the elements in `list_marked` that is not marked or one of the elements in `list_unmarked` that is marked. In the Petri net, the `forbidden` property is a basic safety property (AG). One can note that the opposite property could be easily added.

Quantitative properties. When a Madeus assembly is designed it is not necessarily easy, because of the global coordination with connections between components, to understand the level of parallelism reached during the execution. The method `parallelism` on line 6 of Figure 2 returns the level of parallelism achieved when executing the Petri net. It takes as arguments, first, a boolean indicating if the maximum parallelism level of the entire assembly has to be computed, and second, the list of components for which the parallelism has to be studied separately. To compute the (maximum) level of parallelism, we compute all the reachable markings and then compute for all of them the sum of all tokens in pn-places corresponding to m-transitions belonging to the components to check. The number we want is the maximum of those. The level of parallelism can be completed by a full Gantt diagram of all m-transitions execution. This result is longer to compute but offers a clear view of the behavior of the software commissioning. The function `gantt_boundaries` takes the name of the deployment to check as input, as well as two booleans `mini` and `maxi`. In facts, a Gantt diagram can be computed and drawn by computing either the minimum or the maximum time to reach some markings in the Petri net [6] and by asking the model checker to keep absolute time information in its trace. As a result, computation of a Gantt diagram also computes the minimum and/or the maximum boundaries of the software commissioning. One can note that the trace returned by the model checker has to be processed to be able to draw the Gantt diagram. Moreover, by starting with the last m-transition to execute and backtracking in the Gantt diagram, a clear causality trace can be obtained to identify the critical path that has led to the minimum or maximum execution time [4]. For this reason, the last input of `gantt_boundaries` is a boolean indicating if the critical path has to be extracted from the trace.

4 Evaluation

In this section, we describe how MADA has been used to build the Madeus OpenStack commissioning mentioned in Section 1 from a sequential original one. Before reaching the final efficient and safe version, a total of four versions have been incrementally written by using MADA. Thus, the goal of this evaluation is to illustrate on a real use-case how MADA can be used by a user to avoid safety issues and to identify where efficiency improvements can be performed thanks to quantitative properties. Finally, this section discusses the feasibility of the approach.

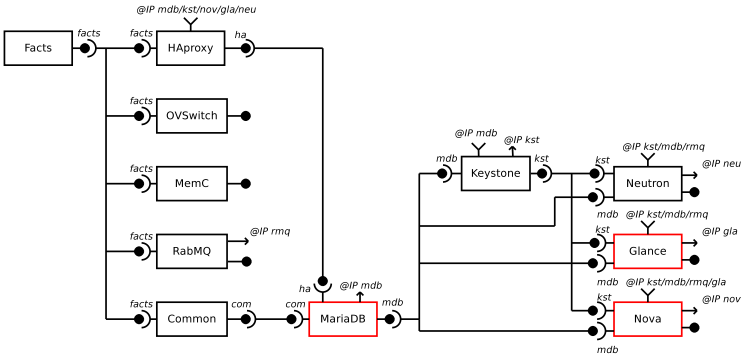


Fig. 3: Coarse-grain Madeus assembly of the OpenStack commissioning.

OpenStack is the de-facto open source solution to address the IaaS (*Infrastructure as a Service*) level of the Cloud paradigm. In other words, OpenStack is the operating system of the Cloud responsible for the management of physical resources of a set of clusters, as well as the management of virtual machines and their life cycle. Following the Ansible installation procedure defined in Kolla, a popular tool to deploy OpenStack[7], we have defined 11 Madeus components: *Nova*, *Glance*, *Neutron*, *Keystone*, *MariaDB*, *RabbitMQ*, *HAProxy*, *OpenVSwitch*, *Memcached*, *Facts*, *Common*. The coarse grain commissioning of OpenStack is depicted in Figure 3, without the detailed internal-nets of components.

All results are shown in Table 2 where the initial Madeus number of elements, the resulting number of elements in the generated Petri net, as well as the execution time of the transformation are given. For space reasons, this evaluation focuses on the most original properties of MADA compared to the related work: `deployability` that checks the inevitable end of the deployment, and the two quantitative properties `parallelism` and `gantt_boundaries` related to the use of time Petri nets. The result of each property as well as its computation time are also shown Table 2. Each Gantt diagram is depicted in separated figures indicating in its caption the obtained critical path. The time intervals used in the experiments have been chosen according to real observed execution time traces. We have used the Roméo model checker for time Petri nets [20] in our experiments. Finally, all the materials used for these experiments are available online[8] (i.e., MADA python files, Petri nets and results).

4.1 MADA evaluation

The first version (*0-deadlock*) has been designed by introducing three straightforward parallel tasks in the three components *Glance*, *Neutron* and *Nova*. The

[7] `https://docs.openstack.org/kolla-ansible/latest/`

[8] `https://gitlab.inria.fr/hcoullon/mada`

	0-deadlock	1-naive	2-nova	3-nova	4-nova-mdb
Madeus places	27	27	28	28	29
Madeus transitions	22	22	25	25	28
Madeus connections	30	30	30	30	30
Petri net places	113	113	124	124	134
Petri net transitions	75	75	84	84	92
Transf. time (ms)	1.6	1.6	1.8	1.7	1.5
Deployability	False	True	True	True	True
Computation time (s)	*0*	*41.6*	*78.7*	*88.7*	*152.6*
Parallelism nova	-	1	2	2	2
Computation time (s)	-	*42.1*	*82.7*	*93.6*	*154.3*
Parallelism full	-	10	11	11	12
Computation time (s)	-	*43.2*	*86.1*	*98.4*	*162.9*
Gantt & critical path	-	Figure 5a	Figure 5b	Figure 5c	Figure 5d
Computation time (s)	-	*130.1*	*266.9*	*275.4*	*588.1*
Boundaries	-	[575, 615]	[518, 554]	[400, 423]	[377, 398]
Computation time (s)	-	*130.1, 128.8*	*266.9, 269.7*	*275.4, 267.6*	*588.1, 580.8*

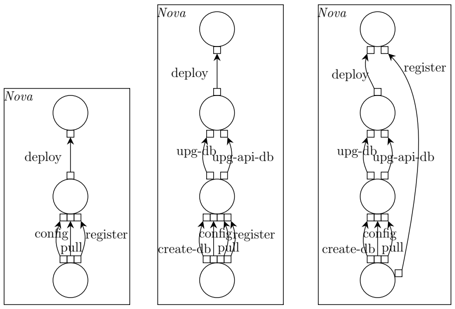
Table 2: Results by using MADA on five different versions of OpenStack commissioning in Madeus.

same parallel tasks have been used in these components: pulling the docker image from the Docker registry at the same time as preparing the configuration of the container, and at the same time as registering the service to the *Keystone* component. This parallelism pattern is illustrated for the component *Nova* on the left of Figure 4.

An error was accidentally introduced in this first version of the Madeus OpenStack commissioning (*0-deadlock*). This error was due to the label "register" of a transition in the component *MariaDB*. In all components containing the same label as a transition, this transition was using the service of the *Keystone* component (for service authentication). However, in *MariaDB* this registering step has to be performed with the component *Common*. In fact *MariaDB* is globally installed before *Keystone*. For this reason, an incorrect sub-assembly has been accidentally built. By executing MADA on the Madeus assembly the deadlock has been detected and a clear trace leading to the problem has been returned. This problem occurs when both *MariaDB* and *Keystone* are waiting for each other.

The version *1-naive* solves the deadlock detected by MADA in *0-deadlock* and keeps the exact same set of components as well as the same assembly. By using MADA it appears that such a design is quite naive as the maximum level of parallelism in *Nova* is actually 1 while we were expecting 3 (Table 2). The Gantt diagram generated by MADA, shown in Figure 5a, explains that the transition *Nova config* cannot be executed at the same time as *Nova pull* because it has to wait for *MariaDB deploy* that ends after *Nova pull*. Similarly, the transition *Nova register* cannot be executed at the same time as the two other tasks because it has to wait for *Keystone deploy* that ends after both *Nova* *pull* and *Nova config*. Furthermore, thanks to the MADA critical path (indicated in the caption of Figure 5a), one can note that *Nova deploy* plays an important part in the overall execution time and could probably be divided into parallel subtasks. This information offers a very useful help to the user to know where to look first in order to improve the efficiency of the commissioning. Without these pieces of information we could have wasted time to understand scripts of *Glance deploy* while it is not in the critical path.

Then, the second version *2-nova* focuses on dividing the transition *Nova deploy* in parallel tasks. It appears that two long tasks responsible for database schemes can be performed in parallel, namely *Nova upg_api_db* and *Nova upg_db*. Moreover, one subtask of *Nova deploy* is independent from the other and can be added as a parallel task with *Nova config*, namely *Nova create-db* (see Figure 4). By using MADA, we verify that the level of parallelism in *Nova* is increased to 2. The Gantt diagram of Figure 5b obtained by MADA shows that both *Nova upg_api_db* and *Nova upg_db* wait for *Nova register* to end which is the reason why *Nova deploy* starts very late. One can also note that *Nova register* is in the critical path. Thanks to these results we have noticed that the dependencies of the parallel task *Nova register* were not well defined as it is entirely independent from other tasks except the last one *Nova deploy*.

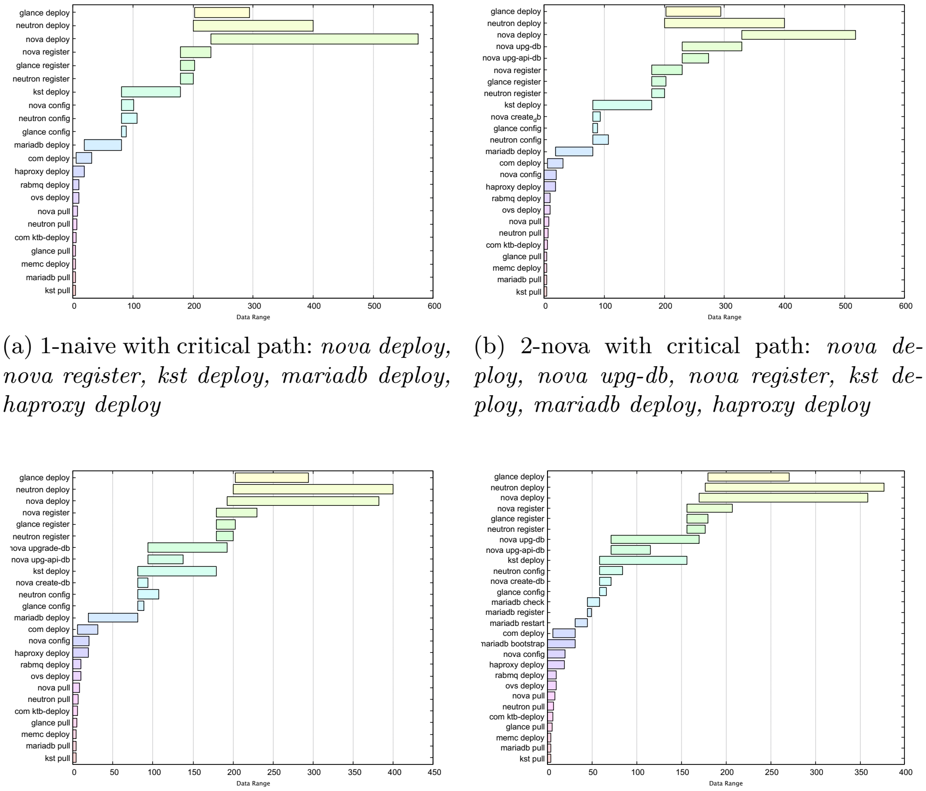


1-naive 2-nova 3-nova

Fig. 4: Nova component evolution thanks to the analysis with MADA.

In the third version *3-nova* we have solved this issue as illustrated in Figure 4, by defining *Nova register* as an overall parallel task. The Gantt diagram shown in Figure 5c demonstrates that tasks of *Nova* are no longer in the critical path. As shown in Table 2 the expected boundaries (i.e., according to time intervals) were $[575, 615]$ for *1-naive* and are $[400, 423]$ for *3-nova*. Finally, in the Gantt diagram and critical path of Figure 5c one can note that the transition *MariaDB deploy* is responsible for the delay of many other tasks such as *Keystone deploy*, for instance.

In the fourth version of the commissioning *4-nova-mdb* the parallelization of *MariaDB deploy* has been studied. This task has been divided in four tasks namely *MariaDB bootstrap*, *MariaDB restart*, *MariaDB register* and *MariaDB check*. The transition *MariaDB bootstrap* can be run at the same time as the already defined *MariaDB pull* transition. Moreover, *MariaDB register* and *MariaDB check* can be executed simultaneously. The resulting Gantt diagram is depicted in Figure 5d with the following expected boundaries: $[377, 398]$. These boundaries are much better than the observed execution time, greater than 750 seconds, of the reference Kolla scripts.



(a) 1-naive with critical path: *nova deploy, nova register, kst deploy, mariadb deploy, haproxy deploy*

(b) 2-nova with critical path: *nova deploy, nova upg-db, nova register, kst deploy, mariadb deploy, haproxy deploy*

(c) 3-nova with critical path: *neutron deploy, neutron register, kst deploy, mariadb deploy, haproxy deploy*

(d) 4-nova-mdb with critical path: *neutron deploy, neutron register, kst deploy, mariadb check, mariadb restart, mariadb bootstrap*

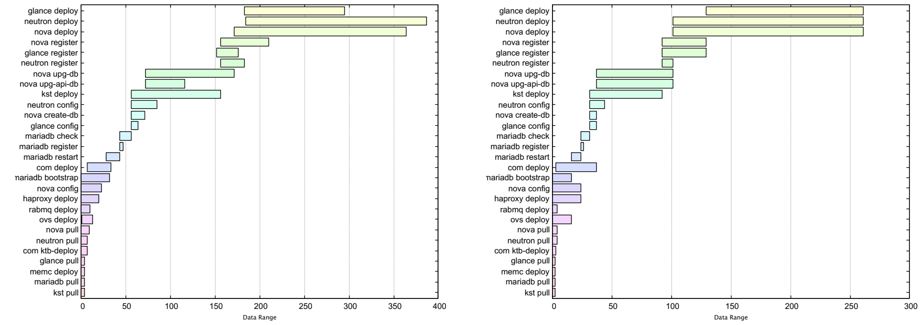
Fig. 5: Gantt diagrams generated by the *gantt_boundaries* property of MADA

4.2 Discussion

Does it work? To answer this question we have conducted real experiments on the OpenStack commissioning. Figure 6a shows the Gantt diagram generated from the traces of one of the real experiments on *4-nova-mdb* (same as the one mentioned in Section 1). One can note that the obtained diagram is very close to the one computed by MADA with time intervals in Figure 5d and that the obtained critical path fits perfectly with the one returned by MADA. This result is also observed for other versions of the commissioning *1-naive*, *2-nova* and *3-nova*, available in the git repository previously indicated. Moreover, the computed boundaries for all versions also perfectly match the real experiments. For instance, the boundaries of *4-nova-mdb* are $[377, 398]$ and the execution time of the run of Figure 6a is 389.8. This confirms that the approach works and that the transformation is valid as well as its implementation. In addition, we would like to emphasize once again that MADA has been very useful in working in the right direction, where tasks can be divided, saving time by avoiding the unnecessary exploration of complex scripts. Finally, by using MADA a separation of concerns is possible between the design phase and the experimental phase of the commissioning. Thus, MADA could be considered as a formal simulator to help the user in the design phase that avoids the burden of real experiments and associated technical issues (e.g., machines provisioning, commands errors, timeouts etc.).

Time intervals. The goal of MADA is not to precisely compute the execution time of the distributed software commissioning but to help identify where efficiency improvements could be made. Accordingly, it is not necessary to be precise about time intervals. However, to provide a good insight of possible efficiency improvements to MADA users, the order of magnitude for each transition is needed. To illustrate this claim, Figure 6b shows the Gantt diagram obtained by using MADA on *4-nova-mdb* with additional errors on the time intervals. Errors are introduced as follows: for tasks representing less than 5% of the total execution time (11 tasks) we have introduced 95% of error on the duration of the interval (e.g., $[6, 7]$ becomes $[3, 10]$). Similarly, for tasks representing respectively between 5-10% (11 tasks), 10-20% (2 tasks), 20-30% (3 tasks), 30-60% (2 tasks) of the execution time we have introduced 90%, 80%, 70%, 40% of error on the duration of the interval (e.g., $[98, 106]$ becomes $[61, 143]$). This method introduces an error that may be substantial (up to 95%) but that does not change the order of magnitude of intervals. One can note that even with this high error on time intervals the Gantt diagram has almost the same shape and the critical path of the execution stays valid. We think that it may be difficult for systems operators to give a precise execution time for a given task. However, giving an order of magnitude through a time interval is much easier as most software installation scripts contain the same types of commands (human expertise). Furthermore, thanks to the high frequency with which system commands appear, machine learning algorithms could also be designed.

Scalability Most computations performed on bounded Petri nets (with or without time) have a PSPACE worst-case complexity. This can be observed in Table 2 where execution time of properties are depicted for each version according to its size. For an increase of the number of places and transitions in the Petri net of 15% the execution time of the *deployability* increases of 73%. We have however to consider a few additional elements in our assessment of the usability of the approach. First, OpenStack is a large system, thus many existing distributed systems can certainly be analyzed with this approach. Second, the generated Petri net itself can be made easier to analyze, depending on the property of interest. Third, the MADA approach is largely independent of the model checker used, and many techniques from the model checking community can be leveraged to improve performance such as partial orders [16], BDD-like data structures [8], SMT-based techniques [10], and lazy model checking [17] for instance. A complete scalability study is left to future work. Finally, compared to the few



(a) Real experiment of 4-nova-mdb on Grid'5000 with critical path: *neutron deploy, neutron register, kst deploy, mariadb check, mariadb restart, mariadb bootstrap*

(b) MADA 4-nova-mdb with errors on time intervals with critical path: *neutron deploy, neutron register, kst deploy, mariadb check, mariadb restart, mariadb bootstrap*

Fig. 6: Experiment on Grid'5000 and experiment with errors on intervals.

contributions of the related work that provide some model checking execution time [14,15] (50 minutes for BPEL, up to 47 minutes for Vercors), MADA execution times can be fairly considered acceptable compared to the burden of testing real experiments on infrastructures each time a modification is made in the assembly (up to 600 seconds). This is due to the fact that Madeus is a relatively small language compared to BPEL and Vercors because of its specificity for distributed software commissioning.

5 Related work

MADA is specifically conceived to help in the design of safe and efficient distributed software commissioning on top of Madeus. Indeed, concurrency and parallelism introduced by Madeus comes at a price. First, parallelism can introduce liveness violation such as deadlocks. Secondly, even if the parallelism *coordination* is ensured by Madeus, designing a parallel software commissioning from a sequential one is difficult. Indeed, subtasks of the commissioning procedures and dependencies between subtasks and software components have to be identified while their exact internal behavior are not well known by operators, because implemented by other developers. Moreover, most of the time commissioning scripts are difficult to read and system commands do not clearly expose their behaviors and functionalities. As far as we know, no existing software commissioning tool or academic model is equipped with a formal way to help the user solving such problems when designing installation procedures. This is in line with the fact that the parallelism capabilities of existing tools and models for automatic software commissioning are very limited compared to Aeolus and Madeus, almost sequential in fact. Thus, in this context liveness issues are impossible, and finding subtasks and dependencies is useless. The contribution of this paper could partially be useful and applicable to Aeolus. However, Aeolus and its satellites [11] are not maintained and were not equipped with such integrated model checking.

Yet, using model checking to help in the design of safe distributed software (not software commissioning) has been extensively studied in the literature. For instance, formal methods have been used in the domain of software components. The *Vercors* framework [14,2] uses parameterized networks of asynchronous automata (pNets) as an intermediate formalism representing behaviors of distributed software components, statically analyzed by the CADP model checker. *BIP* [3] (Behavior, Interaction, Priority) was initially a component-based model for heterogeneous real-time and reactive systems, and has been extended for distributed systems with heterogeneous interactions through *HDBIP* [19]. In BIP, finite state automata are used to represent the behavior of components and their composition. In these contributions, formal methods are used to verify the functional behavior of distributed systems, which is very different from modeling and checking the execution and coordination of software commissioning procedures. In particular, MADA offers quantitative properties related to the concurrency and parallelism introduced within a commissioning execution. This cannot be modeled by Vercors nor BIP. Moreover, this kind of properties can be expressed by using time Petri nets as an intermediate representation of the commissioning execution semantics. Finally, neither Vercors nor BIP clearly explain how verification formulae are expressed by users while MADA automatically generates TCTL formulae from function calls.

Some languages for defining business processes, BPEL [15] and BPMN [13] for instance, have been straightforwardly translated to Petri nets such that a formal semantics can be offered to the user instead of natural language and UML-like semantics. In both of these contributions the Petri net translation has also been used to check liveness and safety properties to detect bugs in user-defined processes. However, quantitative properties such as the one proposed by MADA are not offered because helping the user to improve the process efficiency is not a goal. Moreover, details are missing regarding how translations and properties are integrated in a user-friendly framework.

MACE [18] is another language example that uses formal methods. MACE combines *objects*, *events* and *aspects* programming to address in a single framework concurrency and failures when designing distributed software. Guiding the users in their task and hiding from them the intricate details of static analysis is in line with MADA. However, MACE is more complete and complex than MADA and addresses many different problems at the same time thus raising the complexity of the overall framework. In other words, MADA is specific to software commissioning while MACE solves multiple issues related to distributed software design. The model checking in MACE targets liveness violation only. Causality is addressed by MACE to find failures origins, however, this causality is based on automatic logging instead of model checking. MADA offers an interesting separation of concerns between the design phase and the testing phase on real infrastructures. Quantitative information such as the critical paths for efficiency analysis are studied through model checking thanks to time Petri nets.

Finally, related work has studied how to simplify for a non expert the way properties could be expressed. This is for example one of the contributions around Statemate [7] where timing diagrams are used to generate CTL formulae for verification on reactive systems. The two main differences with MADA are: first the generalization of properties thanks to five function signatures instead of defining one timing diagram for each property to check in Statemate, and second the specific application domain of MADA to distributed software commissioning.

6 Conclusion

In this paper MADA has been presented as a useful approach for introducing model checking to help system operators design their parallel distributed software commissioning. MADA has been evaluated on the real use case of OpenStack which is a very large and complex distributed system. Moreover, comparisons with experiments on real infrastructures have shown the feasibility of the approach. Future work will be focused on using other model checking algorithms to improve the scalability of MADA. Moreover, MADA will be extended to help the user build fault-tolerant distributed software commissioning.

References

1. Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Information and Computation **104**(1), 2–34 (1993)
2. Barros, T., Cansado, A., Madelaine, E., Rivera, M.: Model-checking distributed components: The vercors platform. Electronic Notes in Theoretical Computer Science **182**, 3 – 16 (2007), proceedings of the Third International Workshop on Formal Aspects of Component Software (FACS 2006)
3. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in bip. In: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods. pp. 3–12. SEFM '06, IEEE Computer Society, Washington, DC, USA (2006)
4. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.: Explaining counterexamples using causality. Formal Methods in System Design **40**(1), 20–40 (Feb 2012)
5. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. IEEE trans. on soft. eng. **17**(3), 259–273 (1991)
6. Boucheneb, H., Lime, D., Parquier, B., Roux, O.H., Seidner, C.: Optimal reachability in cost time petri nets. In: Nestmann, U., Wolter, K. (eds.) 15th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2017). Lecture Notes in Computer Science, vol. 10419, pp. 58–73. Springer, Berlin, Germany (Sep 2017)
7. Brockmeyer, U., Wittich, G.: Tamagotchis need not die — verification of statemate designs. In: Steffen, B. (ed.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 217–231. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)

8. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35** (1986)
9. Chardet, M., Coullon, H., Pertin, D., Pérez, C.: Madeus: A formal deployment model. In: 4PAD 2018 - 5th Intl Symp. on Formal Approaches to Parallel and Distributed Systems (hosted at HPCS 2018) (2018)
10. Cimatti, A., Griggio, A.: Software model checking via IC3. In: CAV (2011)
11. Di Cosmo, R., Eiche, A., Mauro, J., et al.: Automatic deployment of services in the Cloud with Aeolus Blender. In: 13th Intl Conf. on Service-Oriented Computing. vol. 9435. Springer (2015)
12. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: a component model for the Cloud. Information and Computation (2014)
13. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of bpmn process models using petri nets. Tech. rep., Queensland University of Technology (2007)
14. Henrio, L., Kulankhina, O., Li, S., Madelaine, E.: Integrated Environment for Verifying and Running Distributed Components. In: Stevens, P., Wasowski, A. (eds.) Fundamental Approaches to Software Engineering. Fundamental Approaches to Software Engineering, vol. 9633. Perdita Stevens and Andrzej Wasowski (2016)
15. Hinz, S., Schmidt, K., Stahl, C.: Transforming bpel to petri nets. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) Business Process Management. pp. 220–235. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
16. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: FORTE (1994)
17. Jezequel, L., Lime, D.: Lazy reachability analysis in distributed systems. In: Desharnais, J., Jagadeesan, R. (eds.) The 27th International Conference on Concurrency Theory (CONCUR 2016). LIPIcs, Dagstuhl Publishing, Québec City, Québec, Canada (2016)
18. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.M.: Mace: Language support for building distributed systems. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '07, ACM (2007)
19. Kobeissi, S., Utayim, A., Jaber, M., Falcone, Y.: Facilitating the Implementation of Distributed Systems with Heterogeneous Interactions. In: IFM 2018 - 14th International Conference on integrated Formal Methods. pp. 1–19. Maynooth, Ireland (Sep 2018)
20. Lime, D., Roux, O.H., Seidner, C., Traonouez, L.: Romeo: A parametric model-checker for petri nets with stopwatches. In: Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (2009)
21. Merlin, P.M.: A study of the recoverability of computing systems. Ph.D. thesis, Dep. of Information and Computer Science, University of California, Irvine, CA (1974)
22. Petri, C.A.: Kommunikation mit Automaten. Dissertation, schriften des iim, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, Bonn (1962)
23. Xu, T., Zhou, Y.: Systems approaches to tackling configuration errors: A survey. ACM Comput. Surv. **47**(4), 70:1–70:41 (Jul 2015)