Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto

Maverick Chardet, Hélène Coullon, Christian Pérez

Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto

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

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

***Abstract*—Dynamic reconfiguration of distributed software systems is nowadays gaining interest because of the emergence of dynamic IoT and smart applications as well as large scale dynamic infrastructures (*e.g.*, Fog and Edge computing). When quality of service and experience is of prime importance, efficient reconfiguration is necessary, as well as performance predictability to decide when a reconfiguration should occur. This paper tackles the problem of efficient execution of a reconfiguration plan and its predictability with Concerto, a reconfiguration model supporting a high level of parallelism. Evaluation performed on synthetic cases and on two real production scenarios show that Concerto provides better performance than state-of-the-art systems with accurate time estimation.**

***Index Terms*—distributed systems, reconfiguration, component model, performance**

I. Introduction

Considering distributed software systems as a set of components or modules interacting together, their reconfiguration consists in dynamically changing any element of their configuration, including their interaction topology and the configuration of their individual components.

With the increasing complexity of both distributed software systems and distributed infrastructures, such reconfiguration procedures have to be automated, and may ideally autonomously react to environmental events. Autonomous reconfiguration is usually modelled with a control loop, such as MAPE-K [1], [2], [3]. It consists in the following four steps: (M) *Monitoring* that returns information about the current configuration of the system as well as events of interest; (A) *Analysis* that, given the information gathered by (M), chooses a new target configuration for the system; (P) *Planning* that determines a reconfiguration plan to go from the current configuration to the new one; and (E) that executes the plan returned by (P). These successive steps are repeated in a loop and they share a common *Knowledge* (K), *i.e.*, a set of models (*e.g.*, reconfiguration language, software modeling etc.).

When reconfiguring critical distributed software systems that should be highly reliable and available or provide a high quality of service or experience (QoS, QoE), the efficiency and safety of the reconfiguration is of prime importance. Indeed, starting a reconfiguration that may interrupt critical services for a long period of time or lead to an undesirable state is prohibited in some cases. Because of this, we argue that reconfiguring should be fast, as well as predictable, so that when choosing a new configuration in (A) and when planning this reconfiguration in (P), the execution time of the reconfiguration should be taken into account. Yet, most reconfiguration models and tools do not pay particular attention to these concerns.

This paper introduces *Concerto*, a new execution reconfiguration model that enhances the following aspects compared to the related work: (1) the performance of reconfiguration through increased parallelism, (2) the estimation of the execution time for a given reconfiguration. Overly formal descriptions are avoided and left to longer publication formats. The second contribution of this paper is to extensively describe and evaluate the algorithm that estimates the performance of Concerto, *i.e.*, the time it takes to execute Concerto reconfigurations.

This work targets all service-, module- or component-oriented software architectures, including in particular legacy and stateful ones. We assume very little knowledge of the internal functional aspect of the modules, using only their control interfaces (*e.g.*, start, stop, backup, etc.), configuration files, management scripts and documentation. Such level of information is often the one found by system administrators when they reconfigure their systems.

The rest of this paper is organized as follows. Section II studies the related work. Section III introduces the Concerto reconfiguration model, while Section IV details the algorithm that estimates the execution time of a Concerto reconfiguration. Section V evaluates the performance of Concerto as well as the accuracy of the time estimations. Finally, Section VI concludes and gives some perspectives.

II. Related Work

This work focuses on performance in reconfiguration and its estimation ahead of execution. Increasing performance can be done in many ways such as command optimizations [4] or by enhancing data management [5]. In this paper we focus on two generic aspects which can increase performance: life-cycle management and parallelism.

A. Life-cycle management

The life-cycle of a system refers to all the configurations in which it can be in, and how to go from one to the others. A programmable life-cycle can lead to faster reconfiguration and better QoS during reconfiguration. In the case of distributed systems, each module usually has its own life-cycle.

Most tools supporting reconfiguration only consider two states for the modules: running/installed and stopped/uninstalled. TOSCA [6] (Topology and Orchestration Specification for Cloud Applications) is an OASIS standard for the deployment and reconfiguration of cloud applications and has multiple implementations, such as OpenTOSCA [7] or Cloudify. TOSCA nodes, which can represent software or resources, are either deployed or not. Platform-specific tools such as AWS CloudFormation[1] or OpenStack Heat[2], or technology-specific tools such as Kubernetes[3] behave in the same way.

Component models provide ways to describe the functional aspects of software modules [8]. In particular BIP [9] allows to fully customize the functional life-cycle of each component. However such level of information cannot be obtained for legacy systems. Some component models also handle the non-functional aspects, such as Fractal [10], GCM [11] or GCM/ProActive [12] which provide the concept of membrane, which contains everything that is not related to the core objective of the component. This can include the handling of the life-cycle of the component as in [12]. However, reconfiguration is fully manual, written with general-purpose components or code, which is not safe. Furthermore, the absence of standard makes it difficult to compose life-cycles coming from different vendors. Tools based on these component models sometimes partially handle life-cycle management. Jade [13] and its evolution Tune [14] give legacy software a Fractal interface, effectively giving them a two-state life-cycle. Deployware [15], which is also based on Fractal, provides each component with six possible states: *configure*, *start*, *manage*, *stop*, *unconfigure* and *uninstall*.

Aeolus [16], [17] is a reconfiguration model in which the life-cycle of each component is modelled as a finite state machine, which makes it fully customisable. Madeus [18], while not handling reconfigurations, also offers customisable life-cycles.

B. Parallelism in generic reconfiguration

More parallelism leads, in most cases, to faster reconfiguration. However, increasing parallelism in reconfiguration is not straightforward. First, one needs information about what can and cannot be done in parallel (*i.e.*, information about dependencies, as precise as possible). In TOSCA, dependencies are given at the node level, implying that parallelism between the reconfiguration of two nodes A and B is never possible if B depends on A. The same issue arises with tools like AWS CloudFormation or OpenStack Heat, and declarative configuration management tools such as Puppet and SaltStack[4]. Kubernetes uses a *fail-and-retry* strategy, running actions to perform on containers in parallel and retrying when a failure occurs because a dependency was not yet solved. Procedural configuration management tools such as Ansible and Chef[5] are limited by the sequential nature of the language in which the reconfigurations (Ansible playbooks and Chef recipes) are expressed. For example, Ansible can only execute one action on multiple nodes if the exact same instruction must be executed. One could argue that parallelism can be manually implemented using general-purpose programming languages, which would be executed by TOSCA artifacts (that contains any piece of code), containers' entry points in the case of Kubernetes, Ansible playbooks, Chef recipes, etc. However, doing so is generally tedious, unsafe and not taken advantage of by the framework.

Aeolus addresses this issue by modeling the life-cycle of the components as a state machine, as well as their dependencies. Thus, given a target configuration for the system, a scheduler for Aeolus can trigger state changes in the life-cycle of the components if all the dependencies are met. Because the dependencies between components are defined in a fine-grained way, a component does not necessarily have to wait for its dependencies to be fully reconfigured before it starts its own reconfiguration. Also, unlike Ansible, each component can perform distinct actions at the same time.

Madeus [18] is a deployment model which, while being restricted to deployment, extends Aeolus to define intra-component dependencies, *i.e.*, dependencies between tasks of the same component, also in a fine-grained way. This allows to increase parallelism even more by executing component actions in parallel.

C. Performance estimation of reconfiguration

Very few publications address reconfiguration performance prediction and its importance when choosing a target configuration in (A), and when planning this reconfiguration in (P). In [19], the authors introduce a framework to reconfigure microservices-based applications by a series of small updates to the architecture following a given strategy in a predictable way. They suggest in conclusion that predicting the reconfiguration plan can help to choose the best fitting strategy for the reconfiguration with respect to a set of service-level agreement requirements. In [20], the authors introduce a middleware to achieve time-bounded reconfiguration in real-time applications. However, to the best of our knowledge, no work has been done on estimating and taking into account the execution time of reconfiguration of legacy distributed systems.

[1] https://aws.amazon.com/cloudformation/
[2] https://docs.openstack.org/heat/latest/
[3] http://kubernetes.io/
[4] https://puppet.com, https://www.saltstack.com/
[5] https://www.ansible.com/, https://www.chef.io/

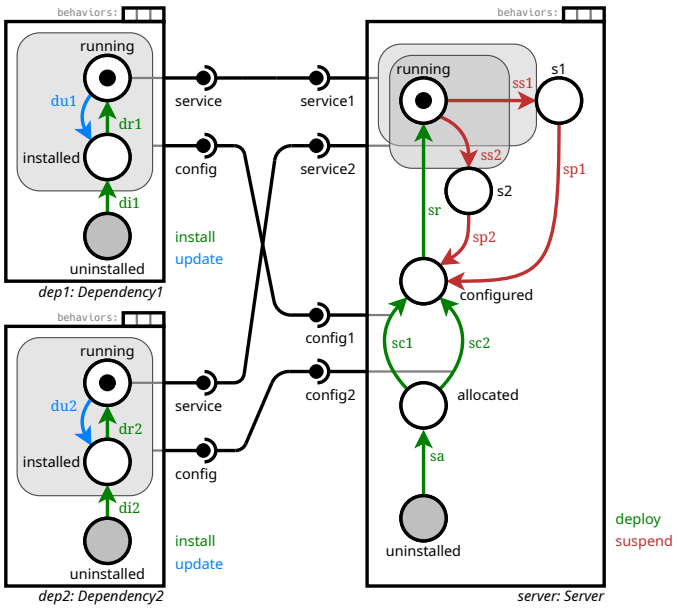


Fig. 1: Example of a Concerto assembly with three component instances: *server*, *dep1* and *dep2*, respectively of types *Server*, *Dependency1* and *Dependency2*. Places are represented by circles, with initial places filled with grey. Transitions are represented by arrows, their color indicating their behavior. Behaviors are listed on the right of a component instance, while its behavior queue is represented on top (here they are all empty). Groups are represented by grey rectangles with rounded corners. Provide ports are represented by black discs outside components. Use ports are represented by semi-circles. Bindings between ports and groups are represented by thin grey lines (if they are connected directly to a place or transition, this means a group containing only this element). Active places or transitions are marked with a token (black disc).

III. Concerto

Concerto is a component-based reconfiguration model focusing on modelling and coordinating the life-cycle of interacting parts of a system (*e.g.*, software module, resource). Typically, each module of the system is modeled with a *control component type*. A *control component type* contains information about the life-cycle and its dependencies. Concerto represents the current configuration of the system by an assembly, *i.e.*, a set of *control component instances* connected together. In the following, the word *component* refers to a control component type while *(component) instance* refers to a control component instance.

A. Control component type

The life-cycle of a component is modelled with a structure inspired by Petri-nets or state machines called *internal-net*. It consists of a set of *places* Π which represent "milestones" in the life-cycle and of a set of *transitions* $\Theta \subseteq \Pi \times \Pi \times \mathcal{A}$ (where $\mathcal{A}$ is a set of actions to perform on the underlying system) which state which actions to perform to go from one place in the life-cycle to another. Note that parallel transitions are allowed. Each transition is associated with a *behavior*, which corresponds to a high-level change one wants to perform on an instance (*e.g.*, commissioning, update). These behaviors are inspired by Mode Automata [21]. The set of behaviors of the component type is denoted B, and the behavior associated with a transition θ is denoted $bhv(\theta)$. One place $init \in \Pi$ is designated as the *initial place* of the component. For example, in Figure 1 component *Dependency1* has three places: *uninstalled* (initial place), *installed* and *running*, as well as three transitions: *di1* and *dr1* associated with behavior *install*, and *du1* associated with behavior *update*. Notice that in the *server* component, transitions *sc1* and *sc2* are parallel, which corresponds to parallel tasks.

Dependencies between components are modelled with the concept of *port* from component-based software engineering [8], adapted to life-cycle interactions. In Concerto provide ports represent the ability for a component to provide something to other components (*e.g.*, service, piece of data) during a part of its life-cycle. Use ports represent the dependency of a component during a part of its life-cycle to something provided by another component. The set of use and provide ports of a component are denoted P_u and P_p. Each port is connected to at least one group of places and/or transitions, indicating which part of its life-cycle is concerned. Groups are subsets of $\Pi \cup \Theta$ and the set of groups of a component is denoted G. The set of bindings between provide ports and groups is denoted $B_u \subseteq P_u \times G$ and the set of bindings between use ports and groups is denoted $B_p \subseteq P_p \times G$. For example, in Figure 1 component *Dependency1* has two provide ports: *config* (bound to a group containing two places and two transitions) and *service* (bound to a group containing only place *running*). Component *Server* has four use ports: *service1*, *service2*, *config1* and *config2*.

B. Component instance and assembly

The configuration of a particular component instance is determined by three elements: its *marking*, its *behavior queue* and its *provide port values*. First, its *marking* $mk \subseteq \Pi \cup (\Theta \times \{\text{beginning}, \text{end}\})$ is the set of its active places and transitions. In Figure 1 instances *dep1*, *dep2* and *server* each have one active place: *running*. To model whether an action associated with a transition has been executed, transitions are either *beginning-active* if their associated action has not yet been executed, or *end-active* otherwise. In Figure 2, in (b) the two blue transitions are beginning-active while in (d) they are end-active. Second, its *behavior queue* bq is a FIFO queue of behaviors in B that determines which transition the instance will execute next, and thus which path it will follow in its life-cycle. In Figure 2, the component queue has one behavior: the blue one, while in Figure 1 all the queues are empty. Third, its *provide port values* are values that the actions of a component can assign to a *provide port*. The value

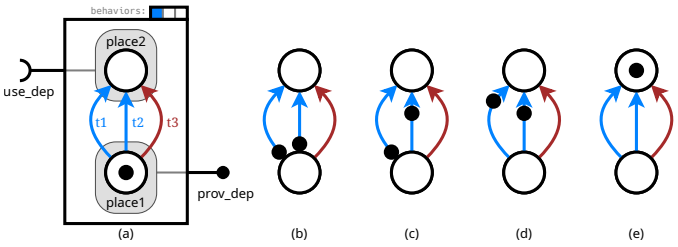


Fig. 2: Possible evolution when the blue behavior is active.

currently held by a provide port p_p is denoted $val(p_p)$. Finally, each instance has an identifier *id*.

An *assembly* $A = (I, L)$ is a set of instances I and a set of connections L between use and provide ports. A connection between port means that the instance owning the use port depends on the instance owning the provide port for a part of its life-cycle. It can also access the value owned by the provide port. For example, the assembly represented in Figure 1 has the three instances *dep1*, *dep2* and *server*, and four connections between their ports, *e.g.*, provide port *service* from instance *dep1* is connected to use port *service1* from instance *server*.

C. Reconfiguration

With Concerto, one can reconfigure the system by adding or removing component instances, modifying the dependencies between them, requesting that an instance changes its configuration by executing a behavior or waiting for an instance to finish the execution of a behavior.

A reconfiguration program is a finite sequence of instructions denoted $P = [\iota_1, \iota_2, \dots] = \iota_1 :: \iota_2 \dots$ It can be applied to an assembly, modifying it to produce a new one. Six types of instructions are available:

- add $(id : c)$ requesting the addition of a new component instance of type c with identifier *id*;
- del (id) requesting the deletion of the component instance with identifier *id*;
- con $(id_u.p_u, id_p.p_p)$, where p_u is a use port of the instance with identifier id_u and p_p is a provide port of the instance with identifier id_p, requesting the addition of a connection between the two ports;
- dcon $(id_u.p_u, id_p.p_p)$ requesting the deletion of a connection;
- pushB (id, b), where *id* is the identifier of a component instance and b one of its behaviors, requesting the addition of b to the instance's queue of behaviors;
- wait (id), where *id* is the identifier of a component instance, blocking the execution of the program until the instance's behavior queue is empty.

Note that all instructions except *wait* are non-blocking, allowing in particular each component to execute one or multiple behaviors in parallel.

D. Semantics overview

Concerto's operational semantics is made of a set of rules. Each rule takes as input the current assembly and the reconfiguration program to be executed and, if its activation conditions are met, the rule can be applied and returns the modified assembly, as well as the modified program. The rules can be applied in any valid order. Each reconfiguration instruction has a dedicated rule, and each rule consumes the first instruction of the reconfiguration program: (1) *add* and *delete* rules respectively add a new component instance (setting its marking to only the initial place) and remove an existing instance; (2) *connect* and *disconnect* rules respectively add or remove a connection; (3) the *push behavior* rule adds a behavior to the queue of behaviors of an instance; and (4) the *wait rule* which can only be applied if the queue of behaviors of a given instance is empty, and does not change the assembly.

Four rules do not correspond to the execution of an instruction and instead describe the evolution of an instance according to its active behavior. In these rules, a group is considered active if and only if at least one of its places or transitions is active. A port is active if and only if all of the groups bound to it are active. A group is said to be *provide-locked* if it is the only active group bound to a provide port which is connected to an active use port. A group is said to be *use-locked* if it is the only inactive group bound to a use port which is not connected to any active provide port. The four rules are:

1. Place to transitions: This rule can be applied if an active place of an instance has outgoing transitions associated with the first behavior in its behavior queue. When applied, the place is marked inactive and the transitions are marked *beginning-active*. In practice, this also starts the execution of the actions associated with each transition. In Figure 2, this corresponds to going from (a) to (b).

2. Transition execution: This rule can be applied if a transition is *beginning-active* and the actions associated with it has ended. When applied, the transition is marked *end-active*. In Figure 2, this corresponds to going from (b) to (c), and then from (c) to (d). Notice that the transitions could have been executed in the other order.

3. Transitions to place: This rule can be applied if, given a place π which is not in a use-locked group, all the incoming transitions of π which are reachable during the execution of current behavior are active. When applied, they are marked inactive and π is marked active. In Figure 2, this corresponds to going from (d) to (e).

4. Finished behavior: This rule can be applied if no transitions are active in an instance, and if there are no outgoing transitions from any active place associated with the current behavior. When applied, the current behavior is removed from the queue. In Figure 2(e), there are no blue transitions from *place2*, the blue behavior can therefore be removed from the behavior queue.

Some restrictions related to the synchronization of instances through their connected ports are applied to the three rules *place-to-transitions*, *transition-execution* and

transitions-to-place. Intuitively, a group connected to a provide port being actively used by another component instance is *provide-locked* because it is forbidden to deactivate such a provide port. A group connected to a non-provided use port is *use-locked* because it is not possible to activate this group until the port is provided. For example, in Figure 2, we can only go from (a) to (b) if *prov_dep* is not connected to an active use port, and we can only go from (d) to (e) if *use_dep* is connected to an active provide port.

The operational semantics of Concerto, in particular its asynchronicity at both the reconfiguration language and the parallel transitions handling levels, is what makes it more efficient than the the related work, in that it takes full advantage of parallelism opportunities.

IV. Performance model

This section deals with Concerto's performance model. Its goal is to estimate the total execution time of a reconfiguration given the initial Concerto assembly and the duration of the transitions in each instance. The performance model is described by a recursive function *DepGraph* which, given an initial assembly and a reconfiguration program, generates a weighted oriented dependency graph (V, A) with A a multiset over $V \times V \times \mathbb{R}_0^+$. Intuitively, this graph encodes all the dependencies between events enforced by the semantics of Concerto, *e.g.*, the execution of a behavior may only start after the corresponding *pushB* instruction or a transition becomes *end-active* only after its execution time has passed since it has become *beginning-active*. All arcs in the dependency graph have a weight of 0, except for those that encode the execution of a transition which have a weight equal to the duration of the transition. The execution time of the reconfiguration corresponds to the longest path between its *source* vertex representing its beginning and its *sink* vertex representing its end. Figure 3 shows the dependency graph corresponding to the reconfiguration presented in Listing 1.

Listing 1: Reconfiguration program updating *dep1* in the assembly of Figure 1.

```
pushB(dep1, update)
pushB(server, suspend)
pushB(server, deploy)
pushB(dep1, install)
wait(server)
```

Section IV-A introduces the *DepGraph* function whereas Section IV-B explains how to obtain the subgraphs corresponding to the execution of a behavior in an instance, which are used by *DepGraph*.

A. Reconfiguration dependency graph

The dependency graph is built by the recursive function *DepGraph*. Each level of recursion corresponds to one instruction and updates the context and current dependency graph for the next instructions. The base case is reached when the program to consume is empty, in which case the

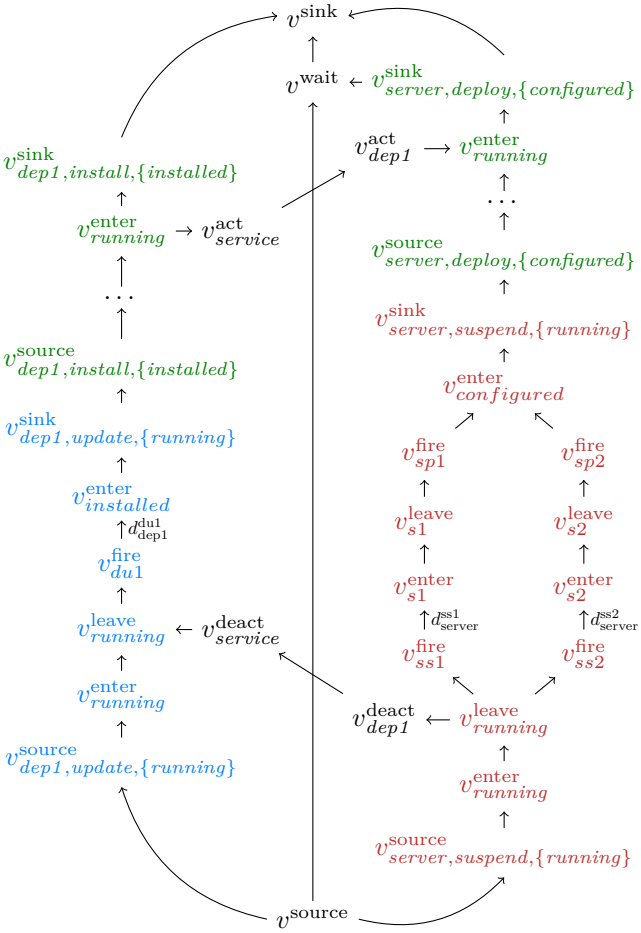


Fig. 3: Dependency graph corresponding to the reconfiguration in Listing 1 applied to the assembly in Figure 1.

function finally returns the dependency graph as-is. The recursive function takes the following inputs:

- P: the sequence of remaining instructions;
- $D = (V, A)$: The current dependency graph;
- $v^{sync} \in V$: The vertex corresponding to the latest synchronous event (*wait* instruction or beginning of the program), used to encode the fact that no behavior pushed after a *wait* instruction can begin its execution before the waited component has finished executing its queue of behaviors;
- $final_\Pi$: A function mapping each instance identifier to the final places of the latest behavior which has been pushed to it. Indeed, when another behavior is pushed to an instance, we need to know the starting places to generate the subgraph representing the execution flow of this behavior;
- end_v: A function mapping each instance identifier to the vertex in the graph corresponding to the end of the execution of its latest behavior. This is to encode the fact that any behavior pushed after it cannot start its execution before the previous one has ended;
- UP: A set of tuples $(id, p_u, v^{\text{start}}_{p_u}, v^{\text{stop}}_{p_u})$ where id is

the identifier of an instance, p_u is a use port and $v_{p_u}^{\text{start}}$ and $v_{p_u}^{\text{stop}}$ are the vertices corresponding to the port in the dependency graph. This is used to encode the list of vertices corresponding to use ports yet to be connected to a provide port;

- PP: A set of tuples $(id, p_p, v_{p_p}^{\text{start}}, v_{p_p}^{\text{stop}})$ where id is the identifier of an instance, p_p is a provide port and $v_{p_p}^{\text{start}}$ and $v_{p_p}^{\text{stop}}$ are the vertices corresponding to the port in the dependency graph. This is used to encode the list of vertices corresponding to provide ports yet to be connected to a use port;
- L: A set of connections between ports. This is used to encode the dependencies between instances of the assembly in the dependency graph.

Given an initial assembly α and a reconfiguration program P, we first call *DepGraph* with the following arguments: D contains a single vertex v^{source} representing the beginning of the P (at the bottom in Figure 3), $v^{sync} = v^{source}$ because no *wait* instruction has occurred yet, $final_\Pi$ is initialized with the active places of the instances present in α, end_v initially has an empty domain, $UP = PP = \emptyset$, and, L is the set of connections in α.

DepGraph handles different cases, depending on P:

Base case: If P is empty, a new vertex v^{sink} is added to D (at the top in Figure 3), representing the end of the program execution. For each identifier id such that $end_v(id)$ is defined, an arc with weight 0 going from $end_v(id)$ to v^{sink} are added to D, encoding the fact that all behaviors need to be over before the reconfiguration finishes. D is then returned.

Add: If $P = add(id : c) :: P'$, we add a new instance of component c with identifier id to the mapping functions $final_\Pi$ and end_v and call the function recursively on P'.

Delete: If $P = del(id) :: P'$, we call the function recursively on P' with no other change, because the deletion of a component does not impact the execution time.

Connect: If $P = con(id_u.p_u, id_p.p_p) :: P'$, we call the function recursively on P' with the new connection added to L.

Disconnect: If $P = dcon(id_u.p_u, id_p.p_p) :: P'$, we call the function recursively on P' with the connection removed from L.

Push behavior: If $P = pushB(id, b) :: P'$, we call the function recursively on P' with D fused with the new subgraph $D_{id,b}^{final_\Pi(id)}$. The way to obtain this graph is given in Section IV-B and corresponds to the execution of the behavior b in the instance with identifier id starting from places $final_\Pi(id)$, *i.e.*, the end places of the behavior executed last in instance with identifier id (or the initial place if no behavior has been executed). In Figure 3, we can see four behavior graphs, their source and sink vertices being of the color representing the corresponding behavior in Figure 1. An arc going from v^{sync} to $v_{source}^{id,b,\Pi_{start}}$ is added to the graph, encoding the fact that the behavior can only start its execution after a previous *wait* instruction has finished. Additionally, if $end_v(id)$ is defined, *i.e.*, if the instance with identifier id has already executed a behavior, an edge from $end_v(id)$ to $v_{source}^{id,b,\Pi_{start}}$ is added, encoding the fact that behavior b can only start its execution after the previous behavior is finished. Finally, we use the set of connections L to determine whether some use or provide ports encoded in the newly added subgraph are connected to other ports. If so, we check whether these ports have already been encoded by vertices in previously added subgraphs using the sets PP and UP. If so, considering a use port p_u and a provide port p_p, we add two arcs respectively going from $v_{p_p}^{\text{start}}$ to $v_{p_u}^{\text{start}}$ and from $v_{p_u}^{\text{stop}}$ to $v_{p_p}^{\text{stop}}$. PP and UP are then updated with the unconnected ports encoded in the newly added subgraph. In Figure 3, the provide port *service* is connected to the use port *dep*1, so two arcs connect the vertices representing these ports.

Wait: If $P = wait(id) :: P'$, we call the function recursively on P' with $D = (V \cup \{v^{wait}\}, A)$ (adding a new vertex v^{wait}) and $v^{sync} = v^{wait}$. This new vertex corresponds to the *wait* instruction, and encodes the fact that no instruction after the *wait* can take effect before it has terminated. Arcs are also added from the previous v^{sync} to the new v^{sync} and from $end(id)$ to v^{sync}. This corresponds to the fact that the *wait* instruction terminates once this instance has finished executing its queue of behaviors. For example, in Figure 3 there is one vertex v^{wait} at the top, corresponding to the only*wait* instruction in Listing 1.

B. Component dependency subgraph

Given an instance with identifier id, its type $c = (\Pi, \Theta, B, bhv, init, G, P_u, P_p, B_u, B_p)$, a set of initially active places $\Pi_{start} \subseteq \Pi$ and a behavior $b \in B$, we can define the dependency subgraph $D_{id,b}^{\Pi_{start}} = \left(V_{id,b}^{\Pi_{start}}, A_{id,b}^{\Pi_{start}}\right)$ corresponding to the execution of b in the instance with identifier id starting from places Π_{start}. In the following, Π_{res} and Θ_{res} correspond respectively to the set of places and the set of transitions reachable from at least one place in Π_{start} by following the transitions of behavior b. For any other notation X, X_{res} corresponds to the restriction of the set or function X to Π_{res} and Θ_{res}. For example, G_{res} is the set containing the non-empty restrictions of all groups in G to $\Pi_{res} \cup \Theta_{res}$. In the subgraph, vertices represent events such as activation or deactivation of places, transitions and ports, while arcs represent the application of Concerto's semantic rules.

Vertices: For each place $\pi \in \Pi_{res}$, we associate two vertices: v_π^{enter} corresponding to the place activation and v_π^{leave} to its deactivation. For each transition $\theta \in \Theta_{res}$, we associate one vertex v_θ^{fire} representing its *beginning-activation*. For each use port $p_u \in (P_u)_{res}$ we associate two vertices: $v_{p_u}^{\text{act}}$ representing its activation and $v_{p_u}^{\text{deact}}$ its deactivation. For each provide port $p_p \in (P_p)_{res}$ we associate two vertices: $v_{p_p}^{\text{act}}$ representing its activation and $v_{p_p}^{\text{deact}}$ its deactivation. Finally, we add one vertex $v_{id,b,\Pi_{\text{start}}}^{\text{source}}$ representing the beginning of the behavior execution and one vertex $v_{id,b,\Pi_{start}}^{\text{sink}}$ representing its end.

Arcs: In the dependency graph, weighted arcs correspond to the application of the rules from the operational semantics. Each individual arc from v_o to v_d with weight w represents a temporal constraint: the event represented by v_d must happen after duration w has passed since event v_o happened. In practice, the weights of all of the arcs are 0, except for those that represent the execution of a transition θ (rule *transition-execution*) which connect v_θ^{fire} to v_π^{enter} (where π is the destination place) with a weight equal to the duration of θ.

Figure 3 shows four subgraphs. Two of these, on the left, are for component type *Dependency1* and correspond respectively to the execution of behavior *update* starting from place *running* and to the execution of behavior *install* starting from place *installed*. The other two, on the right, are for component type *Server* and correspond respectively to the execution of behavior *suspend* starting from place *running* and to the execution of behavior *deploy* starting from place *configured*.

C. Time estimation and restrictions

Given an assembly α and a reconfiguration program P, consider D the dependency graph corresponding to the execution of P on α. Note that D is acyclic. Also, *DepGraph* can be computer in polynomial time and space w.r.t. the number of elements in the component instances taking part in the reconfiguration. The proofs of these claims are however not included in this paper.

We define the time estimation of the execution of P to be the length of a longest path between v^{source} and v^{sink} in D. Note that if the transition durations are expressed as variables, a formula can be built recursively, starting from $v = v^{\text{sink}}$. $formula(v)$ is equal to the maximum for each parent v_p of v connected with an arc of weight w of $formula(v_p)+w$. If v has no parent, then $formula(v) = 0$.

For the time estimation given by this performance model to be correct, we need to consider only deterministic assemblies with respect to transitions durations (the structure of the dependency graph would otherwise depend on the duration of the transitions). To handle arbitrary assemblies, a more complex performance model is needed, which is left as future work. Our restrictions are as follows. First, the components must be well-formed, *i.e.*, have no cycles in their *internal-net* when considering only one behavior. Also, the reconfiguration as well as the successive assemblies it generates must have no deadlocks. Ports must be bound to only one group each, and each port may only be activated up to once, and deactivated up to once during the execution of a single reconfiguration program. Each group must have only one entrance and one exit per behavior. Finally, use and provide ports need to be connected before pushing the behaviors making use of the connected ports. In practice, we expect most real-life assemblies to fulfill these requirements. In particular, all of the assemblies presented in this article do.

Listing 2: Reconfiguration program (1).

```
for i in [1,n]:
  add(dep_i : Dependency_i)
  pushB(dep_i, install)
```

Listing 3: Reconfiguration program (2).

```
for i in [1,n]:
  pushB(dep_i, update)
  pushB(dep_i, install)
```

V. Evaluation

In this section, we evaluate Concerto in three ways. First, we compare the expected performance of Concerto to the expected performance of Ansible and Aeolus on synthetic use cases, showing how and in which cases Concerto allows a gain of performance. Then, we run these synthetic use-cases using the Python proof of concept (PoC) we have developed[6]. By comparing the running times estimated by the performance model to those actually measured, we check that the performance model is accurate. Finally, we analyze a real use case (reconfiguration of a distributed database), checking that the performance matches the estimations of the performance model. The code of the experiments as well as the results we show in this paper are available online[7].

A. Performance analysis on synthetic use-cases

In this section, we make use of synthetic use-cases to show in which cases Concerto increases performance compared to Aeolus and Ansible. To do this, we first need to define the performance models of Aeolus and Ansible.

Aeolus' performance model: We define it to be the same as Concerto's, except for the fact that we can only consider component types with no parallel transitions. This is an accurate performance model for Aeolus as the only difference in terms of parallelism between Aeolus and Concerto is the fact that there can not be parallel transitions inside one Aeolus' component type.

Ansible's performance model: In general, all the transition durations are added up. However, if one transition in each of multiple instances do the same action, they can be executed in parallel using Ansible, so the maximum duration among them is used instead of their sum.

In the following, we consider four reconfiguration use-cases, each featuring different kinds of parallelism. These reconfigurations use the $Dependency_i$ components from Figure 1, where each $Dependency_i$ is assumed to do something different in transition du_i. They also use $Server_n$, a parametric version of the *Server* component which is meant to be connected to n dependencies (hence it has $2 \times n$ use ports and n sc_i, ss_i and sp_i transitions). In Listings 2 to 5, a *for* instruction is used to express parametric reconfiguration programs, with the usual semantics.

Dependencies deployment (1): We consider the deployment of n instances of types Dependency_1, ... Dependency_n. The reconfiguration is given in Listing 2 and the formulas given by the performance models of

[6]https://gitlab.inria.fr/VeRDi-project/concerto
[7]https://gitlab.inria.fr/VeRDi-project/concerto-evaluation

Concerto, Aeolus and Ansible are given in Table I (1). We first notice that the formulas are identical for Aeolus and Concerto. This is because there are no parallel transitions in the components, *i.e.*, that there is only inter-component parallelism, which both of them can handle. As for Ansible, the difference is that the formula is the sum of the maximum durations of the transitions, instead of the maximum of the sum. Recall that we take the maximum of the transition times because the actions associated to respectively transitions di_i and transitions dr_i are assumed to do the same among dep_i instances, even though they are of different types. The gain for Concerto and Aeolus depends on how much the durations differ for these identical transitions. Imagine that two dependencies are hosted in two hosts with distinct hardware specifications, such that one of the two transitions (*e.g.*, performing heavy disk access) takes a long time on one of the host but a short time on the other, and conversely for the other transition(*e.g.*, using a lot of network bandwidth). For instance, take $d_{di_1} = 5$, $d_{di_2} = 50$, $d_{dr_1} = 50$, $d_{dr_2} = 5$. In this case, the execution time for Ansible is $50 + 50 = 100$, while it is $\max(55, 55) = 55$ for Aeolus and Concerto. If we consider however that the instances are running in similar conditions, then we can expect a normal distribution of the durations for each transition. Assuming, for example, $\mu = 60$, $\sigma = 10$, a simulation over 100,000 samples was made for $n = 2$ and for $n = 2000$ to get the probability distribution of the result of both formulas. For $n = 2$, there is a small difference of 3.3 seconds with a mean of 131.3 seconds for Ansible and 128.0 seconds for Concerto/Aeolus, with a standard deviation of respectively 11.7 and 11.6 seconds. However, for $n = 2000$ Concerto/Aeolus is 20.1 seconds faster with a mean of 188.7 seconds for Ansible and a mean of 168.6 seconds for Concerto/Aeolus, both with a standard deviation of 4.7 seconds. Thanks to inter-component parallelism, Concerto and Aeolus are more scalable than Ansible.

Dependencies update no-server (2): We consider, given n dependencies in place *running*, the update of these dependencies. The reconfiguration is given in Listing 3 and the formulas are given in Table I (2). In terms of execution time, the only difference with the previous case is that transition *du* is not assumed to be the same among the dependencies, unlike *di*. This implies that Ansible cannot execute them in parallel, inducing a sum of their execution time instead of a maximum. In this case, assuming $max_i\{d_{dr_i}\}$ is small compared to $\sum_i\{d_{di_i}\}$, the expected gain of Concerto and Aeolus compared to Ansible is proportional to the number of dependencies, showing a better scalability due to inter-component parallelism.

Server deployment (3): We consider, given n dependencies in place *running*, the deployment of an instance of Server using these dependencies. The reconfiguration is given in Listing 4 and the formulas are given in Table I (3). Notice that this time, the formulas are the same for Ansible and Aeolus. This is because we consider actions (transitions sc_i) that cannot be performed in parallel by Ansible because they are different, nor by Aeolus because they are part of the same component. This translates to sum of transitions durations, instead of a maximum for Concerto. In this case, assuming $d_{sa} + d_{sr}$ is small compared to $\sum_i\{d_{sc_i}\}$, the expected gain of Concerto compared to Aeolus and Ansible is proportional to the number of parallel transitions.

Listing 4: Reconfiguration program (3).

```
add(server : Server_n)
for i in [1,n]:
  con(dep_i.config,
      server.config_i)
  con(dep_i.service,
      server.service_i)
pushB(server, deploy)
```

Listing 5: Reconfiguration program (4).

```
for i in [1,n]:
  pushB(dep_i, update)
  pushB(dep_i, install)
pushB(server, suspend)
pushB(server, deploy)
```

Dependencies update with-server (4): We consider, given n dependencies and a server in place *running*, the update of all the dependencies, which implies that the server needs to be suspended. The reconfiguration is given in Listing 5 and the formulas are given in Table I (4). The formula for Ansible is, as expected, the sum of all the durations of the transitions, except for the dr_is which can be executed in parallel. In Aeolus, thanks to inter-component parallelism, the durations do not all add up. The reconfiguration time is the longest time it takes for any component to finish executing its behaviors. The first part of the outer *max* corresponds to the execution of the dependencies. The execution time of dependency i is $du_i + dr_i$ plus the time it takes for the *service* port to be deactivable. There is no intra-component parallelism in Aeolus, so the ss_i transitions execute sequentially. Therefore, the time for use-port $service_i$ to be deactivated is $\sum_{j\leq i}\{d_{ss_j}\}$, which hence is the time it takes for the *service* port to be deactivable. The second part of the outer *max* corresponds to the execution of the server in which all the transitions have been sequentialized, hence giving the sum of all the transition durations. In Concerto, the ss_i and sp_i transitions can be executed in parallel which, compared to Aeolus, decreases signigicantly the time required before the dependencies may leave the *running* place (for the i^{th} dependency it roughly divides it by i), and divides by roughly n the execution time of the ss_i and sp_i transitions. If we set the duration of all transitions to 5 seconds, this reconfiguration with 10 dependencies would take 160 seconds for Ansible, 105 seconds for Aeolus and 15 seconds for Concerto. With 100 dependencies, it would take 1510 seconds for Ansible, 1005 seconds for Aeolus and still 15 seconds for Concerto.

B. Validation of the performance model

In order to ensure that the performance model matches the experimental results, the four synthetic use-cases presented above were implemented using our Python prototype of Concerto. Each transition performs a Python *time.sleep*, allowing to make the duration of all the

Framework	Formula
(1) *Dependencies* deployment	
Ansible	$\max_i \left\{d_{di_i}\right\} + \max_i \left\{d_{dr_i}\right\}$
Aeolus	$\max_i\{d_{di_i} + d_{dr_i}\}$
Concerto	$\max_i\{d_{di_i} + d_{dr_i}\}$
(2) *Dependencies* update no-server	
Ansible	$\sum_i \left\{d_{du_i}\right\} + \max_i \left\{d_{dr_i}\right\}$
Aeolus	$\max_i\{d_{du_i} + d_{dr_i}\}$
Concerto	$\max_i\{d_{du_i} + d_{dr_i}\}$
(3) *Server* deployment	
Ansible	$d_{sa} + \sum_i \{d_{sc_i}\} + d_{sr}$
Aeolus	$d_{sa} + \sum_i \{d_{sc_i}\} + d_{sr}$
Concerto	$d_{sa} + \max_i \{d_{sc_i}\} + d_{sr}$
(4) *Dependencies* update with-server	
Ansible	$\sum_i \left\{d_{du_i} + d_{ss_i} + d_{sp_i}\right\} + \max_i \left\{d_{dr_i}\right\} + d_{sr}$
Aeolus	$\max\left(\max_i \left\{d_{du_i} + \sum_{j\leq i} \left\{d_{ss_j}\right\} + d_{dr_i}\right\}, d_{sr} + \sum_i \{d_{ss_i} + d_{sp_i}\}\right)$
Concerto	$\max(\max_i\{d_{du_i} + d_{ss_i} + d_{dr_i}\}, d_{sr} + \max_i\{d_{ss_i} + d_{sp_i}\})$

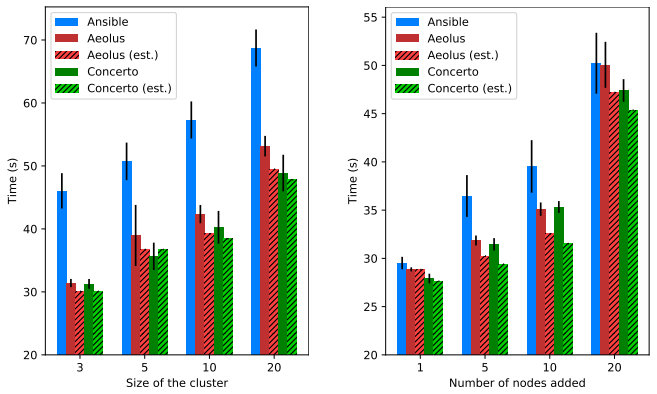
TABLE I: Formulas estimating the run-time of each reconfiguration by Ansible, Aeolus and Concerto.

transitions parameters of the component types. Given a reconfiguration, we randomly selected a duration for each transition (continuous uniform distribution between 0s and 10s), used the performance model to estimate the running time and executed the reconfiguration using the prototype, comparing the actual execution time to the estimation. The durations in this experiment do not need to be realistic because what we want to prove is that the performance model matches the actual execution times: the more scenarios encountered, even unrealistic, the better. We ran this experiment 250 times for each reconfiguration, for 1, 5 and 10 dependencies. This represents a total of $250 \times 4 \times 3 = 3000$ executions. The difference between the estimated time and the measured time was never above 0.05 seconds and never represented more than 5.7% of the estimated time (in this instance the total execution time was 0.4 seconds, which explains the relatively high percentage). The median execution times were 14.5 seconds for reconfiguration (1), 14.8 seconds for (2), 17.3 seconds for (3) and 21.7 seconds for (4). The measured time was always slightly larger than the estimated time. This is explained by the small overhead introduced by the Concerto prototype. The performance model therefore matches what is observed in reality.

C. Performance estimation on real use-cases

We consider two reconfigurations. First, the *decentralisation* from a MariaDB instance to a Galera cluster of size 3, 5, 10 or 20. Second, the *scaling* of a cluster of size 3, with the number of nodes added equal to 1, 5, 10 or 20. This can for example be part of a multi-region deployment of OpenStack, as discussed at the 2018 Vancouver OpenStack summit[8]: originally all regions use the same centralized database; then OpenStack is reconfigured so that multiple regions have a local instance of the database synced in a Galera cluster; and nodes are added when increasing the number of regions having their own local instances.

[8]https://www.openstack.org/videos/summits/vancouver-2018/highly-resilient-multi-region-keystone-deployments



(a) Decentralisation (b) Scaling

Fig. 4: Estimated and measured running times for Ansible, Aeolus and Concerto of the *decentralisation* and *scaling* reconfigurations (error bars: standard deviation).

Evaluations have been carried out on the Uvb cluster of the experimental platform Grid'5000 (www.grid5000.fr). Uvb is composed of 43 nodes equipped with two 6-core Intel Xeon X5670 CPUs, 96 GB RAM, 250 GB HDD and a 1 Gbps network + 40 Gbps InfiniBand.

The use case has been implemented with Concerto, Ansible (which is widely used by devops), and Aeolus (which is the model that offers the highest level of parallelism in the literature, *i.e.*, *inter-component*). As the development of Aeolus was discontinued, the Aeolus execution has been emulated by transforming the Concerto components so that there is no *intra-component* parallelism. The Concerto and Aeolus implementations use *SSH* calls to execute *Bash* scripts on the remote server, while the *Ansible* one uses the corresponding commands provided by Ansible.

For each tool and each parameter, the experiment was repeated 15 times. The total execution time as well as the durations of the individual transitions for Concerto and Aeolus were recorded (Ansible does not allow to measure this for each individual host). The transition durations measured for Aeolus were used along with the performance model to estimate the running time of the reconfigurations for Aeolus and Concerto. Figure 4 shows these estimations as well as the measured running times.

We observe that the performance model gives a slightly lower execution time for Concerto compared to Aeolus, which is confirmed by the actual execution times. These are comprised in the ranges of possible execution times for both Concerto and Aeolus. In terms of pure performance, Concerto's gains compared to Aeolus range from -0.6% (scaling, 10 nodes) to 8.5% (decentralisation, 5 nodes), and from 5.4% (scaling, 1 node) to 32.1% (decentralisation, 3 nodes) compared to Ansible. In terms of precision of the time estimation using the average duration for each transition, the maximum error is 10.8% (3.8s) for the

scaling with 10 additional nodes with Concerto. This is explained by the fact that taking the average durations of the transitions tends to underestimate the influence that a single transition has on the total execution time. When using respectively the minimum and maximum durations instead, the measured execution time is always between the two estimations.

VI. Conclusion

In this paper we have introduced Concerto, a reconfiguration model which increases performance in reconfiguration thanks to a higher level of parallelism. We have also presented its performance model which, given a Concerto configuration and a reconfiguration program, is able to express the execution time of this reconfiguration as a function of the execution time of the elementary actions performed. In the future, this will allow the reconfiguration choices made autonomically to take into account the expected execution time of this reconfiguration, which is important in applications with a high QoS requirement.

Concerto has sound foundations for good software engineering properties. Future work includes building on these foundations to have strong separation of concerns between actors of the reconfiguration, such as component developers and system administrators. Concerto is also formally defined, and we plan to use formal methods, like it was done in [22] for the Madeus deployment model, to provide strong safety guarantees as well as automatic reconfiguration plan inference to the users. Finally, we consider the execution of concurrent reconfigurations and the decentralization of the execution to be ways in which Concerto could better apply to Fog/Edge scenarios.

Acknowledgment

This work was funded by the Discovery project (see https://beyondtheclouds.github.io/). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several universities as well as other organizations (see https://www.grid5000.fr). These experiments also used the EnosLib library (see https://gitlab.inria.fr/discovery/enoslib).

References

[1] D. M. Chess and J. O. Kephart, "The vision of autonomic computing," *Computer*, vol. 36, no. 01, pp. 41–50, jan 2003.

[2] R. Boujbel, S. Rottenberg, S. Leriche, C. Taconet, J. Arcangeli, and C. Lecocq, "Muscadel: A deployment dsl based on a multiscale characterization framework," in *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, July 2014, pp. 708–715.

[3] L. Cudennec, G. Antoniu, and L. Bougé, "CoRDAGe: towards transparent management of interactions between applications and ressources," in *Intl Workshop on Scalable Tools for High-End Computing (STHEC 2008)*. Kos, Greece: Michael Gerndt and Jesus Labarta and Barton Miller, Jun. 2008, pp. 13–24.

[4] T. L. Nguyen, R. Nou, and A. Lebre, "YOLO: Speeding up VM and Docker Boot Time by reducing I/O operations," in *EURO-PAR 2019 - European Conference on Parallel Processing*. Göttingen, Germany: Springer, Aug. 2019, pp. 273–287.

[5] J. Darrous, S. Ibrahim, A. C. Zhou, and 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*. Washington D.C., United States: IEEE, May 2018, pp. 553–562.

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

[7] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "Opentosca – a runtime for tosca-based cloud applications," in *Service-Oriented Computing*, S. Basu, C. Pautasso, L. Zhang, and X. Fu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 692–695.

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

[9] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in bip," in *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, Sep. 2006, pp. 3–12.

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

[11] M. Aldinucci, C. Bertolli, S. Campa, M. Coppola, M. Vanneschi, and C. Zoccolo, "Autonomic grid components: the gcm proposal and self-optimising assist components," in *Joint workshop on HPC grid programming environments and components and component and framework technology in high-performance and scientific computing at HPDC*, vol. 15, 2006.

[12] F. Baude, L. Henrio, and C. Ruz, "Programming distributed and adaptable autonomous components—the gcm/proactive framework," *Software: Practice and Experience*, vol. 45, no. 9, pp. 1189–1227, 2015.

[13] S. Bouchenak, N. D. Palma, D. Hagimont, and C. Taton, "Autonomic management of clustered applications," in *2006 IEEE Intl Conference on Cluster Computing*, Sep. 2006, pp. 1–11.

[14] L. Broto, D. Hagimont, P. Stolf, N. Depalma, and S. Temate, "Autonomic management policy specification in tune," in *Proceedings of the 2008 ACM Symposium on Applied Computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 1658–1663.

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

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

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

[18] M. Chardet, H. Coullon, D. Pertin, and C. Pérez, "Madeus: A formal deployment model," in *4PAD 2018 - 5th Intl Symp. on Formal Approaches to Parallel and Distributed Systems (hosted at HPCS 2018)*, Orléans, France, Jul. 2018, pp. 1–8.

[19] F. Boyer, X. Etchevers, N. de Palma, and X. Tao, "Poster: A declarative approach for updating distributed microservices," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, May 2018, pp. 392–393.

[20] M. Garcia Valls, I. R. Lopez, and L. F. Villar, "iland: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 228–236, Feb 2013.

[21] F. Maraninchi and Y. Rémond, "Mode-automata: About modes and states for reactive systems," in *Programming Languages and Systems*, C. Hankin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 185–199.

[22] H. Coullon, C. Jard, and D. Lime, "Integrated model-checking for the design of safe and efficient distributed software commissioning," in *Integrated Formal Methods*, W. Ahrendt and S. L. Tapia Tarifa, Eds. Cham: Springer International Publishing, 2019, pp. 120–137.