Toward Safe and Efficient Reconfiguration with Concerto

Maverick Chardet, Hélène Coullon, Simon Robillard

Toward Safe and Efficient Reconfiguration with Concerto

Maverick Chardet[a], Hélène Coullon[a], Simon Robillard[a]

[a]*IMT Atlantique, Inria, LS2N, UBL, F-44307 Nantes, France*

Abstract

For large-scale distributed systems that need to adapt to a changing environment, conducting a reconfiguration is a challenging task. In particular, efficient reconfigurations require the coordination of multiple tasks with complex dependencies. We present Concerto, a model used to manage the lifecycle of software components and coordinate their reconfiguration operations. Concerto promotes efficiency with a fine-grained representation of dependencies and parallel execution of reconfiguration actions, both within components and between them. In this paper, the elements of the model are described as well as their formal semantics. In addition, we outline a performance model that can be used to estimate the time required by reconfigurations, and we describe an implementation of the model. The evaluation demonstrates the accuracy of the performance estimations, and illustrates the performance gains provided by the execution model of Concerto compared to state-of-the-art systems.

Keywords: reconfiguration, component-based models, coordination, parallelism, distributed software

1. Introduction

With the advent of cloud computing, service-oriented distributed software systems have become commonplace. The complexity and scale of these systems is expected to keep increasing, in particular through the development of fog and edge architectures [1, 2, 3, 4] and their associated applications [5]. A key ingredient for the long-term success of these systems is the capacity to adapt to a changing environment (e.g., availability of hardware resources, variations in usage load), to evolving operational requirements (e.g., quality of service, energy, security), or to evolving software services and dependencies (e.g., smart-* applications, microservice architectures), without unduly affecting functionality.

Dynamic evolution of distributed software systems has been widely studied in the literature under different forms and designations, such as self-adaptive systems [6, 7], reactive programming [8], or systems reconfiguration [9, 10, 11, 12] for instance. In this paper, we specifically target the design and execution of dynamic reconfigurations of component-based software systems. We intend to provide a formal model with well-defined semantics and enough expressivity to design generic reconfigurations of software systems at their behavioral and structural levels. The notion of component used here is very general, and is for instance applicable to component-based software systems, service-oriented applications, microservices, etc. A reconfiguration in this context basically consists in moving from one configuration of the software system to another, while guaranteeing safety properties. By configuration, we mean the topology of the distributed system (i.e., components and their connections), and the state of its various components (either started/stopped, or a finer-grained characterization that may be specific to each component type).

The reconfiguration of a complex distributed system is a difficult task. It becomes even more challenging when efficiency is taken into account, which is needed to minimize disruption time and maintain a good user experience. Furthermore, there is room to improve reconfiguration efficiency, as the associated operations can be time-consuming, especially when a large number of components are involved. Performing these operations sequentially is very inefficient and largely negates the benefit of running a system on distributed hardware. On the other hand, manually programming the parallel execution and synchronization of operations across multiple software components is an arduous task, not to mention the challenge of maintaining this type of reconfiguration script. Instead, reconfiguration solutions should promote efficient execution through parallelism (both between components and within a given component) without adding to the burden of either developers or system administrators. Additionally, the performance of reconfigurations should be predictable, in order to guide the choices of system administrators or autonomic tools conducting them (e.g., analysis and planning phases of the MAPE-K loop [13, 14]). Unfortunately, no existing solution offers both a formal model for generic reconfiguration and a high level of parallelism in reconfiguration execution. For instance, production tools such as Ansible [15], Chef [16], or Puppet [17] offer various degrees of parallelism between software components (sometimes using a declarative approach for dependencies) but necessitate the use of brittle scripts for a finer degree of control, in particular to introduce fine-grained parallelism. Additionally, these tools usually lack well-documented semantics, making them unsuitable for static analysis, performance estimation or formal verification. Furthermore, academic models such as Aeolus [18] or Engage [19] lack the ability to describe independent reconfiguration tasks for a given component, which is needed to execute them in parallel. Other models may offer the right level of parallelism, but be unsuitable to represent reconfigurations. This is the case of Madeus [20], which is meant to model deployments and is therefore unable to represent the dynamic evolution of components and their assembly during a reconfiguration.

This paper presents Concerto, a formal model for efficient reconfiguration of component-oriented software systems. The objectives of Concerto are: first, allowing **highly parallel execution** of reconfigurations through fine-grained representation of component lifecycles and their dependencies; second, providing **accurate performance estimation** of the reconfiguration; and finally, defining a **formal semantics** of its execution model, paving the way for further research in formal verification techniques. Specifically, the model presented in this paper offers the following contributions:

- Concerto is a model that offers the possibility to model efficient reconfiguration procedures (Section 3), with inter-component *and* intra-component parallelism, as well as asynchronous control of the components;
- Concerto has a clearly-defined semantics (Section 4), as needed for reliable implementation and use of static analysis techniques;
- Concerto has a predictable execution model (Section 5), which can be used to assess the performance of reconfiguration programs before their execution, with accurate results;
- Concerto has been implemented as a tool in Python and validated on use cases (Section 6), demonstrating the accuracy of its performance estimation model and showing that it out-performs both production tools and existing academic solutions in terms of performance.

Concerto has previously been presented in [21], focusing on its performance estimation capabilities, but with only an overview of the model itself. In this article, we now provide a thorough description of the semantics of Concerto and extend the experiments to measure the performance benefits of its execution model.

2. Related Work

In this section, we first present the literature regarding dynamic reconfiguration. We classify the related work into two main categories: contributions related to Component-Based Software Engineering (CBSE), and contributions of the DevOps community. Then, we give an analysis of the related work based on two main criteria: the ability to model the lifecycle of a distributed software system, and the ability to model different levels of dependencies, and thereby to allow parallel execution of reconfiguration tasks. Finally, we discuss our contribution relative to the work presented here and to the given criteria.

2.1. Reconfigurations in the literature

Dynamic adaptation of distributed software systems is a broad research topic [22]. This paper focuses on mechanisms to execute a reconfiguration in the context of component-based systems. Building software in a compositional manner enhances, on the one hand, software engineering properties such as maintainability and reusability [18], and on the other hand, the ability to dynamically adapt the components of the software system dynamically [23]. Furthermore, a compositional approach is suited to distributed software systems, in which independent pieces of software run on different machines across a network [24, 25, 26]. For this reason, we mainly study two categories of contributions: those arising from the CBSE and AOP (Aspect-Oriented Programming) communities, and those arising from the DevOps community. The latter include many deployment, provisioning and orchestration tools developed in recent years that also follow a compositional approach.

Component-Based Software Engineering - In CBSE, a system is built by combining several independent components. The interface of components is provided by *ports* that expose both the provisions and the requirements of each component. Components can be instantiated and compatible ports can be connected to form an *assembly* of components. The runtime engines of component models are responsible for instantiating and starting component instances and ensuring their communications. Many component-based models have been proposed, such as Corba [27], CCM [28], CCA [29], SCA [30], L2C [26], BIP [31] or Fractal [32, 33]. In the specific domain of reconfiguration (dynamic evolution of component-based software), Fractal has been one of the main contributions for two main reasons. First, Fractal is a reflective component model, i.e., it offers APIs and controllers to introspect and modify the structure and behavior of components, thus enabling dynamic control of software systems. Second, a Fractal component consists of a programmable membrane (implemented itself as a component assembly) that represents the non-functional control part of the encapsulated content (functional code), thus enabling the customization of reconfiguration procedures (or protocols). Many contributions on reconfiguration have been based on Fractal: Frascati [34] uses Fractal to extend SCA with reconfiguration capabilities; the languages Fpath and Fscript were designed to easily introspect and reconfigure Fractal applications [35]; GCM/ProActive [24, 25] extends Fractal with Active objects and allows behavioral reconfiguration [36, 12]; Jade [37] and its evolution Tune [38] build a framework for autonomous legacy software management on top of Fractal; SAFRAN [39] combines Fractal with AOP to better design adaptations. From this line of work, it appears that the flexibility of the programmable membrane, i.e., the programmable control of components, is one of the keys toward generic reconfigurations. Fractal controllers include the lifecycle management of components (e.g., starting, stopping components) and the ability to modify the assembly on the fly by adding or removing components and connections between them. Other component models also include mechanisms to perform reconfigurations. For instance, in [40, 41] L2C has been extended with reconfiguration capabilities for the specific case of high performance numerical simulations. Studies have also been conducted for reconfiguration mechanisms in BIP [42].

Some contributions reuse the concepts of components and their dependencies or connections to specifically model the non-functional part of legacy distributed software systems. For example, a Fractal component can wrap an existing piece of software and add a membrane to control it (e.g., Jade and Tune approaches). SmartFrog [43], Engage [19], Aeolus [44] and the contribution detailed in [45, 46, 47] offer ways to program the lifecycle of any distributed software systems, including reconfiguration steps, by using finite state automata.

DevOps contributions - In recent years, the DevOps community has improved the automatic management of distributed software systems and distributed infrastructures. The deployment — or commissioning — of distributed software systems has been enhanced by the usage of containers (e.g., Docker, LXC), or by languages such as Ansible [15] and Chef [16], which allow the definition of sequences of actions to perform, or Puppet [17] and SaltStack [48], which use a declarative approach to define and reach target states for a system. This last class of solutions is often identified as Infrastructure-As-Code (IAC) or Software-Configuration-Management (SCM). These efforts are all based on a compositional approach, each tool using its own denomination for components, e.g., *roles*, *nodes*, *resources* etc. These tools can also be used for reconfiguration. Ansible, which is the closest to a scripting language, is very flexible, and any sequence of actions can be programmed on components to perform an adaptation. Declarative approaches offer a higher level of abstraction, with developers describing the expected results of changes, rather than the procedure to perform them. This approach makes Puppet easier to use but also more limited in its reconfiguration capabilities. Complex reconfiguration actions within Puppet have to be handled manually in general-purpose languages such as Bash or Perl.

Tools such as Kubernetes [49] or Docker Swarm [50] offer to automatically orchestrate and manage a set of components assimilated to a set of containers. These tools offer interesting reconfiguration capabilities, in particular the automatic management of containers faults, the automatic management of new component/service discovery, and the automatic management of scalability rules.

DevOps contributions towards reconfiguration are typically not generic. Instead, each solution targets a very specific reconfiguration case. For instance, some tools are specific to containers without any dependencies, others are specifically made to handle scalability rules.

In the case of cloud computing, efforts were made to standardize the orchestration of applications. OCCI [51] (Open Cloud Computing Interface) is a standard for the unification of cloud providers APIs. TOSCA [52] (Topology and Orchestration Specification for Cloud Applications) is an OASIS standard for the deployment and reconfiguration of cloud applications, with multiple implementations, such as OpenTOSCA [53] or Cloudify. Although it is not an implementation of TOSCA, Apache Brooklyn [54] works in a similar way when it comes to the execution of reconfiguration. TOSCA nodes represent software or resources, connected to one another by relationships (for example, a server may contain an OS which contains a piece of software, a piece of software may require another one to be present, etc.). Nodes can be added or removed, thus handling some forms of reconfiguration. Additionally, nodes may define custom operations applicable to them (for example to change their configuration).

2.2. Comparison of existing solutions

This paper focuses on the efficiency of reconfigurations. Among features of reconfiguration models, two are particularly important for efficiency: lifecycle management, and declarations of dependencies, a key to parallel execution.

Lifecycle management - The lifecycle of a component refers to the stages of its lifespan, and the transitions between them. The lifecycle of a distributed system is thus related to the lifecycles of its components and their coordination. Most tools that support reconfiguration only consider two states for the components: running/installed and stopped/uninstalled. In TOSCA, each node has two main possible states: *deployed* or *not deployed*. By default, only those two states are taken into account when coordinating lifecycles of components. Platform-specific tools such as AWS CloudFormation[1] and OpenStack Heat[2], or technology-specific tools such as Kubernetes[3] behave in the same way in their basic mode.

However, this basic lifecycle limits coordination control, hence efficiency. A more flexible and finer-grained lifecycle management is necessary to reduce reconfiguration execution time. Fractal [33] (and its offshoots [24, 25]), with its programmable membrane, is a good candidate for flexible lifecycle management. However, a controller for the lifecycle of each component must be implemented manually in the membrane, which is a tedious task. As a result, dedicated models have been proposed, in which all the components share the same lifecycle. For instance, Jade [37] and its evolution Tune [38] give legacy software a Fractal interface with a two-state lifecycle (on/off). The Deployware [55] framework was built on top of Fractal and offers six different actions for any component: *configure*, *start*, *manage*, *stop*, *unconfigure* and *uninstall*. Although the specification of OCCI [51] only defines a 2-state lifecycle for the components, some work has been done to increase this number and provide finer control. For example, in [56] the authors present an extension of the standard to support a 4-state lifecycle: *undeployed*, *deployed*, *inactive* and *active*.

Engage [19], Aeolus [57, 44], and the contribution of Brogi et al. [45, 46, 47] are component-oriented reconfiguration models in which the lifecycle of each component is modeled as a fully customizable finite state machine. In addition to featuring programmable lifecycles, these solutions automatically coordinate the lifecycles of components through ports or dependencies. Madeus [20], although it does not handle reconfigurations, also offers customizable lifecycles and automatic coordination. Finally, some DevOps tools have been extended to model more complex lifecycles. For example, TOSCA provides the notion of workflow, which could be assimilated to a customized lifecycle made of a sequence of operations that are usually hidden within artifacts of nodes. This allows the definition of specific lifecycles within TOSCA, but requires a specification of workflows of operations to be provided by the user.

Dependency declarations and parallelism - Minimizing the execution time of a reconfiguration, assuming that reconfiguration actions have already been individually optimized [58, 59], comes down to increasing the level of parallelism between these actions. This is directly correlated to the ability to express dependencies between operations of several components, in other words between the lifecycles of components.

[1]`https://aws.amazon.com/cloudformation/`
[2]`https://docs.openstack.org/heat/latest/`
[3]`http://kubernetes.io/`

First, parallelism can be introduced manually with general-purpose programming languages. For instance, when using Fractal, the developer is free to manually handle Java threads inside reconfiguration actions. This obviously requires a great deal of effort on the part of the user, and is subject to a high propensity for errors. Higher-level models and solutions are required to introduce parallelism transparently and automatically.

In TOSCA, by default there can be no parallelism between nodes that have dependencies to one another. One must be started or configured before the other. Custom workflows address this limitation by introducing dependencies between operations of nodes, thus creating something close to an execution graph, or a scheduling. However, as previously detailed, workflows have to be specified manually, which could be a tedious process. TOSCA-based tools (such as AWS CloudFormation or OpenStack Heat), as well as declarative DevOps management tools (such as Puppet and SaltStack), suffer from a similar lack of fine-grained dependencies. Dependencies are declared at the component level, so that parallelism between two components is never possible if one depends on the other. This is also the case for instance when using Argo Workflows within Kubernetes. Argo Workflows [4] is an open source container-native workflow engine for orchestrating parallel jobs on Kubernetes. The use of Argo within Kubernetes makes it possible to add dependencies between containers, unlike the basic usage that launches all containers simultaneously and retries starting operations upon failure. However, one task in Argo models the starting of one container, and the lifecycle of a container is still limited to started/stopped. As a result, when two containers have a dependency, one must stop before the other can start.

Procedural DevOps management tools like Ansible and Chef offer some forms of automatic parallelism but are limited by the sequential nature of their languages. For instance, Ansible can execute one action on multiple nodes in parallel only if the action is the same on all nodes. Additionally, because they work by executing a series of instructions directly on the hosts, these tools do not offer a formal framework.

From these descriptions, one can note that the level of parallelism reachable during execution is directly correlated to the granularity of component lifecycles, as well as the granularity of the dependencies that can be declared between reconfiguration operations and between lifecycles. Indeed, a fine-grained programmable lifecycle is required to express finer dependencies and to improve efficiency. In [45, 46, 47] and in Aeolus, parallel execution of components is allowed even if one depends on the other (after the dependency has been satisfied) and even if the actions to perform are distinct. This is possible thanks to the fine-grained dependencies that can be specified between the finite state automata that model the lifecycles of the components. However, parallelism within a single finite state automata is not possible in Aeolus. Madeus increases the level of parallelism in comparison to Aeolus, by allowing multiple actions to be

[4]https://argoproj.github.io/argo/examples/

performed concurrently within a component. However, Madeus is specific to deployment and does not support reconfiguration in general.

2.3. Discussion

This paper presents Concerto, a model for reconfiguration. Concerto combines the CBSE concepts of components and ports with state machines to model the lifecycle of each software component in a given distributed software. The fine-grained customizable modeling of lifecycles as well as their connections (i.e., dependencies) and their automatic coordination leads to faster reconfigurations. By leveraging fine-grained, dependencies Concerto allows a very high level of parallelism compared to the related work.

Among the efforts mentioned above, Concerto is closest to (i) the imperative language Ansible, as it offers a flexible way to specify complex lifecycles and reconfiguration procedures and (ii) the component-based model Aeolus, as it represents component lifecycles with a finite state machine. Unlike Aeolus however, Concerto allows the declaration of independent reconfigurations tasks inside a component, and their execution in parallel.

Furthermore, Concerto complements many other contributions on reconfiguration mentioned above. For instance, Concerto could be used by tools such as TOSCA, CloudFormation, OpenStack Heat, or Kubernetes to specify their fixed lifecycles, or to turn to more flexible lifecycles and their automatic coordination, thus opening the way to better performance.

Finally, although some academic contributions are equipped with formal semantics of models and associated properties (i.e., Fractal-based contributions and Aeolus), most DevOps tools lack well-documented semantics, making them unsuitable for static analysis, performance prediction or formal verification

3. Overview of Concerto

This section explains the main concepts of Concerto in a simple way, while the full formalism is given in Section 4.

3.1. Components and Assemblies

Concerto targets DevOps scenarios, in which system administrators have limited control over the functional behavior of the distributed software systems but should have full control over their lifecycles. In Concerto, this is achieved through the concept of *control component*. Each *control component* represents the lifecycle of a piece of distributed software, as well as its possible evolution over time.

A control component is typically written by a component developer and may have dependencies that can be fulfilled by other components during their lifecycle, or fulfill the dependencies of other components. Formally, each component is characterized by an *internal net* that describes its evolution, and a set of *coordination ports* (or simply *ports*) that describes its coordination interface (provisions and requirements) towards other components.

The internal net is composed of a set of *places*, which represent the internal states of the component, and *transitions*, which allow passage from one place to another. The semantics of this internal net is inspired by Petri nets, with tokens marking reached places and fired transitions. Unlike Petri nets however, firing a transition is not an atomic event: after it is fired, a transition can remain active while other events take places, before finally ending. Indeed, in the implementation, a transition typically corresponds to an action that is required to reach a different state, for example installing a package or setting up parameters. The duration of actions is represented by the non-atomicity of transitions, and the end of a transition after a period of activity represents the completion of the action. The specific nature and effects of these actions is out of the scope of the formal model, as it focuses on the coordination of actions.

For the purpose of verification, it is preferable to consider all possible executions without assumptions on the time required by actions to complete. For this reason, the semantics presented in Section 4 do not have numerical values for action durations, and simply assume that an action may terminate at any point after having been started. For performance estimation however, it is useful to consider the time taken by actions. This is the object of Section 5, in which transitions are associated with a numerical value representing an estimation of the execution time of the action.

The internal net allows the modeling of parallelism inside a component: multiple places may be active, and multiple transitions may be fired simultaneously. This is a distinguishing characteristic of Concerto, setting it apart from models based on finite state machines, such as Aeolus.

Externally, the interface of a component is given by its *ports*. Each port is bound to a subset of the lifecycle that is called a *group*. A group contains places and transitions between them. A port is *active* when at least one of the elements of its group holds a token. Because Concerto is intended to model dynamic systems, a port that is active can later become inactive, once its group no longer holds any token.

Ports can be of two types, namely *provide ports* and *use ports*. A provide port represents a service or piece of information that the component offers when the port is active. On the other hand, a use port indicates that a component requires some service or information to perform. The elements of a group bound to a use port can only acquire a token when that port is *provided*. The active status of use port is defined just as for provide ports, by the presence of tokens on the bound group, but instead signifies that the service or piece of information is being used: as long as this is the case, the providing port should not become inactive. Thus, exchanges of service and data between components are modeled and implicitly synchronized.

A *control component type* (or *component type* for short) is a template that defines an internal net and ports. It also defines the state of the internal net upon creation, through its *initial places*. Multiple *component instances* may be created based on this template. A collection of component instances and their links is called an *assembly*.

Figure 1 depicts an assembly comprising two components. The component on the left represents a server, with two provide ports: one represents its IP address (that other components need in order to set up their configurations) and the other its main service. On the right, the component corresponding to a client has two use ports, linked to the provide ports of the server. Both components are here depicted in their starting configurations: the server is `undeployed`, and the client `uninstalled`, as denoted by the tokens on the places of the same name. Firing transitions will eventually lead to different configurations. In the client component, two transitions can be fired from the place holding a token. Recall that transitions correspond to actions, and these two transitions represent independent actions that can be executed in parallel. Conversely, two transitions reach the place labeled `configured`, indicating that two actions need to be completed before that place can be reached, thus representing a local synchronization point. Some places of the client belong to groups of its use ports, and therefore may be reached only when the corresponding provide port of the server is active. For example, to become `configured`, the client must obtain the IP address of the server: this requirement is modeled by the use port `server_ip`. Correspondingly, the provide port of the server represents it making its IP address available. Here, the port is activated when the server is either `allocated` or `running`, or executing the transition between those places. When this is the case, the places `installed` and `configured` of the client become reachable. To be reachable, the place `running` of the client requires not only the port `server_ip` to be provided, but also the port `server`.

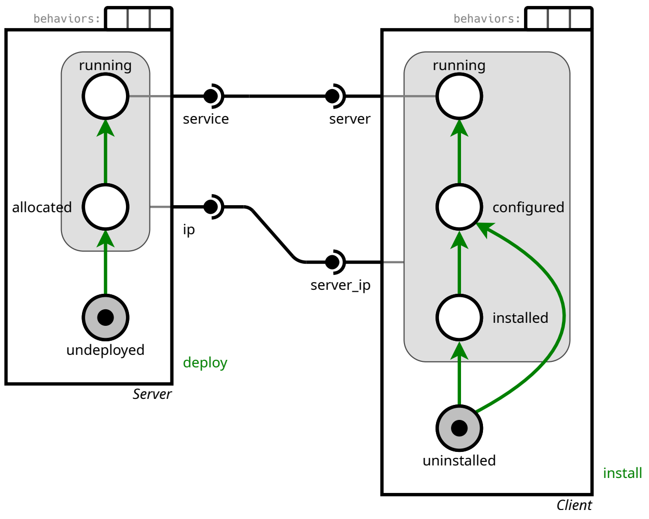


Figure 1: A Concerto assembly representing a server-client software.

3.2. Behaviors

In a changing environment, components need to adapt to new conditions. In Concerto, this dynamic nature of components is captured by *behaviors*. At any given point, a component instance executes a behavior that specifies the subset of its transitions that may be fired. Intuitively, changing the behavior of a component enables or disables some transitions in its internal net, and thereby modifies the sequence of actions (transitions) that it can perform and states (places) that it can reach.

The set of behaviors of a component is akin to its user interface. For instance, a component might include a behavior that executes setup actions and ultimately leads the component to a working state, and another behavior that executes actions required before the component deletion. Concerto does not impose any requirements on the behaviors that should be offered. Instead, it is up to the component developer to determine what set of behaviors constitutes an adequate user interface for any particular component.

Behaviors can be controlled from outside the component (e.g., by an automated controller or, as explained in the following subsection, a reconfiguration program) but this control is asynchronous. Requests to execute specific behaviors are sent to a component instance, that will queue the requests until it can execute them. A component instance keeps executing in its current behavior until it reaches a configuration where all places that hold tokens have no outgoing transitions active in that behavior. In this configuration and behavior, no progress can occur, therefore the behavior is discarded, and the component behavior is changed according to the next queued request, if any.

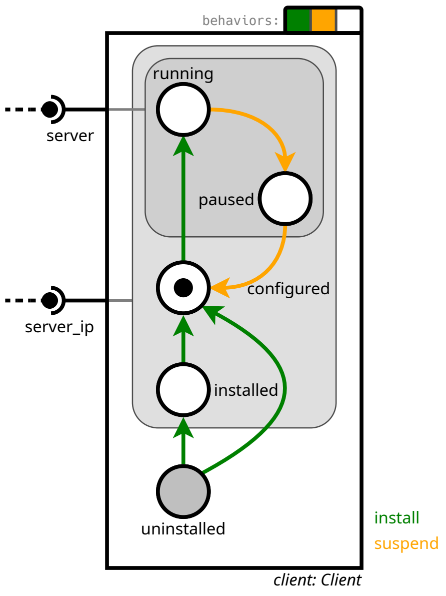


Figure 2: A Concerto component with multiple behaviors.

Figure 2 shows another client component. Unlike the previous example, this component is equipped with two different behaviors, that are depicted using two different colors for the transitions. The behavior `install`, represented by green transitions, allows the component to ultimately reach the state `running` (if the use port conditions are satisfied). The other behavior, `suspend`, represented by yellow transitions, allows a running component to go back to the state `configured`. This can be used, for example, to stop using the port `server`, before further modifications to the assembly. The queue of behavior requests is depicted at the top of the component, with the leftmost color corresponding to the current behavior. The component depicted here is in the `configured` state, which has an active outgoing transition under the behavior `install`, therefore the component must remain in that behavior. Eventually, the component, while still in the behavior `install`, will reach the state `running`, which has no active outgoing transitions in that behavior. Then, the queued request for behavior `suspend` will be taken into account, and the component switched to that behavior.

A real-world component would likely implement more behaviors, notably one to go back to the place `uninstalled`, putting the component in a state where all use ports are inactive, a requirement in order to disconnect the component.

Listing 1 illustrates how the component of Figure 2 can be declared in our Python implementation of Concerto. It has the following structure: lines 3 to 9 declare the places of the component and line 11 the initial place; lines 13 to 16 declare groups of places and give them names; lines 18 to 21 declare the possible behaviors; lines 23 to 30 declare the transitions with their source and destination places, the behavior to which they belong, and a function corresponding to the action to perform when the transition is fired; lines 32 to 35 declare the dependencies, or ports, of the component with a type (use or provide) and the group to which they are bound; and finally after line 37, the functions that are executed during transitions are defined. These functions perform concrete reconfiguration actions, including communication with remote machines if necessary, while Concerto offers a way to coordinate their executions.

3.3. Reconfiguration Program

In order to allow the reconfiguration of assemblies, Concerto is equipped with a basic language. While the specification of the components and their internal nets is carried out by component developers, the Concerto reconfiguration language is intended for use by system administrators or third-party developers writing a dynamic reconfiguration. Alternatively, it could be used by an autonomous tool (control loop) generating reconfiguration plans. The primitives of the language allow the following operations:

- *creation of components*: a uniquely named instance of a component is created, according to a given component type;
- *connection of component ports*: a use port that is not already linked, becomes linked to a provide port;

Listing 1: Declaration of the *Client* component type from Figure 2 in our implementation

```
class Client(Component):
    def create(self):
        self.places = [
            'uninstalled',
            'installed',
            'configured',
            'running',
            'paused'
        ]

        self.initial_place = 'uninstalled'

        self.groups = {
            'using_server_ip': ['installed','configured','running','paused'],
            'using_server': ['running','paused']
        }

        self.behaviors = [
            'install',
            'suspend'
        ]

        self.transitions = {
            'install1': ('uninstalled','installed','install',self.install1),
            'install2': ('uninstalled','configured','install',self.install2),
            'configure': ('installed','configured','install',self.configure),
            'start': ('configured','running','install',self.start),
            'suspend1': ('running','paused','suspend',self.suspend1),
            'suspend2': ('paused','configured','suspend', self.suspend2)
        }

        self.dependencies = {
            'server_ip': (DepType.USE, 'using_server_ip'),
            'server': (DepType.USE, 'using_server')
        }

    # Definition of the actions
    def install1(self):
        remote = SSHClient()
        remote.connect(host, user, pwd)
        remote.exec_command(cmd)
        ...
```

- *disconnection of component ports*: a link between a use and provide port is deleted, or if the use port is bound to an active transition, the execution is suspended until this is no longer the case;
- *deletion of components*: a component that is not linked to any other is deleted;
- *request to execute a behavior*: a component instance is requested to execute a behavior, with the request being handled asynchronously, as described above;
- *synchronization*: further execution of the reconfiguration program is suspended until a given component instance has executed all behavior requests submitted to it.

These operations only target control components. For instance, component creation is a purely logical operation at the control level, and does not trigger concrete tasks typically associated with starting a component (e.g., downloading an image, starting a virtual machine). These actions must be associated to transitions in a dedicated behavior, and the execution of that behavior must be requested after the control component creation.

Listings 2 and 3 give two examples of reconfiguration programs, as written in our Python implementation of Concerto. The execution of these programs is detailed in the next subsection. The first example creates an instance of `Reconfiguration` named `deploy`, that performs the deployment of an assembly composed of one client and one server connected together. The second creates a `Reconfiguration` instance named `maintain` that performs a maintenance of the server in that assembly. In both cases, instructions to perform are added to the `Reconfiguration` instance, then the execution of this reconfiguration over the assembly is initiated by a call to `run_reconfiguration`. The execution of the Python script is then suspended until the reconfiguration has finished, with a call to `synchronize`.

Listing 2: Declaration of a Concerto reconfiguration program corresponding to a deployment. The target assembly and the steps of the execution are depicted in Figure 3

```
deploy = Reconfiguration()
deploy.add("client", Client)
deploy.add("server", Server)
deploy.connect("client", "server_ip","server", "ip")
deploy.connect("client", "server","server", "service")
deploy.push_behavior("client", "install")
deploy.push_behavior("server", "deploy")
deploy.wait("client")

assembly = Assembly()
assembly.run_reconfiguration(deploy)
assembly.synchronize()
assembly.terminate()
```

Listing 3: Declaration of a Concerto reconfiguration program to maintain a server in the assembly. Figure 4 shows the steps of the execution.

```
maintain = Reconfiguration()
maintain.pushB("server", "maintain")
maintain.pushB("server", "deploy")
maintain.pushB("client", "suspend")
maintain.pushB("client", "install")
maintain.wait("client")

assembly.run_reconfiguration(maintain)
assembly.synchronize()
assembly.terminate()
```

3.4. Execution

The evolution of a Concerto assembly is based on two levels of parallelism. As detailed previously, one component can execute multiple transitions simultaneously (*intra-component parallelism*). In addition, the executions of components behaviors are carried out independently from each other (*inter-component parallelism*). Global synchronization can be specified manually with the aforementioned primitive, but synchronization between two components occurs automatically as a result of their connection through use and provide ports. The model ensures that a part of the lifecycle of a component that has an external dependency (encoded by a use port) is executed only when that dependency is satisfied by another component. Conversely, a component that provides a service during part of its lifecycle (encoded by a provide port) is prevented from interrupting the service while it is in use. When a component is ready to perform an action (transition) that will interrupt a service, only use ports that are already using the corresponding provide port may continue to do so, while additional usage is blocked, thus allowing the providing component to eventually fire the transition.

The impact of these synchronization points on performance is reduced by the capacity for internal parallelism, as components may perform independent actions while they wait for other components to activate or deactivate ports. This model greatly facilitates the coordination of components, while avoiding unnecessary idle time. Likewise, the asynchronous nature of behavior requests means that it is not necessary to test the state of a component in order to initiate reconfiguration operations, and that these operations can be carried out as early as possible, as determined by the model. These different elements of the execution model of Concerto combine to allow the description of very efficient reconfiguration procedures with minimum need for manual synchronization and explicit parallelization of tasks.

3.5. Examples

In the following, we detail the execution of the two examples presented in Listings 2 and 3, in order to illustrate the semantics of Concerto. Each item of the following enumerations represent one configuration at a given point in the execution (i.e., a snapshot of the execution). Each of these configurations is also depicted respectively in Figures 3 and 4 with the corresponding numbers indicated on three types of elements: the red tokens (indicating the position of tokens for this configuration), the use and provide ports (indicating the configuration numbers associated to the activation of provide ports and the usage of use ports), and on the corresponding state of the behavior queue.

First, here is an example of execution for the reconfiguration `deploy` of Listing 2 represented in Figure 3.

0. This first configuration corresponds to the state of the assembly after the reconfiguration instructions from Line 2 to Line 7 have been executed. Components have been added, connected, and behaviors `deploy` (for component `server`) and `install` (for component `client`) have been pushed,

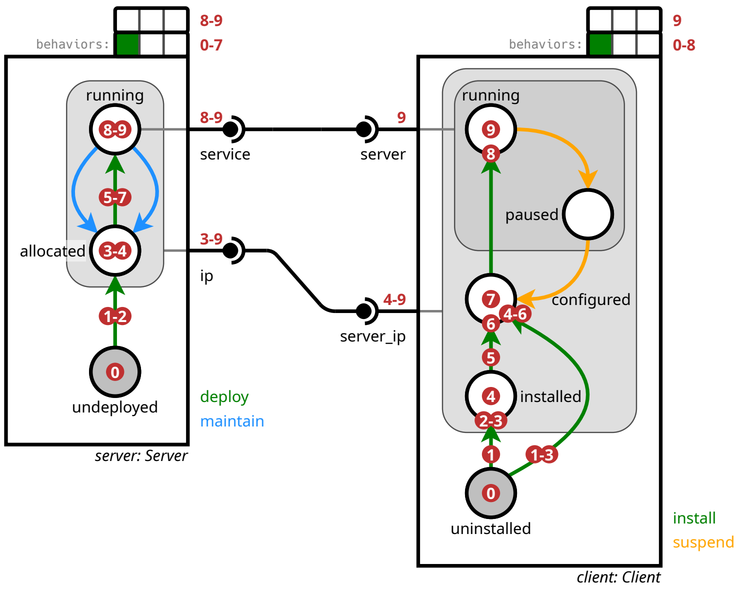


Figure 3: Example of execution for the reconfiguration of Listing 2.

but their execution has not started yet. Initial places `undeployed` and `uninstalled` hold tokens.

1. Tokens are removed from the initial places and one token is placed on each associated outgoing transition. In this configuration, two transitions of the component `client` start their execution simultaneously.
2. One of the transitions of the component `client` reaches its ending, (in the implementation, this happens when the associated function returns, i.e., when a concrete reconfiguration action is completed). However, the destination place `installed` cannot be reached yet, as this requires external information about the IP address of the server. This is represented by the use port `server_ip` bound to a group containing the place `installed`.
3. The first transition of the component `server` ends, and the place `allocated` is reached. Hence, the provide port `ip` becomes active.
4. The use port `server_ip`, which is connected to the provide port `ip`, is now provided and the place `installed` of the component `client` is reached.
5. Next, transitions are fired in both components. Notice that the transition started from the place `uninstalled` in step 1 has reached its ending. However, the place `configured` cannot be reached as one of its incoming transitions is not finished yet.
6. The transition from `installed` to `configured` in the component `client` reaches its ending.
7. After the termination of the actions associated with both incoming transitions of the place `configured`, tokens are removed from the transition

endings, and the place is reached.

8. In this configuration, the place `deploy` of the component `server` is reached, thus activating the provide port `service` and the associated connection. Furthermore, as no more transitions can be fired for the current behavior, that behavior is removed from the behavior queue, which becomes empty.
9. With the activation of the connection between the ports `service` and `server`, the place `running` of the component `client` is reached. As no more transitions can be fired for the current behavior, the behavior is removed from the queue, which is empty. This triggers the last instruction (Line 8) of the reconfiguration program, a synchronization barrier blocking the execution until the component instance `client` had executed all its behavior requests.

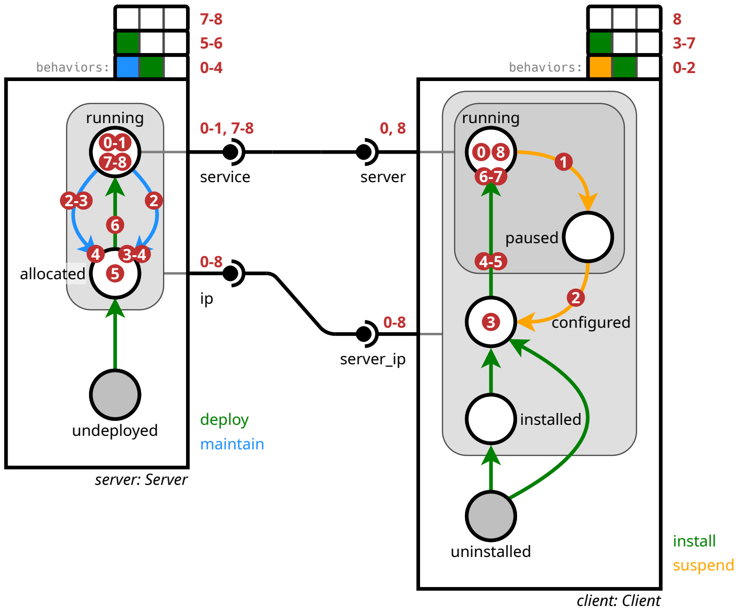


Figure 4: Example of execution for the reconfiguration of Listing 3.

Second, here is an example of execution for the reconfiguration of Listing 3 represented in Figure 4. This reconfiguration will typically be executed some time after the deployment in Listing 2. We describe its execution, with the assembly initially in configuration 9 of the previous example.

0. Places `running` of both components hold tokens. Following the execution of instructions from Line 2 to Line 5, behaviors `maintain` (blue) and `deploy` (green) have been pushed to the behavior queue of the instance `server`, and behaviors `suspend` (yellow) and `install` (green) have been pushed to the behavior queue of the instance `client`.

1. The transition from `running` to `paused` in the instance `client` is fired. It is the only active transition that can be fired, as firing transitions from the place `running` of instance `server` would interrupt the provide port `service` while it is used.
2. The transition from `paused` to `configured` in the instance `client` is fired. The port `service` of `server` is not in use anymore. As a result, transitions from `running` to `allocated` in the instance `server`, which are in the current behavior `maintain`, are now fired.
3. In this configuration, the place `configured` of `client` is reached. Thus, no more transitions in the behavior `suspend` can be fired, and it is removed from the behavior queue of `client`, making `install` (green transitions) the behavior of that component. One of the two transitions leading to `allocated` in the instance `server` has reached its ending.
4. Both transitions of the blue behavior `maintain` have reached their endings in the instance `server`. Furthermore, the transition from `configured` to `running`, which is now in the current behavior of the instance `client`, has been fired.
5. The place `allocated` of `server` is reached, hence the behavior `maintain` is removed from the behavior queue, making `deploy` (green transitions) the behavior of that component.
6. The transition from `allocated` to `running`, which is now in the current behavior of `server`, is fired, and the transition leading to `running` in `client` ends. However, the place `running` cannot be reached as the provide port `service` is not active.
7. The place `running` of `server` is reached, thus activating the provide port and the associated connection.
8. The place `running` of `client` is reached, thus ending the behavior `install`. This triggers the last synchronization barrier in the reconfiguration program (Line 6).

3.6. Verification

Concerto was designed with formal semantics in mind, making it a suitable platform for static analysis and formal methods. These techniques allow the verification of the correctness of a given reconfiguration on a given assembly. Properties that can be used to describe correctness include deadlock-freedom (all dependencies are eventually satisfied, allowing the reconfiguration to progress), reachability (the reconfiguration leads to a specified state), invariant preservation (a given property remains true throughout the reconfiguration) and temporal properties (some events happen in a specified order).

In addition to the correctness properties, we are also interested in the analysis of quantitative properties that distinguish functionally equivalent reconfigurations. Notably, the degree of parallelism and the execution time of a reconfiguration are useful metrics that can guide component developers as well as system administrators. For these quantitative properties, it may be helpful to consider additional information that is not strictly included in the formal model, such as the execution time of actions or the underlying infrastructure used to execute the reconfiguration.

Our performance estimation tool (Section 5) can be seen as a first step in this direction, as it can be used to detect deadlock and to estimate execution time in some reconfigurations. However this tool remains limited, and for more complex reconfigurations or assemblies, or for properties other than deadlock-freedom, more sophisticated techniques are required. Model-checking is a natural candidate for this task. It has notably been used to analyze assemblies in Madeus [60], a model for deployment that shares similarities with Concerto. This work relies on a translation of assemblies into time Petri nets, in order to provide a unified framework for verification of both correctness properties and qualitative properties. It demonstrates the viability of model-checking techniques on systems similar to those targeted by Concerto. However, reconfiguration is a more general problem than deployment, hence more challenging. To compensate for the added complexity, and to improve scalability, it would be useful to optimize the translation, e.g., with goal-driven simplifications. However, the aforementioned techniques are outside the scope of this paper and left to future work.

4. The Concerto Formal Model

While the previous section gave an overview of the concepts used in Concerto, this section presents them in a formal way. After that, the operational semantics of the model is also detailed.

4.1. Control Component Type

A *control component type* (or simply *component type*) is a template used to construct various instances. Formally, a component type is defined as a tuple $(\Pi, \pi_{\text{init}}, \Delta, \Theta, B, P_u, P_p)$. Π is the set of *places* in the internal net, with a distinguished element π_{init} called the *initial place*. To handle synchronization of parallel transitions, places are equipped with *stations*. Each station corresponds to a set of incoming transitions: once all these transitions have been executed, the place can be reached. Δ is the set of stations, and the place to which a station δ is attached is denoted $\pi(\delta)$. Θ is the set of *transitions* in the internal net, where each element of Θ is a pair (π, D) with $\pi \in \Pi$ the source place from which the transition originates, and $D \in \mathcal{P}(\Delta)$ the non-empty set of destination stations of the transition. Note that a transition has a single source but one or more destinations, because it can operate as a *switch*, modeling an action with various possible outcomes. In order to distinguish these possible outcomes during the execution, we use the notion of *transition ending*, a pair (θ, δ) composed of a transition $\theta = (\pi, D) \in \Theta$ and a station $\delta \in \Delta$, such that $\delta \in D$. B is the set of behaviors of the component type, where each element of B is a subset of Θ, representing the usable transitions under that behavior. Transitions in a given behavior are such that they do not form a cycle in the internal net: behaviors are meant to represent a set of operations that will terminate if the necessary use ports are eventually provided. Finally, P_p is the set of *provide ports* and

P_u the set of *use ports* of the component type. Each (use or provide) port p is bound to a set of places, called its *group* and denoted $G(p)$, that is used to determine when the port is active.

When we need to distinguish the elements of various component types, we will use a superscript notation, e.g., Π^c to refer to the places of the type c.

4.2. Component Instance

A component instance is defined as a tuple $(id, c, Q, \mathcal{M})$, where id is a unique identifier, c is a component type, Q is a sequence of elements of B^c, and $\mathcal{M}$ a subset of $\Pi^c \cup \Theta^c \cup (\Theta^c \times \Delta^c)$. This tuple describes the state of the instance at some point in the execution. In particular, Q represents a queue of behaviors to be successively executed, and $\mathcal{M}$ is the set of places, transitions and transition endings that hold tokens. Tokens are central in the semantics, where they denote a current state (tokens on places), an ongoing action (tokens on transitions) or the result of an action (tokens on transition endings).

4.3. Assembly and Reconfiguration Program

An *assembly* is a set of component instances and links between their ports. Formally, it is defined as a pair (I, L), where I is a finite set of component instances, and L is a set of tuples (i_1, u, i_2, p), where $i_1 \in I$ is a component of type c_1, $i_2 \in I$ is a distinct component of type c_2, $u \in P_u^{c_1}$ and $p \in P_p^{c_2}$.

A reconfiguration program targets an assembly and is a sequence of reconfiguration instructions. In the rest of this section, for some element e and some sequence of elements S, we denote $e \cdot S$ (respectively $S \cdot e$) the sequence constructed by adding e at the beginning (respectively the end) of S. The empty sequence is denoted $[]$.

The available instructions are $\texttt{add}(id, c)$, $\texttt{del}(id)$ (creation and deletion of component), $\texttt{con}(id_1, u, id_2, p)$, $\texttt{dcon}(id_1, u, id_2, p)$ (connection and disconnection), $\texttt{pushB}(id, b)$ (request to execute a behavior), and $\texttt{wait}(id)$ (synchronization), where id must be an instance identifier, c a component type, u a use port, p a provide port and b a behavior. The semantics of these instruction is detailed below.

4.4. Operational Semantics

We now define the rules that describe the evolution of Concerto assemblies and the effect of reconfiguration instructions. To that end, we introduce the notion of *configuration*. A configuration is a tuple $\langle (I, L), R\rangle$ where (I, L) is an assembly and R is a reconfiguration program. The semantics are given by a binary relation over configurations, denoted $\rightsquigarrow$, such that $C_1 \rightsquigarrow C_2$ if the configuration C_1 can lead to the configuration C_2. The semantics are not deterministic, in particular, there is no priority when multiple events are possible (firing a transition, reaching a place, etc.). Consequently, $\rightsquigarrow$ is not a function.

4.4.1. Statuses of Ports

We first give some definitions related to ports, that will later be needed to define the synchronization conditions in the semantic rules. The status of a port is decided by the group (set of places) to which it is bound. A port is active if at least one of the places of its group holds a token, or if at least one transition (or transition ending) located between two of the places of the group holds a token. Formally, for a given component c and group $G \subseteq \Pi^c$, we define the elements of the group to be

$$\mathit{elements}(G) \equiv \bigcup \begin{cases} G \\ \{(s, D) \in \Theta \mid s \in G \land \forall \delta \in D,\, \pi(\delta) \in G\} \\ \{((s, D), \delta) \in \Theta \times \Delta \mid \delta \in D \land s \in G \land \forall \delta' \in D,\, \pi(\delta') \in G\} \end{cases}$$

A port p in an instance $i = (c, id, Q, \mathcal{M})$ is active if at least one element of its group holds a token.

$$\mathit{active}(i, p) \equiv \mathit{elements}(G(p)) \cap \mathcal{M} \neq \emptyset$$

For active provide ports in an instance $i = (c, id, b \cdot Q, \mathcal{M})$, we also need to consider the special case where the component is ready to fire transitions that will lead to the deactivation of the port. In this case, the port only provides to the use ports that are already using it, and refuses new usage. This state is defined by

$$\mathit{refusing}(i, p) \equiv \forall e \in \mathit{elements}(G(p)), e \in \mathcal{M} \to \mathit{exit}(e, G(p))$$

where $\mathit{exit}(e, G)$ holds when e is a place and has outgoing transitions in the current behavior of the instance that leave the group G, and no transitions that do not leave G

$$\mathit{exit}(e, G) \equiv e \in \Pi^c \land \big(\exists D, (e, D) \in b\big) \land \big(\forall D \forall \delta, (e, D) \in b \land \delta \in D \to \pi(\delta) \notin G\big)$$

4.4.2. Evolution of Components

We now present the rules that describe the evolution of components outside of reconfiguration instructions. Each of these rules affects exactly one component instance in the assembly, but some of them consider the state of the provide and use ports linked to the instance, and therefore depend on the state of the whole assembly. We use the notation $A \uplus B$ to denote the union of two disjoint sets A and B.

Firing transitions.

$$\frac{\pi \in \Pi^c \cap \mathcal{M} \qquad \forall (j, u, i, p) \in L,\ \mathit{active}(j, u) \to \mathit{active}(i', p)}{\langle (I \uplus \{i\}, L), R\rangle \rightsquigarrow \langle (I \uplus \{i'\}, L), R\rangle}$$

where

- $i = (id, c, b \cdot Q, \mathcal{M})$

- $i' = (id, c, b \cdot Q, \mathcal{M} \cup \{(\pi, D) \in b\} \setminus \{\pi\})$

Intuitively, when a place holds a token, its outgoing transitions in the current behavior of the component may be fired simultaneously. This is represented by removing the token on the place, and placing tokens on each of these transitions instead. The second condition prevents the rule from being applied in situations where it would deactivate a provide port (in particular, one whose group contains π) that is being used by another component.

Ending transition.

$$\frac{\theta = (\pi, D) \in \Theta^c \cap \mathcal{M} \qquad \delta \in D}{\langle (I \uplus \{i\}, L), R\rangle \rightsquigarrow \langle (I \uplus \{i'\}, L), R\rangle}$$

where

- $i = (id, c, Q, \mathcal{M})$
- $i' = (id, c, Q, \mathcal{M} \cup \{(\theta, \delta)\} \setminus \{\theta\})$

This rule represents the end of a transition by transferring the token from the transition to one of its endings. The choice of the transition ending (in the case where the transition is a switch, i.e., has multiple endings) is non-deterministic. This non-determinism represents the fact that a transition may have various outcomes, depending on parameters (e.g., user input, time) that are not specified in the semantic model.

Entering place.

$$\frac{\delta \in \Delta \qquad E_\delta \subseteq \mathcal{M} \qquad \forall u,\ \pi(\delta) \in G(u) \rightarrow \mathit{provided}(i, u) \land \mathit{allowed}(i, u)}{\langle (I \uplus \{i\}, L), R\rangle \rightsquigarrow \langle (I \uplus \{i'\}, L), R\rangle}$$

where

- $E_\delta = \{\big((s, D), \delta\big) \mid (s, D) \in \Theta^c \land \delta \in D\}$, i.e., the set of transition endings that reach the station δ
- $i = (id, c, Q, \mathcal{M})$
- $i' = (id, c, Q, \mathcal{M} \cup \{\pi(\delta)\} \setminus E_\delta)$

and

- $\mathit{provided}(i, u) \equiv \exists (i, u, j, p) \in L,\ \mathit{active}(j, p)$, indicating that the requirement of the use port u is satisfied,
- $\mathit{allowed}(i, u) \equiv \forall (i, u, j, p) \in L,\ \mathit{refusing}(j, p) \land \neg\, \mathit{active}(i, u) \rightarrow \neg\, \mathit{active}(i', u)$, ensuring that the change does not initiate usage to a port that is currently refusing it.

If all the transition endings that reach a station hold a token, and if the use port conditions are satisfied, the tokens can be removed from the transition endings and a token added to the place to which the station is attached. Intuitively, this represents a synchronization point between multiple transitions before reaching a place.

Finishing behavior.

$$\frac{\mathcal{M} \subseteq \Pi^c \qquad \forall(\pi, D) \in b,\ \pi \notin \mathcal{M}}{\langle (I \uplus \{i\}, L), R\rangle \rightsquigarrow \langle (I \uplus \{i'\}, L), R\rangle}$$

where

- $i = (id, c, b \cdot Q, \mathcal{M})$
- $i' = (id, c, Q, \mathcal{M})$

If a component has tokens only on places that have no outgoing transitions active in the current behavior, then this behavior is discarded. This leads to a modification of the current behavior, and therefore of the active transitions of the component.

4.4.3. Reconfiguration Instructions

Lastly, we present the semantic rules describing the instructions of the reconfiguration language. Each of these rules depends on the first instruction in the reconfiguration program, and describes how the instances of the assembly or their links are modified as a result.

Add component instance.

$$\frac{\iota = \texttt{add}(id, c) \qquad \neg\exists(id, c', Q, \mathcal{M}) \in I}{\langle (I, L), \iota \cdot R\rangle \rightsquigarrow \langle (I \uplus \{(id, c, [], \{\pi^c_{\text{init}}\})\}, L), R\rangle}$$

The instruction $\texttt{add}(id, c)$ creates a component instance of type c, provided that the identifier is not already attached to another instance in the assembly. The created instance has an empty behavior queue, and its initial place holds a token.

Delete component instance.

$$\frac{\iota = \texttt{del}(id) \qquad i = (id, c, Q, \mathcal{M}) \in I \qquad \neg\exists(i_1, u, i_2, p) \in L,\ i = i_1 \lor i = i_2}{\langle (I, L), \iota \cdot R\rangle \rightsquigarrow \langle (I \setminus \{i\}, L), R\rangle}$$

The instruction $\texttt{del}(id)$ deletes a component identified by id from the assembly, provided that the component is not connected to any other.

Connect ports.

$$\frac{\iota = \texttt{con}(id_1, u, id_2, p) \qquad \{i_1, i_2\} \subseteq I \qquad i_1 \neq i_2 \qquad \neg\exists(i_1, u, i', p') \in L}{\langle (I, L), \iota \cdot R\rangle \rightsquigarrow \langle (I, L \uplus \{(i_1, u, i_2, p)\}), R\rangle}$$

where i_1 is an instance of type c_1 identified by id_1, and i_2 is an instance of type c_2 identified by id_2, such that $u \in P^{c_1}_u$ and $p \in P^{c_2}_p$.

The instruction $\texttt{con}(id_1, u, id_2, p)$ adds a connection between the use port u and provide port p of two distinct instances identified by id_1 and id_2. Use ports can be connected to at most one provide port, therefore the instruction is executed only if the port u is not already connected.

Disconnect ports.

$$\frac{\iota = \texttt{dcon}(id_1, u, id_2, p) \qquad \neg\, \mathit{active}(i_1, u)}{\langle (I, L), \iota \cdot R\rangle \rightsquigarrow \langle (I, L \setminus \{(i_1, u, i_2, p)\}), R\rangle}$$

where i_1 is an instance of type c_1 identified by id_1, and i_2 is an instance of type c_2 identified by id_2, such that $u \in P_u^{c_1}$ and $p \in P_p^{c_2}$.

The instruction $\texttt{dcon}(id_1, u, id_2, p)$ removes the connection between the ports u and p of the components identified by id_1 and id_2. This can only happen when the use port u is inactive.

Pushing behavior.

$$\frac{\iota = \texttt{pushB}(id, b) \qquad b \in B^c}{\langle (I \uplus \{i\}, L), \iota \cdot R\rangle \rightsquigarrow \langle (I \uplus \{i'\}, L), R\rangle}$$

where

- $i = (id, c, Q, \mathcal{M})$
- $i' = (id, c, Q \cdot b, \mathcal{M})$

The instruction $\texttt{pushB}(id, b)$ corresponds to an asynchronous behavior request directed at the component identified by id. That request is added to the behavior queue of the component.

Waiting.

$$\frac{\iota = \texttt{wait}(id) \qquad (id, c, [], \mathcal{M}) \in I}{\langle (I, L), \iota \cdot R\rangle \rightsquigarrow \langle (I, L), R\rangle}$$

The instruction $\texttt{wait}(id)$ acts as a synchronization barrier until the component identified by id has executed all the behavior requests submitted to it.

5. Performance Model of Concerto

This section describes a performance model for Concerto reconfigurations. Its goal is to estimate the total execution time of a reconfiguration given (i) the reconfiguration program, (ii) the initial assembly, and (iii) the duration of the transitions in each instance. The computed result is the time required to execute the critical path in the reconfiguration, i.e., the longest sequence of events that has to be performed to complete the reconfiguration. This corresponds to the execution time of the reconfiguration in the optimal case where all transitions are fired as early as possible, and there is no restriction on the number of transitions that can be executed in parallel.

The main element of the performance model is a weighted oriented dependency graph (V, A) with A a multiset over $V \times \mathbb{R}_{\geq 0} \times V$. Intuitively, the vertices of the graph represent events occurring during the execution (e.g., the activation of a port, the execution of an instruction) and the arcs represent the dependencies between events, e.g., the fact that the execution of a behavior may only start after the corresponding `pushB` instruction has been executed. Arcs that correspond to the execution of a transition are weighted with a positive value encoding the duration of the transition. All other arcs have a weight of 0. The critical path corresponds to the longest path between the vertex representing the beginning of the reconfiguration and the vertex representing its end.

Cycles in the dependency graph correspond to deadlocks in the reconfiguration. In this case, the performance analysis correctly indicates that the reconfiguration will not end, as the longest path in the graph has infinite length. However, some other types of non-progressing states (e.g., where a use port has become deactivated before being needed) are not captured: a finite critical path only indicates that there exists a terminating execution of the reconfiguration, but it does not guarantee termination of all executions.

5.1. Assumptions

For the purpose of performance estimation, the outcome and execution time of actions must be known, therefore we assume that:

- all transitions have exactly one ending (no switches);
- transition durations are given as values of $\mathbb{R}_{\geq 0}$ by a function $time(id, \theta)$.

Transition durations depend on an instance identifier, so that the execution of a similar transition may require different times in various instances (as a result of hardware discrepancies or data locality, for example). In practice, these durations are often estimations provided by system administrators.

This performance model is applicable to many commonly occurring reconfiguration scenarios, however it is not meant to be a complete analysis tool. In particular, it is restricted to:

- assemblies such that all groups have at most one entrance and one exit (i.e., places connected by a transition to another place outside of the group) in each behavior;
- reconfigurations that may lead to at most one activation and one deactivation of a given port.

The first condition implies that the activation of a port matches the activation of a single place (the entrance to the group of the port) and the deactivation of that port matches the deactivation of the exit to the same group.

The second condition ensures that the port provision condition required to enter places can be mapped to an event in the dependency graph. For more complex reconfiguration scenarios, where ports have multiple periods of activation, it is possible to split the reconfiguration script in multiple parts and analyze them separately, however this may require the insertion of global synchronization points, with an effect on performance.

In practice these assumptions are compatible with many real cases of reconfiguration, as illustrated in Section 6.

5.2. Reconfiguration dependency graph

The dependency graph is constructed by Algorithm 1, which considers each instruction of the reconfiguration program iteratively and extends the graph accordingly. Besides the graph $D = (V, A)$ being constructed, the algorithm maintains the following auxiliary variables:

- a vertex $v^{\text{sync}} \in V$ that corresponds to the latest synchronization barrier (beginning of the program or last blocking instruction `wait` or `dcon`);
- a function tokens_{Π} that maps identifiers id to the places that will hold tokens when the last behavior of the instance identified by id has been executed;
- a function end_v that maps identifiers id to the vertex in the graph that represents the end of the execution of the last behavior of the instance identified by id.

We denote by $f_{\perp}$ the function that is undefined everywhere, and by $f[y := v]$ the functional update of f, i.e., the function that maps y to v, and all other values x to $f(x)$.

The construction of the graph, as described in Algorithm 1, begins with a vertex v^{source} that represents the beginning of the execution of the reconfiguration program. The auxiliary function end_v initially maps each component identifier to v^{source}, while the function tokens_{Π} initially maps each identifier to the marked places $\mathcal{M}$ of the corresponding instance, in the initial state of the assembly I. Vertices representing the activation and deactivation of each port are also added to the graph. The graph is then extended, by considering instructions in the order in which they occur.

Wait. Given an instruction `wait`(id), the graph is extended with a unique vertex v^{wait}, representing the synchronization event. An arc is added from the vertex v^{sync} to v^{wait} (synchronization occurs after the previous synchronization barrier) and another from $\mathit{end}_v(id)$ to v^{sync} (synchronization occurs after the component identified by id has executed all its behavior requests). The auxiliary variable v^{sync}, which represents the last synchronization point, is updated.

Connection. Given an instruction `con`(id_1, u, id_2, p), edges are added to represent the order in which the connected ports can be (de)activated, as well as an edge to represent the fact that the activation of the use port cannot occur before the connection, i.e., the last synchronization point represented by v^{sync}.

Disconnection. An instruction `dcon`(id_1, u, id_2, p) is a synchronization point, and is therefore treated similarly to wait, except that the synchronization condition is the deactivation of the port u represented by $v^{\text{deact}}_{id_1,u}$.

```
Data: assembly $(I, L)$, reconfiguration $\iota_1 \cdot \iota_2 \cdot \ldots \cdot \iota_n$
Result: graph $(V, A)$
$V \leftarrow \{v^{\text{source}}\}$ ;
$A \leftarrow \emptyset$ ;
$\mathit{tokens}_\Pi, \mathit{end}_v \leftarrow f_\bot$ ;
for $(id, c, Q, \mathcal{M}) \in I$ do
    $\mathit{tokens}_\Pi \leftarrow \mathit{tokens}_\Pi[id := \mathcal{M}]$ ;
    $\mathit{end}_v \leftarrow \mathit{end}_v[id := v^{\text{source}}]$ ;
    for $p \in P^c_u \cup P^c_p$ do
        $V \leftarrow V \cup \{v^{\text{act}}_{id,p}, v^{\text{deact}}_{id,p}\}$ ;
    end
end
for $(i, u, j, p) \in L$ do
    $A \leftarrow A \cup \{(v^{\text{act}}_{id_j,p}, 0, v^{\text{act}}_{id_i,u}), (v^{\text{deact}}_{id_i,u}, 0, v^{\text{deact}}_{id_j,p})\}$ ;
end
$v^{\text{sync}} \leftarrow v^{\text{source}}$ ;
for $i$ from $1$ to $n$ do
    match $\iota_i$ with
        case wait($id$) do
            $V \leftarrow V \cup \{v^{\text{wait}}_i\}$ ;
            $A \leftarrow A \cup \{(v^{\text{sync}}, 0, v^{\text{wait}}_i), (\mathit{end}_v(id), 0, v^{\text{wait}}_i)\}$ ;
            $v^{\text{sync}} \leftarrow v^{\text{wait}}_i$ ;
        case con($id_1, u, id_2, p$) do
            $A \leftarrow A \cup \{(v^{\text{sync}}, 0, v^{\text{act}}_{id_1,u}),(v^{\text{act}}_{id_2,p}, 0, v^{\text{act}}_{id_1,u}),(v^{\text{deact}}_{id_1,u}, 0, v^{\text{deact}}_{id_2,p})\}$ ;
        case dcon($id_1, u, id_2, p$) do
            $V \leftarrow V \cup \{v^{\text{dcon}}_i\}$ ;
            $A \leftarrow A \cup \{(v^{\text{sync}}, 0, v^{\text{dcon}}_i), (v^{\text{deact}}_{id_1,u}, 0, v^{\text{dcon}})\}$ ;
            $v^{\text{sync}} \leftarrow v^{\text{dcon}}_i$ ;
        case pushB($id, b$) do
            $\mathit{behaviorGraph}(id, b)$ ;
        end
    end
end
$V \leftarrow V \cup \{v^{\text{sink}}\}$ ;
for $(id, c, Q, \mathcal{M}) \in I$ do
    $A \leftarrow A \cup \{(\mathit{end}_v(id), 0, v^{\text{sink}})\}$ ;
end
$A \leftarrow A \cup \{(v^{\text{sync}}, 0, v^{\text{sink}})\}$ ;
return $(V, A)$ ;
```

Algorithm 1: The construction of the dependency graph.

```
Procedure $\textit{behaviorGraph}(id, b)$ is
    let $c$ be the type of the instance identified by $id$
    $V \leftarrow V \cup \{v^{\text{source}}_{id,b}, v^{\text{sink}}_{id,b}\}$ ;
    $A \leftarrow A \cup \{(end(id), 0, v^{\text{source}}_{id,b}), (v^{\text{sync}}, 0, v^{\text{source}}_{id,b})\}$ ;
    for $\pi \in \Pi^c$ do
        $V \leftarrow V \cup \{v^{\text{enter}}_{\pi}, v^{\text{leave}}_{\pi}\}$ ;
        $A \leftarrow A \cup \{(v^{\text{enter}}_{\pi}, 0, v^{\text{leave}}_{\pi})\}$ ;
        if $\pi \in tokens_{\Pi}(id)$ then
            $A \leftarrow A \cup \{(v^{\text{source}}_{id,b}, 0, v^{\text{enter}}_{\pi})\}$ ;
        end
    end
    for $\theta = (s, \{d\}) \in b$ do
        $V \leftarrow V \cup \{v^{\text{fire}}_{\theta}\}$ ;
        $A \leftarrow A \cup \{(v^{\text{leave}}_{s}, 0, v^{\text{fire}}_{\theta}), (v^{\text{fire}}_{\theta}, time(id, b), v^{\text{enter}}_{\pi(d)})\}$ ;
    end
    for $p \in P^c_p$ do
        $A \leftarrow A \cup \{(v^{\text{enter}}_{in(G(p),b)}, 0, v^{\text{act}}_{id,p}), (v^{\text{deact}}_{id,p}, 0, v^{\text{leave}}_{out(G(p),b)})\}$ ;
    end
    for $u \in P^c_u$ do
        $A \leftarrow A \cup \{(v^{\text{act}}_{id,u}, 0, v^{\text{enter}}_{in(G(u),b)}), (v^{\text{leave}}_{out(G(u),b)}, 0, v^{\text{deact}}_{id,u})\}$ ;
    end
    remove elements of $V$ and $A$ that are not reachable from $v^{\text{source}}_{id,b}$ ;
    $tokens_{\Pi} \leftarrow tokens_{\Pi}[id := \{\pi \mid v^{\text{enter}}_{\pi} \in V \wedge \neg\exists(\pi, D) \in b\}]$ ;
    $end_v \leftarrow end_v[id := v^{\text{sink}}_{id,b}]$ ;
    for $\pi \in tokens_{\Pi}(id)$ do
        $A \leftarrow A \cup \{(v^{\text{enter}}_{\pi}, 0, v^{\text{sink}}_{id,b}\}$ ;
    end
end
```

Algorithm 2: The construction of the dependency subgraph for each instruction.

Creation and deletion of components. The creation or deletion of a control component does not measurably contribute to the execution time. Indeed, any action to perform on a given control component is achieved through behaviors in Concerto, and the creation and the deletion only refer to instances of control components. Furthermore, these are not blocking operations: deletion requires the component to be disconnected, but this is a well-formedness condition that can be checked statically. Therefore these two instructions are not taken into account during the construction of the dependency graph (the list of components in the assembly should also include those created during the reconfiguration).

Push behavior. The case for the instruction $\texttt{pushB}(id, b)$ is handled in the procedure *behaviorGraph* (Algorithm 2). Given a component identifier *id* and a behavior b, *behaviorGraph* extends the graph with vertices representing the events occurring during the execution of that behavior. The construction of this subgraph depends on the component instance identified by *id* and the behavior b, but also the set of places $tokens_{\Pi}(id)$, i.e., the places that hold tokens at the beginning of this execution of b.

Two vertices $v_{id,b}^{\text{source}}$ and $v_{id,b}^{\text{sink}}$ are added to represent the beginning and end of the behavior. The former is connected to $end_v(id)$ to ensure that behaviors are executed in the order in which they are requested, and to v^{sync} to ensure that the last synchronization point is taken into account.

For each place π in the component type, a vertex v_{π}^{enter} representing the place being entered is added to the graph. If the place holds a token at the beginning of the behavior b, this vertex is connected to $v_{id,b}^{\text{source}}$. Another vertex v_{π}^{leave} is also added, representing the point at which the outgoing transitions are ready to be fired, after the place has been reached and any provide port bound to that place has been deactivated.

For each transition $\theta = (s, \{d\})$ in the requested behavior, one vertex v_{θ}^{fire} is added, connected to v_{s}^{leave} to encode the fact that the transition may only start after a token leaves its source place, and to $v_{\pi(d)}^{\text{enter}}$ to represent its outcome. The latter connection is weighted $time(i, \theta)$ to represent the time taken by the execution of the action associated to θ.

For each provide port p, we consider the group $G(p)$, and in particular the entrance place $in(G(p), b)$ and exit place $out(G(p), b)$ of that group, under the behavior b. Two arcs are added. The first, from $v_{in(G(p),b)}^{\text{enter}}$ to v_{p}^{act}, represents the fact that p becomes active after a token has been added to the entrance of the group. The second arc, from v_{p}^{deact} to $v_{out(G)}^{\text{leave}}$, represents the fact that the group may be deactivated only after p is not in use anymore. Conversely, for each use port u, two arcs are added, one from v_{u}^{act} to $v_{in(G(u),b)}^{\text{enter}}$, and another from $v_{out(G(u),b)}^{\text{leave}}$ to v_{u}^{deact}.

Note that the subgraph that was just constructed to describe the events of the behavior b may not be connected if some places and transitions are not reachable in a given behavior and starting configuration. For this reason, the vertices and arcs that are not reachable from $v_{id,b}^{\text{source}}$ are removed. It is then easy to determine the final places of the behavior and update $tokens_{\Pi}$ accordingly.

The vertices corresponding to the final places are connected to $v_{id,b}^{\text{sink}}$, denoting the end of the behavior when all final places are reached.

5.3. Example

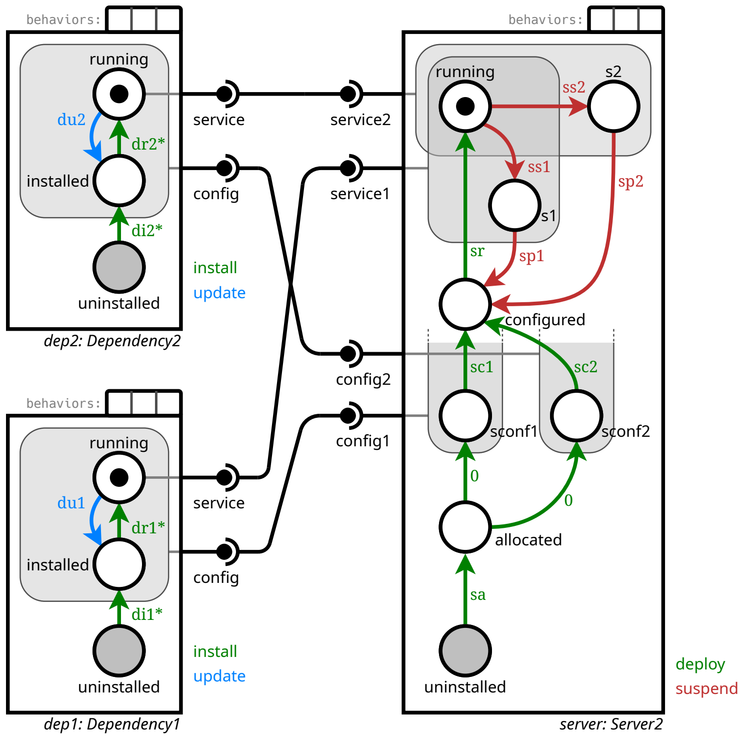


Figure 5: A Concerto assembly with two components Dependency$_i$ and one component Server$_2$. In the server component, the two groups containing either place `sconf1` or place `sconf2` also contain all the places represented above them (`configured`, `running`, `s1` and `s2`).

Listing 4: Reconfiguration program updating *dep*1 in the assembly of Figure 5.

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

Let us consider the reconfiguration presented in Listing 4, targeting the assembly depicted in Figure 5 in its initial configuration. The main goal of this reconfiguration is to run an update in component instance `dep1` (the transition `du1`), then go back to the state `running`. Since this interrupts the provide port `service` of that component, the component `server` must transition to a state where it does not use that port, then go back to its own state `running` after the update. Note that most synchronization is done implicitly via ports: the transition `du1` of component `dep1` can only be fired after the use port `service1` has been deactivated, and the transition `sr` of component `server` can only be fired after the provide port `service` has been reactivated. In addition, when the component `dep1` executes the behavior `update`, its port `service` becomes *refusing*, meaning that use ports cannot initiate usage with it. This prevents the scenario where the component `server` executes both behaviors `suspend` and `deploy` before the component `dep1` has had a chance to fire the transition `du2`, which would cause non-termination of the reconfiguration. Without this mechanism, a synchronization barrier would be needed before requesting the behavior `deploy`.

The corresponding dependency graph is presented in Figure 6. Colors are used to distinguish subgraphs corresponding to behaviors: the vertices introduced by the call to $\mathit{behaviorGraph}(\mathit{dep}1, \mathit{update})$ are shown in blue, those introduced by $\mathit{behaviorGraph}(\mathit{srv}, \mathit{suspend})$ in red, and those introduced by $\mathit{behaviorGraph}(\mathit{srv}, \mathit{deploy})$ and $\mathit{behaviorGraph}(\mathit{dep}1, \mathit{install})$ in green. These last two subgraphs are only partially shown. The graph also contains a vertex v^{wait} corresponding to the last instruction in the reconfiguration, and vertices v^{source} and v^{sink} generated in the initial phase of the graph construction. For vertices corresponding to transitions, the weight is indicated. All other vertices have weight 0.

Given concrete values for the function *time*, this graph can be used to determine the longest sequence of events in the reconfiguration the transition, hence the minimum time required to execute it.

6. Evaluation

In this section, we first describe the implementation of Concerto, and illustrate its usage on a real use case. Then we compare the expected execution time of reconfigurations under the execution models of Ansible, Aeolus and Concerto. This is done in three steps: (i) we provide ways to estimate execution times under the models of Ansible and Aeolus, (ii) we empirically validate our performance estimations for Concerto on synthetic use cases, and (iii) we compare the expected performances of the three models on those use cases.

6.1. Implementation

In this section, we present the Python implementation of Concerto that we use in our experiments. The full source code is available online[5].

Figure 7 shows the Python types and classes that are available in our implementation of Concerto. The class `Component` represents a Concerto control component type. A new component type is created by declaring a new class that inherits `Component`. The class must override the abstract method `create`, in

[5]`https://gitlab.inria.fr/VeRDi-project/concerto`

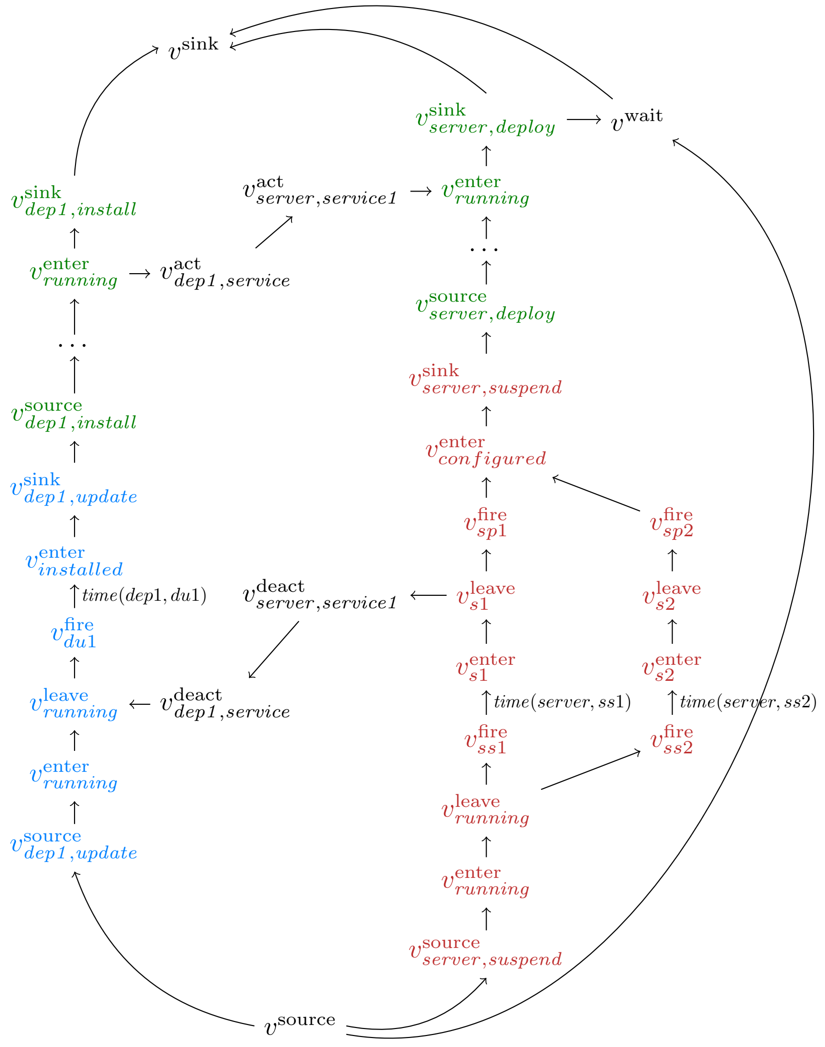

v^{sink}
v^{wait}
$v^{\text{sink}}_{server,deploy}$
$v^{\text{act}}_{server,service1}$
$v^{\text{enter}}_{running}$
$v^{\text{sink}}_{dep1,install}$
$v^{\text{enter}}_{running}$
$v^{\text{act}}_{dep1,service}$
$v^{\text{source}}_{server,deploy}$
$v^{\text{sink}}_{server,suspend}$
$v^{\text{source}}_{dep1,install}$
$v^{\text{enter}}_{configured}$
$v^{\text{sink}}_{dep1,update}$
v^{fire}_{sp1}
v^{fire}_{sp2}
$v^{\text{enter}}_{installed}$
$time(dep1, du1)$
$v^{\text{deact}}_{server,service1}$
v^{leave}_{s1}
v^{leave}_{s2}
v^{fire}_{du1}
v^{enter}_{s1}
v^{enter}_{s2}
$v^{\text{leave}}_{running}$
$v^{\text{deact}}_{dep1,service}$
$time(server, ss1)$
$time(server, ss2)$
v^{fire}_{ss1}
v^{fire}_{ss2}
$v^{\text{enter}}_{running}$
$v^{\text{leave}}_{running}$
$v^{\text{source}}_{dep1,update}$
$v^{\text{enter}}_{running}$
$v^{\text{source}}_{server,suspend}$
v^{source}

Figure 6: Dependency graph corresponding to the reconfiguration in Listing 4 applied to the assembly in Figure 5.

particular, the following attributes must be initialized: `places`, `initial_place`, `groups`, `transitions` and `dependencies` (which correspond to ports). Each `Transition` contains an element `action`, which is expected to be a Python function, and will be called when the transition is executed, possibly with arguments. Listing 1 shows the declaration of the component type *Client* of Figure 2.

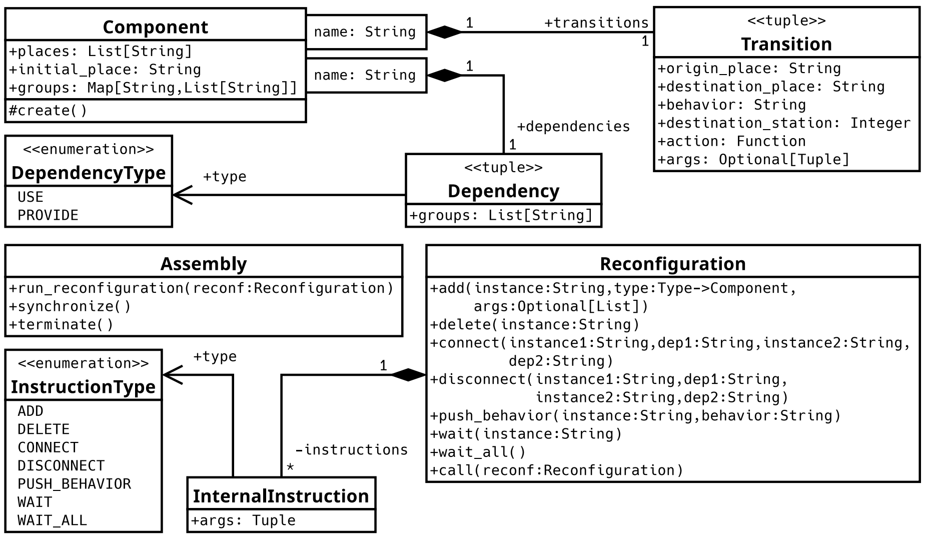


Figure 7: UML class diagram of our implementation of Concerto

The task of declaring a component type would typically be carried out by a component developer, or someone encapsulating existing code into a component for use in a Concerto assembly.

An instance of the class `Assembly` corresponds to an environment in which to execute reconfigurations. It keeps track of unique component identifiers and connections, and also manages the Python threads dedicated to executing reconfigurations. The user (typically, a system administrator) defines an assembly object and can then call its method `run_reconfiguration`. This asynchronously starts the reconfiguration in a new Python thread. The method `synchronize` stops the calling thread until the reconfiguration has been fully executed, then the method `terminate` can be used to destroy the thread.

An instance of the class `Reconfiguration` stores a list of reconfiguration instructions (instances of `InternalInstruction`). Instructions can be appended to the list with dedicated methods, one for each instruction. Note two additional instructions compared to the formal model: `wait_all` prevents further instructions to be executed until all existing instances have finished executing their behaviors, equivalent to a sequence of `wait` instructions, while `call` provides a way to compose reconfigurations: it takes a `Reconfiguration` object as argument and adds all its instructions to the internal list of the current object. Listings 2 and 3 give examples of reconfiguration programs.

When a reconfiguration is executed over an instance of `Assembly`, a new Python thread is started to perform the following actions in a loop:

1. try to apply the first instruction in the reconfiguration program (if successful, discard the instruction);
2. for each instance with at least one behavior in its queue:
    (a) check if any place has been reached,

(b) check if the final places of the current behavior have been reached, update the behavior accordingly,
(c) check transitions conditions, and if satisfied, start the corresponding action in a new thread,
(d) check if any of the previously started actions has terminated.

When a transition is fired, the action (i.e., Python function) corresponding to the transition is called in its own thread. Note that Python threads do not take advantage of hardware parallelism capabilities, but because the actions usually run other (possibly remote) processes to do the heavy work, this is not an issue. For instance, in the experiments presented further, actions initiate `ssh` connections to apply commands to remote machines, hence starting new processes. The termination of the action function corresponds to the end of the transition in the semantic model.

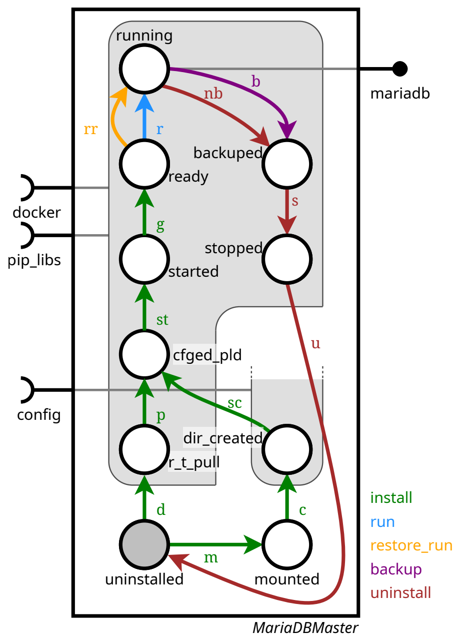
6.2. Real use case: database reconfigurations

We now describe two reconfigurations of a database that was presented in the context of a multi-region deployment of OpenStack at the 2018 Vancouver OpenStack summit[6]. OpenStack is a software solution to operate private clouds. It acts as an operating system for a cloud service and manages its infrastructure. In a multi-region deployment of OpenStack, the infrastructure is split into regions, and some OpenStack control modules are replicated to make each region partially autonomous.

We focus on the database module used within OpenStack. In our scenario, we consider one initial configuration that corresponds to the initial deployment (*DeployInit*) and two reconfigurations: *Decentralization* and *Scaling*. During the initial deployment (*DeployInit*), dependencies are installed on all the hosts and a single database instance is deployed on one host called *initial host*. During the *Decentralization*, the database is reconfigured so that multiple hosts (each representing a different region) have a local instance of the database. The hosts other than the *initial host* are called *additional hosts*. These instances are configured as a Galera cluster in order to synchronize their content. During the *Scaling*, additional database instances are deployed on other *additional hosts*, effectively increasing the size of the cluster.

Modules. The main module is the database module *MariaDB*. We use a containerized version of the MariaDB software which can be booted in three modes. The *standalone* mode corresponds to a standard instance of MariaDB. The *cluster-init* mode makes it possible to initiate a new Galera cluster so that other instances of MariaDB can join it. Finally, the *cluster-join* mode allows the instance to join an already existing Galera cluster. The lifecycle of the MariaDB module is as follows. To deploy MariaDB, a dedicated directory must be mounted, which will contain configuration files and the data stored by the

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



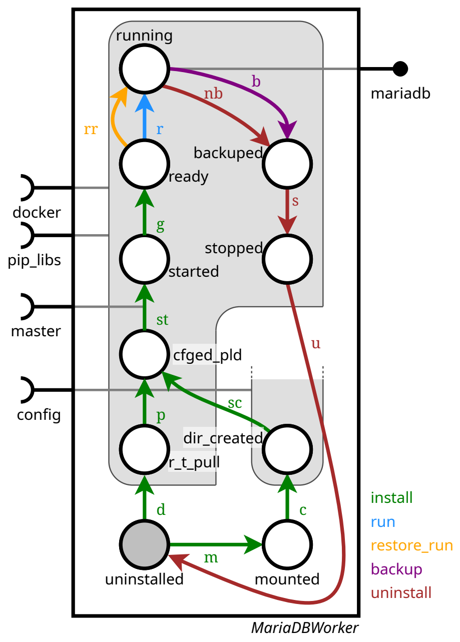


Figure 8: Two possible Concerto component type for MariaDB, one for the *initial host* (Master) and one for the *additional hosts* (Worker). The only difference is the presence of a *master* use port in the Worker implementation, allowing for proper coordination when joining a Galera cluster.

database. The configuration files, which state among other things in which mode MariaDB will be run, must be placed at the appropriate location. In parallel, the MariaDB docker image may be downloaded (once Docker is available on the host). Then, the Docker image may be started. Once the database service is operational, a previously created backup present on the host may be restored to populate the database. The database can then be used by other modules.

When running, the MariaDB service may be stopped, after possibly saving a backup of the content of the database on the host. The directories containing the configuration of MariaDB and its data can then be unmounted, effectively resetting the database.

Note that many other operations could be performed on this module, but these are the ones needed for the use case considered here. A database developer may build a much more complex and complete control component for MariaDB. Figure 8 shows a possible implementation of this module by two Concerto control component types, one to work in *standalone mode* or *cluster-init mode* (MariaDBMaster) and one to work in *cluster-join mode* (MariaDBWorker).

Other modules support the database, either directly or indirectly. Docker must be installed on every host running the database, as well as appropriate Python libraries. In turn, these require software packages to be installed through the package manager of the host OS. The lifecycle of these modules is entirely

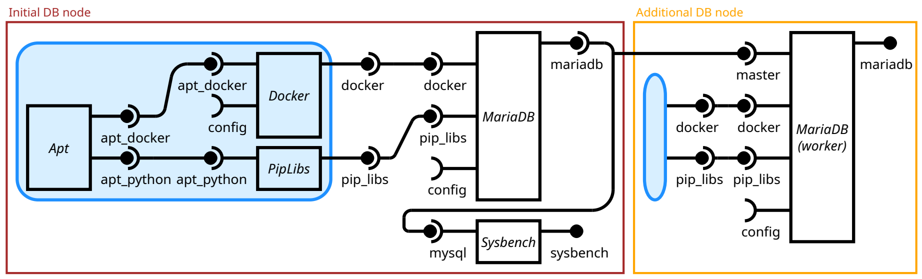


Figure 9: Overview of the Concerto assembly of a Galera distributed database with one initial and one additional host. The content of the blue rectangle represented for the initial host is hidden in the additional host for the sake of readability.

sequential. Finally, the `Sysbench` benchmarking software is used to act as a client of the distributed database in our scenario. It can be installed and run, and then suspended and restarted when necessary.

Figure 9 shows an overview of a Concerto assembly corresponding to the deployed state of two instances of a MariaDB database in a Galera cluster, one on the *initial host* and one on an additional host.

Reconfigurations. We consider the initial deployment *DeployInit* and the two reconfigurations *Decentralization* and *Scaling* to have two integer parameters n and m ($n \geq 2$, $m > n$), n and m being numbers of hosts on which instances of MariaDB are going to be deployed in the reconfigurations presented below.

The initial configuration, namely *DeployInit*, deploys the initial host with the database in *standalone mode*, by instantiating the component types *MariaDBMaster*, and *StandaloneConfig* (used to put MariaDB in the *standalone mode*). It also deploys Docker and the Python libraries on m additional hosts, emulating the fact that these hosts are already under usage in regions but do not have replicas of the database. Listing 5 gives a possible implementation in Concerto to reach this first configuration. This Concerto program is not part of the evaluation, as it only serves to reach the initial configuration in our scenario.

The first reconfiguration, namely *Decentralization*, starts from the initial configuration resulting from the execution of Listing 5. It replaces the standalone database by a distributed, or decentralized, database with $n+1$ instances (one on the initial host and one on each of the n additional hosts). The reconfiguration effectively performs a backup of the content of the database on the initial host, restarts the database container in *cluster-init mode* and restores the backup once the database is running. In addition, it deploys one instance of the database on each of the n additional hosts in *cluster-join mode*. Listing 6 gives a Concerto program for this reconfiguration. Component types *ClusterInitConfig* and *ClusterJoinConfig* provide configuration information to set MariaDB in *cluster-init mode* and *cluster-join mode*, respectively.

The second reconfiguration, namely *Scaling*, scales the distributed database up by increasing the number of additional hosts (from n to m). Listing 7 gives one possible Concerto program for this reconfiguration.

Listing 5: Deployment program.

```
add (m_mariadb, MariaDBMaster)
add (m_apt, Apt)
add (m_docker, Docker)
add (m_piplibs, PipLibs)
add (m_sysbench, Sysbench)
add (m_sconf, StandaloneConfig)
con (m_docker, apt_docker,
     m_apt, apt_docker)
con (m_piplibs, apt_python,
     m_apt, apt_python)
con (m_mariadb, config,
     m_sconf, data)
con (m_mariadb, docker,
     m_docker, docker)
con (m_mariadb, pip_libs,
     m_piplibs, pip_libs)
pushB (m_mariadb, install)
pushB (m_mariadb, run)
pushB (m_apt, install)
pushB (m_docker, install)
pushB (m_piplibs, install)
pushB (m_sysbench, install)
pushB (m_sconf, install)
for i in [1..m]:
  add (w{i}_apt, Apt)
  add (w{i}_docker, Docker)
  add (w{i}_piplibs, PipLibs)
  con (w{i}_docker, apt_docker,
       w{i}_apt, apt_docker)
  con (w{i}_piplibs, apt_python,
       w{i}_apt, apt_python)
  pushB (w{i}_apt, install)
  pushB (w{i}_docker, install)
  pushB (w{i}_piplibs, install)
```

Listing 6: Decentralization program.

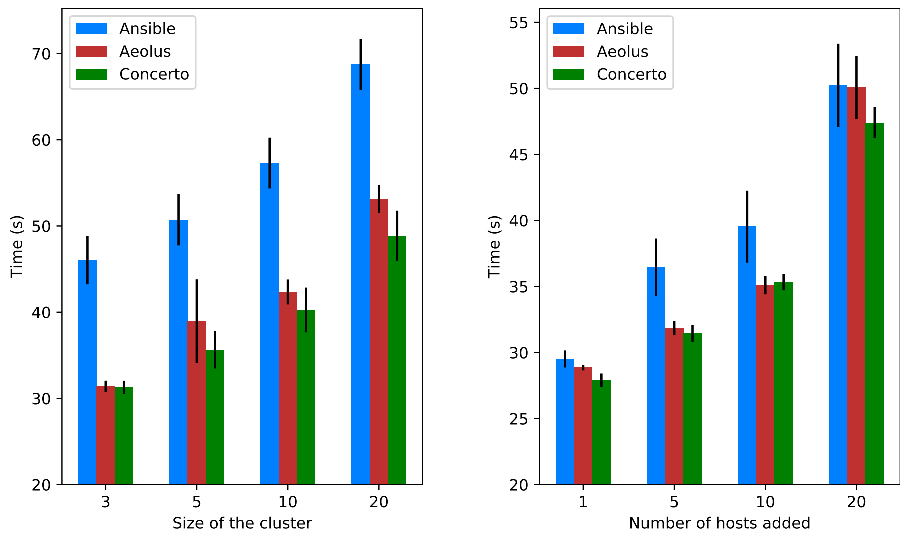
```
pushB (m_sysbench, suspend)
pushB (m_mariadb, backup)
pushB (m_mariadb, uninstall)
con (m_docker, apt_docker,
     m_apt, apt_docker)
add (w_gconf, ClusterJoinConfig)
for i in [1,n]:
  add (w{i}_mariadb, MariaDBWorker)
  con (w{i}_mariadb, config,
       w_gconf, data)
  con (w{i}_mariadb, docker,
       w{i}_docker, docker)
  con (w{i}_mariadb, pip_libs,
       w{i}_piplibs, pip_libs)
  pushB (w{i}_mariadb, install)
wait(m_mariadb)
dcon (m_mariadb, config,
      m_sconf, data)
del (m_sconf)
add (m_gconf, ClusterInitConfig)
con (m_mariadb, config,
     m_gconf, data)
pushB (m_mariadb, install)
pushB (m_mariadb, restore_run)
pushB (m_sysbench, install)
for i in [1,n]:
  con (w{i}_mariadb, master,
       m_mariadb, mariadb)
```

Listing 7: Scaling program.

```
for i in [n+1,m]:
  add (w{i}_mariadb, MariaDBWorker)
  con (w{i}_mariadb, config,
       w_gconf, data)
  con (w{i}_mariadb, docker,
       w{i}_docker, docker)
  con (w{i}_mariadb, pip_libs,
       w{i}_piplibs, pip_libs)
  con (w{i}_mariadb, master,
       m_mariadb, mariadb)
  pushB (w{i}_mariadb, install)
```

Executing the use case. We execute the two reconfigurations described above, using an implementation in Concerto and another in Ansible. A third implementation is meant to emulate Aeolus, as its code is no longer maintained. It also relies on Concerto, but the component types do not execute any transitions in parallel (i.e., they are sequential versions of the component types used in the main Concerto implementation). The Concerto and Aeolus implementations use SSH calls to execute Bash scripts on the remote hosts, while Ansible relies on corresponding built-in commands.

First, we execute the reconfiguration *Decentralization* from a MariaDB instance to a Galera cluster of size 3, 5, 10 or 20. Second, we execute the reconfiguration *Scaling* of a cluster of size 3, with the number of hosts added equal to 1, 5, 10 or 20.



(a) Decentralization (b) Scaling

Figure 10: Average running times for Ansible, Aeolus and Concerto of the *Decentralization* and *Scaling* reconfigurations (error bars: standard deviation).

Evaluations were carried out on the Uvb cluster of the experimental platform Grid'5000 (`www.grid5000.fr`). Uvb is composed of 43 hosts equipped with two 6-core Intel Xeon X5670 CPUs, 96 GB RAM, 250 GB HDD, an internal network interface with a transfer rate of 40 Gbps InfiniBand, and an external network interface with a transfer rate of 1 Gbps.

For each solution 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 provides no way to measure this for each individual host). Figure 10 shows the average running times.

The execution time of the Concerto implementation is slightly shorter than that of the Aeolus implementation. The gains with Concerto compared to Aeolus range from -0.6% (scaling, 10 hosts) to 8.5% (decentralization, 5 hosts), and from 5.4% (scaling, 1 host) to 32.1% (decentralization, 3 hosts) compared to Ansible. As expected, the gain compared to Ansible is even greater, especially in the decentralization scenario where it ranges from 28.9% (20 hosts) to 32.1% (3 hosts). In the scaling scenario, the gain ranges from 5.4% (1 host) to 13.7% (5 hosts).

This use case illustrates the applicability of Concerto in real life and its potential gains. We will show later in this section that the relatively low gain compared to Aeolus is predicted by our performance model, and that it results from the limited level of parallelism in this use case.

6.3. Synthetic use cases

We now introduce additional synthetic use cases. They will be used to validate the performance model of concerto against real executions, and to compare

Listing 8: Reconfiguration program (1).

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

Listing 9: Reconfiguration program (2).

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

Listing 10: 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 11: Reconfiguration program (4).

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

the theoretical performance of Ansible, Aeolus and Concerto.

We consider four reconfiguration use cases, each featuring different levels of parallelism. They are all based on a server with some requirements/dependencies, e.g., a database or the installation of a software library. Throughout this evaluation, the parameter n indicates the number of dependencies of the server. A server with n dependencies is modeled by the component type $Server_n$. The dependencies are modeled by the component types $Dependency_i$, where i is an identifier for the type of dependency component. In this use case, all component types for dependencies have the same lifecycle. Additionally, the transitions in their behavior *install* have the same transition actions, but one transition in their behavior *update* (transition du_i) may perform different actions depending on i. Figure 5 shows the assembly for $n = 2$.

The reconfigurations that we analyze are given in Listings 8 to 11. Note that in each of them, a loop is used to express parametric reconfiguration programs, but this can be unrolled as the loop parameter is known prior to execution. The first reconfiguration, namely *DeployDeps*, deploys n dependencies and features inter-component parallelism with identical reconfiguration actions on all components. The second, namely *UpdateNoServer*, updates these n dependencies and features inter-component parallelism with different reconfiguration actions across components. The third, namely *DeployServer*, deploys the server (with its dependencies already deployed) and features intra-component parallelism. Finally, the fourth, namely *UpdateWithServer*, updates the n dependencies, which requires suspending of the server. It features both inter-component and intra-component parallelism.

6.4. Validation of the performance model for Concerto

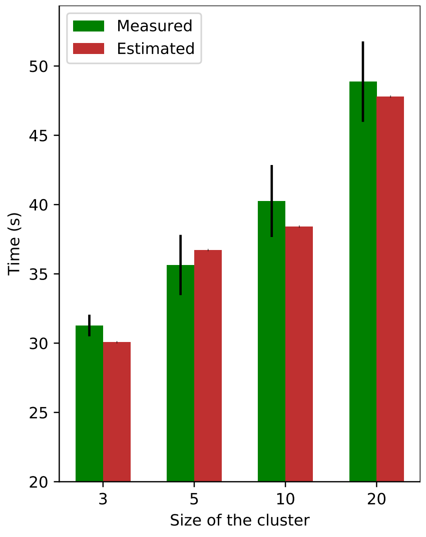
To validate the performance model of concerto we both use the synthetic use cases and the real use case.

Synthetic use cases. In order to ensure that the performance model matches the experimental results, the execution time of our four synthetic reconfigurations were measured using our Python implementation of Concerto. Each transition calls the Python function `time.sleep` to simulate the time required by an arbitrary reconfiguration action. Given a reconfiguration, we randomly selected a duration for each transition (continuous uniform distribution between 0 and 10 seconds), and compared the execution time of the implementation to the predicted time given by the performance model. The durations in this experiment do not need to be realistic, as the aim is to test the accuracy of the performance model in a variety of situations.

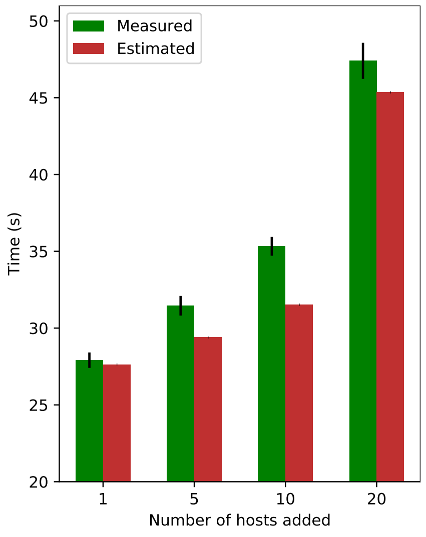
Nb. deps.	Med.	Avg.	Min.	Max d.(%)	Max d.(s)	(to min.)	(to avg.)
(1) *DeployDeps*							
1	10.09s	10.15s	0.37s	5.7% (of 0.37s)	0.04s (0.2%)	10.8%	0.4%
5	15.01s	14.84s	7.76s	0.4% (of 7.76s)	0.04s (0.2%)	0.5%	0.3%
10	16.16s	16.01s	10.94s	0.3% (of 10.94s)	0.05s (0.3%)	0.5%	0.3%
(2) *UpdateNoServer*							
1	10.41s	10.35s	0.67s	2.5% (of 0.67s)	0.04s (0.2%)	5.3%	0.3%
5	15.01s	14.84s	7.22s	0.3% (of 7.27s)	0.04s (0.2%)	0.5%	0.3%
10	16.52s	16.36s	10.46s	0.3% (of 10.46s)	0.04s (0.2%)	0.4%	0.2%
(3) *DeployServer*							
1	14.59s	14.58s	0.90s	2.5% (of 0.90s)	0.05s (0.2%)	5.4%	0.3%
5	18.34s	17.95s	8.49s	0.4% (of 8.49s)	0.05s (0.2%)	0.6%	0.3%
10	19.19s	19.13s	9.64s	0.3% (of 9.64s)	0.05s (0.2%)	0.5%	0.3%
(4) *UpdateWithServer*							
1	17.64s	17.49s	5.57s	0.5% (of 5.57s)	0.05s (0.2%)	0.9%	0.3%
5	21.96s	22.04s	11.57s	0.3% (of 11.57s)	0.05s (0.2%)	0.4%	0.2%
10	24.21s	23.89s	16.80s	0.2% (of 16.80s)	0.05s (0.2%)	0.3%	0.2%

Table 1: Summary of the results obtained with the implementation of Concerto on the synthetic use cases, with either 1, 5 or 10 dependencies. Each row of the table corresponds to 250 runs. For each, the median, average and minimum execution times over all the runs are given. Then, the maximum difference between the time predicted by the performance model and the actual measured time in both percentage and seconds are given. Finally, the maximum difference in seconds is compared to the minimum and average execution times.

We ran this experiment 250 times for each reconfiguration, for 1, 5 and 10 dependencies, for a total of 3000 executions. Table 1 summarizes the results obtained. The full results as well as the code to reproduce these experiments



(a) Decentralization



(b) Scaling

Figure 11: Average running times and estimation with the theoretical performance model (using average transition durations) of the *Decentralization* and *Scaling* reconfigurations (error bars: standard deviation).

are available online [7].

The difference between the estimated time and the measured time is at most 0.05 seconds, or 5.7% above the estimated time (in this instance the total execution time was 0.37 seconds, which explains the relatively high percentage). The measured time was always slightly longer than the estimated time, which is explained by the small overhead introduced by the Concerto implementation.

Real use case. Regarding the real use case for which the experimental results have been presented in Figure 10, the theoretical performance model has also been executed by using the average duration for each transition. Results given by the performance model are depicted in Figure 11. The error for the *Decentralization* ranges from 2.2% (1.1s, 20 nodes) to 4.6% (1.8s, 10 nodes). The error for the *Scaling* ranges from 1.0% (0.3s, 1 additional node) to 10.8% (3.8s, 10 additional nodes). This error is explained by the fact that taking the average durations of the transitions leads to underestimating the influence that a single transition can have on the total execution time. When using respectively the minimum and maximum durations instead, the measured execution time is always between the min/max estimations.

As a result, it appears that the experimental results previously obtained in Figure 10 can be precisely estimated by our performance model. The relatively modest performance improvement over Aeolus that was observed is correlated to the limited intra-component parallelism level of the use case.

[7]`https://gitlab.inria.fr/VeRDi-project/concerto-evaluation` (directory *synthetic*)

This validates our performance model for Concerto both on synthetic use cases and one real use case.

6.5. Theoretical performance analysis on synthetic use cases

Before presenting the theoretical performance comparison between Ansible, Aeolus and Concerto, we introduce the performance models for Ansible and Aeolus.

Performance model for Ansible. In the rest of this section, the performance of Concerto is compared theoretically to Ansible. First, we devise a theoretical performance model for Ansible and validate it empirically.

In Ansible, a reconfiguration consists in a sequence of tasks, each task being composed of a reconfiguration action and metadata, indicating in particular the hosts on which the action should be executed. Tasks are executed sequentially, but one task may execute the same action in parallel on multiple hosts. The task is complete only when the actions have terminated on all hosts, thus introducing a synchronization barrier before the next task can be executed. Therefore, given a sequence of reconfiguration tasks $t_1, t_2, \ldots, t_n$, where t_i executes action a_i on a set of hosts H_i, the total reconfiguration time is

$$\sum_{i=1}^{n} \max_{h \in H_i} \left(d\left(h, a_i\right) \right)$$

where $d\left(h, a_i\right)$ is the duration of action a_i on host h.

To validate this model, we executed an Ansible reconfiguration composed of two tasks, each executing the Bash command `sleep` for a randomly determined time between 0 and 10 seconds (with a potentially different time across hosts). The facts gathering feature of Ansible was disabled to minimize overhead during the execution.

We performed this experiment 1500 times both for $n = 2$ and $n = 5$. For $n = 2$, the difference between predicted and measured execution time ranged from 0.8s to 5.1s with a mean of 1.0s and a median of 1.0s. Note that only two measurements had a difference of more than 1.3s. For $n = 5$ it ranged from 0.9s to 7.2s with a mean of 1.2s and a median of 1.2s also. Note that only two measurments had a difference of more than 1.6s. In these experiments, a small overhead is observed when executing Ansible.

This shows that our execution and performance modeling for Ansible matches reality, and is even somewhat optimistic as it does not take into account the slight overhead. The code to reproduce these experiments is available online[8].

Performance model for Aeolus. Aeolus is no longer under active development, and we could not run it in our experiments. However, its execution model is similar to Concerto, except that transitions cannot be executed in parallel

[8]`https://gitlab.inria.fr/VeRDi-project/concerto-evaluation` (directory *ansibleperf*)

Framework	Formula
(1) ***DeployDep***	
Ansible	$\max_i \{d_{di_i}\} + \max_i \{d_{dr_i}\}$
Aeolus	$\max_i\{d_{di_i} + d_{dr_i}\}$
Concerto	$\max_i\{d_{di_i} + d_{dr_i}\}$
(2) ***UpdateNoServer***	
Ansible	$\sum_i \{d_{du_i}\} + \max_i \{d_{dr_i}\}$
Aeolus	$\max_i\{d_{du_i} + d_{dr_i}\}$
Concerto	$\max_i\{d_{du_i} + d_{dr_i}\}$
(3) ***DeployServer***	
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) ***UpdateWithServer***	
Ansible	$\sum_i \{d_{du_i} + d_{ss_i} + d_{sp_i}\} + \max_i \{d_{dr_i}\} + d_{sr}$
Aeolus	$\max\left(\max_i \left\{d_{du_i} + \sum_{j\leq i} \{d_{ss_j}\} + d_{dr_i}\right\},\ d_{sr} + \sum_i\{d_{ss_i} + d_{sp_i}\}\right)$
Concerto	$\max(\max_i\{d_{du_i} + d_{ss_i} + d_{dr_i}\},\ d_{sr} + \max_i\{d_{ss_i} + d_{sp_i}\})$

Table 2: Theoretical formulas estimating the run-time of each reconfiguration by Ansible, Aeolus and Concerto.

inside a component. For this reason, we emulate the execution of Aeolus by replacing the Concerto component types with versions that do not have parallel transitions (i.e., we sequentially order the reconfiguration actions). Thus, we use the performance model of Concerto to estimate the performance of Aeolus.

Theoretical comparison. We now make use of synthetic use cases and performance models to compare the expected performance of Concerto to those of Aeolus and Ansible. Results are given in Table 2.

(1) ***DeployDeps*****:** We consider the deployment of n instances of n component types $Dependency_1$, ..., $Dependency_n$. The reconfiguration is given in Listing 8 and the formulas given by the performance models of Concerto, Aeolus and Ansible are given in Table 2 (1). This use case only considers the behavior `install`. As all component types $Dependency_i$ are supposed to perform the same actions within this behavior (denoted with a star on Figure 5), Ansible is able to perform the same action simultaneously on all instances. As a result, the transitions di_i can be executed in parallel across the instances dep_i, then the transitions dr_i. The formulas are identical for Aeolus and Concerto, because there are no parallel transitions in the components, i.e., there is only inter-component parallelism. The gain for Concerto and Aeolus depends on the difference of duration for similar transitions across instances. Figure 12 illustrates what happens when this difference is large, which may happen when

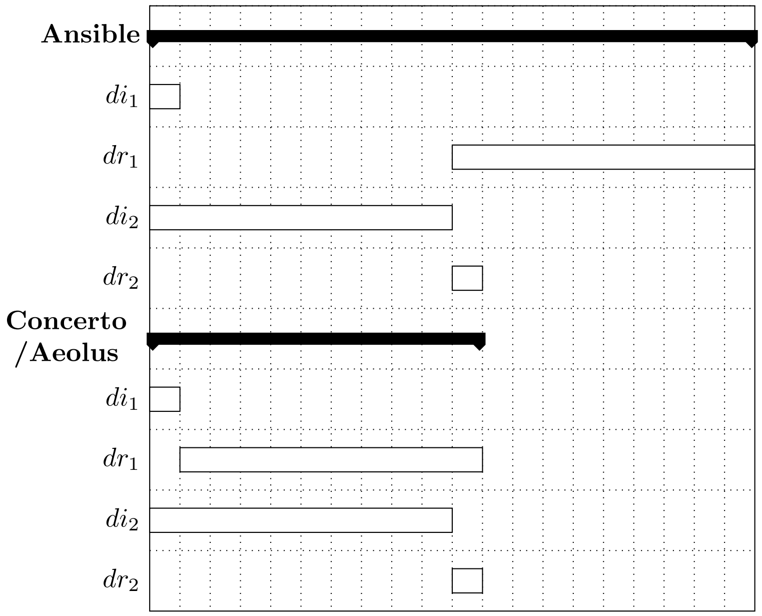


Figure 12: Gantt chart representing the difference in parallelism between Ansible on the one hand and Concerto and Aeolus in the other hand.

hosts have different hardware, different network bandwidth, etc. For instance, with $d_{di_1} = 5$, $d_{di_2} = 50$, $d_{dr_1} = 50$, $d_{dr_2} = 5$, the execution time for Ansible is $50 + 50 = 100$, whereas it is $\max(55, 55) = 55$ for Aeolus and Concerto. However, when running the same actions on similar hardware, we expect a normal distribution $\mathcal{N}(\mu, \sigma^2)$ (of mean μ and standard deviation σ) for the durations of the transitions di_i and dr_i respectively. Table 3 shows the distribution of the total execution time for different values of the number of dependencies n and σ. Without loss of generality, the mean duration for each transition is set to $\mu = 60$. With a standard deviation $\sigma = 10$ and $n = 2$ dependencies, there is a small difference in mean execution time of 3.3 seconds between Ansible and Concerto/Aeolus. For $n = 2000$ however, Concerto/Aeolus is 20.1 seconds faster. This shows that inter-component parallelism helps the scalability of Concerto and Aeolus on bigger assemblies. If we choose $\sigma = 20$, the difference in mean execution time is 6.7 seconds for $n = 2$ and 40.2 seconds for $n = 2000$, twice as much as a similar case with $\sigma = 10$. With $\sigma = 100$, the difference in mean execution time is 32.3 for $n = 2$ and 201.2 seconds for $n = 2000$, a tenfold increase compared to the case with $\sigma = 10$. Thus, the absolute gain in time of Concerto and Aeolus compared to Ansible seems to be proportional to the standard deviation of transitions durations σ.

(2) *UpdateNoServer*: Given n dependencies in place `running`, we consider the update of these dependencies. The reconfiguration is given in Listing 9 and the formulas are given in Table 2 (2). Unlike the previous case, the transi-

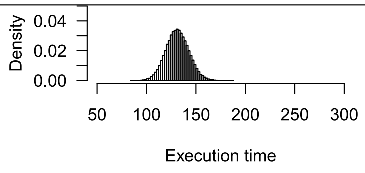
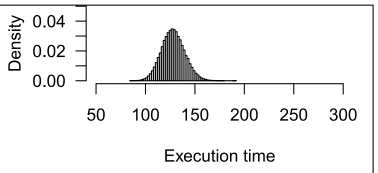
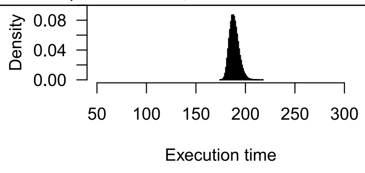
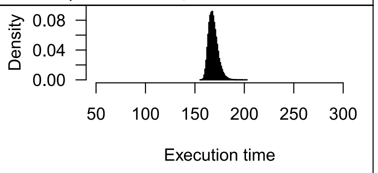
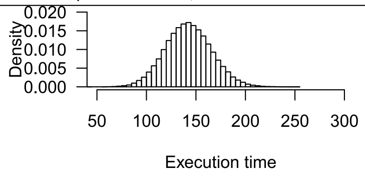
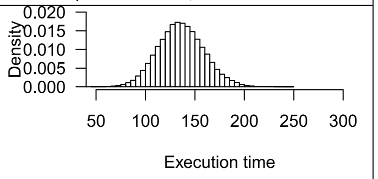
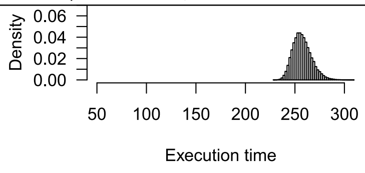
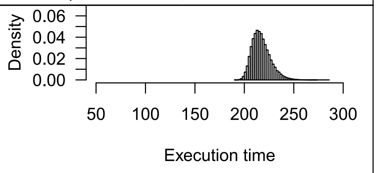
σ	n	Ansible	Concerto/Aeolus
10	2	 $\mu = 131.3$, $\sigma = 11.7$	 $\mu = 128.0$, $\sigma = 11.6$
	2000	 $\mu = 188.7$, $\sigma = 4.7$	 $\mu = 168.6$, $\sigma = 4.7$
20	2	 $\mu = 142.6$, $\sigma = 23.4$	 $\mu = 135.9$, $\sigma = 23.3$
	2000	 $\mu = 257.4$, $\sigma = 9.4$	 $\mu = 217.2$, $\sigma = 9.5$
100	2	$\mu = 232.2$, $\sigma = 116.6$	$\mu = 199.9$, $\sigma = 116.7$
	2000	$\mu = 807.0$, $\sigma = 47.3$	$\mu = 605.8$, $\sigma = 47.4$

Table 3: Distributions of the total execution time when the transitions follow a normal distribution $\mathcal{N}(\mu, \sigma^2)$ depending on the number of dependencies n and σ (with $\mu = 60$). Each histogram was obtained by simulation over 100,000 samples. Histograms were omitted for $\sigma = 100$ for the sake of readability.

tions du_i of the behavior `update` are not assumed to be the same among the dependencies (different types of components), therefore Ansible cannot execute them in parallel (transitions di, as before, can be executed in parallel). In this case, assuming that $max_i\{d_{dr_i}\}$ 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 improved scalability resulting from inter-component parallelism.

(3) *DeployServer*: Given n dependencies in place `running`, we consider the deployment of an instance of *Server* that uses these dependencies. The reconfiguration is given in Listing 10 and the formulas are given in Table 2 (3). Here, the formulas are similar for Ansible and Aeolus, as the transitions sc_i cannot be performed in parallel by Ansible (because they are different), nor by Aeolus (because they are part of the same component). In this case, assuming that $d_{sa}+d_{sr}$ 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.

(4) *UpdateWithServer*: Given n dependencies and a server in place `running`, we consider the update of all the dependencies, which requires the suspension of the server. The reconfiguration is given in Listing 11 and the formulas are given in Table 2 (4). In Ansible, the transitions are executed sequentially, except for the transitions dr_i that can be executed in parallel (same action for all component types in the behavior `install`). In Aeolus, thanks to inter-component parallelism, the reconfiguration time is the longest time required by any component to execute 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 required before the port `service` may be deactivated. There is no intra-component parallelism in Aeolus, so the transitions ss_i execute sequentially. Therefore, the time required before use port `service_i` is no longer in use is $\sum_{j\leq i}\{d_{ss_j}\}$, which is also the time required before the provide port `service` can be deactivated. The second part of the outer *max* corresponds to the execution time for the server in which all the transitions have been sequentially ordered, i.e., the sum of durations of all transitions.

In Concerto, the transitions ss_i and sp_i can be executed in parallel. Compared to Aeolus, this significantly decreases the time required before the dependencies may leave the place `running` (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. For instance, 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, the time would increase to 1510 seconds for Ansible and 1005 seconds for Aeolus, while Concerto would still take only 15 seconds.

In this last section, after validating Concerto on a real use case and performance models on both synthetic and real use cases, we have shown the gains theoretically possible thanks to Concerto, compared to both Ansible and Aeolus, according to the different levels of parallelism. As expected, gains compared to Ansible are substantial in almost all cases (both inter-component parallelism and intra-component parallelism), while gains compared to Aeolus are conditioned by the presence of intra-component parallelism within control components. These results also illustrate that Concerto outperforms other solutions even on a relatively small scale. A large-scale study is left to future work.

7. Conclusion

In this paper, we have presented Concerto, a formal model for the reconfiguration of component-based applications, with an emphasis on efficiency through parallel execution of reconfiguration tasks. We have formally presented the semantics of Concerto and described a performance estimation model for its reconfiguration programs. Further, we have empirically evaluated the accuracy of this performance model and used it to measure the performance advantage offered by Concerto over existing management tools and academic models.

For further work, we plan on facilitating the writing of reconfiguration programs. Currently, to use and reconfigure a component, one must know about its ports but also its internals. In particular, in order to avoid deadlocks, the dependency relations between use and provide ports must be determined, for each behavior. However the internal net of a component exposes more information than what is strictly required for this task, as it also shows the internal parallelism. It is possible to infer, from the internal net of a component, a summary of port and behavior relations, for example in the form of a transition system. This information could be used by system administrators or by autonomic reconfiguration tools to build reconfiguration programs.

We also plan to simplify the design of control components by leveraging common lifecycle patterns. For instance we have noticed that container-based lifecycle have a common basis that could be directly offered to the user through a programming skeleton. This is probably true for other kinds of components.

Having a reconfiguration model with well-defined semantics, we wish to develop formal methods for the verification of reconfiguration programs. Progress (absence of deadlock) and preservation of invariants are among the most relevant properties to verify automatically. In addition to the correctness properties, we are also interested in the analysis of quantitative properties that distinguish functionally equivalent reconfigurations. Notably, the degree of parallelism and the execution time of a reconfiguration are useful metrics that can guide component developers as well as system administrators. Model-checking is a suitable candidate for this task, as demonstrated by verification efforts on Madeus [60], a model for deployment that shares similarities with Concerto.

Lastly, we wish to integrate Concerto in an autonomic reconfiguration loop. In the context of the MAPE-K model, Concerto offers a basis for knowledge representation and takes care of the execution step. The remaining steps to close the autonomic loop are to track the evolution of Concerto applications during their lifespan (monitoring), use that information to determine target configurations (analysis) and synthesize reconfiguration programs to reach those targets (planning).

References

[1] P. Hu, S. Dhelim, H. Ning, T. Qiu, Survey on fog computing, J. Netw. Comput. Appl. 98 (C) (2017) 27–42.

[2] M. Iorga, L. Feldman, R. Barton, M. J. Martin, N. S. Goren, C. Mahmoudi, Fog Computing Conceptual Model, Tech. rep., NIST (march 2018).

[3] R. Mahmud, R. Kotagiri, R. Buyya, Fog Computing: A Taxonomy, Survey and Future Directions, in: Internet of Everything, Springer, 2018, pp. 103–130.

[4] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, P. A. Polakos, A comprehensive survey on fog computing: State-of-the-art and research challenges, IEEE Communications Surveys & Tutorials 20 (1) (2017) 416–464.

[5] A. V. Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh, R. Buyya, Fog computing: Principles, architectures, and applications, in: Internet of Things, Elsevier, 2016, pp. 61–75.

[6] O. Nierstrasz, M. Denker, L. Renggli, Model-Centric, Context-Aware Software Adaptation, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 128–145.

[7] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D. B. Smith, J. P. Sousa, L. Tahvildari, K. Wong, J. Wuttke, Software Engineering for Self-Adaptive Systems: A Second Research Roadmap, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 1–32.

[8] Z. Wan, P. Hudak, Functional reactive programming from first principles, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00, Association for Computing Machinery, New York, NY, USA, 2000, p. 242–252. `doi:10.1145/349299.349331`.
URL `https://doi.org/10.1145/349299.349331`

[9] J. Buisson, F. Dagnat, E. Leroux, S. Martinez, Safe reconfiguration of Coqcots and Pycots components, Journal of Systems and Software 122 (2016) 430–444.

[10] F. Boyer, O. Gruber, D. Pous, A robust reconfiguration protocol for the dynamic update of component-based software systems, Software: Practice and Experience 47 (11) (2017) 1729–1753. `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2499`.

[11] F. Durán, G. Salaün, Robust and reliable reconfiguration of cloud applications, Journal of Systems and Software 122 (2016) 524–537.

[12] N. Gaspar, L. Henrio, E. Madelaine, Formally reasoning on a reconfigurable component-based system – A case study for the industrial world, in: Formal Aspects of Component Software (FACS), Vol. 8348 of LNCS, Springer, 2013, pp. 137–156.

[13] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.

[14] M. Litoiu, M. Shaw, G. Tamura, N. M. Villegas, H. Müller, H. Giese, R. Rouvoy, E. Rutten, What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems?, in: Software Engineering for Self-Adaptive Systems 3: Assurances, Vol. 9640 of LNCS, Springer, 2017.

[15] Ansible, `https://www.ansible.com/`.

[16] chef, `https://www.chef.io/`.

[17] Puppet, `https://puppet.com/`.

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

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

[20] M. Chardet, H. Coullon, D. Pertin, 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, 2018, pp. 1–8.

[21] M. Chardet, H. Coullon, C. Perez, Predictable efficiency for reconfiguration of service-oriented systems with concerto, in: Proceedings 20Th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, IEE, To appear.

[22] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, C. Becker, A survey on engineering approaches for self-adaptive systems, Pervasive and Mobile Computing 17 (2015) 184 – 206, 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian. `doi:https://doi.org/10.1016/j.pmcj.2014.09.009`.
URL `http://www.sciencedirect.com/science/article/pii/S157411921400162X`

[23] P. Oreizy, N. Medvidovic, R. N. Taylor, Architecture-based runtime software evolution, in: Proceedings of the 20th International Conference on Software Engineering, 1998, pp. 177–186. `doi:10.1109/ICSE.1998.671114`.

[24] F. Baude, D. Caromel, C. Dalmasso, M. Danelutto, V. Getov, L. Henrio, C. Pérez, GCM: a grid extension to fractal for autonomous distributed components, Annals of telecommunications 64 (1-2) (2009).

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

[26] J. Bigot, Z. Hou, C. Pérez, V. Pichon, A low level component model easing performance portability of hpc applications, Computing 96 (12) (2014) 1115–1130. `doi:10.1007/s00607-013-0368-3`.
URL `http://dx.doi.org/10.1007/s00607-013-0368-3`

[27] Object Management Group, CORBA Component Model (Apr. 2006).
URL `https://www.omg.org/spec/CCM/4.0/PDF`

[28] Object Management Group, Deployment and configuration of component-based distributed applications (Apr. 2006).
URL `https://www.omg.org/spec/DEPL/4.0/PDF`

[29] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, T. Dahlgren, The cca component model for high-performance scientific computing, Concurrency and Computation: Practice and Experience 18 (2) (2006) 215–229. `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.911`, `doi:10.1002/cpe.911`.
URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.911`

[30] H.-y. Paik, A. L. Lemos, M. C. Barukh, B. Benatallah, A. Natarajan, Service Component Architecture (SCA), Springer International Publishing, Cham, 2017, pp. 203–250. `doi:10.1007/978-3-319-55542-3_8`.
URL `https://doi.org/10.1007/978-3-319-55542-3_8`

[31] A. Basu, M. Bozga, J. Sifakis, Modeling heterogeneous real-time components in bip, in: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06), 2006, pp. 3–12. `doi:10.1109/SEFM.2006.27`.

[32] G. Blair, T. Coupaye, J.-B. Stefani, Component-based architecture: the Fractal initiative, Annals of telecommunications 64 (Feb 2009).

[33] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The fractal component model and its support in java, Software: Practice and Experience 36 (11-12) (2006) 1257–1284. `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.767`, `doi:10.1002/spe.767`.
URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.767`

[34] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, J.-B. Stefani, A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures, Software: Practice and Experience 42 (5) (2012) 559–583. `doi:10.1002/spe.1077`.
URL `https://hal.inria.fr/inria-00567442`

[35] P.-C. David, T. Ledoux, T. Coupaye, M. Léger, FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures, Annals of Telecommunications - annales des télécommunications Volume 64 (Numbers 1-2 / février 2009) (2008) 45–63.
URL `https://hal.archives-ouvertes.fr/hal-00468474`

[36] L. Henrio, M. Rivera, Stopping Safely Hierarchical Distributed Components: Application to GCM, in: Proc. of the CompFrame/HPC-GECO Workshop on Component Based High Performance, CBHPC, ACM, NY, USA, 2008, pp. 8:1–8:11.

[37] S. Bouchenak, N. D. Palma, D. Hagimont, C. Taton, Autonomic management of clustered applications, in: 2006 IEEE Intl Conference on Cluster Computing, 2006, pp. 1–11. `doi:10.1109/CLUSTR.2006.311842`.

[38] L. Broto, D. Hagimont, P. Stolf, N. Depalma, S. Temate, Autonomic management policy specification in tune, in: Proceedings of the 2008 ACM Symposium on Applied Computing, SAC '08, ACM, New York, NY, USA, 2008, pp. 1658–1663. `doi:10.1145/1363686.1364080`.

[39] P.-C. David, T. Ledoux, An aspect-oriented approach for developing self-adaptive fractal components, in: W. Löwe, M. Südholt (Eds.), Software Composition, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 82–97.

[40] V. Lanore, C. Pérez, A reconfigurable component model for HPC, in: CBSE 2015, ACM, Montréal, Canada, 2015, p. 10.
URL `https://hal.inria.fr/hal-01142606`

[41] V. Lanore, C. Perez, A reconfigurable component model for hpc, in: 2015 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE), 2015, pp. 1–10. `doi:10.1145/2737166.2737169`.

[42] R. E. Ballouli, S. Bensalem, M. Bozga, J. Sifakis, Programming Dynamic Reconfigurable Systems, in: Formal Aspects of Component Software - 15th International Conference, FACS 2018, Pohang, South Korea, 2018.
URL `https://hal.archives-ouvertes.fr/hal-01888550`

[43] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, P. Toft, The smartfrog configuration management framework, SIGOPS Oper. Syst. Rev. 43 (1) (2009) 16–25. `doi:10.1145/1496909.1496915`.
URL `https://doi.org/10.1145/1496909.1496915`

[44] R. Di Cosmo, J. Mauro, S. Zacchiroli, G. Zavattaro, Aeolus: a component model for the Cloud, Information and Computation (2014) 100–121.

[45] A. Brogi, A. Canciani, J. Soldani, Fault-aware application management protocols, in: Lecture Notes in Computer Science (including

subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 9846 LNCS, Springer Verlag, 2016, pp. 219–234. doi:10.1007/978-3-319-44482-6_14.
URL https://link.springer.com/chapter/10.1007/978-3-319-44482-6_14

[46] A. Brogi, A. Canciani, J. Soldani, Modelling the dynamic reconfiguration of application topologies, faults included, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 10319 LNCS, Springer Verlag, 2017, pp. 178–196. doi:10.1007/978-3-319-59746-1_10.
URL https://link.springer.com/chapter/10.1007/978-3-319-59746-1_10

[47] A. Brogi, A. Canciani, J. Soldani, Fault-aware management protocols for multi-component applications, Journal of Systems and Software 139 (2018) 189–210. doi:10.1016/j.jss.2018.02.005.

[48] Saltstack, https://www.saltstack.com/.

[49] Kubernetes, http://kubernetes.io/.

[50] Docker Swarm, https://docs.docker.com/engine/swarm/.

[51] A. Edmonds, T. Metsch, A. Papaspyrou, A. Richardson, Toward an open cloud standard, IEEE Internet Computing 16 (4) (2012) 15–25.

[52] 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).

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

[54] Apache Brooklyn, https://brooklyn.apache.org/ (2017).

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

[56] F. Korte., S. Challita., F. Zalila., P. Merle., J. Grabowski., Model-driven configuration management of cloud applications with occi, in: Proceedings of the 8th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER,, INSTICC, SciTePress, 2018, pp. 100–111. doi:10.5220/0006693001000111.

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

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

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

[60] H. Coullon, C. Jard, D. Lime, Integrated model-checking for the design of safe and efficient distributed software commissioning, in: International Conference on Integrated Formal Methods, Springer, 2019, pp. 120–137.