# Reconciling Parallelism Expressivity and Separation of Concerns in Reconfiguration of Distributed Systems

Maverick Chardet

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# THESE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPERIEURE MINES-TELECOM ATLANTIQUE
BRETAGNE PAYS DE LA LOIRE - IMT ATLANTIQUE
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
*Mathématiques et Sciences et Technologies de l'Information et de la Communication*
Spécialité : *Informatique*

Par

**Maverick CHARDET**

# Reconciling Parallelism Expressivity and Separation of Concerns in Reconfiguration of Distributed Systems

**Thèse présentée et soutenue à Nantes, le 03 décembre 2020**
**Unité de recherche : LS2N**
**Thèse N° : 2020IMTA0194**

**Rapporteurs avant soutenance** :

| | |
|---|---|
| Daniel Hagimont | Professeur (HDR), INP Toulouse / ENSEEIHT |
| Eric Rutten | Chargé de recherche Hors Classe (HDR), Inria |

**Composition du Jury :**

| | | |
|---|---|---|
| Présidente : | Laurence Duchien | Professeure (HDR), Université de Lille |
| Examinateurs : | Françoise Baude | Professeure (HDR), Université Côte d'Azur |
| | Daniel Hagimont | Professeur (HDR), INP Toulouse / ENSEEIHT |
| | Eric Rutten | Chargé de recherche Hors Classe (HDR), Inria |
| Directeur de thèse : | Christian Perez | Directeur de recherche (HDR), Inria |
| Co-encadrants de thèse : | Hélène Coullon | Maîtresse assistante, IMT Atlantique |
| | Adrien Lebre | Professeur (HDR), IMT Atlantique |

# Résumé

Les systèmes informatiques distribués, qui fonctionnent sur plusieurs ordinateurs, sont désormais courants et même utilisés dans des services critiques, tels que les hôpitaux, le contrôle aérien ou ferroviaire, les réseaux de télécommunication et désormais les voitures autonomes. Cependant, ces systèmes deviennent de plus en plus complexes, en terme d'échelle, de dynamicité et de qualité de service attendue.

La reconfiguration de systèmes distribués consiste à modifier leur état durant leur exécution. Les systèmes distribués peuvent être reconfigurés pour plusieurs raison, parmi lesquelles la mise à jour de certaines de leurs parties, leur adaptation pour obéir à de nouvelles contraintes (en termes de capacité utilisateurs, d'efficacité énergétique, de fiabilité, de coûts, etc.) ou même changer leurs fonctionnalités. Du fait de la complexité croissante de ces systèmes, de plus en plus d'acteurs interagissent avec eux (développeurs spécifiques à des parties précises du système, architectes de logiciels distribués, développeurs de reconfigurations, gérants d'infrastructures, etc.). Ainsi, il y a un besoin d'outils d'automatisation pour aider ces acteurs à déployer et reconfigurer les systèmes distribués.

Lors de la reconfiguration d'un système, les changements à effectuer (le plan de reconfiguration) doivent être spécifiés, que ce soit par un développeur humain ou par un outil d'automatisation, et ensuite exécutés. Cependant, les systèmes distribués sont composés de différents modules qui interagissent entre eux, et ces interactions diffèrent en fonction de l'état de ces modules. Cela induit des contraintes sur l'ordre dans lequel les actions de reconfiguration qui composent le plan de reconfiguration doivent être exécutées.

L'objectif de cette thèse est de fournir un système pour que des développeurs ou des outils puissent exprimer des plans de reconfiguration efficaces pour les systèmes distribués tout en prenant en compte les interactions variables entre leurs modules. Il est attendu qu'il respecte les compétences de chaque type d'acteur, sans pour autant faire de compromis sur l'efficacité. De plus, il est défini formellement, ce qui rend possible le fait de fournir des garanties à ses utilisateurs telles que la terminaison d'un plan de reconfiguration ou la préservation d'un invariant pendant la reconfiguration, ce qui augmente la sûreté de la reconfiguration.

Les contributions suivantes sont faites dans ce travail. Premièrement, un modèle de déploiement formel, Madeus (le déploiement est un type spécifique de reconfiguration qui consiste à rendre un service opérationnel à partir de rien). Deuxièmement, Concerto, un modèle formel qui étend Madeus pour supporter la reconfiguration de manière générale. Troisièmement, un modèle de performance, à la fois pour Madeus et Concerto, ce qui rend possible d'estimer les performances attendues d'un plan de reconfiguration. Quatrièmement, une implantation des deux modèles en Python. Enfin, une évaluation théorique et expérimentale de ces modèles est présentée en termes de performance, de séparation des préoccupations, de sûreté et de précision des modèles de performance.

# Abstract

Distributed computer systems, which run on multiple computers, are now commonplace and used even in critical systems, such as hospitals, train or air traffic control, telecommunication networks and now autonomous cars. However, these systems are becoming more and more complex, in terms of scale, dynamicity and expected quality of service.

Reconfiguration of distributed systems consists in changing their state at runtime. Distributed systems may be reconfigured for many reasons, including updating some of their parts, adapting them to fulfill new requirements (in terms of user capacity, energy efficiency, reliability, costs, etc.) or even changing their capabilities. Because of these systems' growing complexity, more and more actors interact with them (developers specific to given parts of the system, distributed software architects, reconfiguration developers, infrastructure managers, etc.). Therefore, there is a need for automated tools to assist these actors in deploying and reconfiguring distributed systems.

When reconfiguring a system, the changes to be made (the reconfiguration plan) need to be specified, either by a human developer or by an automated tool, and then executed. However, distributed systems are composed of multiple modules which interact with each other, and these interactions differ depending on the state of each module. This induces constraints on the order in which the reconfiguration actions that compose the reconfiguration plan have to be executed.

The focus of this thesis is to provide a framework for developers or tools to express efficient reconfiguration plans for distributed systems while taking into account the varying interactions between their modules. It is intended to respect the skills of each kind of actor, while not compromising on the efficiency. Additionally, it is defined formally, making it possible to provide guarantees to its users such as the termination of a reconfiguration plan or the preservation of an invariant during reconfiguration, which increases the safety of reconfiguration.

The following contributions are made in this work. First, a formal deployment model, Madeus (deployment is a specific type of reconfiguration which consists in setting the service up and running from scratch). Second, Concerto, a formal model which extends Madeus to support general reconfiguration. Third, a performance models for both Madeus and Concerto to be able to estimate the expected performance of a reconfiguration plan. Fourth, an implementation of both models in Python. Finally, theoretical and experimental evaluations of these models are presented in terms of performance, separation of concerns, safety and precision of the performance models.

# Remerciements (acknowledgements)

J'aimerais tout d'abord remercier les membres de mon jury, à qui j'ai eu l'honneur de présenter mes travaux. Tout d'abord, je remercie Mme Laurence Duchien qui a présidé ce jury. Ensuite, je remercie MM. Daniel Hagimont et Eric Rutten, rapporteurs, qui m'ont fait des retours extrêmement pertinents. Je remercie Mme Françoise Baude pour sa participation à ce jury. Je les remercie tous d'avoir accepté d'évaluer mes travaux, ainsi que pour leurs questions très intéressantes et leurs encouragements pour la suite. Il me reste à remercier mes encadrants, mais j'y reviendrai.

Mes trois années à Nantes ont été incroyablement enrichissantes. Cependant, tout le monde sait que les trois années de thèse ne sont pas un fleuve tranquille, et ces années auraient été bien différentes sans les incroyables membres de l'équipe Stack avec qui j'ai partagé travail mais aussi moments de détente, que ce soit autour d'un repas, d'un café (mais pas pour moi), d'un jeu de société ou sur un canapé. Merci d'abord à Émile, ou plutôt dankon pour le dire en espéranto, pour toutes ces discussions dans le bureau et en dehors, ainsi que pour ta présence alors que nous vivions chaque étape de la vie d'un doctorant à peu près en même temps. Merci à Fatima-Zahra pour ta bonne humeur constante. Merci à Simon et Charlène qui ont apporté un souffle nouveau à mes travaux, j'ai adoré travailler avec vous et j'aurais aimé avoir eu le temps de mieux vous connaître en dehors du travail. Merci à Dimitri P. pour m'avoir accueilli très chaleureusement à mon arrivée et pour toutes les discussions, de travail ou non, toujours avec le sourire. Merci à Marie pour toutes les discussions, ta bonne humeur, les soirées jeux et tout le reste. Merci aussi à Florent, bien que ne faisant pas partie de l'équipe Stack tu faisais pour moi partie de mon équipe. Merci à Dimitri S. avec qui on ne s'ennuie jamais et qui a toujours le mot pour le faire rire. Merci à Ronan pour toutes les discussions extrêmement intéressantes et pour l'aide que tu m'as apportée sans compter. Merci à Thomas avec qui les discussions pourraient ne jamais s'arrêter. Merci à Jolan pour la bonne humeur que tu as apportée depuis ton arrivée. Merci à Yewan pour les moments très agréables passés en conférence. Merci à David, Jad, Linh, Sirine, Twinkle et Jacques pour nos agréables conversations. Merci à tous les autres membres de l'équipe Stack, avec qui je n'ai pas eu la chance d'échanger beaucoup mais qui ont toujours été agréables. Enfin, un merci tout particulier à Jonathan qui à Chicago comme à Nantes a toujours été un vrai ami et qui a rendu ces trois années bien plus agréables, un troll après l'autre.

Je remercie également les membres de l'équipe Avalon, que j'ai eu l'occasion de croiser quelques fois mais avec qui j'aurais aimé faire plus ample connaissance.

Un grand merci aux membres de mon comité de suivi individuel, MM. Sébastien Limet et Gabriel Antoniu, qui m'ont encouragé et m'ont fait des remarques pertinentes lors de nos réunions.

Je remercie également MM. Romuald Debruyne et Hervé Grall pour m'avoir fait confiance pour enseigner dans leurs cours, expérience qui a été décisive dans mes choix d'orientation postdoctoraux.

Je remercie évidemment les membres de ma famille, qui m'ont soutenu tout au long de mes études, et en particulier mes parents qui m'ont permis d'étudier le domaine de mon choix et m'ont fait confiance toutes ces années. Je remercie aussi mes proches qui m'ont soutenu et qui se reconnaîtront.

Je terminerai par remercier mes encadrants. M. Adrien Lebre tout d'abord, qui m'a toujours fait confiance et encouragé, dans mon travail de recherche comme pour l'enseignement, et qui a été pour moi un incroyable chef d'équipe. Surtout, je remercie profondément Mme Hélène Coullon et M. Christian Perez, qui tout au long de ces trois années m'ont encadré de la meilleure manière qui soit. Leurs retours extrêmement constructifs et bienveillants, leurs conseils et encouragements, leur soutien et leur compréhension dans les moments difficiles ainsi que leur détermination à me faire réussir ont été essentiels à l'aboutissement de ces travaux, et je leur en suis sincèrement reconnaissant.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Listings

# Chapter 1

# Introduction

Over the past few decades, computer systems which run on multiple computers, called distributed systems, have become more and more common and relied on. Today, they are used even in critical systems, such as hospitals, train or air traffic control, telecommunication networks or autonomous cars. The management of these systems is a complex task: one needs not only to consider a single piece of software running on a single machine, but multiple pieces of software interacting with each other running on multiple machines. This is becoming even more true as distributed systems become larger (putting together more pieces of software running on more machines), more dynamic (having the ability to evolve and react to external changes over time), run on more heterogeneous infrastructures (e.g., fog computing, edge computing) and are relied upon in critical systems.

An essential management task of dynamic distributed systems is their *deployment*. Deploying a distributed system consists in putting it in a functioning state by executing a set of actions such as transferring files, installing software, changing configuration files or executing shell commands.

Nowadays, these systems usually function in a highly dynamic environment: they must deal with a constantly changing number of users, changes in the price of backend services or energy providers, geographical constraints for servers due to laws in certain countries or latency requirements, among others. Additionally, the developers or managers of the applications may want to add new features or update existing ones without having to interrupt the whole system. Modifying a system at run-time is called *reconfiguration*, and, like deployment, consists in executing a set of actions to perform the desired change. Because of this similarity, in the rest of this thesis we consider deployment as a special case of reconfiguration.

Reconfiguration of distributed systems is complex because of the interactions between all the modules that compose these systems. Performing changes on one module usually impacts other modules too: for example, if the database used by a web server is suspended, the web server itself is impacted, and the service that it can provide to its clients is degraded. All this has to be taken into consideration when reconfiguring

distributed systems.

Additionally, many different kinds of actors interact with distributed systems and their reconfiguration. In practice, the modules which compose a given distributed system often come from distinct providers, and yet another actor uses them as building blocks to design the full system. It may even happen that yet another actor needs to reconfigure the system at a later time. All of these actors have specific skills and knowledge, which must be acknowledged when addressing the reconfiguration of distributed systems.

In addition to involving a large number of actors, performing reconfiguration also requires several operations to be performed. Autonomic computing is a field that studies the self-management capabilities of systems. While not restricted to this setting, reconfiguration can be modeled using a the common loop of autonomic computing: MAPE-K. In this model, the adaptation (or reconfiguration) of a system is broken down to four steps: *monitor*, *analyze*, *plan* and *execute*. The *monitor* step consists in gathering data about the system such as the current number of users, energy consumption, event logs, etc. The *analyze* step consists in deciding, using the data gathered in the previous step, whether reconfiguration should be performed, i.e., if changes should be made to the system. The *plan* step consists in, if reconfiguration should occur, determining the set of actions that should be performed to obtain the desired result. Finally, the *execute* step consists in performing these actions. All these steps share a common *knowledge* about the system and the models used to describe it.

In this work, we focus on the *execute* step, as well as the models in the common *knowledge* which are related to the execution of reconfiguration (in particular, a way of describing the set of reconfiguration actions to be performed needs to be provided to the *plan* step). Solutions addressing this part of the reconfiguration process exist to assist the different actors to cope with the complexity that comes with reconfiguring distributed systems. They may be more or less generic in terms of types of reconfiguration actions that can be described. They may also provide more or less expressivity in terms of parallelism between these actions (to increase performance in reconfiguration and limit the time during which the system is down or operates with reduced capabilities). Finally, they may provide separation of concerns between the actors of reconfiguration to various degrees.

## Research Objectives

While many solution exist for the execution of reconfiguration, most of them do not provide at the same time high levels of (a) genericity, (b) parallelism expressivity and (c) separation of concerns between the different types of actors of reconfiguration. Usually, two of them come together: (a) and (c) by generic solutions which do not provide optimal performance in terms of time of reconfiguration, (b) and (c) by solution specific to a given type of reconfiguration and, finally, (a) and (b) by low-level

solutions which are impractical to use in the general case.

In this work, we argue that it is possible to reconcile these three properties. We hence aim at defining a generic framework for the execution of reconfiguration which at the same time allows to express a high degree of parallelism and provides a good separation of concerns between the actors of reconfiguration. This framework should also be defined formally and the level of parallelism precisely defined, so that it can be analyzed, evaluated and provide safety guarantees.

## Contributions

In this work, we first consider the specific case of deployment (recall that we consider deployment a special case of reconfiguration), which lays the foundations for a more general solution on reconfiguration. Then, we consider reconfiguration in general. In this spirit, the contributions of this thesis are the following:

- a formal component model called *Madeus* which reconciles genericity, parallelism expressivity and separation of concerns in the case of deployment;
- a formal component model called *Concerto* which extends Madeus to support reconfiguration in general while conserving these good properties;
- performance models for Madeus and Concerto, which define the exact level of parallelism that can be achieved;
- an implementation of both Madeus and Concerto in Python;
- a comprehensive evaluation of Madeus and Concerto in terms of parallelism expressivity and separation of concerns.

Note that the Madeus model was already under study when the author joined the team and has mainly been designed by Dimitri Pertin who is a former post-doctoral researcher. However, the author of this dissertation has created the formal model of Madeus entirely as well as its performance model. Furthermore, the author has coded the most recent version of the implementation of Madeus as an abstraction of Concerto. This version is the one used in experiments performed on Madeus, including for papers in which the author did not directly contribute to.

## Publications

The above contributions have been the subject of three publications, one in a national conference, one in an international workshop, and one in an international conference. Moreover two journal papers have been submitted, one currently undergoing minor revision and one currently undergoing major revision.

**International conferences**

- **Maverick Chardet**, Hélène Coullon, Christian Pérez. *Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto.* In CCGrid 2020 - 20th International Symposium on Cluster, Cloud and Internet Computing, Melbourne, Australia, 2020.
- **Maverick Chardet**, Hélène Coullon, Christian Pérez, Dimitri Pertin. *Madeus: A formal deployment model.* In 4PAD 2018 - 5th International Symposium on Formal Approaches to Parallel and Distributed Systems (hosted at HPCS 2018), Jul 2018, Orléans, France.

**National conference**

- **Maverick Chardet**, Hélène Coullon, Christian Pérez. *Interfaces comportementales pour la reconfiguration de modèles à composants.* In ComPas 2018, Toulouse, France.

**Submissions to journals**

- **Maverick Chardet**, Hélène Coullon, Christian Pérez, Dimitri Pertin, Charlène Servantie, Simon Robillard. *Enhancing Separation of Concerns, Parallelism, and Formalism in Distributed Software Deployment with Madeus.* In Journal of Systems and Software (JSS) [Under minor revision].
- **Maverick Chardet**, Hélène Coullon, Simon Robillard. *Toward Safe and Efficient Reconfiguration with Concerto.* In Science of Computer Programming (SCP), special issue FOCLASA 2019 "Coordination and Self-Adaptiveness of Software Applications" [Accepted].

## Organization of the dissertation

This document is organized in six chapters after this introduction.

Chapter 2 defines several distributed systems-related concepts which will be used in the rest of the document. Chapter 3 studies the state of the art on deployment and reconfiguration, extracts missing properties of the related work, and presents the challenges of the thesis. Chapter 4 presents our first contribution, the deployment model Madeus. After presenting its general concepts, we give a formal definition of the model, followed by the presentation of a performance model and a discussion. Chapter 5 presents our second contribution, the reconfiguration model Concerto, using a similar plan as Chapter 4. Additional information is also provided on a new concept introduced to maintain a high separation of concerns: *behavioral interfaces*. Chapter 6 first presents our Python implementation of Madeus and Concerto, the one of Madeus being programmed as an abstraction over the one of Concerto. Design

decisions are discussed and examples are provided. This chapter then evaluates both models in terms of parallelism expressivity and separation of concerns through a real production use-case as well as synthetic use-cases. Finally, Chapter 7 discusses the contents of this thesis, concludes and provides ideas of future works.

# Chapter 2

# Distributed Systems and their Modeling

## Contents



In this chapter, we introduce fundamental concepts used in the rest of this document. In Section 2.1, we present distributed infrastructures, classify the different

types of computing resources that they provide and how to access them. In Section 2.2, we give a definition for distributed applications, provide a way to represent them using the concept of component and introduce the concept of life-cycle of a software module. In Section 2.3, we explain what it means to reconfigure a distributed application, present typical types of reconfiguration and position reconfiguration in the context of autonomic computing. Finally, in Section 2.4, we explain how parallelism can occur during reconfiguration, how this parallelism can be modeled and classified.

## 2.1 Distributed Infrastructures

### 2.1.1 Types of resources

Distributed infrastructures are sets of *computing resources* interconnected by one or several *networks*, which can also be considered as resources. In this document, we focus on computing resources and consider physical networks to be the medium in which they live and virtual networks, which are software-defined, to be pieces of software like any other.

We break down computing resources into the following categories.

**Physical machine** A physical machine is a physical computer with hardware resources like CPUs, RAM, storage and other peripherals such as sensors, imaging devices, etc. An *operating system* (OS) is in charge of handling these resources which are shared among the programs and users of the machine.

Physical machines can be assembled into larger structures. For example, a *cluster* of machines usually refers to a set of machines which have similar specifications and are interconnected by a local network. According to the type of hardware that equips a cluster (high speed network, GPUs, many-cores etc.) and to the level of abstractions that hides the hardware and administration aspects from the end-user, a cluster may be given a more specific name, such as *supercomputer* or *cloud infrastructure*. Distinct (and possibly distant) clusters may also be organized as a "cluster of clusters" (e.g., grids, clouds).

**Virtual machine (VM)** A virtual machine is a virtual computer which does not use physical hardware resources directly but emulated ones instead. An *hypervisor* is in charge of running one or multiple virtual machines and emulating their CPU, RAM, storage, etc. using another machine's (usually physical) resources. An *operating system* (OS) also runs on a virtual machine and allows to access the emulated resources with software running on the machine itself.

Virtual machines are usually used to co-locate multiple distinct OSs using the same physical hardware while ensuring a strong isolation between them. Multiple

users can therefore share the same physical machine without compromising security. Cloud providers, which will be introduced later, make heavy use of this.

Examples of virtualization solutions are Linux KVM[1], VMware[2] or Microsoft Hyper-V[3].

**Container** A container is an isolation layer running on an OS. Containers co-located on the same OS share their access to the kernel but have distinct software, storage, libraries, configuration files, etc. Containers are more lightweight than virtual machines (they all share the same OS).

A *container management system*, like Docker, is in charge of running these containers. Container management systems usually provide advanced management functionalities such as replicas (running multiples of the same container).

Examples of container solutions are Docker[4], Linux containers[5] (LXC, LXD, LXCFS) or rkt[6].

### 2.1.2 Provisioning computing resources

Computing resources are made available in different ways.

#### 2.1.2.1 Direct access

Sometimes, there is no need to reserve computing resources because they are directly available. For example, private servers that run uninterrupted, workstations, etc.

#### 2.1.2.2 Clusters and grids

Recall that clusters and grids are groups of physical computing resources. While they are shared among multiple users, most of the time the users require that they be the only user using the subset of resources they use to avoid interference. Reservations therefore need to be made (on a subset of the cluster or grid). *Batch schedulers* such as *oar*[7] or *slurm*[8] are used to manage these infrastructures and reservations. A user typically submits a reservation for a subset of nodes in the cluster or grid (possibly all the nodes) and for a given period of time. When this period starts, the user is given access to the resources, and when it stops, access is withdrawn. Examples of grids with this type of access are the Computing Center for Research and Technology[9] and

[1] https://www.linux-kvm.org/
[2] https://www.vmware.com/
[3] https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/
[4] https://www.docker.com/
[5] https://linuxcontainers.org/
[6] https://coreos.com/rkt/
[7] https://oar.imag.fr/
[8] https://slurm.schedmd.com/
[9] http://www-hpc.cea.fr/en/complexe/ccrt.htm

the Barcelona Supercomputing Center[10]. Experimental platforms for research such as Grid'5000[11] and Chameleon[12] also provide this kind of access.

#### 2.1.2.3 Cloud computing

The American National Institute of Standards and Technology[13] defines cloud computing [1] as follows: "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

Cloud infrastructures can be public or private. Large public cloud actors include Amazon Web Services[14], Microsoft Azure[15], Google Cloud[16], and OVH[17].When it comes to private clouds, OpenStack[18] is the de-facto open-source standard to operate the underlying resources.

Cloud infrastructures use virtualization technology [2] (in particular virtual machines) to provision and release computing resources rapidly. They provide languages or APIs (e.g., Amazon CloudFormation [3] or OpenStack Heat [4]) to let the users specify their needs. Three main kinds of services are usually available. First, IaaS (Infrastructure as a Service) allows users to request the creation (or the deletion) of new virtual machines, which is usually done in a matter of seconds or minutes. The user is then granted access to each of them. The other two, PaaS (Platform as a Service) and SaaS (Software as a Service) allow users to create or destroy instances of ready-to-use services or software, but they do not have to handle virtual machines directly. The user is then provided with a way to use the services (access point, credentials, etc.).

The virtual machines can give access to different kinds of hardware (number of CPU cores, GPU, capacity of storage, type of storage, etc.) and be hosted on machines physically located in a given location (which can be desirable to have low latency or for legal/privacy/security reasons). A recent trend is the development of *fog computing* [5, 6, 7] which consists in providing computing resources very close to the end users (to allow very low latency or high-volume communication without network congestion). *Edge computing* [8] is also developing and consists in using the (usually small) computing resources at the very edge of the network (e.g., sensors, chips) to pre-process the data, and therefore lowering the quantity of data to be sent

[10] https://www.bsc.es/
[11] https://www.grid5000.fr/
[12] https://www.chameleoncloud.org/
[13] https://www.nist.gov/
[14] https://aws.amazon.com/
[15] https://azure.microsoft.com/
[16] https://cloud.google.com/
[17] https://us.ovhcloud.com/
[18] https://www.openstack.org/

to cloud or fog computing resources for processing. A typical example is video analytics [9], in which hundreds or thousands of surveillance cameras in a smart city process the information locally before sending only the relevant information to the cloud, instead of streaming the whole video feed.

### 2.1.3 Accessing remote computing resources

Once computing resources are provisioned, one needs to be given access to them. Depending on their intended usage, this access can have different forms.

#### 2.1.3.1 Direct shell access

In some cases, it is beneficial for the users to have direct shell access to the computing resources they use. This access can have different levels of privilege, from read-only to root. This kind of access, with a high-enough level of privilege, gives the most control to the user who is able to install anything on the machine, start and stop services, run any kind of software task at any time.

In the case of physical or virtual machines, this access is usually given through the *SSH* (Secure SHell) protocol. In the case of containers, the virtualized OS directly provides a shell accessible from the hosting machine.

#### 2.1.3.2 Batch access

Sometimes, giving direct access to the machines to users is not desirable, either for security purposes or to maximize the utilization of the computing resources, for example to force releasing the machines when the computing task requested by the user is finished. Batch access to a machine or group of machines allows users to submit a job to be executed (e.g., a Bash script). This job can have metadata attached to it, such as a priority or hardware requirements (e.g., number of CPU cores, RAM and storage capacity). A *scheduler* then uses this information to decide the order in which the tasks are executed and on which computing resources. This kind of access gives less freedom to the user compared to direct shell access as there is no way to control when the task will be executed, and everything needs to be planned in advance. Note that the level of privilege on the machine can also vary.

This kind of access is usually provided by computing-oriented platforms such as clusters, grids or super-computers. It is also common in the case of federations (infrastructures made in part of user hardware which they can add and remove to the federation at any time). Examples of federations are Folding@home[19] or Rosetta@home[20]. In these cases the only objective of the task is to compute a result using input data, and not to provide services to other systems. For example, batch access is unsuitable to run web services.

[19] https://foldingathome.org/
[20] https://boinc.bakerlab.org/

Note that batch access and direct shell access are not incompatible, and sometimes both can be used depending on the use-case.

## 2.2 Distributed Software

### 2.2.1 Architecture of distributed applications

*Distributed software* can be defined as software which runs on distributed infrastructures. In distributed software, multiple software modules interact with each other. Each module is located on a single compute resource, but a given compute resource may be the host of multiple modules.

A *distributed application* is a piece of distributed software which is defined to achieve a specific goal. Examples of distributed applications are web services [10], composed of one or multiple web servers, proxies to distribute the user load over these servers, databases to store persistent data and other back-end web services (file storage, image processing, etc.). A typical example would be a social networking service.

This application may be *static*, in which case its set of modules is defined ahead of use and remains the same during the life-time of the application, or *dynamic*, in which case its set of modules and their interactions may evolve over time, and in particular at run-time. Example of such evolutions are the increase of the number of web servers in case of high user load, the update of modules of the application or a change in how the modules interact.

In order to fulfill its purpose, a distributed application must be *deployed* to a distributed infrastructure. Deployment is the process which consists in installing, configuring and running each module of the application so that they all work together as intended. This process is complex because modules need to run on appropriate hardware, have proper network connectivity depending on their requirements, and dependencies between modules need to be taken into account. These dependencies can be of many types, such as for instance: temporal or functional when a service cannot start before another service of another module has started, information-related when a module needs information from another module, such as an IP address, to configure itself.

For example, let us consider a web server and a MySQL database which it needs to function properly. To deploy this application, compute resources first have to be provisioned for both modules. Then, both of them have to be installed on these resources, and configured. Note that the web server needs the IP address of the database as a part of its configuration, information which is not known prior to the provisioning process. Then, the server cannot run before the database is operational.

Notice that the complexity grows with the number of modules and their diversity in a distributed application. A large research community exists to solve problems related to deployment of distributed software. In this work, we focus on one possible

cause for complexity (dependencies between modules) and assume that the placement problem (choosing where to deploy each module) has already been solved.

Many kinds of distributed applications exist. *Server-client* applications consist in a set of modules (called the *clients*) interacting with a single module (called the *server*) to get information they need to function. *Master-worker* applications are the opposite: a module called the *master* provides other modules called *workers* with some tasks to perform. Data-oriented applications sometimes use more complex topologies such as *map-reduce* (often abstracted to the user by dedicated frameworks such as Hadoop[21]) where the application is split into layers of modules (splitting, mapping, shuffling, reducing), each layer sending tasks to modules of the next layer. A recent trend is the development of *micro-services* applications, where each module can provide a well-defined service and defines an interface for other modules to use this service. Web services are often structured this way. *Peer-to-peer* applications consist in a set of similar modules, usually each deployed on a different compute resource, which can communicate with any other module if need be (for example to exchange data).

We can classify the modules of distributed applications into two categories. They can be *stateless*, which means that they do not store data other than their configuration files and possibly cache, or *stateful*, which means that their behavior depends on data which depends from the previous interactions it had with other modules. A typical example of stateful module is a database.

In this work we focus on distributed applications with the following characteristics:

- each module may provide one or more services;
- each module has a set of *dependencies* to a given set of services provided by other modules;
- each module may be stateful or stateless;
- the mapping between the modules providing the services and those which use them can be known at any given time, in particular by the system managing the distributed application.

### 2.2.2 Component-based representation

The component paradigm is used both as a representation method (e.g., in UML[22] - Unified Modeling Language) and a software development method (e.g., Component-Based Software Engineering [11, 12]). It consists in describing applications as sets of *components* with clearly-defined interfaces. These interfaces are represented by *ports* (each port corresponding, for example, to a service that is provided or a requirement

[21] https://hadoop.apache.org/
[22] https://www.uml.org/

to a given service), and the ports of different components are *connected* when they interact (for example, when a component that requires a service uses the one provided by another component). A set of inter-connected components is usually called an *assembly*.

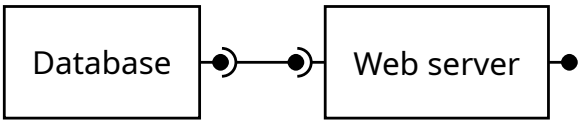



Figure 2.1: Example of a component assembly with two components: a web server and a database. The database provides a service through its provide port and the web server uses this service through its use port, while itself providing a service through a provide port.

*Component-based models* (or *component models*) are particularly well-suited to model service-oriented distributed systems, as each module can be represented by a component, the dependencies of the module can be represented by *use ports* and the services that it provides by *provide ports*. Figure 2.1 shows an example of a distributed application composed of a web server and a database, the web server using the service provided by the database.

Component models usually come with an Assembly Description Language (ADL) which allows its users (among other things) to define assemblies by listing its list of components and connections.

### 2.2.3 Life-cycle of distributed applications

#### Life-cycle

The *life-cycle* of a piece of software corresponds to the set of *configurations* in which it can be, as well as how and when it can go from one configuration to another. A *configuration* designates everything that determines how the piece of software behaves internally and consequently how it interacts with the outside world, directly (e.g., through an API) or indirectly (e.g., through what actions must be performed to put it in a given configuration).

For example, let us consider the MySQL database in our previous example. Two main configurations first come to mind: *running* (when one can use its MySQL API) and *not running* (otherwise). Both of these can be refined, however. When the database is running, there are plenty of settings for a MySQL database, which can affect whether some SQL requests are allowed, the speed at which requests will be treated, etc. Likewise, when the database is not running, it may be because: the service has not been started on the machine hosting the database, it is in an error state, it has not yet been configured, some dependencies of MySQL are missing, the database is not currently existing, etc. Given any two configurations, one can provide

a set of actions to go from one to the other (such as starting the MySQL service or installing dependencies) or a set of alternatives in case of errors.

One might notice that it is possible to consider a very large number of states to take into consideration everything that might affect the interface of a piece of software to a slight degree. For example, in the case of a database, we could consider that a `write` requests effectively changes the interface of the database because its responses to the following requests will be different. However, the life-cycles of software are usually considered through models which abstract away the parts which are not relevant or practical. The most common models used to represent life-cycles are state-machines, which are detailed later in this section.

### Extension to distributed applications

Distributed applications are composed of multiple interacting modules, which each has its own life-cycle. For example, if we consider a web server using a database, the web server has its own life-cycle, as well as the database. Notice that these life-cycles are not independent: the life-cycle of the web server depends on the one of the database. This is because the web server may not provide some of its services when the database is not running.

### Representation using state-machines

Life-cycles are usually modeled using state machines or their derivatives. In the following, three of them are introduced: basic state machines, Petri nets and UML state-charts.

**Basic state machines** State machines [13, 14] are defined by a set of *states*, and a set of *transitions* between those states, each labeled with a *symbol* from a given *alphabet*. In the following, we only consider deterministic state machines, meaning that given a state and a symbol, there is always at most one transition labeled with this symbol leaving this state. One state is declared to be the *initial state*. The state machine is always in one given state, initially the initial state. It changes states when it receives a symbol of the *alphabet* as input: it follows the transition from its current state to a new state labeled with this symbol (if there is no such transition, the machine halts).

State machines can be used to model life-cycles by representing configurations with states, possible actions by symbols of the alphabet and the fact that one action leads from one configuration to another by a transition between two states labeled with this action. Figure 2.2 shows how the life-cycle of a database might be modeled using an a basic state machine.

**Petri nets** While basic state-machines capture the notion of configurations and actions to go from one to the other, they do not capture the potential parallelism

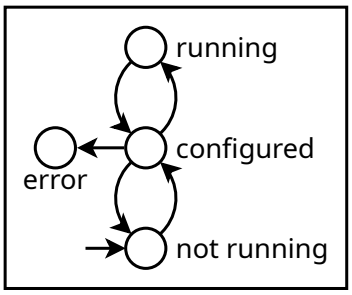



Figure 2.2: Example of a life-cycle defined with a basic state machine.

between actions. For example, to go from *uninstalled* to *configured*, it may be possible to parallelize actions such as downloading the dependencies and setting up the local file-system.

Petri nets [15, 16] are developments of state machines in which there may be multiple current states (called places), and where transitions go from a set of source places to a set of destination places. Active places hold one or more *tokens*. A transition executes by consuming a given number of tokens from each source place and producing a given number of tokens in each destination place.

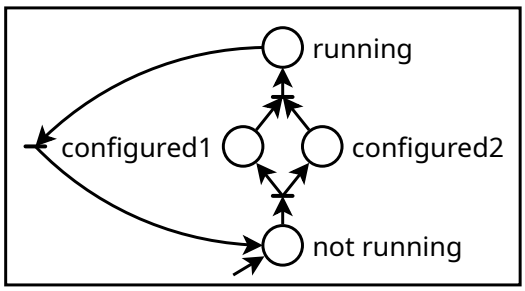



Figure 2.3: Example of a life-cycle defined with a Petri net.

By considering transitions consuming a single token per source place and consuming a single token per destination place, it is possible to represent parallel actions. Figure 2.3 shows how the life-cycle of a database might be modeled using a Petri net, with parallel actions. Note that more complex scenarios may be modeled by Petri nets, but they are not relevant to our definition of a life-cycle.

**UML state charts** UML state charts [17, 18] are an extension to basic state machines, with additional features to improve their expressivity (in particular to model parallelism like Petri nets), but also to reduce the explosion in number of states and transitions as the system to model grows in size. The two main additions are *nested states* and *orthogonal regions*. First, states can themselves be UML state charts (in which case they are called *composite*, as opposed to *simple*). When receiving an input

symbol, the deepest state chart in the hierarchy attempts to process it, and if no suitable transition exists, the state chart above it in the hierarchy makes an attempt, etc. Second, a composite state can be composed of multiple state charts instead of one, in which case they are considered to be orthogonal regions. When inputs are treated at this level of the hierarchy, they are processed by both state charts in parallel.

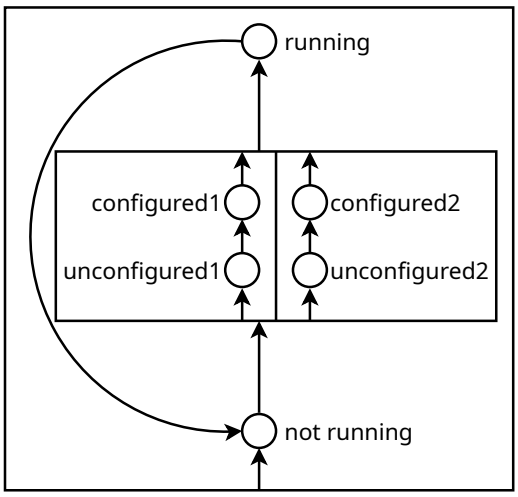



Figure 2.4: Example of a life-cycle defined with an UML state-chart. It is composed of two basic states (`not running` and `running`) and one composite state, itself composed of two orthogonal regions: one containing states `unconfigured1` and `configured1`, and the other containing states `unconfigured2` and `configured2`.

Figure 2.4 shows how the life-cycle of a database might be modeled using an UML state chart, with parallel actions and nested states. The transition leaving from state `not running` leads to a composite state (i.e., state which contains its own state-machine), composed of two parallel regions. In practice, after leaving the `not running` state, both the `unconfigured1` and `unconfigured2` states become active. The two regions then progress independently from each other. When both states `configured1` and `configured2` are active, the transition to go to state `running` may be used.

## 2.3 Reconfiguration of Distributed Applications

### 2.3.1 Overview

Reconfiguring distributed applications consists in changing their configuration, i.e., interacting with their life-cycles. Recall that the life-cycle of a distributed application is made of the life-cycles of all of its modules, and that these life-cycles are, in the general case, not independent from each other.

### 2.3.2 Typical types of reconfiguration

While an infinite number of reconfigurations can be imagined, some reconfiguration problems are faced very often by users, which leads to some typical types of reconfiguration being addressed by many solutions in the literature. Here we present the most common ones.

#### 2.3.2.1 Deployment

The deployment of an application consists in moving it from a non-existing state to a functional state, i.e., a state in which it operated properly. A deployment could also be called a commissioning procedure. In the case of distributed software: each module must be deployed, respecting its dependencies to other modules.

The typical way to do this is to choose a host on which to deploy each module (placement) and then to deploy each module when the modules it depends on have already been deployed. A deployment procedure is thus highly correlated to the type of resources targeted, the provisioning of resources, and their access, all of which are described in Section 2.1. Many academic contributions, production tools and languages have been proposed to automate and ease the conception of deployment procedures. A large part of these solutions is described and classified in Chapter 3. Recall that in this document, we consider deployment as a specific case of reconfiguration.

#### 2.3.2.2 Scaling

Let us consider a distributed application which includes a number of replicas of the same module (for example compute nodes or instances of a web server). The scaling of this distributed application consists in increasing or decreasing the number of replicas of this module. This is usually done to maintain a good balance between costs and quality of service, for example in the case of an application with a number of users which varies over time.

Typically, rules are given by the user to trigger an increase or decrease in scale depending on given metrics such as user load, CPU load of given machines, energy consumption, etc. [19, 20, 21].

#### 2.3.2.3 Update

Updating an application consists in replacing its code and configuration with a new version. In the case of distributed applications, each module may be updated individually. However, while a module is being updated, the other modules that depend on it have to stop using it during the update process.

This is typically done by suspending them or putting in a state in which they do not use the module currently being updated [22, 23, 24, 25, 26, 27].

The challenges that comes with updating distributed applications are first to minimize the propagation of the service interruption, and then to minimize the downtime

of the application, i.e., the time during which the services it provides are unavailable or limited. Some techniques exist to address this. In particular, rolling software update consists in, when updating replicas of a same module, updating only a part of them while the other ones continue serving requests. This is done for example in OpenStack[23].

#### 2.3.2.4 Migration

*Migration* consists in moving one or more modules from their compute resources to other ones. Sometimes, in a cloud computing context, this can be done by migrating the virtual machines themselves [28], but other times the software modules themselves are moved [29]. Sometimes, migration also refers to moving data handled by a module to another module, possibly needing a conversion [30, 31]. This is usually done when the hardware capabilities of the compute resources currently used do not match the current workload (if the latter is too high, more compute power is required; if it is too low, money and/or energy can be saved by using less powerful machines), to reduce latency or to use another system (for example another type of database).

The main challenge with migration is to minimize the downtime, which is more difficult when the modules are stateful (e.g., it is not possible to run another copy on the new resource and stop the old one once the new one is already functional).

### 2.3.3 Autonomic computing

Autonomic computing consists in giving any software system (distributed or not) the capability to autonomously adapt (or self-adapt) to its evolving environment. This process includes the monitoring of the system to control, decision-making to decide whether changes are required, which ones, when to apply them and how, and the execution of these changes itself. There exist several models for autonomic computing. The one that is usually adopted in computer science and distributed systems is the MAPE-K model introduced by IBM in 2003 [32]. MAPE-K models the autonomic process as a loop comprised of four steps: *Monitor*, *Analyze*, *Plan* and *Execute*, all sharing a common *Knowledge*.

The Monitor step consists in gathering metrics on the system, such as the CPU load of the machines in the distributed infrastructure, the number of current users, the energy consumption, logs, etc. The *Analyze* step consists in, given the metrics collected, deciding whether or not a change should occur in the application. The *Plan* phase consists in generating a reconfiguration plan to achieve the change decided in the Analyze phase. The *Execute* step consists in applying the reconfiguration plan to the application. The common *Knowledge* is made of all the models, languages and information about the application shared between the four steps.

[23] https://docs.openstack.org/ironic/train/contributor/rolling-upgrades.html

While all the steps of MAPE-K are related to reconfiguration, in this document we address specifically the execution of reconfiguration (which takes place in step (E) of MAPE-K), as well as the models used for the execution itself and its interface with the other steps (K).

## 2.4 Parallelism in Reconfiguration

### 2.4.1 Overview

Executing a reconfiguration plan consists in performing multiple reconfiguration actions. These actions usually have dependencies so that one action must have to be executed after some other actions, but before others.

The traditional way of scheduling these actions is to pick any total order that satisfies these dependencies, and execute them sequentially in that order. However, this is not optimal in terms of performance. Intuitively, executing tasks in parallel makes the total execution time shorter. This is for example mentioned in the conclusion of [33]. While this claim has to be put in perspective (for example, a task may take longer to execute if it is done in parallel with another task), increasing parallelism is one way in which the performance of reconfiguration can be improved.

Increasing parallelism can be done in two ways: executing tasks in parallel if they are known to be independent and increasing the number of tasks which are known to be independent. Intuitively, more detailed models of the life-cycles of the modules leads to more information about the dependencies between reconfiguration actions. This claim will be explored further in Chapter 3.

In the following, we discuss a way to model dependencies between reconfiguration actions, and then introduce a classification of the different types of parallelism in reconfiguration of distributed applications.

### 2.4.2 Modeling sets of reconfiguration actions with dependency graphs

When trying to model in particular the set of actions to perform to go from one configuration to another, a tool at our disposal are *dependency graphs*. In particular, they can be used to model the partial order of actions to perform. In the following, we consider a particular type of dependency graphs: connected weighted directed acyclic graph in which each arc corresponds to an action (i.e., a task), and the weight of this arc is the time it takes for the task to complete. There are always at least two vertices in a dependency graph: the *source* vertex, which is an ancestor of all the other vertices, and the *sink* vertex, which is a descendant of all the other vertices. A task $t$ depends on another task $t'$ if there exists a path from the *source* vertex to the *sink* vertex going through $t'$ and $t$ in this order. A task cannot be executed before a task it depends on is over, so that the vertices in the graph act as synchronization

points for tasks. Two tasks which do not depend on each other can be executed in parallel.

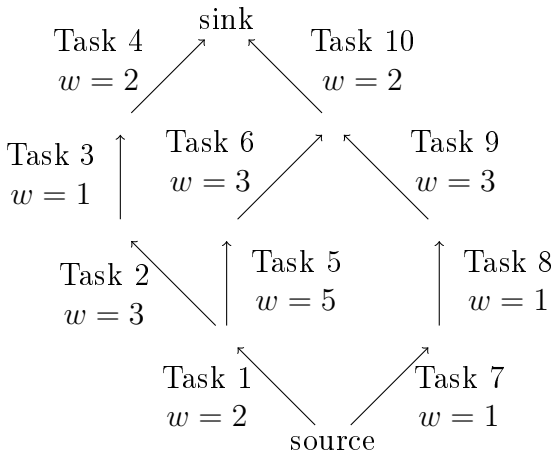



Figure 2.5: An example of dependency graph.

Figure 2.5 shows an example of dependency graph. In this graph, tasks 1 and 7 can be executed in parallel because they do not depend on each other. However, task 10 depends on tasks 1, 5, 6, 7, 8 and 9 and has to wait for those to be completed before it can execute. By definition, a task can execute after a duration equal to the maximum weight of the paths from the source to the parent of its corresponding arc. In this example, task 10 can execute after a duration equal to $\max(2{+}5{+}3, 1{+}1{+}3) = \max(10, 5) = 10$. The path or paths with the largest weight (in this case the path going through tasks 1, 5 and 6) is/are called *critical path(s)*. By definition, the critical paths of a vertex $v$ are the longest paths between the source vertex and $v$. The *critical paths* of the graph are defined to be the critical path(s) of the sink vertex, i.e., the longest paths between the source vertex and the sink vertex. In our example, the critical path of the graph is the one going through tasks 1, 5, 6 and 10. The time complexity to find the critical path in a dependency graph $(V, E)$ is $\mathcal{O}(|V| + |E|)$.

### 2.4.3 Types of parallelism in reconfiguration of distributed systems

We introduce below three types of parallelism which may occur during reconfiguration. Note that these are not mutually exclusive. They serve as a basis for the classification of parallelism we introduce in Chapter 3.

#### 2.4.3.1 At the host level

Distributed systems consist in software running on multiple compute resources, called *hosts*. Parallelism at the host level consists in multiple hosts executing reconfiguration

actions at the same time. These actions may be identical or different across hosts.

#### 2.4.3.2 At the module level

While parallelism at the host level depends on the infrastructure, parallelism at the module level depends on the software architecture, and in particular how the distributed system is split into modules. It consists in reconfiguration actions concerning distinct modules being executed at the same time.

#### 2.4.3.3 Within modules

The model of the life-cycle of a module includes which reconfiguration actions may be executed and lead from one state to another. If the life-cycle is precise enough, it may include information about which reconfiguration actions can be executed in parallel. Parallelism within modules refers to reconfiguration actions affecting the same module being executed in parallel.

## 2.5 Conclusion

In this chapter we have defined several concepts, in particular distributed infrastructures (and how they can be interacted with), distributed software and applications, deployment and reconfiguration of distributed applications, parallelism in reconfiguration as well as autonomic loops (and how the execution of reconfiguration is related to other steps of autonomic behavior). We have seen that distributed applications are made of modules interacting with each other, each module having its own life-cycle. We have also introduced ways to represent these modules (component models), their life-cycles (state machines and their derivatives) and parallelism of reconfiguration actions (dependency graph). We have also suggested that parallelism of reconfiguration actions is not independent from the life-cycles of the modules being reconfigured. This will be studied in details in the following chapter, which covers the state of the art for this document.

# Chapter 3

# Reconfiguration of Distributed Systems: State of the Art

## Contents



Both reconfiguration and distributed systems are vast research areas. In this chapter, we analyze works at the intersection of these areas, i.e., works that address the reconfiguration of distributed systems. We compare these works in there current state

as of the publication of this thesis and do not consider their history. In Section 3.1 we define the scope of this state of the art. In Section 3.2 we list and define the criteria that we use to analyze each contribution. In Sections 3.3 and 3.4 we list and present the contributions in our scope, grouped into two different categories: *configuration management tools* and *control component models*. Finally, in Section 3.5 we discuss the overall picture and draw some conclusions.

## 3.1 Scope

While many types of reconfiguration exist, this work focuses on distributed systems, and mostly on service-oriented systems. Moreover, we focus on two particular aspects of reconfiguration: the execution of a given reconfiguration plan and the models used to express the elements involved, the reconfiguration plan, the reconfiguration tasks and their dependencies. In the context of a MAPE-K loop, this would correspond to E (execute) and the part of K (knowledge) which concerns E or its interface to P (plan). Finally, we aim at providing a general and generic reconfiguration solution for the targeted systems, i.e., which is not specific to a particular technology, language or scenario.

### Exclusions

Following these guidelines leads us to exclude the following categories of work from this state of the art.

**Software package managers** Software package managers, like *apt*, *yum* or *nix* [34, 35], manage the set of packages installed on a given machine. Because package managers consider the problem of package dependencies and package updates they can be considered as reconfiguration systems. However, they do not handle distributed elements and remote dependencies. We hence do not consider them to perform reconfiguration of distributed systems. Instead, we consider them as a tool which can be used by a reconfiguration framework to apply some change on a given machine.

**Continuous integration, deployment and delivery** DevOps is a recent trend in software development. In [36] it is defined as "a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality". Among these practices are continuous integration, deployment and delivery. *Continuous integration* consists in sharing the code of an application in a common repository using a version control system, and keep it updated with the changes made by developers on a daily basis. *Continuous deployment* consists in testing, after any change has been made, if the application still works as intended by automatically deploying the application and

running a set of tests. When all the tests pass, the changes may be pushed to a production environment through *continuous delivery*, effectively publishing a new version of the software. This software may then be deployed in its most recent version, or updated if already deployed.

While these practices may be part of a larger system to automatically update software modules, or even add or remove resources and software modules (e.g., by using paradigms such as *infrastructure as code*), they do not address the actual execution of such a reconfiguration and are therefore out of our scope. However, the work presented in this document may be integrated to DevOps practices to enhance the deployment and update processes.

**Dynamic Software Update (DSU) frameworks** DSU frameworks, like Ginseng [37], [38] or Coqcots/Pycots [39] allow to update the code of a running application written in a specific coding language. Most of the time, the dependencies they take into consideration are those in the call stack, because in most cases updating a function which is currently executing is problematic. However, external work is required most of the time to handle dependencies between distinct executables running on distinct machines. In any case, each of these frameworks target applications written in a specific language (e.g., C for Ginseng, Python for Coqcots/Pycots). Indeed, because of the techniques used in DSU, it is not possible to write a light wrapper to encapsulate something written in another language. This makes any DSU framework specific to a technology. Again, DSU may be used by the frameworks we do consider as a way to apply some change to a given piece of software.

**Standard APIs** Over time, standards have been created to unify the APIs of multiple services such as cloud providers, virtual machines or containers. For example, OCCI [40] (Open Cloud Computing Interface) is a common interface for cloud providers. Similarly, OVF [41] (Open Virtualization Format) is a common interface for virtual machines and containers. These contributions may be used by reconfiguration frameworks, but in themselves they do not address the execution of reconfiguration.

**Autonomic frameworks addressing specific needs** Some autonomic frameworks define a full MAPE-K workflow to achieve specific goals. For example MuScADeL [42] allows to express constraints over a large-scale system by defining multiple scales on which the constraints can apply. These constraints consist in specifying the system requirements of the devices where software modules must be deployed, their required geographical location, etc. The whole system can then autonomously adapt to changes, in particular to new users entering the system, users disconnecting from the system, devices changing location, etc. However, the complexity of MuScADeL is mainly located in the *monitoring*, *analysis* and *planning* steps of the MAPE-K loop, not in the *execution* step. Similarly, Calvin [43] (and its extension Calvin Constrained [44] to support resource-constrained devices) provides a framework for the

automatic management of data-driven distributed IOT applications, but does not address specific issues in the execution of reconfiguration.

**Solutions addressing other parts of the MAPE-K loop** As previously stated, this work focuses on the *execute* step of the MAPE-K loop. While it has many connections with the other steps, in particular the *plan* step, all the work consisting in deciding which reconfiguration should occur is out of the cope of this state of the art. For example, Ctrl-F [45, 46] models the life-cycle of software components and the policies of a given distributed system as a way to avoid executing reconfiguration actions that would lead the system in an inconsistent or undesirable state. It does not address the execution of the reconfiguration actions themselves, however, which makes it complementary to the work presented in this document. In [47], the authors provide a declarative language to express the desired configurations for the *analyze* step to know when to perform reconfiguration. Another example is the approach taken in [48], [49] or in Safran [50], which consists in using the aspect-oriented approach to decouple the concern of writing functional code and writing reconfiguration rules (analyze/plan) and the concern of writing functional code.

**Software component models** We call software component model a model in which the functional parts of the software modules are defined inside components, following the principles of CBSE [11]. This category includes CCA [51], CCM [52], Fractal [53], SCA [54], DirectMOD [55] or BIP [56]. They provide a way to implement software in a modular and maintainable way. They include at least an *abstract model*, i.e., a list of concepts which can be used to define the software modules (such as components, ports, connectors, membranes or states).

However, this abstract model does not account for practical concerns such as how to deploy components on an infrastructure, run them or making components communicate, either on the same computer or remotely. The core of the components model therefore does not define how to execute reconfiguration. In this document, we hence consider reconfiguration frameworks which are based on component models, but not the component models themselves.

## 3.2 Analysis criteria

In this section, we introduce six criteria that are used in the rest of this chapter to analyze the literature.

### 3.2.1 Reconfigurable elements

Given a reconfiguration framework, the reconfigurable elements are the objects that can be manipulated during a reconfiguration. The more types of elements can be

reconfigured, the more expressivity the solution has.

In this document, we focus on software reconfiguration. When any command, script or program can be executed by a reconfiguration framework, any software-related reconfiguration can be expressed. In this case, we say that *general software* is reconfigurable.

Some frameworks provide dedicated abstractions for more specific categories of software reconfiguration (e.g., management of containers). This dedicated support improves convenience and safety. This support may be provided in addition to general software reconfiguration, or instead of it (in which case the expressivity is lower).

We observe that the following elements may have dedicated support:

- Containers - Dedicated support for containers takes the form of an abstraction of a container service. It may support operations such as creation of containers or groups of replicas of containers, container image management, etc.
- Virtual machines - Dedicated support for virtual machines takes the form of an abstraction of an hypervisor. Changing the number of virtual machines used in a distributed system or their locations are a common tasks and one of the main arguments for the cloud paradigm.
- Custom elements - Some frameworks can be extended with additional types of elements by the community (e.g., storage systems, database).

In the following, any natively supported element considered as an entity that can be managed by a reconfiguration framework is called a *module*.

### 3.2.2 Types of reconfiguration operations supported

A reconfiguration may range from deploying a couple of distributed software modules to completely changing the architecture of a running stateful distributed system. Reconfiguration frameworks may support only a subset of these types of reconfiguration operations. While it is always possible to manually perform reconfiguration tasks with custom code, we only consider the reconfigurations which are natively supported by each solution. Note that for the moment we consider the reconfiguration operations on the modules that can be manipulated (presented in the previous section), and not how the dependencies between these modules are handled. We classify the types of reconfiguration operations as follows:

- Deploy modules - Given the description of an application, ability to put in service the modules that constitute it.
- Add modules - Given an already existing application, ability to add additional modules to this application.

- Remove modules - Given an already existing application, ability to remove modules.
- Custom module operations - Ability to execute operations defined by the developer of this modules. These operations are usually made to change the configuration of the module or how it interacts with the other modules.
- Terminate the application - Ability to terminate gracefully a distributed application. This is a subset of *Remove modules* without the ability to keep part of the application alive.

Note that we consider here the capabilities at the execution level, meaning that any planning-related feature, such as *scaling* (increasing or decreasing the number of instances of a module, most likely a VM) or *rolling updates* (updating a set of instances of a module in two or more steps in order to continue providing a service) are designated by what is actually executed when they are performed (adding or removing modules in the case of scaling, and custom module operations performing updates in the case of rolling updates).

### 3.2.3 Life-cycle handling

The life-cycle of modules may be modeled and taken advantage of to different extents. In Subsection 2.2.3, we stated that life-cycles can be modeled using various kinds of automata. We therefore use the concepts of automata to classify the different types of life-cycle models observed in the literature:

- On/off - This is when an module is either deployed or not deployed. The life-cycle consists of two states, *on* and *off*. An action (or transition) may be executed to deploy the module or delete it.
- Fixed-$n$ - This is when an module has $n$ possible states, but all of them are identical for all modules. An action may be executed to go from one state to another. Note that *on/off* is the particular case *fixed*-2.
- Custom-seq - This is when developers may customize the life-cycle of each module: each module can have any finite number of states, and an action may be executed to go from one state to another.
- Custom-par - Similar to *custom-seq*, except that a module may be in multiple states and/or execute multiple actions to change states at the same time.

Additionally, as soon as there are at least three possible states, the life-cycle model is:

- Single-path - If there is never more than one path to go from a state to another (not counting cycles).
- Multi-path - Otherwise. This effectively allows for choices to be taken at run-time on which path to choose to go from one state to another.

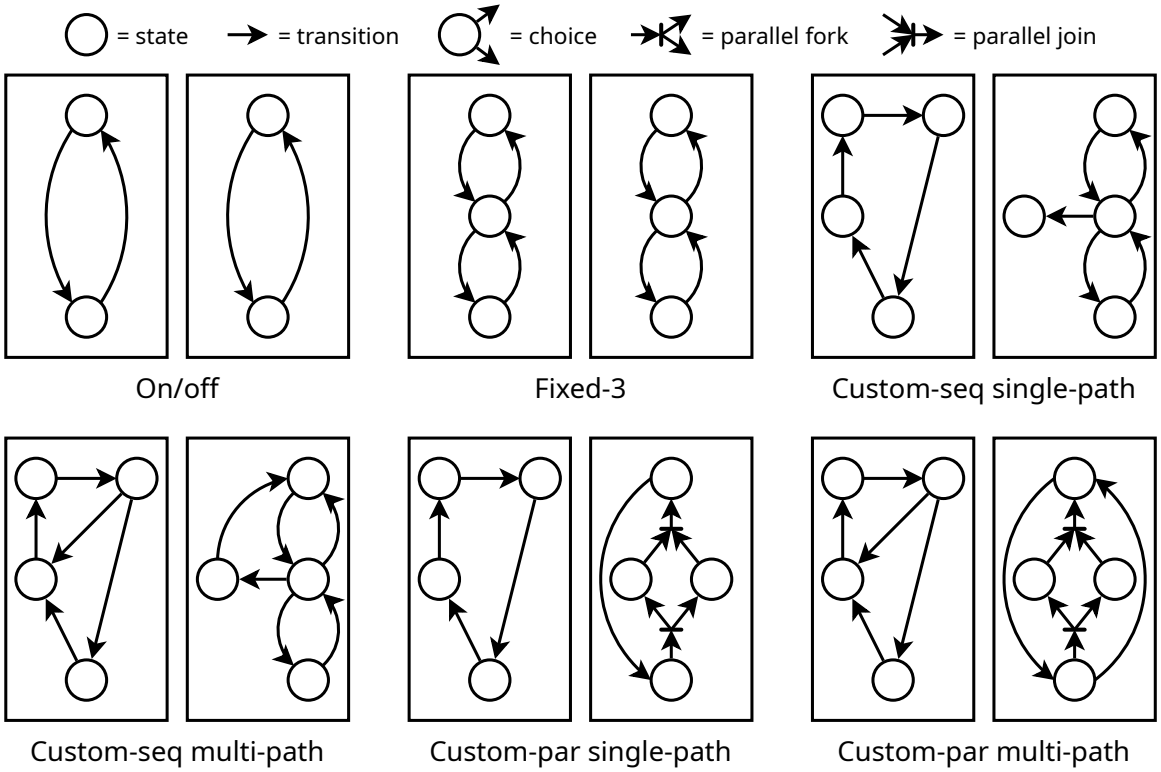



Figure 3.1: Different types of life-cycle modeling. Each type is exemplified by a state-machine-like representation of the life-cycles of two modules.

Figure 3.1 shows different ways in which the life-cycle of a server and a database may be modeled.

### 3.2.4 Parallelism of reconfiguration tasks

One way to improve the performance of reconfiguration, i.e., to decrease the time it takes to execute a reconfiguration plan, is to perform reconfiguration tasks in parallel. Recall that in Section 2.4.3, we have identified three main types of parallelism during reconfiguration: at the host level, at the module level and within modules (which are not mutually exclusive). In this section we introduce different levels of parallelism, i.e., combinations of the three types of parallelism, that we can observe in the literature. Figure 3.2 illustrates these levels of parallelism through an example where the green action depends on the blue action of another module.

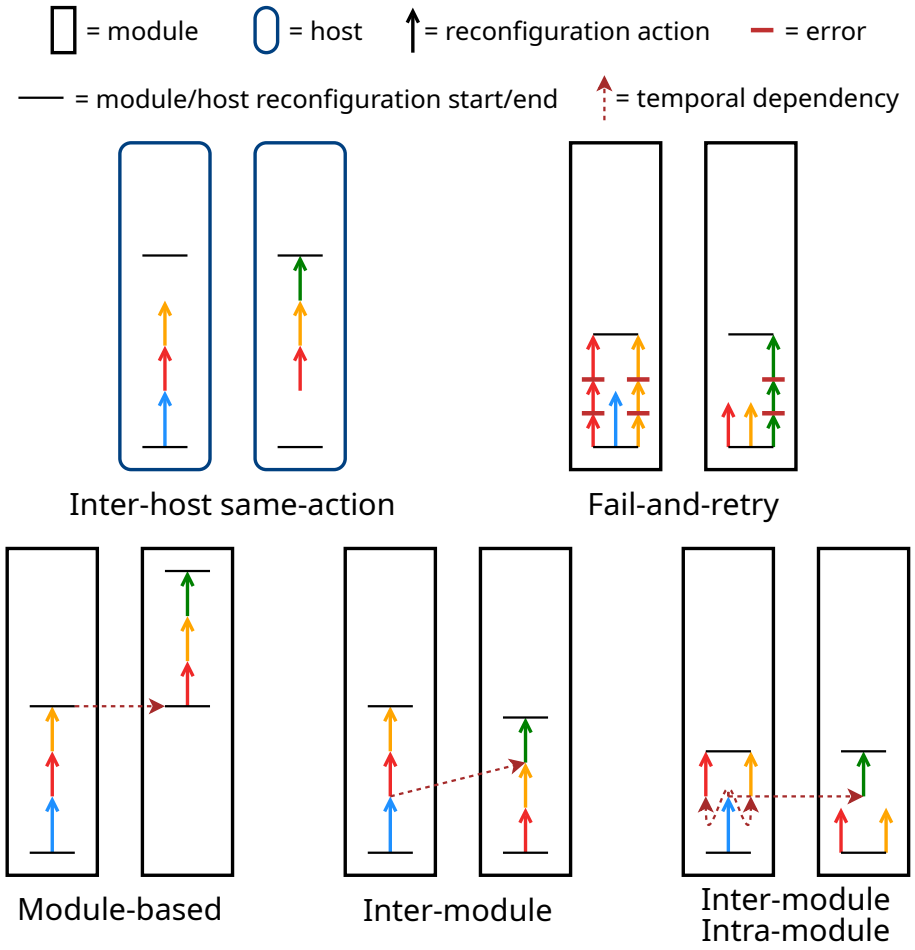



Figure 3.2: Different levels of parallelism. Arrows represent reconfiguration actions to be executed, and the vertical axis represents time in each of the sub-figures. We consider that the red and yellow actions of the component on the left, as well as the green action of the component on the right have a dependency to the blue action of the component on the left. Note that *inter-host same-action* is the only level defined in terms of hosts instead of modules. This is equivalent if each module of the system is deployed on its own host, but in the general case it is less expressive.

In distributed software reconfiguration framework, parallelism at the host level exists in two forms. It can be *same-action*, in which case reconfiguration actions can be performed in parallel on multiple hosts only if they are identical. This is for example the type of parallelism given by a multi-ssh connection as offered by *cluster-ssh*. This is our first level of parallelism: *inter-host same-action*. In Figure 3.2, we can see that the actions in red and yellow are performed at the same time on both hosts, as they are identical. The second type of host-based parallelism is when which action each host execute does not matter. In this case, the level of parallelism is

determined by how the dependencies between modules are handled.

One strategy is to not handle dependencies at all, performing all actions at the same time, and relying on an error occurring when a dependency is missing. When the dependency of a module is missing, the reconfiguration actions are attempted again until no error is raised, i.e., until all the dependencies are satisfied. This is our second level of parallelism: *fail and retry*. In Figure 3.2, we can see that the green reconfiguration action of the module on the right and the red and yellow actions of the module on the left are attempted three times, the two first of which failed because the blue action of the module on the left had not yet completed.

Another strategy is to use information about dependencies to coordinate the reconfiguration actions. Depending on the precision of the modeling of the life-cycles of the modules and how dependencies are defined, we can observe three levels of parallelism.

First, when dependencies are only defined at the module level, we have the *module-based* level: modules execute their reconfiguration actions in parallel only if they have no dependency to one another. In Figure 3.2, we see that because the module on the right depends on the module on the left, the former's actions are executed only once all the latter's ones have finished executing.

Second, when dependencies between modules are defined in a fine-grained way, i.e., at the action level, we have the *inter-module* level: reconfiguration actions of two distinct modules can be performed in parallel even when a dependency exists between these modules, as long as the dependencies between actions are respected. In Figure 3.2, we see that contrary to in the *module-based* case the red and yellow actions of the module on the right can be executed directly, because they do not have dependencies. The green action can be executed just after because its only dependency, the blue action of the other component, has already finished executing.

Third, when dependencies between the actions of a single module are declared, parallelism can occur within that module. In this case, we have the *intra-module* level. In Figure 3.2, we can see that the red an yellow actions of both modules are executed in parallel when *intra-module* parallelism is provided in addition to *inter-module* parallelism.

### 3.2.5 Separation of concerns

We define three categories of actors which may interact with a reconfiguration framework for distributed applications (recall that deployment is a kind of reconfiguration):

- Module developers - They are experts in a type of module used in distributed software (e.g., databases). They develop and package the module to conform to the requirements of the reconfiguration framework, and they provide documentation to use the module.

- Reconfiguration developers - They are experts in making modules work together

to form a distributed application. They write reconfigurations to deploy the system, but also to make it evolve depending on new specifications. They provide the reconfigurations (possibly parametric) with documentation. Generally such a role is carried out in a company by a DevOps engineer.

- System administrators - They make distributed systems work over an infrastructure they own.

Harold Ossher and Peri Tarr define separation of concerns in [57] to be the ability to identify, encapsulate and manipulate only those parts of software that are relevant to a particular concept, goal or purpose. We argue that in distributed systems, which involve many types of human actors with distinct fields of expertise, each type of actor should manipulate only elements of a reconfiguration solution which are directly relevant to the goals and purposes specific to their areas of expertise.

Ideally, in our case, reconfiguration developers should not have to look at the modules' implementation, and the documentation provided by the module developers should have minimal information about the modules' internals. Similarly, system administrators should not have to look at the details of the reconfigurations provided by the reconfiguration developers.

### 3.2.6 Formal modeling

Formal modeling allows to define properties on the reconfiguration process in a formal way and prove them without having to execute it. This make it possible to provide guarantees to the reconfiguration developers or system administrators, such as the termination of a reconfiguration or the conformity of the system to some invariants when performing reconfiguration. We consider here the modeling of the reconfiguration framework itself, not the modeling of the code of the modules provided by the module developers.

We distinguish three levels of modeling in the literature:

- No modeling - The users are free to perform basically anything, so that it is not possible for the framework to provide any guarantees.

- Abstraction - The users manipulate documented concepts such as nodes, components and pre-defined reconfiguration actions. While this is not enough to provide formal guarantees, the possible outcomes are limited and easier to envision.

- Formal model - The users manipulate concepts which are formally defined and obey a formally defined semantics so that it is possible to precisely predict the outcome of executing a reconfiguration on a system. Note that this is at the framework level and does not need to predict, for instance, hardware faults.

## 3.3 Configuration management tools

Configuration management (CM) tools are widely used in the industry and are an essential part of the DevOps[36] techniques. They allow system administrators to deploy and configure virtual infrastructures or software on physical infrastructures or in the cloud. We categorize CM tools using two criteria. The first is one of our analysis criteria: which types of elements can be reconfigured. The following options exist in the literature:

- Software only: In this case, they are called Software Configuration Management (SCM) tools and do not have built-in support for containers or VMs. However, one could argue that it is possible to encapsulate infrastructure management with code.
- Containers only: Everything that is manipulated is encapsulated into containers which are run using a container service.
- VMs only: These are usually the services provided with hypervisors by cloud providers to allow infrastructure managers to define their infrastructure as code.
- A combination of the above.

The second criteria to categorize CM tools is their declarative or imperative nature. Most of them are declarative, meaning that the users define their target system, and the reconfiguration is inferred from the difference between this target system and the currently existing system. However, some SCM tools are imperative, meaning that the users declare the set of reconfiguration actions to be performed directly.

Because they work in a substantially different way compared to the other CM tools, we start by presenting imperative SCM tools. Then, we introduce declarative CM tools (including declarative SCM tools).

### 3.3.1 Imperative SCM tools

**Ansible** Ansible [58] is an imperative SCM which is built on top of SSH. An Ansible reconfiguration consists in a sequence of instructions to be executed on one or more hosts. Such a sequence is called a *playbook*. Each instruction represents a reconfiguration action and states a type of instruction (called *module* in Ansible, but this notion is distinct from what we call *module* in this document), parameters and the name of the group hosts on which it should be executed. The groups of hosts are defined by the system administrator prior to the execution of a playbook. Sequences of instructions can be packaged into roles, which is the Ansible equivalent of modules. Instruction types are used to abstract away the underlying operating system and offer *idempotent instructions* for the most common configuration operations. For example, the *mount* instruction type takes as parameter which directory should be mounted to

which path, but the actual mounting operation will not be applied if the directory is already mounted. Instruction types can be added by the user, or the *shell* instruction type can be used to perform operations using shell commands directly. When an instruction must be executed on multiple hosts, it is done in parallel. However, two distinct instructions are always executed sequentially.

Ansible can be seen as an abstraction of SSH with syntactic sugar for the most common commands. It therefore reconfigures software. Even though dedicated instruction types exist to start or manage *Docker* containers, containers or virtual machines are not treated as first class elements. Because there is no restriction on the commands that can be executed, all types of reconfiguration operations are possible. The life-cycle of the modules are equivalent to *custom-seq multi-path*. The parallelism is *inter-host same-action*. Because any code may be executed when performing reconfiguration actions, the separation of concerns is limited. This is because Ansible playbooks do not have information about the current state of the system, and because the instructions may be low-level system instructions, the system administrator must ensure that executing the playbook will actually do what is expected. Ansible's idempotent instructions help in certain cases, but using only idempotent instructions reduces greatly the expressive power of Ansible. To the best of our knowledge, no formal operational semantics exists for Ansible, and because any instruction can be performed by the playbooks there are no abstractions to reason about besides the playbooks themselves.

**Chef** Chef [59] is another imperative SCM tool, with a more complex architecture than Ansible. It has a server-client architecture, and uses recipes, the Chef equivalent of Ansible playbooks. Recipes are submitted to a central server after being tested by an additional element, the *workstation*. The clients can then retrieve the recipes from the server when they need to apply them, and the execution is performed locally instead of remotely. Chef allows to define dependencies (to other recipes) inside recipes, which is not the case of Ansible (in Ansible all the reconfiguration actions must be declared in the same playbook). When a recipe B which depends on a recipe A is executed by a client, it can retrieve the recipe A directly from the server and execute it before B.

Similarly to Ansible, any kind of reconfiguration action which can be defined by code may be used, and therefore there is not restriction on the types of reconfiguration operations. In terms of life-cycle handling, it is also equivalent to *custom-seq multi-path*. When it comes to parallelism, the execution of the reconfiguration tasks by distinct hosts are independent, because they each have their own client. However, if a dependency exists between recipes (which can be considered as modules), one must wait for the recipe that is depended on to be fully executed before the other recipe's execution. We therefore have *module-based* parallelism. When it comes to separation of concerns, one might argue that it is better than Ansible's because of the dependencies between playbooks. However, the main issue of low-level reconfiguration

actions that are not aware of the context remains. There is no formal model nor abstractions to reason about besides recipes.

### 3.3.2 Declarative CM tools

We start by introducing the TOSCA standard, which defines many concepts which are also used by most of the declarative CM tools with small variations. We then introduce the other solutions, comparing them to TOSCA when it makes sense.

**TOSCA** TOSCA [60] (Topology and Orchestration Specification for Cloud Applications) is an OASIS standard for the deployment and reconfiguration of cloud applications and has multiple implementations, such as OpenTOSCA [61] or Cloudify [62]. The system administrator describes the target architecture in a graph where each *node* represents an element, which may be virtual machines, containers, software or other things depending on the implementation. Note that in the following, a *node* refers to a TOSCA node, corresponding to a software module. Each edge between nodes represents a *relationship* (e.g., a node representing a VM contains a node representing an OS). The nodes and the relationships are typed, and each type of node comes with a set of *artifacts* describing its deployment and destruction processes, and possibly other operations that can be performed on the node.

The TOSCA standard does not restrict what elements may be represented as nodes. Cloudify, for example, can represent VMs on popular public and private cloud infrastructures, containers and software. It is also extensible so that custom node types can be created. TOSCA supports the addition and removal of elements and offers the possibility to define custom node operations, i.e., reconfiguration actions to perform on a node in addition to deployment and removal. By default, TOSCA defines a *fixed-10* life-cycle with the following states for each node: *initial*, *creating*, *created*, *configuring*, *configured*, *starting*, *started*, *stopping*, *deleting* and *error*. Also, by default, dependencies may only be precise up to the node level, which means that this life-cycle is not used to increase the parallelism between reconfiguration tasks by default. TOSCA does not specify precisely how these life-cycles should be orchestrated and leave it to the implementation. Cloudify for example defines a default workflow (i.e., orchestration of reconfiguration actions) for node startup and termination, which executes the different steps sequentially for each node in parallel, unless a dependency exists, in which case the node that is depended on must finish its execution before the other one can start its own. This corresponds to *node-based* parallelism. Custom workflows may be defined by the users, but they must use information which is not provided in TOSCA to improve the parallelism, i.e., information contained in the implementation of the node. In this case, this corresponds to *inter-module* parallelism. When performing reconfiguration, the description of the target infrastructure is made by system administrators using nodes provided by module developers. They can use templates written by reconfiguration developers to simplify the process. The separa-

tion of concerns in TOSCA is therefore very high between these actors when using the default workflows. When using custom workflows, the reconfiguration developer (or system administrator) must know the details of the nodes' implementation, resulting in a low separation of concerns. When it comes to formalism, TOSCA defines abstractions which can be used to reason about the reconfiguration process like nodes and relationships. But TOSCA does not come with an operational semantics, which must be defined by its implementations. However, external work has been done to model formally the TOSCA standard [63, 64], which can give guarantees if the users take the time to formalize the implementation of each TOSCA node used.

**Brooklyn** Brooklyn [65] is a declarative CM tool developed by Apache. While it is not an implementation of TOSCA, it is very similar to it and uses the same concepts as TOSCA, including user-defined workflows with their advantages in terms of parallelism but disadvantages in terms of separation of concerns. By default, the life-cycles are *on/off*, however. Other than that, the analysis is the same as TOSCA's.

**Juju** Juju [66, 67] is a declarative CM tool developed by Canonical. Similarly to Brooklyn, Juju is not an implementation of TOSCA but uses similar concepts, without user-defined workflows and with an *on/off* life-cycle. The analysis is therefore the same as Brooklyn's with the default workflows.

**Terraform** Terraform [68] is an open-source CM tool which focuses on virtual infrastructure (VMs and containers). A graph similar to what is done using TOSCA is defined by the infrastructure manager using a DSL (effectively implementing the infrastructure as code paradigm). It does not come with software support, but it is extensible so that custom nodes can be defined. MySQL is supported out of the box however. Additionally, it does not support custom node operations but only addition and deletion. Other than this, the analysis is the same as TOSCA's with the default workflows.

**PaaSage** PaaSage [69, 70] is an academic project which led to the development of a platform for the deployment and the management of cloud applications. It essentially works in a similar way to Terraform, except that its DSL, CAMEL [71], allows for additional steps in the development process to increase separation of concerns when it comes to cloud providers. The nodes representing virtual machines may correspond to abstract specifications, which can then be replaced automatically by matching virtual machines offered by the chosen cloud provider. Other than this, the analysis is the same as Terraform's.

**AWS CloudFormation** Amazon Web Services (AWS) CloudFormation [3] is a CM tool specific to AWS and allows to define an infrastructure running on Amazon's

cloud. This infrastructure is defined as a graph, similarly to TOSCA, using a DSL. However, the elements that can be reconfigured are only AWS VMs, and they may only be added or removed. It offers support for autonomic policies such as auto-scaling or rolling updates which can be defined by using specific node types provided by AWS, however this is not in the scope of this state of the art. Other than this, the analysis is the same as TOSCA's with the default workflows.

**OpenStack Heat** OpenStack Heat [4] is similar to AWS CloudFormation, but targets infrastructures provided by an instance of OpenStack instead of AWS. It uses its own DSL called HOT (Heat Orchestration Template) to define the expected infrastructure and define autonomic behavior. The analysis is the same as AWS CloudFormation's.

**Kubernetes** Kubernetes [72] is an open-source container orchestration service maintained by Google. It allows to define container-based infrastructures in a declarative way using its DSL, similarly to TOSCA. However, only containers are supported as reconfigurable elements. Custom node operations are implemented using container access points, or simply execution of arbitrary commands inside the containers. The fixed-5 life-cycle of each container is managed by a *pod*, which can be in the following states: *pending*, *running*, *succeeded*, *failed*, *unknown*. Kubernetes uses the *fail-and-retry* strategy when it comes to parallelism. Separation of concerns is limited when performing custom node operations, because this is done by executing arbitrary code inside the VMs. Finally, Kubernetes provides abstractions like *pods*, *resources* and *controllers* to reason about reconfiguration, but no formal semantics is provided.

**Docker Swarm** Docker Swarm [73] is Docker's own container orchestrator. When it comes to our analysis criteria, Docker Swarm behaves similarly to Kubernetes and therefore the analysis is the same as Kubernetes'.

**MoDEMO** MoDEMO [74] is a reconfiguration framework for cloud computing. It focuses on elasticity management (i.e., scaling) of virtual machines and containers while being agnostic to the cloud provider thanks to the use of OCCI [40].

MoDEMO can reconfigure containers and virtual machines. It supports addition, removal (and from a planning viewpoint, scaling, which in terms of execution comes down to adding and removing modules). The life-cycle handling is the one of OCCI, i.e., *on/off*. The parallelism is *module-based*. In terms of separation of concerns, because only scaling is permitted, reconfiguration developers and infrastructure managers do not need to know implementation details, so there is a high separation of concerns. MoDEMO has been formalized in UML, which corresponds to documented abstractions, as to the best of our knowledge there is no formalization of the execution semantics.

**Puppet** Puppet [75] is a platform-independent SCM tool (the elements that can be reconfigured are only software). It has a similar architecture to the one of Chef, i.e., a master/worker architecture. Users describe the expected status of the software as a *manifest*, which is similar to a TOSCA graph, written in Puppet's own DSL. *Modules*, similar to TOSCA node types, are available for the system administrator to use. The manifest is then sent to a server, and each node in the system to configure downloads it and applies the necessary changes. This is done by the *Puppet agent* which must be installed on all the nodes. Puppet agents are similar to *Chef* agents, but work in a declarative way instead of an imperative one. It is possible to execute custom shell commands however if necessary, effectively guaranteeing that all reconfiguration types can be performed. Puppet has an *on/off* life-cycle for the software elements. Similarly to *Chef*, it has *module-based* parallelism. It has good separation of concerns when not using custom shell commands (if used, it is not only about reaching a declared state but executing custom code). It has not been formalized and provides only abstractions.

**SaltStack** SaltStack [76] is similar to Puppet, the main difference being the language used to describe an architecture (YAML and Python instead of Ruby), and the persistent TCP connection kept between the Master node and the Worker nodes. The analysis is therefore the same as Puppet's.

**Jolie Redeployment Optimiser** Jolie Redeployment Optimiser [77] (JRO) is a tool to deploy and re-deploy modules described in the Jolie language [78, 79]. JRO's major contribution is to provide an optimal deployment (adding modules) or re-deployment plan (adding or removing modules) given some metrics.

In terms of execution, it handles *software* modules described in Jolie. It is able to add and remove modules. The modules have an *on/off* life-cycle and the parallelism is *module-based* (dependencies can be declared between modules). The separation of concerns is high as reconfiguration developers and software administrators do not need to know the implementation details of the modules. While Jolie has been formalized, the JRO provides abstraction but not a formalism of its execution semantics.

## 3.4 Control component models

An other approach taken in the literature is to detail the life-cycle of the modules, possibly using them to improve on parallelism. We observed that this is done using component-based approaches, which leads to good properties such as composability, reusability and separation of concerns. The components are not used to describe the functional parts of the modules but focus on their control, i.e., modeling and handling their life-cycles. In the following, we call these components *control components*.

**Jade** Jade [80] is a middleware to manage distributed applications. It requires each piece of software to be wrapped in a Fractal component so that they all have a uniform interface. A module developer is in charge of wrapping legacy code with a Fractal component to implement the common interface. This way, that piece of software can be managed, and in particular started and stopped. Introspection and reconfiguration can be performed thanks to Fractal's reconfiguration capabilities.

The elements that can be reconfigured in Jade are the pieces of software that have been wrapped inside Fractal components. Thanks to Fractal's reconfiguration capabilities, components may be added or removed. Custom operations may be defined by exposing additional interfaces compared to just those required by Jade. The life-cycle of Jade components is *on/off*. It is possible to manually encode a more complex life-cycle inside the membrane of a Fractal component, i.e., an area of the component dedicated to non-functional concerns, but this is not taken advantage of by the framework. This on/off life-cycle induces *module-based* parallelism. In terms of separation of concerns, Jade brings to legacy software some good properties of component-based software engineering. By increasing composability and reusability, reconfiguration developers do not have to worry about individual module's business code. However, module and reconfiguration developers are required to learn Fractal to be able to respectively produce a wrapper for the modules and write reconfigurations. Finally, work has been done to write formal specifications for Fractal [81]. Additionally, while vanilla Fractal exposes low-level APIs for reconfiguration, which is error-prone and difficult to check as they are called from user code, one can benefit from contributions to the Fractal ecosystem like FScript [82], a DSL which allows to express Fractal reconfigurations while providing some guarantees such as termination.

**Tune** Tune [83] is an evolution of Jade and addresses the complexity for component developers to wrap legacy code into Fractal components. This is done by providing a DSL called *wrapping description language* (WDL) to describe the wrappers, as well as a UML profile to describe reconfigurations. Even though the WDL concepts are ultimately translated to Fractal, the user does not have to know and understand Fractal.

Tune makes the work of component developers easier and improves separation of concerns compared to Jade by not requiring module nor reconfiguration developers to know Fractal. Other than this, the analysis is the same as Jade's.

**DeployWare** DeployWare [84] is another wrapping-based middleware. Similarly to Tune, a DSL allows software experts to define the wrapper for a piece of software. However, DeployWare manages the life-cycle of the modules by allowing developers to write *procedures* for each component. These procedures must be sequential (there is only one sequence of procedures to go from the undeployed state to the running state and conversely) and symmetric (if a procedure leads from a state A to a state B, a procedure leading from state B to state A must exist, e.g., *install*/*uninstall*). A

DeployWare assembly can then be defined by a user, which is automatically transformed to a Fractal assembly handling all of the orchestration of the procedures to deploy the full assembly.

According to our analysis criteria, DeployWare is similar to Tune. Two differences exist though. First, the DeployWare framework hides the actual Fractal components. DeployWare itself only allows for *deployment and termination of the application*, it is not possible to remove only one part of it. Second, the granularity of the life-cycle of the control components, which is *custom-seq single-path*. However, this does not affect the parallelism, as two components may only be deployed sequentially (if there is a dependency) or in parallel (if not), which corresponds to *module-based* parallelism. Separation of concerns is high. No formal semantics is provided, but there are abstractions.

**Adage** Adage [85, 86] is a deployment framework for applications on computation grids. It features a DSL called GADe, allowing users to describe their grid application to be deployed (directly in GADe, or by using another DSL specific to a programming model, like MPI, which is then translated to GADe). Taking as input the description of the application in GADe, the description of the available resources and control parameters, it generates a deployment plan and executes this deployment. Even though Adage was developed for grids, it could be adapted for more general use.

Adage can deploy general *software* described in GADe. It also supports plugins to provide technology-specific support, i.e., *custom elements*. It supports the *deployment* and *addition* of modules. Each module has a *fixed* sequential deployment life-cycle (*single-path*). In terms of parallelism, the first part of the deployment (sending files) is done in *inter-host same-action* fashion, while the execution of processes is done in a *module-based* fashion. The separation of concerns is high as the declaration of the available resources, the creation of a plugin for a specific technology and description of an application are completely distinct. Finally, Adage has not been formally, but abstractions are provided thanks to GADe.

**SmartFrog** SmartFrog [87, 88, 89] (Smart Framework for Object Groups), created by HP and later open-sourced, is a framework in which software components include a functional part called *managed entity*, some *configuration* data and a *life-cycle manager*, which corresponds to our notion of control component. A life-cycle manager is a Java class which represents a state-machine and exposes methods corresponding to transitions in this state-machine. These methods can then be called by a scheduler which manages the life-cycles of a set of components. Data dependencies between components are handled through *lazy bindings*, which can be resolved at run-time.

SmartFrog is able to reconfigure software elements provided with a life-cycle manager. The creation and deletion of elements as well as performing custom operations is done by instantiating Java classes and calling their methods. SmartFrog provides *custom-seq multi-path* life-cycles under the form of state-machines. SmartFrog does

not come with schedulers which must be implemented by the user. In that sense, the user is responsible for handling the fine-grained life-cycle dependencies between the components. The lazy bindings mechanism, if used correctly by the programmer, allows a component to wait until some information is available. Because of this, we can consider SmartFrog to feature *inter-module* parallelism. Separation of concerns is quite high as what is exposed to the reconfiguration developer are the state-machines of the life-cycle managers produced by the module developers, provided appropriate documentation comes with the Java classes. However, this state-machine still exposes some of the internals of the modules, and unless every time-dependency is encoded as a lazy binding, which is not necessarily the case, the reconfiguration developer needs to know the time-dependencies between the modules' life-cycles to write an appropriate scheduler. Finally, some work has been done to formalize the core of SmartFrog [90].

**Engage** Engage [91] is an academic framework which has similar concepts to SmartFrog's. In particular, the modules are associated with a state-machine to manage their life-cycles called *driver*. Drivers contain at least three special states: *uninstalled*, *inactive* and *active*. The component developer is then free to add addition states, as long as there always exists a single path going from either one of these special states to another. The modules are put together to from an application in a hierarchical way. The transitions correspond to actions described with code to change the state of the modules and can be guarded by a requirement of the form: all upstream/-downstream modules must be in this special state. Engage also has mechanisms to increase separation of concerns by deriving full specifications of an application from partial ones, but this is part of the planning phase and not the execution phase of the autonomic loop, so it is out of our scope.

What differentiates Engage from SmartFrog is the nature of the life-cycles (*custom-seq single-path* instead of *custom-seq multi-path*) and how dependencies are handled: some dependencies are declared between modules themselves, resulting in *module-based* parallelism, and some dependencies are declared using the three special states of the life-cycles of the modules in the same branch as the module considered, resulting in a restricted form of *inter-module* parallelism. Note that using the latter reduces the separation of concerns as the module developers needs to think about how the modules will be composed with other modules when declaring their life-cycles. Finally, the semantics of Engage has been formally defined.

**Aeolus** Aeolus [92] is another framework with similar concepts to SmartFrog's. The life-cycle of each module is modeled by a state-machine inside a control component. Each state of the life-cycle corresponds to an action being performed by the module. One major difference with Engage though is that dependencies are represented by ports of the component, which are themselves associated with states of the life-cycle. This allows to declare in the model the part of the module's life-cycle in which the

dependency is used. Reconfigurations are then performed using a DSL allowing to add and remove components, and change the current state of the component following a transition declared by the module developer.

Each control component models the life-cycle of a software module. The reconfiguration DSL allows to add and remove modules, and also to change the current state of existing modules. This allows in particular to perform custom module operations. The life-cycle handling is *custom-seq multi-path*. Because ports represent dependencies between parts of the life-cycles, Aeolus features *inter-module* parallelism. However, the scheduler which applies reconfiguration actions using the DSL needs to take advantage of this. Similarly to SmartFrog, the separation of concerns is pretty high given that the reconfiguration developers are provided with a state-machine for each module and their dependencies. However, this state-machine still exposes some internals of the module. Aeolus has been fully defined in a formal way.

## 3.5 Analysis

We have presented two main types of contributions in the literature. First, in Section 3.3 we have introduced configuration management (CM) tools, which we can split into two groups: imperative CM tools and declarative CM tools. Second, in Section 3.4, we studied what we designate by *control component models*, i.e., models which provide functional modules with life-cycle managers which we call *control components*. In this section, we discuss the state of the art as a whole to try and identify trends and weaknesses. Table 3.1 summarizes the analysis presented before according to the criteria given in Section 3.2.

### 3.5.1 The special case of imperative SCM tools

Imperative software configuration management tools stand out because the life-cycles they define are composed of sequences of operations to apply, either on one or multiple hosts. They simply execute sequences of reconfiguration commands, called playbooks in the case of Ansible and recipes in the case of Chef. Because these commands execute concrete actions on real machines, separation of concerns is low as reconfiguration developers need to check that the code applying to different modules will not interfere with each other, and system administrators need to ensure that the code will actually perform the intended actions. These tools can be considered to be at a lower level compared to other solutions, meaning that they could be used by these other solutions as a mean to execute commands on actual machines.

Finally, among the selected solutions, we can notice the uniqueness of Ansible in terms of parallelism, because it is the only one to have *inter-host same-action* parallelism.

| Solution | Reconfigurable | Types of rec. | Life-cycle | Parallelism | Sep. of conc. | Formalism |
|---|---|---|---|---|---|---|
| Ansible | software | all | custom-seq multi-path | inter-host same-action | low | no modeling |
| Chef | software | all | custom-seq multi-path | module-based | dependencies | no modeling |
| TOSCA (default) | implem. dep. | all | fixed-10 | module-based | high | third-party |
| TOSCA (custom) | implem. dep. | all | custom-seq multi-path | inter-module | module-based dep. | third-party |
| Brooklyn (default) | all | all | on/off | module-based | high | abstractions |
| Brooklyn (custom) | all | all | custom-seq multi-path | inter-module | module-based dep. | abstractions |
| Juju | all | all | on/off | module-based | high | abstractions |
| Terraform | VMs, containers (extensible) | addition, deletion | on/off | module-based | high | abstractions |
| PaaSage | VMs, containers (extensible) | addition, deletion | on/off | module-based | high | abstractions |
| CloudFormation | AWS VMs | addition, deletion | on/off | module-based | high | abstractions |
| OpenStack Heat | OpenStack VMs | addition, deletion | on/off | module-based | high | abstractions |
| Kubernetes | containers | all | fixed-5 | fail-and-retry | high | abstractions |
| Docker Swarm | containers | all | fixed-5 | fail-and-retry | high | abstractions |
| MoDEMO | VMs, containers | addition, deletion | on/off | module-based | high | abstractions |
| Puppet | software | all | on/off | module-based | high (no custom com.) | abstractions |
| SaltStack | software | all | on/off | module-based | high (no custom com.) | abstractions |
| JRO | software | addition, deletion | on/off | module-based | high | abstractions |
| Jade | software | all | on/off | module-based | high | third-party |
| Tune | software | all | on/off | module-based | high | third-party |
| DeployWare | software | deployment, termination | custom-seq single-path | module-based | high | abstractions |
| Adage | software | addition | fixed | module-based / inter-host same-action | high | abstractions |
| SmartFrog | software | all | custom-seq multi-path | inter-module | internals exposed | formal semantics |
| Engage | software | all | custom-seq single-path | module-based / inter-module | internals exposed | formal semantics |
| Aeolus | software | all | custom-seq multi-path | inter-module | internals exposed | formal semantics |

Table 3.1: Comparison of the solutions of the literature.

### 3.5.2 Correlations between analysis criteria

We can observe connections between the different analysis criteria presented in Section 3.2 in Table 3.1. In the following, we exclude imperative SCM tools which have been covered previously.

**Life-cycles and separation of concerns** We notice that there is a strong correlation between how life-cycles are handled and separation of concerns. All the solutions with very high separation of concerns feature *on/off* or *fixed-n* life-cycles. This can be explained by the simplicity introduced by a *fixed-n* life-cycle for the software modules: they all have the same interface. Moreover, when the life-cycle is not *on/off*, we observe that the parallelism is either *module-based* or *fail-and-retry*, meaning that it is actually treated as *on/off* when it comes to dependencies between modules. Because the dependencies clearly refer to when the module is on, the startup procedure can be hidden to the reconfiguration developer, resulting in a high separation of concerns.

**Life-cycles and parallelism** Because *inter-module* parallelism requires by definition for software modules to have at least 3-state life-cycles, all the solutions with *inter-module* parallelism need to have *fixed-n* or *custom-seq* life-cycle models. In practice, they all have *custom-seq multi-path* life-cycles. Note that the more precise and custom the life-cycles can be for each module, the more precise the dependencies can be between their life-cycles, and therefore the more opportunities for parallelism there are. We can notice however that not all that have these kinds of life-cycle models feature *inter-module* parallelism, such as Deployware for instance. We can also notice that none of the solutions presented, and to the best of our knowledge no solution in the literature, features *intra-module* parallelism, which requires *custom-par* life-cycles (in which reconfiguration actions of a single module can be executed in parallel).

### 3.5.3 Problem: how to reconcile separation of concerns and performance?

We can observe that in the literature, separation of concerns and performance seem to be incompatible. Declarative CM tools in general seem to favor separation of concerns by providing simple and mostly unified interfaces for software modules, which abstracts away their complexity and the one of their life-cycles. This results in more separation of concerns and less things to worry about to execute reconfigurations. This simplicity is important for declarative CM tools, which in addition to the execution of reconfiguration also handle the monitoring, analysis and planning phases of the autonomic loop to offer features such as auto-scaling, auto-recovery or rolling updates.

In the case of control component models, multiple compromises have been made. While Jade, Tune and Deployware only provide *module-based* parallelism, SmartFrog,

Engage and Aeolus provide different degrees of *inter-module* parallelism, introducing better performance at the cost of more complex module interfaces and lower separation of concerns.

In this thesis, we address the problem of providing a framework for the execution of reconfiguration which shows good separation of concerns between the different actors while improving performance over the state of the art by providing parallelism of reconfiguration tasks inside modules, in addition to between modules. Formal specifications and semantics are also targeted in this thesis.

## 3.6 Conclusion

In this chapter, we have first set the scope of the thesis and excluded some works addressing other aspects of reconfiguration. We then have presented a set of analysis criteria, and used them to analyze the related work.

We have seen that a high level of parallelism expressivity in reconfiguration is never associated with a high level of separation of concerns. This is due to the complexity introduced by custom per-module life-cycles. Additionally, we have noticed that none of the solutions achieve the maximum level of parallelism expressivity that we have identified, when reconfiguration actions can be performed in parallel both between and inside modules, using fine-grained life-cycle dependencies to perform actions in parallel even if one module depends from the other. In conclusion, we have observed the lack of a general solution for reconfiguration which provides both this high level of parallelism expressivity, and a high level of separation of concerns between module developers, reconfiguration developers and infrastructure administrators.

# Chapter 4

# The Madeus Deployment Model

## Contents



This chapter presents our first contribution: the Madeus deployment model. The first objective of Madeus is to allow module developers to define the life-cycle of software modules during their *deployment*. The second objective is to coordinate the deployment of a distributed application composed of these modules efficiently. The third objective of is to have a high separation of concerns between the module developer and the deployment process developers. Finally, Madeus' concepts and semantics are defined formally.

In the first section we give an overview of Madeus based on a simple example. Then, in the second section we define all the concepts of Madeus formally. Finally, in the third section we present a performance model for Madeus.

**Work context** At the beginning of my PhD, an informal prototype of Madeus had already been developed by the team, along with a basic implementation. My contribution was to define a formal model for Madeus, along with a performance model (presented in sections 4.2 and 4.3 respectively). This led to some significant changes, and ultimately I also wrote a new implementation.

## 4.1 Overview

Madeus is a control component model, in the sense introduced in Chapter 3. The deployment life-cycle of each software module is managed by a *control component* (*component* in the following). Considering a distributed application to be deployed, Madeus coordinates the deployment life-cycles of its modules by executing an *assembly* made of the components managing these modules. In this section, we introduce the concepts used by Madeus in an informal way, based on an example. This example consists in deploying a database and a server which uses this database and is illustrated by Figure 4.1.

### 4.1.1 Component

In Madeus, each module is managed by a *component*, with its deployment life-cycle encoded by a Petri-net like structure called *internal-net*. An *internal-net* is composed of *places*, representing "milestones" in the deployment process, and *transitions* representing actions that must be executed to go from one place to another. The component's places which have no incoming transitions are called *initial places*. They correspond to the state in which the module is not deployed nor being so. It is important to realize that while in many component models components contain operational code, i.e., address the functional aspects of the modules they represent, in Madeus the components only contain code that controls their life-cycles. Madeus is therefore agnostic to the technology used to implement the modules themselves, as long as they provide an API to interact with their life-cycle.

Let us analyze the example pictured in Figure 4.1. There are two components representing respectively a database (`db`) and a server (`server`). The `db` component has three places (`undeployed`, `allocated` and `running`) and two transitions (`allocate` and `run`). The `server` component has five places (`uninstalled`, `installed`, `configured`, `running` and `providing`) and five transitions (`ins`, `conf1`, `conf2`, `run` and `wait`). Notice that transition `conf1` on the one hand, and transitions `ins` and `conf2` on the other hand can be executed in parallel. In this example, the transitions may be associated with scripts performing deployment tasks of the database and the server.

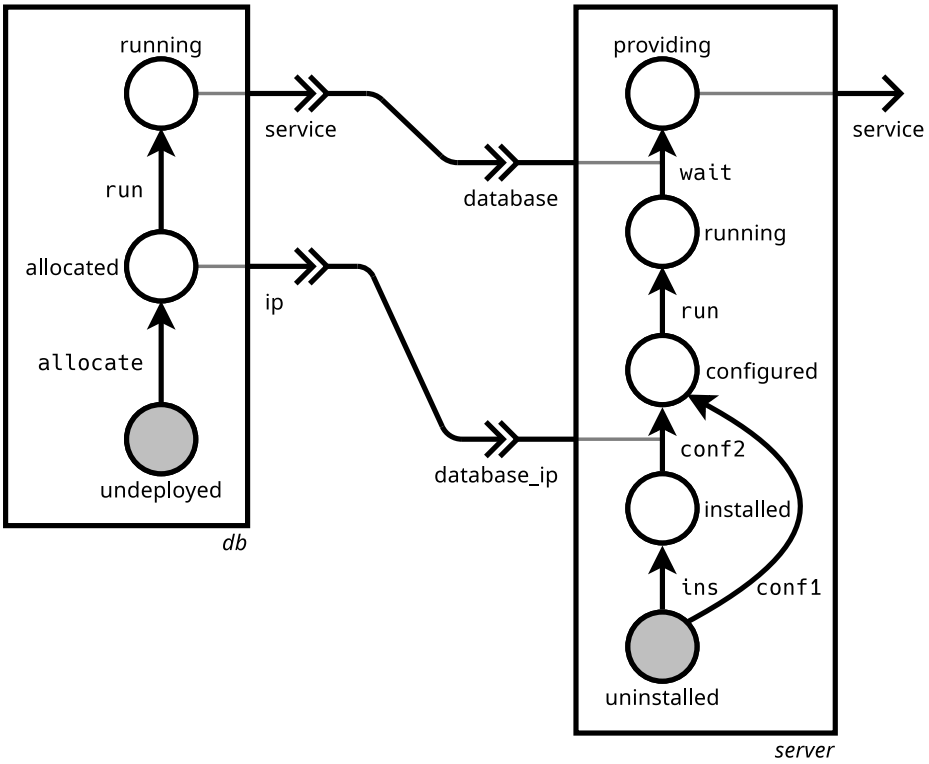



Figure 4.1: Madeus assembly describing the deployment procedure of a server and the database used by the server. Components are represented by rectangles. Inside components, places are represented by circles, initial places being filled with gray, and transitions are represented by arrows between places. Outside components, provide ports are represented by arrows and use ports are represented by inverse arrows.

For example, in the `db` component, the `allocate` transition may be associated with a script allocating a virtual machine to host the database (and boot it with a disk image with the database already installed), while `run` would start the service. In the `server` component, `ins` would be associated with a script installing packages such as *Apache*, `conf1` with a script performing configuration steps which do not need for the packages to be installed, `conf2` with a script finishing the configuration, etc.

In Madeus, dependencies to other deployment life-cycles are represented by *coordination ports* (in the following: *ports*), which differ from the usual notion of port in component models by the fact that they are used for coordination instead of data transfers or remote calls. Use ports are associated with transitions and correspond to a requirement that must be fulfilled by another Madeus component before these transitions can execute. Usually, this requirement corresponds to the other component being able to provide some piece of information (e.g., an IP address) or to some kind of service being served by the software module it represents (e.g., a MySQL server

accepting connections). Provide ports correspond to a signal that a component has reached a given milestone in its deployment process, and is therefore capable of fulfilling some kind of requirements, such as providing an IP address or ensuring that some service is running.

Coming back to the example of Figure 4.1, component `db` has two provide ports `ip` and `service`, respectively associated with places `allocated` and `running`. Component `server` has two use ports `database_ip` and `database`, respectively associated with transitions `conf2` and `wait`, and one provide port `service` associated with place `providing`.

### 4.1.2 Assembly

A Madeus assembly is a set of components for which each provide port may be connected to one or more use ports. A connection between a use port and a provide port associates the dependency (represented by the use port) to the signal ensuring it has been fulfilled (represented by the provide port). An assembly effectively models the deployment of distributed software.

In Figure 4.1, components `server` and `db` are connected to form an assembly. Two couples of ports are connected, indicating that transition `conf2` in component `server` may only be executed once place `allocated` has been reached in component `db`. Similarly, transition `wait` in component `server` may only be executed once place `running` has been reached in component `db`.

Notice that Madeus ports are represented using arrows and not circles (which is the UML standard to represent use/provide ports) because active provide ports in Madeus can never be deactivated, and merely correspond to a signal.

### 4.1.3 Execution

A Madeus assembly can be executed: this corresponds to performing the deployment process of some distributed software, i.e., coordinating all individual deployment tasks (represented by the components' transitions) to make it operational. At each step of the execution, the current progress in the deployment process of a Madeus component is encoded by a set of *tokens*. At each moment, a component has one or more tokens (multiple tokens correspond to parallel actions within a component). These tokens are located on specific objects of the internal-net. For this purpose, we consider the transitions to be complex objects, composed of three parts: *beginning*, *in progress* and *end*.

A token may be on:

- a place, which means that the corresponding milestone has been reached and the next deployment actions may be executed;

- the *beginning* part of a transition, which means that the action encoded by the transition is about to start its execution;
- the *in progress* part of a transition, which means that the action is being executed;
- the *end* part of a transition, which means that the action has finished its execution.

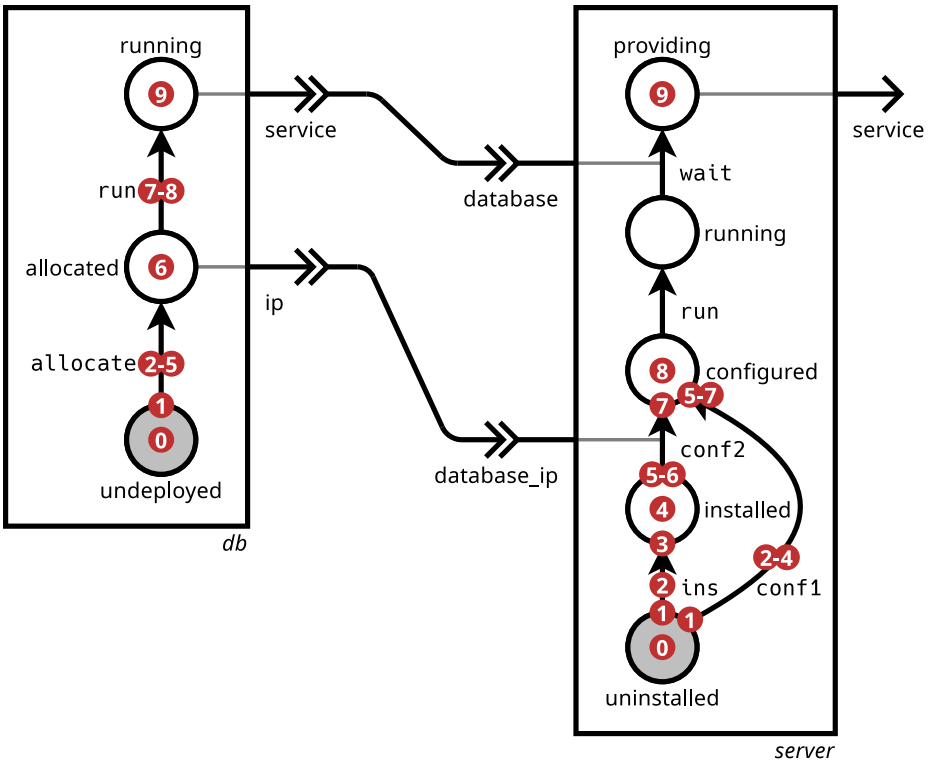



Figure 4.2: Ten snapshots of the execution of the deployment of the Madeus assembly of Figure 4.1. Each red circle is a token, and the number inside it corresponds to which snapshot it belongs to. Two adjacent circles with two numbers separated with a dash represent a token corresponding to multiple snapshots (from the first number to the second). Graphically, the tokens represented at the border of a place and a transition are either on the *beginning* part of the transition (if the place is the origin) or on the *end* part (of the place is the destination). The tokens represented in the middle of a transition are on the *in progress* part of that transition.

Intuitively, these tokens evolve in a similar way to those of Petri nets, but taking into account the specificities of Madeus. The complete operational semantics of

Madeus is given in Section 4.2, however we now present it informally through the example presented in Figure 4.2. This figure shows a series of 10 snapshots of a possible execution of the deployment process of the assembly presented in Figure 4.1:

0. At the start, one token is put in the initial place of each component.

1. When a token is in a place, it can be removed in favor of one token at the *beginning* of each transition going out of that place. In this snapshot, tokens have been removed from the `undeployed` and `uninstalled` places and some have been put at the *beginning* of their outgoing transitions.

2. When a token is at the start of a transition, it may be moved to the *in progress* part of the transition (starting the corresponding deployment action) if nothing prevents the transitions from executing. In this snapshot, transitions `allocate`, `ins` and `conf1` have started their executions.

3. When the action corresponding to a transition has finished its execution, the token on the *in progress* part of the transition may be moved to its *end*. In this snapshot, transition `ins` was finished and its corresponding token was moved.

4. When there is a token at the *end* of all the incoming transitions of a place, they may be removed, in which case a token is put on the place. This corresponds to a synchronization point for parallel actions. In this snapshot, there was only one incoming transition for place `installed` which had a token at its *end*, so it was moved to the place.

5. In this snapshot, component `server` has two tokens: one at the *beginning* of transition `conf2` and one at the *end* of transition `conf1`. None of them can be moved at the moment. The first one cannot be moved because transition `conf2` is bound to use port `database_ip`, which is connected to component `db`'s `ip` port which is itself bound to the `allocated` place. This means that as long as this place has not been reached by a token, transition `conf2` cannot be executed. The second cannot be moved because transition `conf2` should have a token at its *end* for both tokens to be removed and one put in place `configured`.

6. In this snapshot, place `allocated` in component `db` has been reached, which makes transition `conf2` in component `server` legal to execute.

7. In this snapshot, both transitions `conf1` and `conf2` have finished their execution and have a token at their *end*.

8. Because both incoming transitions to place `configured` had a token at their *end*, they were removed and one was put in the place.

9. This snapshot represents the end of the deployment process: no further actions can be executed because only places with no outgoing transitions have a token.

## 4.2 Formal Model

In the previous section, Madeus was presented informally with the help of an example. In this section, we define all the concepts used by Madeus formally and present the operational semantics used to perform a Madeus deployment. These concepts as well as the notations we use to refer to them in this chapter are gathered in Table 4.1 for convenience.

### 4.2.1 Component

A Madeus control component can be seen as an *internal-net* (comprised of places and transitions) and an *interface* (comprised of use and provide coordination ports). The ports are bound to places and transitions, connecting these two parts. Formally, a component is defined as a couple $(N, I)$ where $N = (\Pi, \Theta)$ is an internal-net and $I = (P_\mathrm{u}, P_\mathrm{p}, \rightleftharpoons_\mathrm{u}, \rightleftharpoons_\mathrm{p})$ is the interface and its bindings.

#### 4.2.1.1 Internal-net

We consider a set $\mathbb{A}$ of deployment actions that users can define using a given implementation of Madeus. This set is common to all components and may correspond, for example, to the set of functions that can be written in a given programming language.

We define the internal-net $N$ as a directed acyclic graph (DAG) $(\Pi, \Theta)$, in which each vertex is called a place ($\Pi$ is a set of places) and each arc is tagged with a deployment action and is called a transition ($\Theta \subseteq \Pi \times \mathbb{A} \times \Pi$ is a set of transitions). The set of incoming (respectively outgoing) transitions of a given place $\pi \in \Pi$ are denoted $\Theta_\mathrm{in}(\pi)$ and $\Theta_\mathrm{out}(\pi)$. Formally:

$$\Theta_\mathrm{in}(\pi) = \{(\pi_\mathrm{source}, \alpha, \pi_\mathrm{dest}) \in \Theta \mid \pi = \pi_\mathrm{dest}\}$$
$$\Theta_\mathrm{out}(\pi) = \{(\pi_\mathrm{source}, \alpha, \pi_\mathrm{dest}) \in \Theta \mid \pi = \pi_\mathrm{source}\}$$

A place $\pi$ such that $\Theta_\mathrm{in}(\pi) = \emptyset$ is said to be *initial*, while if $\Theta_\mathrm{out}(\pi) = \emptyset$, the place is called *final*.

#### 4.2.1.2 Interface and bindings

The interface of a component is comprised of a set $P_\mathrm{u}$ of use ports and a set $P_\mathrm{p}$ of provide ports. Use ports may be bound to one or more transitions, indicating that these transitions have a dependency to another module's life-cycle. The binding relation $\rightleftharpoons_\mathrm{u} \subseteq P_\mathrm{u} \times \Theta$ is such that $p_\mathrm{u} \rightleftharpoons_\mathrm{u} \theta$ if and only if $p_\mathrm{u}$ is bound to $\theta$.

Provide ports may be bound to one or more places. The binding relation $\rightleftharpoons_\mathrm{p} \subseteq P_\mathrm{p} \times \Pi$ is such that $p_\mathrm{p} \rightleftharpoons_\mathrm{p} \pi$ if and only if $p_\mathrm{p}$ is bound to the place $\pi$. The detailed semantics is given later but intuitively, a provide port becomes *active* when at least one of its bound places has been reached.

| Internal-net | |
|---|---|
| $\Pi$ | set of places of a component |
| $\Theta \subseteq \Pi \times \mathbb{A} \times \Pi$ | set of transitions |
| **Interface and bindings** | |
| $P_{\mathrm{u}}$ | set of use ports |
| $P_{\mathrm{p}}$ | set of provide ports |
| $\rightleftharpoons_{\mathrm{u}} \subseteq P_{\mathrm{u}} \times \Theta$ | binding relation between use ports and transitions |
| $\rightleftharpoons_{\mathrm{p}} \subseteq P_{\mathrm{p}} \times \Pi$ | binding relation between provide ports and places |
| **Assembly** | |
| $C$ | set of components of an assembly |
| $L$ | set of use-provide connections of an assembly |
| **Operational semantics** | |
| $\mathcal{M}$ | subset of elements holding a token |
| $\mathcal{R}$ | subset of places that have been reached |
| $\mathcal{E}$ | set of ongoing actions |

Table 4.1: Elements used by Madeus and their notations.

### 4.2.2 Assembly

A Madeus *assembly* is a tuple $(C, L)$ comprised of a set $C$ of components and a set $L$ of connections between ports. We remind that each element of $C$ is a tuple $((\Pi, \Theta), (P_{\mathrm{u}}, P_{\mathrm{p}}, \rightleftharpoons_{\mathrm{u}}, \rightleftharpoons_{\mathrm{p}}))$. We denote $\Pi^c$ the set of places of component $c \in C$, $\Theta^c$ its set of transitions, etc. We define:

$$\Pi^* = \bigcup \{\Pi^c \mid c \in C\}$$
$$\Theta^* = \bigcup \{\Theta^c \mid c \in C\}$$
$$\ldots$$

A connection is comprised of a use port of a component and a provide port of another component. We hence have $L \subseteq {P_{\mathrm{u}}}^* \times {P_{\mathrm{p}}}^*$. In particular, a provide port can be connected to one or multiple use ports, and a use port can be connected to one or multiple provide ports.

A Madeus assembly is *well-formed* if all the use ports in its components are connected to provide ports.

### 4.2.3 Operational semantics

Given a Madeus assembly $A$, we now define how the deployment procedure is performed. First, we introduce the notion of *configuration* of an assembly which corresponds to the state of the deployment it represents. Then, we describe the execution of the deployment as a sequence of configurations, representing the evolution of the state of the deployment over time which obeys a set of *semantic rules*.

#### 4.2.3.1 Configuration

At each moment in the execution of a Madeus deployment assembly $(C, L)$, the *configuration* of this assembly is defined by a set of tokens, the set of reached places and which actions are currently executing. A token can be on a place or on the *beginning*, *in progress* or *end* parts of a transition. Formally, a configuration is a tuple $\langle \mathcal{M}, \mathcal{R}, \mathcal{E} \rangle$ where:

- $\mathcal{M} \subseteq \Pi^* \cup (\Theta^* \times \{\text{beg}, \text{inp}, \text{end}\})$ is the marking, i.e., the set of elements holding a token;
- $\mathcal{R} \subseteq \Pi^*$ is the set of places that have been reached;
- $\mathcal{E} \subseteq \mathbb{A}$ is the set of actions that are being executed.

Note that $\mathcal{R}$ can be deduced from $\mathcal{M}$, but we make it part of the configuration for the sake of simplicity.

The initial configuration of an assembly is given by the tuple $\langle I, I, \emptyset \rangle$, where $I = \bigcup_{c \in C} \{\pi \in \Pi^c \mid \Theta_{\text{in}}(\pi) = \emptyset\}$ (i.e., the set of initial places).

#### 4.2.3.2 Execution

The execution of a Madeus assembly corresponds to performing the underlying deployment procedure. It is described as a finite sequence of configurations:

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

where $\rightsquigarrow$ is a binary relation which states that the direct evolution from one configuration to another is legal. The *semantic rules* describing this relation are explained right after this paragraph and are formally defined in Figure 4.7. An execution is *complete* if there exists no configuration $\langle \mathcal{M}', \mathcal{R}', \mathcal{E}' \rangle$ such that $\langle \mathcal{M}_n, \mathcal{R}_n, \mathcal{E}_n \rangle \rightsquigarrow \langle \mathcal{M}', \mathcal{R}', \mathcal{E}' \rangle$.

#### 4.2.3.3 Semantic rules

Five rules define the binary relation $\rightsquigarrow$, i.e., state when the direct evolution from one configuration to another is legal. Each rules has a set of hypotheses which, if they hold, imply that the conclusion holds. The conclusion is always that one configuration can directly evolve to another. In the following, we describe the conclusion by stating the changes that are applied to the first configuration to obtain the second.

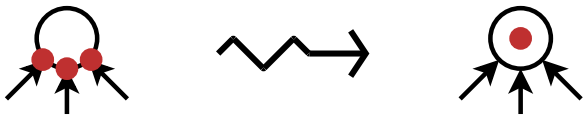


Figure 4.3: Illustration of the rule $\mathsf{Reach}_\pi$.

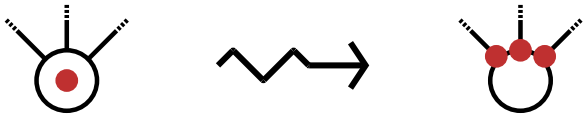


Figure 4.4: Illustration of the rule $\mathsf{Leave}_\pi$.

**Reaching place** The rule $\mathsf{Reach}_\pi$ describes a token reaching a place $\pi$. It requires that all the incoming transitions of $\pi$ have a token at their end. In the conclusion, those tokens are removed and one token is placed on $\pi$, as illustrated in Figure 4.3. The place is also added to the set $\mathcal{R}$ of reached places.

**Leaving place** The rule $\mathsf{Leave}_\pi$ describes how a token can leave a place $\pi$. There are no other requirements than $\pi$ holding a token. The token is removed from $\pi$ and a token added on the beginning of each of $\pi$'s outgoing transitions. This rule is illustrated in Figure 4.4.

**Firing transition** The rule $\mathsf{Fire}_\theta$ corresponds to the firing of a transition $\theta = (\pi_{\text{source}}, \alpha, \pi_{\text{dest}})$. The *beginning* of the transition must hold a token, and any use port bound to $\theta$ must be provided. A use port is provided if it is connected to at least one active provide port of another component. A provide port is active if at least one place bound to that port has been reached. If this is the case, then the token is moved from the *beginning* of the transition to its *in progress* part, and the action $\alpha$ starts its execution, as illustrated in Figure 4.5.

**Terminating action** The rule $\mathsf{Termin}_\alpha$ corresponds to the termination of an action $\alpha$. It requires that $\alpha$ be in the set $\mathcal{E}$ of executing actions. In practice, this rule is used by an implementation of Madeus when the action has actually finished executing.

**Ending transition** The rule $\mathsf{End}_\theta$ formally describes the end of a transition $\theta = (\pi_{\text{source}}, \alpha, \pi_{\text{dest}})$. It requires that the *in progress* part of $\theta$ holds a token and that the action $\alpha$ has terminated. If that is the case, the token is moved from the *in progress* part of $\theta$ to its *end*. Figure 4.6 illustrates this rule.

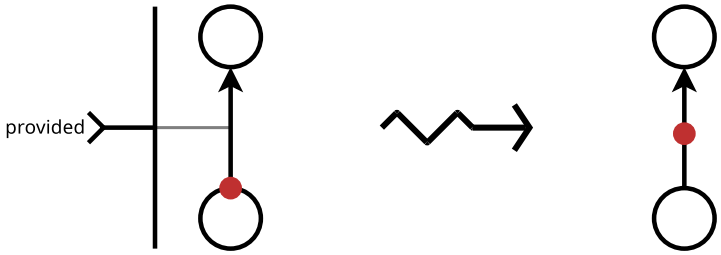



Figure 4.5: Illustration of the rule $\mathsf{Fire}_\theta$.

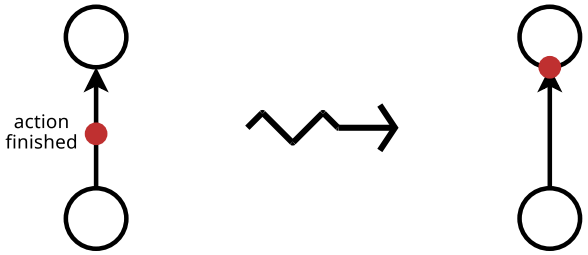



Figure 4.6: Illustration of the rule $\mathsf{End}_\theta$.

#### 4.2.3.4 Discussion

An execution of a Madeus assembly consists in a sequence of configurations. One can also see this as applying semantic rules to a starting configuration to make it evolve, until no rule can be applied, meaning that the deployment has finished. These rules are applied sequentially. One may therefore wonder whether or not this impacts the parallel execution of deployment actions. It is important to distinguish the execution of the deployment actions from the application of the semantic rules. The deployment actions associated with each transition are being executed while a token is on the *in progress* part of the transitions. Even though the application of the semantic rules is sequential, it is possible to have tokens on the *in progress* part of multiple transitions. Moreover, the execution of a semantic rule is considered to be atomic, or at least extremely fast compared to the execution of deployment actions. The sequential nature of the Madeus semantics does not limit the parallelism of deployment actions.

## 4.3 Performance Model

In the previous section we have presented Madeus in a formal way. In this section, we provide a model to express the total duration of a Madeus deployment as a function of the duration of the transitions of each component of an assembly. The duration of a transition is defined as the time it takes for its associated deployment action to complete. The formula that we obtain can be used for two purposes. First, to study

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

where

$$T = \{(\theta, \text{end}) \mid \theta \in \Theta_{\text{in}}(\pi)\}$$

$$\frac{\pi \in \Pi^* \qquad \Theta_{\text{out}}(\pi) \neq \emptyset \qquad \pi \in \mathcal{M}}{\langle \mathcal{M}, \mathcal{R}, \mathcal{E} \rangle \rightsquigarrow \langle (\mathcal{M} \setminus \{\pi\}) \cup \{(\theta, \text{beg}) \mid \theta \in \Theta_{\text{out}}(\pi)\}, \mathcal{R}, \mathcal{E} \rangle} \ \mathsf{Leave}_\pi$$

$$\frac{\theta \in \Theta^* \qquad (\theta, \text{beg}) \in \mathcal{M} \qquad \forall p_{\text{u}} : p_{\text{u}} {\rightleftharpoons_{\text{u}}}^* \theta \implies \mathit{provided}(p_{\text{u}})}{\langle \mathcal{M}, \mathcal{R}, \mathcal{E} \rangle \rightsquigarrow \langle (\mathcal{M} \setminus \{(\theta, \text{beg})\}) \cup \{(\theta, \text{inp})\}, \mathcal{R}, \mathcal{E} \cup \{\alpha\} \rangle} \ \mathsf{Fire}_\theta$$

where

$$\mathit{provided}(p_{\text{u}}) \equiv \exists p_{\text{p}}, (p_{\text{u}}, p_{\text{p}}) \in L : (\forall \Pi_{\text{b}}, p_{\text{p}} {\rightleftharpoons_{\text{p}}}^* \pi_{\text{b}} : \pi_{\text{b}} \in \mathcal{R})$$

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

$$\frac{\theta = (\pi_{\text{source}}, \alpha, \pi_{\text{dest}}) \in \Theta^* \qquad (\theta, \text{inp}) \in \mathcal{M} \qquad \alpha \notin \mathcal{E}}{\langle \mathcal{M}, \mathcal{R}, \mathcal{E} \rangle \rightsquigarrow \langle (\mathcal{M} \setminus \{(\theta, \text{inp})\}) \cup \{(\theta, \text{end})\}, \mathcal{R}, \mathcal{E} \rangle} \ \mathsf{End}_\theta$$

Figure 4.7: The operational semantics of Madeus.

how Madeus handles parallelism during deployment. Second, provided we can obtain a reasonably precise estimation (respectively a lower/upper bound) of the duration of the transitions, we can obtain an estimation (respectively a lower/upper bound) of the full Madeus deployment procedure. These estimations or bounds can be useful for a system administrator or, in the case of an automated deployment, for the algorithm that decides if the deployment time is reasonable before executing the process, in particular for critical systems and services.

### 4.3.1 Dependency graph

Our goal is to obtain a formula expressing the duration of the deployment process as a function of the duration of the deployment actions as variables. To that end, we use the formal semantics of Madeus to encode the execution flow of a Madeus assembly into a dependency graph. This problem can be divided in two: first, encoding the execution flow of each component within the assembly in their respective dependency graph, and second, connect them together to form the dependency graph of the whole assembly.

The general idea to obtain a dependency graph for a Madeus assembly is to generate a dependency graph for each component of the assembly in which the tasks comprise the execution of the transitions (in which case their weight is equal to the duration of the given transition) and other dependencies (in which case their weight is equal to 0). Then, we merge them together to form the dependency graph of the whole assembly, adding some vertices and transitions to encode the dependencies between the components. Once we have the dependency graph for the whole assembly, we can define the time it takes for the deployment process to complete as the length of the critical path of this graph.

### 4.3.2 Assumptions

The performance model given here is valid only for assemblies such that any set of places bound to a provide port contains exactly one place. This is because if a provide port is bound to a set containing multiple places, the requirement imposed by this binding is satisfied as soon as one of the places is reached. The total duration then depends on the earliest time at which one of these places is reached, which cannot be encoded in a dependency graph. In practice this is not a severe restriction, as ports requirements very rarely include disjunctions. However, we still consider the possibility of having multiple sets (of one place) bound to a port. In this case, the port is enabled when all the places have been reached, which can be encoded in the dependency graph.

### 4.3.3 Dependency graph of a component

Given a component $c = ((\Pi, \Theta), (P_{\mathrm{u}}, P_{\mathrm{p}}, \rightleftharpoons_{\mathrm{u}}, \rightleftharpoons_{\mathrm{p}}))$, we define the dependency graph $(V_c, E_c)$ corresponding to its execution flow. We define the set of vertices $V_c$ and the set of weighted arcs $E_c$ in the following.

#### 4.3.3.1 Vertices

In the dependency graph, vertices represent events related to places, transitions or ports. We define $V_c$ as the union of several sets of vertices defined next, plus one source and one sink vertices.

$$V_c = V^{\Pi} \cup V^{\Theta} \cup V^{P_{\mathrm{u}}} \cup V^{P_{\mathrm{p}}} \cup \left\{v_c^{\mathrm{source}}, v_c^{\mathrm{sink}}\right\}$$

**Places** For each place $\pi \in \Pi$, we introduce one vertex that represents the event of the place being reached.

$$V^{\Pi} = \bigcup_{\pi \in \Pi} \left\{v_{\pi}^{\mathrm{reach}}\right\}$$

**Transitions** For each transition, we introduce one vertex that represents its firing.

$$V^{\Theta} = \bigcup_{\theta \in \Theta} \left\{v_{\theta}^{\mathrm{fire}}\right\}$$

**Use ports** For each use port we introduce one vertex representing the instant when it starts being provided.

$$V^{P_{\mathrm{u}}} = \bigcup_{p_{\mathrm{u}} \in P_{\mathrm{u}}} \left\{v_{p_{\mathrm{u}}}^{\mathrm{provided}}\right\}$$

**Provide ports** For each provide port we introduce one vertex representing the instant when it starts providing.

$$V^{P_{\mathrm{p}}} = \bigcup_{p_{\mathrm{p}} \in P_{\mathrm{p}}} \left\{v_{p_{\mathrm{p}}}^{\mathrm{providing}}\right\}$$

#### 4.3.3.2 Arcs

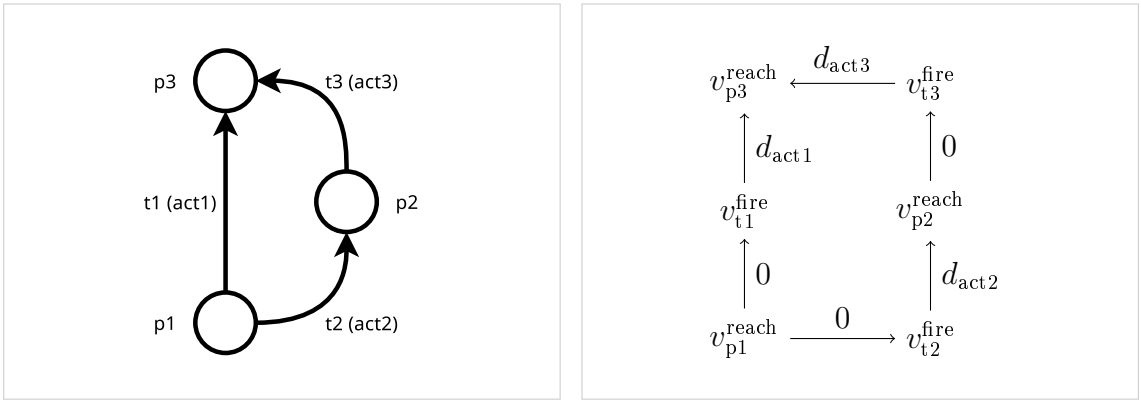
In the dependency graph, arcs represent the tasks that Madeus must perform. In practice, the arcs corresponding to the execution of deployment actions are weighted with the corresponding duration, while the other arcs have a weight of 0 and merely represent dependencies between the application of the rules of the semantics. For example, a token may enter a place only after all its incoming transitions have finished executing, i.e., have a token at their end. We define $E_c$ as the union of several sets of arcs defined next.

$$E_c = E^{\Theta} \cup E^{P_{\mathrm{u}}} \cup E^{P_{\mathrm{p}}} \cup E^{I} \cup E^{F}$$

**Transitions** For each transition $\theta = (\pi_{\text{source}}, \alpha, \pi_{\text{dest}})$, we introduce two arcs. The first, from $v^{\text{reach}}_{\pi_{\text{source}}}$ to $v^{\text{fire}}_{\theta}$, represents the fact that $\theta$ may only be fired after $\pi_{\text{source}}$ has been reached. Its weight is 0. The second, from $v^{\text{fire}}_{\theta}$ to $v^{\text{reach}}_{\pi_{\text{dest}}}$ represents the fact that place $\pi_{\text{dest}}$ may be reached only after the action $\alpha$ has finished executing. Supposing this requires a time $d_\alpha$, the weight of the arc is set to $d_\alpha$. $E^{\Theta}$ is therefore defined as follows.

$$E^{\Theta} = \bigcup_{\theta=(\pi_{\text{source}},\alpha,\pi_{\text{dest}})\in\Theta} \left\{ \left(v^{\text{reach}}_{\pi_{\text{source}}}, 0, v^{\text{fire}}_{\theta}\right), \left(v^{\text{fire}}_{\theta}, d_\alpha, v^{\text{reach}}_{\pi_{\text{dest}}}\right) \right\}$$

Figure 4.8 illustrates how a section of Madeus internal-net with three places `p1`, `p2` and `p3` and three transitions `t1`, `t2` and `t3` (associated to actions `act1`, `act2` and `act3` respectively) is transformed to a part of a dependency graph.




(a) Section of Madeus internal-net

(b) Section of dependency graph

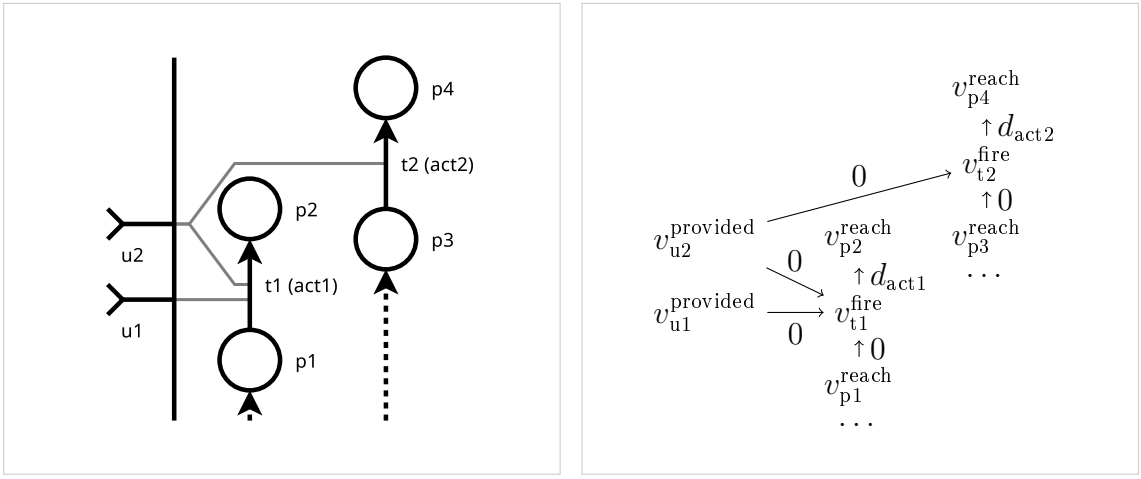
Figure 4.8: Transformation of places and transitions to a dependency graph.

**Use ports** For each use port $p_{\text{u}}$ and each transition $\theta$ such that $p_{\text{u}} \rightleftharpoons_{\text{u}} \theta$ (i.e., $p_{\text{u}}$ is bound to $\theta$), we introduce one arc from $v^{\text{provided}}_{p_{\text{u}}}$ to $v^{\text{fire}}_{\theta}$ with weight 0. This represents the fact that $\theta$ may only start once $p_{\text{u}}$ is provided. $E^{P_{\text{u}}}$ is therefore defined as follows.

$$E^{P_{\text{u}}} = \bigcup_{(p_{\text{u}},\theta)\in\rightleftharpoons_{\text{u}}} \left\{ \left(v^{\text{provided}}_{p_{\text{u}}}, 0, v^{\text{fire}}_{\theta}\right) \right\}$$

Figure 4.9 illustrates how a section of Madeus internal-net with two use ports `u1` and `u2` (associated to transition `t1` and transitions `t1` and `t2` respectively) is transformed to a part (i.e., sub-graph) of dependency graph.

**Provide ports** For each provide port $p_{\text{p}}$ and each place $\pi$ such that $p_{\text{p}} \rightleftharpoons_{\text{p}} \pi$ (i.e., $p_{\text{p}}$ is bound to the place $\pi$), we introduce one arc from $v^{\text{reach}}_{\pi}$ to $v^{\text{providing}}_{p_{\text{p}}}$ with weight




(a) Section of Madeus internal-net (b) Section of dependency graph

Figure 4.9: Transformation of use ports and their bindings to a dependency graph.

0. This represents the fact that $p_{\text{p}}$ starts providing once all the places bound to it have been reached. $E^{P_{\text{p}}}$ is therefore defined as follows.

$$E^{P_{\text{p}}} = \bigcup_{(p_{\text{p}},\pi)\in\rightleftharpoons_{\text{p}}} \left\{ \left( v_{\pi}^{\text{reach}}, 0, v_{p_{\text{p}}}^{\text{providing}} \right) \right\}$$

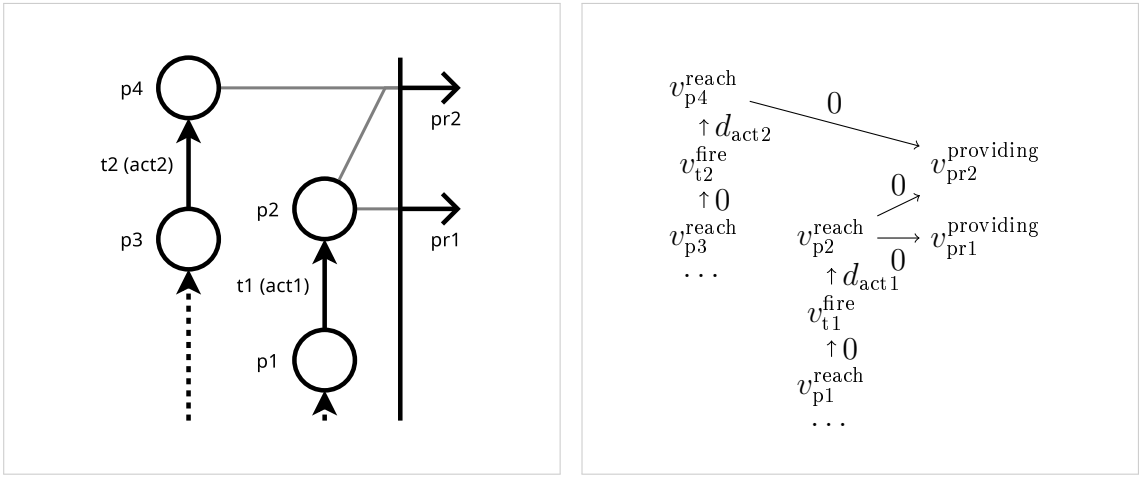
Figure 4.10 illustrates how a section of Madeus internal-net with two provide ports `pr1` and `pr2` (associated to places `p2` and places `p2` and `p4` respectively) is transformed to a section of dependency graph.

**Initial places** For each initial place $\pi$ (recall that $\pi$ is initial if it has no incoming transitions, i.e., $\Theta_{\text{in}}(\pi) = \emptyset$) we introduce one arc from $v_c^{\text{source}}$ to $v_{\pi}^{\text{reach}}$. It represents the fact that a token is placed in each initial place in the initial configuration of the component.

$$E^{I} = \bigcup_{\pi\in\Pi,\, \Theta_{\text{in}}(\pi)=\emptyset} \left\{ \left( v_c^{\text{source}}, 0, v_{\pi}^{\text{reach}} \right) \right\}$$

**Final places** For each final place $\pi$ (recall that $\pi$ is final if it has no outgoing transitions, i.e., $\Theta_{\text{out}}(\pi) = \emptyset$) we introduce one arc from $v_{\pi}^{\text{reach}}$ to $v_c^{\text{sink}}$. It represents the fact that the deployment procedure of the component terminates only after all final places have been reached.

$$E^{F} = \bigcup_{\pi\in\Pi,\, \Theta_{\text{out}}(\pi)=\emptyset} \left\{ \left( v_{\pi}^{\text{reach}}, 0, v_c^{\text{sink}} \right) \right\}$$




(a) Section of Madeus internal-net (b) Section of dependency graph

Figure 4.10: Transformation of provide ports and their bindings to a dependency graph.

### 4.3.4 Dependency graph of an assembly

We have seen in the previous section how to generate a dependency graph for a given component. In this subsection, we describe how to combine them to generate a dependency graph $(V, E)$ for a whole assembly.

Recall that an assembly is a tuple $A = (C, L)$. In the following, we consider that $C = \{c_1, c_2, \ldots c_n\}$.

#### Vertices

Given the $n$ dependency graphs $(V_{c_i}, E_{c_i})$, we add a global source vertex $v^{\text{SOURCE}}$ and a global sink vertex $v^{\text{SINK}}$.

$$V^S = \left\{v^{\text{SOURCE}}, v^{\text{SINK}}\right\}$$

This gives us the set $V$ of vertices for the dependency graph of the whole assembly.

$$V = V^S \cup \bigcup_{i=1}^{n} V_{c_i}$$

#### Arcs

We then add arcs between the global source vertex and the source vertices of the dependency graphs of the components. Similarly, we add arcs between their sink vertices and the global sink vertex. These arcs all have weight 0.

$$E^S = \bigcup_{i=1}^{n} \left\{ \left( v^{\text{SOURCE}}, 0, v_{c_i}^{\text{source}} \right), \left( v_{c_i}^{\text{sink}}, 0, v^{\text{SINK}} \right) \right\}$$

Finally, for each connection $(p_{\text{u}}, p_{\text{p}})$ in $L$, we add one arc from the vertex representing the fact that the $p_{\text{p}}$ provide port starts providing to the vertex representing the fact that the $p_{\text{u}}$ use port starts being provided.

$$E^P = \bigcup_{(p_{\text{u}}, p_{\text{p}}) \in L} \left\{ \left( v_{p_{\text{p}}}^{\text{providing}}, 0, v_{p_{\text{u}}}^{\text{provided}} \right) \right\}$$

This gives us the set $E$ of arcs for the dependency graph of the whole assembly.

$$E = E^S \cup E^P \cup \bigcup_{i=1}^{n} E_{c_i}$$

Figure 4.11 depicts the dependency graph corresponding to the server-database assembly of Figure 4.1. We notice the sub-graph of component `db` on the left, and the one of component `server` on the right. In the latter, notice in particular the arcs corresponding to the execution of transition `conf1` parallel to those corresponding to the execution of transitions `ins` and `conf2`. Notice also that these two sub-graphs are connected by the global source and global sink vertices, and by the arcs between the vertices corresponding to their connected ports.

### 4.3.5 Duration of the deployment process

We have seen in the previous subsection how to obtain the dependency graph corresponding to a Madeus assembly. We now detail how to express the total duration of the deployment process as a function of the duration of the individual deployment actions.

Because the dependency graph is weighted with the duration of the actions, intuitively the duration of the deployment process corresponds to the length of one of its critical paths, i.e., one of its longest paths from vertex $v^{\text{SOURCE}}$ to vertex $v^{\text{SINK}}$. If the edges are weighted with values, then the duration of the deployment process can be computed in polynomial time (relative to the size of the graph), e.g., by finding a topological order, because $(V, E)$ is acyclic. Note that the size of the graph is linear to the sum of numbers of places, transitions, ports and connections in the assembly. However, if the edges are weighted with variables, we can express the duration of the deployment process as a function of these variables in the following way. We define $LCP(v)$ to be the length of a critical path from vertex $v^{\text{SOURCE}}$ to vertex $v$. We can express $LCP$ recursively in the following way:

$$LCP(v) = \begin{cases} 0 & \text{if } v = v^{\text{SOURCE}} \\ \max_{(v_{\text{parent}}, w, v) \in E} \left( w + LCP(v_{\text{parent}}) \right) & \text{otherwise} \end{cases}$$

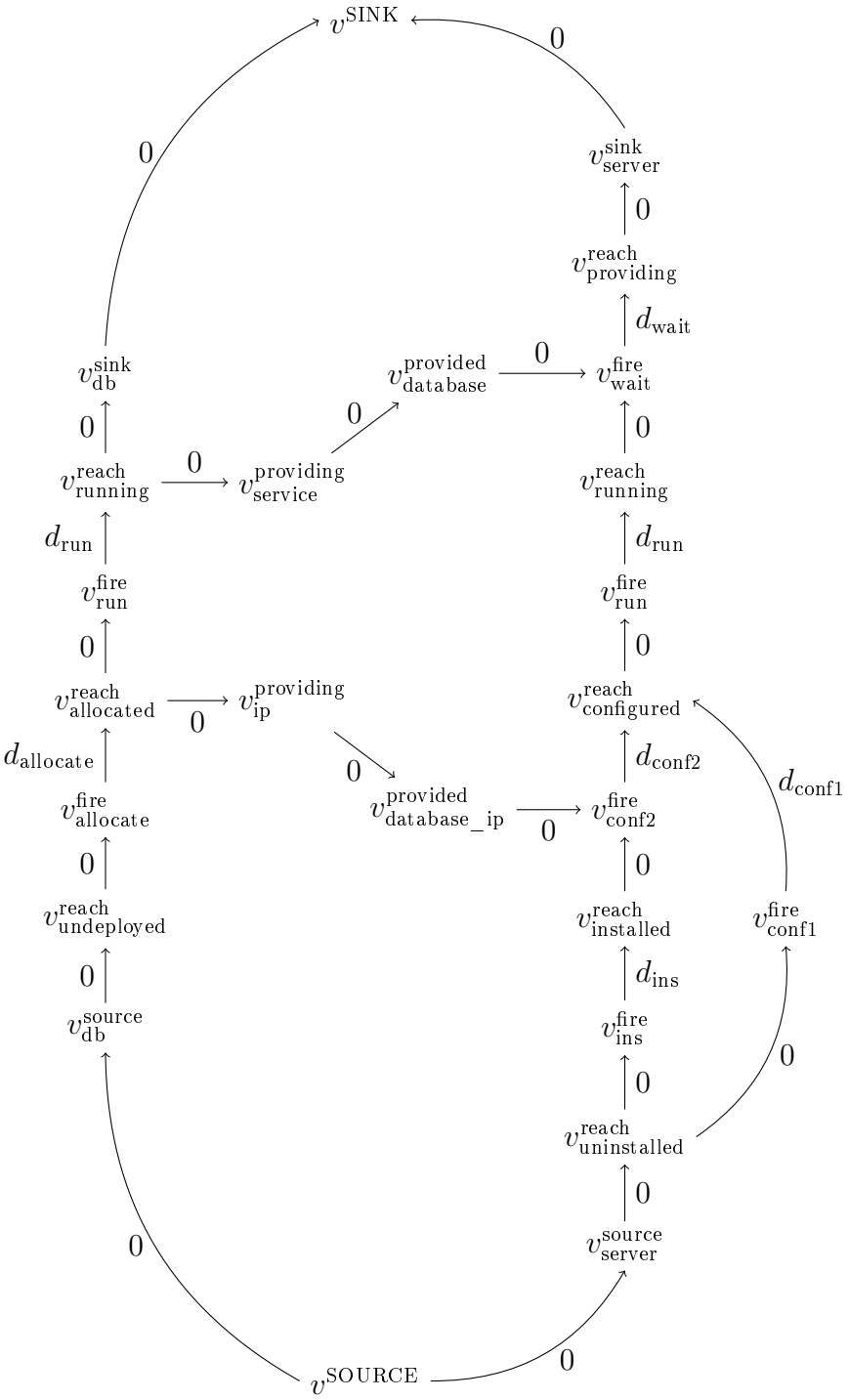



Figure 4.11: Dependency graph corresponding to the assembly of Figure 4.1.

The duration of the deployment process is equal to $LCP(v^{\text{SINK}})$, which is well-defined because $(V, E)$ is acyclic and $v^{\text{SOURCE}}$ is an ancestor of all the other vertices.

Using the dependency graph of Figure 4.11, we obtain the following formula for the server-database assembly shown in Figure 4.1:

$$\begin{aligned}
&\max(d^{\text{db}}_{\text{run}} + d^{\text{db}}_{\text{allocate}},\\
&\quad d^{\text{server}}_{\text{wait}} + \max(d^{\text{db}}_{\text{run}} + d^{\text{db}}_{\text{allocate}},\\
&\qquad d^{\text{server}}_{\text{run}} + \max(d^{\text{server}}_{\text{conf1}},\\
&\qquad\quad d^{\text{server}}_{\text{conf2}} + \max(d^{\text{db}}_{\text{allocate}},\\
&\qquad\qquad d^{\text{server}}_{\text{ins}}))))
\end{aligned}$$

Which can be simplified, because the second element of the most outer max is necessary larger than its first element, the proof of which is left to the reader. The final formula is:

$$\begin{aligned}
&d^{\text{server}}_{\text{wait}} + \max(d^{\text{db}}_{\text{run}} + d^{\text{db}}_{\text{allocate}},\\
&\qquad d^{\text{server}}_{\text{run}} + \max(d^{\text{server}}_{\text{conf1}},\\
&\qquad\quad d^{\text{server}}_{\text{conf2}} + \max(d^{\text{db}}_{\text{allocate}},\\
&\qquad\qquad d^{\text{server}}_{\text{ins}})))
\end{aligned}$$

## 4.4 Discussion

Madeus is a deployment model conceived from the ground up to have as much parallelism expressivity as possible. Each module of a distributed application has its life-cycle modeled by the *internal-net* of a Madeus *component*. This internal-net is a Petri-net-like structure which allows to express parallelism of deployment actions within the module's own life-cycle. Using the classification criteria introduced in Chapter 3, Madeus has the *custom-par multi-path* level of life-cycle modeling.

The coordination between deployment actions of distinct modules is made thanks to Madeus assemblies. Connections between the ports of components are used to model dependencies between deployment actions of these distinct modules. Because the ports of the components are bound to precise parts of the life-cycle, these dependencies are extremely fine-grained. This, in addition to the possibility to express parallelism within modules, results in a *state-based intra-module* level of parallelism. The performance model of Madeus allows to get a formula expressing the total deployment time as a function of the duration of the individual deployment actions. This allows to precisely define the gain introduced by this high level of parallelism.

In terms of separation of concerns, module developers are intended to develop Madeus components, reconfiguration developers are intended to assemble them as an assembly, and finally system administrators are intended to execute these assemblies. While the precise evaluation of the separation of concerns is discussed in Chapter 6, we argue that the distinction between the internal-net and the interface (made of ports) of the components has important benefits in this regard.

Madeus is modeled formally and in particular has a formally defined operational semantics. This opens the door to the use of formal methods to provide some guarantees, such as what was done in [93].

However, Madeus comes with some limitations. First, it is strictly restricted to deployment and does not handle general reconfiguration. This will be addressed by our second contribution introduced in Chapter 5. Madeus also assumes a single coherent representation of the assembly, which entails a centralized execution.

## 4.5 Conclusion

In this chapter has been presented *Madeus*, a formal model for the deployment of distributed software with a high degree of parallelism. The *deployment life-cycle* of each module as well as its *external dependencies* are described in a *component*. The life-cycle is defined as a parallel state-machine called *internal-net*, and the dependencies are defined as a set of use and provide *ports*. Madeus components can be connected together by their ports to form an *assembly*, describing the complete deployment process for a (possibly complex) distributed system.

The assembly can be executed to perform the associated deployment process. The complete *operational semantics* of Madeus has been introduced. This semantics leads to a *state-based intra-module* level or parallelism (i.e., parallel actions within the deployment of a module as well as across modules).

Finally, a performance model has been presented to express the total deployment time of a Madeus assembly as a function of the duration of each individual deployment action. In the experiments presented in Chapter 6, this will be used to predict thanks to historical data, which can be useful to system administrators or automated systems to make better decisions.

# Chapter 5

# The Concerto Reconfiguration Model

## Contents



This chapter presents our second contribution: the Concerto reconfiguration model. Concerto extends the Madeus deployment model (presented in Chapter 4) for reconfiguration.

The first objective of Concerto is to allow module developers to define detailed life-cycles for the modules, not only during their deployment but also during the rest of their lifespans. The second objective is to let reconfiguration developers define fine-grained dependencies between the reconfiguration actions of distinct modules. The third objective is to show a high level of separation of concerns between module developers, reconfiguration developers and system administrators. Finally, Concerto's concepts and semantics are defined formally.

In the first section we give an overview of Concerto based on a simple example. In the second section we define all the concepts and the semantics of Concerto in a formal way. In the third section we present a performance model for Concerto. Then, in the fourth section we present an addition to Concerto called behavioral interfaces, which aim at improving separation of concerns for users. Finally, in the fifth section we discuss the benefits and limits of Concerto.

## 5.1 Overview

Concerto extends the Madeus deployment model to support reconfiguration. Concerto and Madeus hence share many concepts like *internal-nets*, *places* or *transitions*, sometimes altered to support reconfiguration. In particular, Concerto also uses the control component approach. When differences exist, these concepts are explained again for Concerto, while the main changes from Madeus to Concerto are outlined at the end of this section (in Sub-section 5.1.4).

### 5.1.1 Component type

In Concerto, each kind of module is represented by a component type. Each component type has its life-cycle encoded by an *internal-net* and its dependencies encoded by *ports*, similarly to Madeus components.

#### Internal-net

Concerto's internal-nets are similar to Madeus', in that they are composed of places and transitions. Additionally, each transition is associated to a *behavior*. Behaviors intuitively correspond to a set of actions (i.e., transitions) to go from one state of the life-cycle of the module to another (like *deploy*, *suspend*, *update*, etc.). One transition is associated to a single behavior, but a behavior may be associated to any number of transitions. Transitions with multiple endings, called *switches*, exist to model multiple possible evolutions during the life-cycle of a components. One place in the *internal-net* is designated as *initial place* and corresponds to the starting configuration of the module. Unlike Madeus, this place must be explicitly stated by the module developer.

For example, Figure 5.1 shows two component types representing respectively a database (`Db`) and a proxy (`Proxy`). The `Proxy` component type has seven places

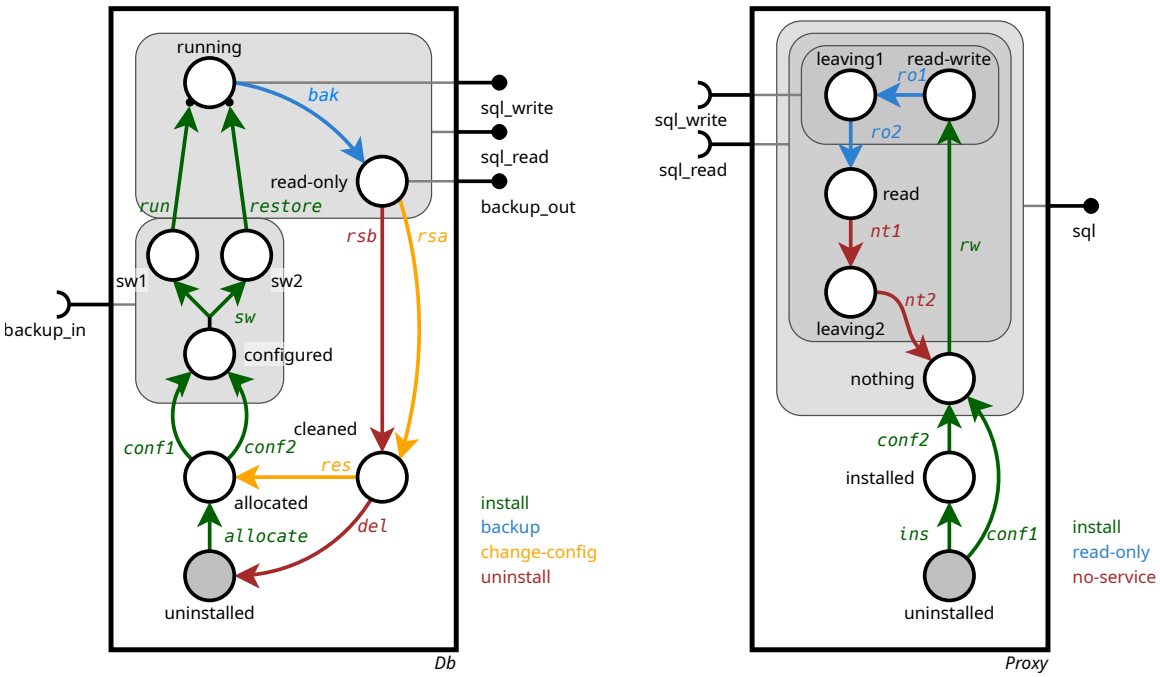



Figure 5.1: Two Concerto component types describing the life-cycle of a database and a proxy (which can be seen as a client). Components are represented by rectangles. Inside components, places are represented by circles, initial places being filled with gray, and transitions are represented by colored arrows between places. The color of an arrow indicates to which behavior the transition it represents is associated (the matching between color and name of behavior is made on the right of the component type). Switches are represented like transitions with multiple endings. Groups are represented by gray rounded rectangles. Outside components, provide ports are represented by semi-circles and use ports are represented by discs. Bindings between ports and groups are represented by a thing gray line. When the line is connected to a place instead of a group, it designates a group containing only this place.

(`uninstalled`, `installed`, `nothing`, `read-write`, `leaving1`, `read` and `leaving2`) and eight transitions. The transitions `ins`, `conf1`, `conf2` and `rw` are associated with the behavior `install`, the transitions `ro1` and `ro2` are associated with the behavior `read-only`, and the transitions `nt1` and `nt2` are associated with the behavior `no-service`. Notice that transition `conf1` on the one hand, and transitions `ins` and `conf2` on the other hand can be executed in parallel as in Madeus. The `Db` component type has a switch (transition with multiple endings) `sw`, going from place `configured` to either place `sw1` or place `sw2`.

#### Ports

In Concerto, and similarly to Madeus, dependencies to other life-cycles are represented by ports. While provide ports in Madeus could only be activated, in Concerto they can also be deactivated. For this reason, we need to specify which part of the life-cycle corresponds to when the service related to the port is used or provided (depending on the type of port). A *group* is a part of internal-net and corresponds to a part of the module's life-cycle.

In Concerto, use and provide ports are bound to a group. A use port of a component $c$ hence represents a requirement that must be fulfilled by another Concerto component so that $c$ can be in the part of its life-cycle designated by the group it is bound to. A provide port of $c$ represents the fact that $c$ fulfills some requirements while it is in the part of its life-cycle designated by the group it is bound to.

For example, in Figure 5.1, component type `Proxy` has two use ports (`sql_write` and `sql_read`) and one provide port (`sql`). In particular, it requires some components to provide the `sql_write` service to be in places `read-write` or `leaving1` (or in the transition between them). It also provides the `sql` service while it is in places `read-write`, `leaving1`, `read`, `leaving2` or `nothing`, or any transition between them. Notice that ports are represented with discs for provide ports and semi-circles for use ports in Concerto, instead of arrows and inverse arrows like in Madeus. This is to signify that these ports may be deactivated unlike in Madeus.

### 5.1.2 Assembly

A Concerto *assembly* is made of a set of *component instances* and a set of *connections* between the ports of the instances.

#### Component instance

A *component instance* (referred in the following as just *instance*) is made of a component type and a state for this component type. Intuitively, an instance corresponds to the life-cycle of an actual piece of software, while a component type is merely a blueprint. This entails that multiple instances can have the same component type but different states. The state of an instance is determined by a *marking* (a set of tokens located on places, transitions or transition endings of the component type) and a *behavior queue* (a list of behaviors of the component type). Intuitively, the marking designates in which part of its life-cycle a piece of software is, while the behavior queue designates the set of transitions that are to be executed by the instance (those associated with the behaviors in the queue). The first behavior in the queue is the *active behavior*, which designates which transitions in the internal-net of the component type can be executed.

Figure 5.2 is an example of assembly containing two instances, an instance `db` of type `Db`, and an instance `proxy` of type `Proxy`. Instance `db` has two tokens on

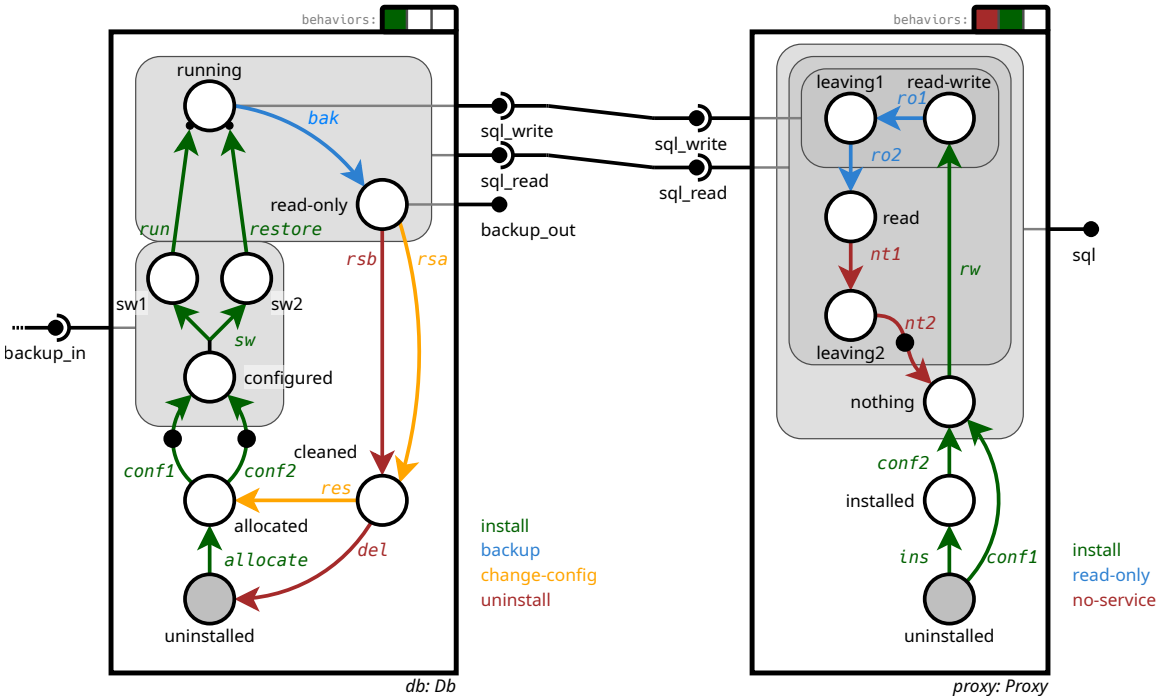



Figure 5.2: Concerto assembly composed of one instance of each component type of Figure 5.1. The tokens of the marking are represented by black discs. The behavior queue is represented by an area at the top-right corner of each component instance where the behaviors in the queue are listed from left to right with their respective colors. The names of each behavior is associated to their color thanks to a list on the right of each component instance.

transitions `conf1` and `conf2` and has one behavior in its queue: `install`. Because it is its active behavior, the green transitions can be executed. Instance `proxy` has one token on transition `nt2`. It has two behaviors in its queue: `no-service` and `install`. Because `no-service` is its active behavior, only the red transitions can be executed.

### Connections

Concerto's connections are similar to Madeus': a connection may exist between any use port of a component instance and any provide port of another instance. They represent a dependency between these two instances.

### Execution semantics

A Concerto assembly can evolve on its own by following a set of rules governing the location of the tokens and the evolution of the behavior queue. At each moment, a

| Status | Definition |
|---|---|
| **Place, Transition, Transition ending** | |
| Active | Holds a token |
| Inactive | Not active (= does not hold a token) |
| **Group** | |
| Active | At least one of the places of the group, transitions between the places of the group or transition endings of these transitions is active |
| Inactive | Not active |
| **Provide port** | |
| Active | The group it is bound to is active |
| Deactivating $\Longrightarrow$ Active | The group it is bound to is active but would not be active if all the active places in the group which have outgoing transitions associated with the active behavior transferred their token to these outgoing transitions |
| Used $\Longrightarrow$ Active | Connected to at least one active use port |
| Inactive | Not active |
| **Use port** | |
| Provided | Connected to at least one active provide port |
| Active $\Longrightarrow$ Provided | The group it is bound to is active |
| Inactive | Not active |

Table 5.1: Possible statuses of Concerto elements.

component has one or more tokens, like in Madeus. In Concerto, a token may be on:

- a place, which means that the corresponding milestone has been reached;
- a transition, which means that the corresponding reconfiguration action is being executed;
- a transition ending, which means that the action has finished its execution (in the case of a switch, a transition may have multiple endings, in which case only one can have a token).

These tokens evolve in a similar way to those of Madeus. However, Concerto features more complex interaction between the components through their ports. The complete operational semantics of Concerto is given in Section 5.2, however we present it informally through the example presented in Figure 5.3 (Table 5.1 defines some notions used in the following example). In the following, each step corresponds to one of the snapshots presented in Figure 5.3:

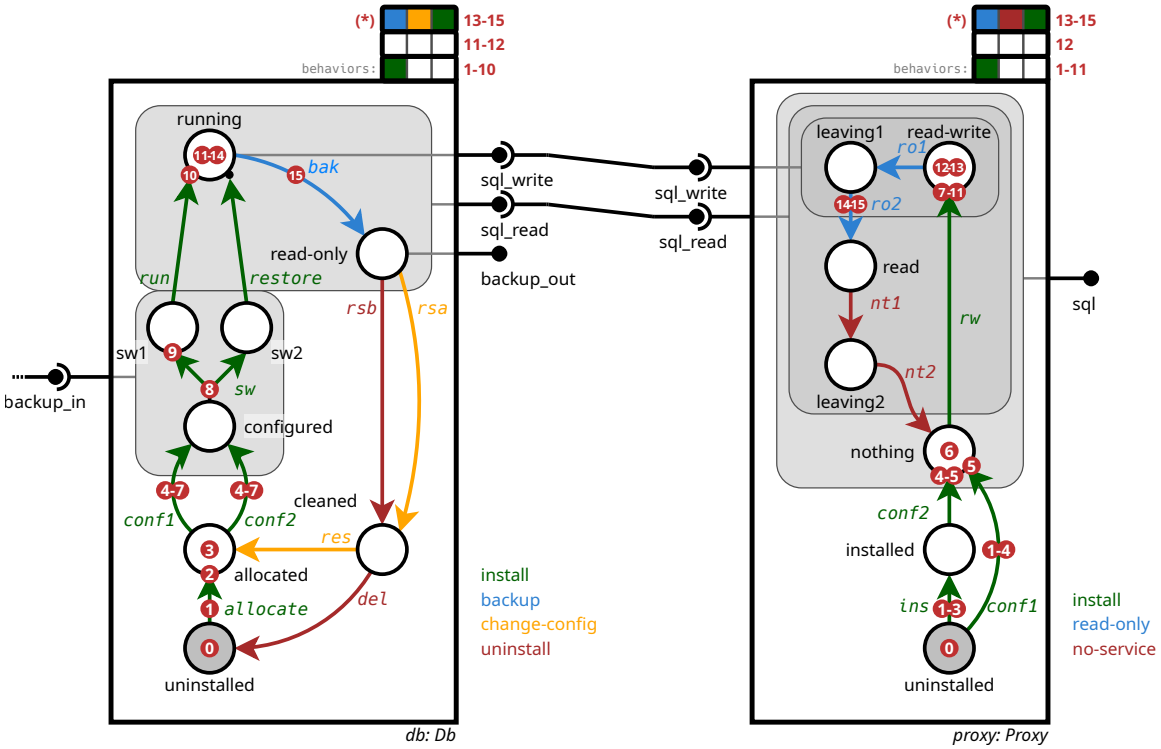



Figure 5.3: Sixteen snapshots of the execution of a Concerto assembly. Each red circle is a token, and the number inside it corresponds to which snapshot it belongs to. Two adjacent circles with two numbers separated with a dash represent a token corresponding to multiple snapshots (from the first number to the second). The numbers in red on the right of the behavior queues at the top-right of each component instance have a similar meaning: each queue represented corresponds to a set of snapshots.

0. At the start, we consider the case where the starting place of each component instance is active. Both instances have a single behavior in their behavior queue: `install`.

1. When a token is in a place, it can be removed and one put on each outgoing transition of the place associated with the current behavior. In this snapshot, tokens have been removed from the two `uninstalled` places and put in the `allocate`, `ins` and `conf1` transitions which are all associated to the active behavior (`install`) of their respective component instances.

2. When the action corresponding to a transition has finished its execution, the token may be moved from the transition to its ending (or one of its endings in the case of a switch). In this snapshot, the transition `allocate` has finished its

execution and the token it had was moved to its only transition ending.

3. When all the transition endings coming to a given place and associated to a given behavior hold a token, i.e., when the associated reconfiguration actions have finished their execution, these tokens can be removed and one token assigned to the place. In this snapshot, the token was removed from the transition ending between transition `allocate` and place `allocated` of instance `db` and put in this last place.

4. Skipping a few steps, in this snapshot transition `conf` of instance `db` is holding a token. In instance `proxy`, transitions `inf` and `conf2` have finished executing. However, place `nothing` cannot yet have a token because transition `conf1` has not finished executing.

5. In this snapshot, transition `conf1` has finished its execution.

6. In this snapshot, because both `conf1` and `conf2` of instance `proxy` have tokens in their transition endings, these can be removed and one token put on place `nothing`.

7. Skipping a few steps, transition `rw` of instance `proxy` has finished its execution but it is not possible to put a token in place `read-write` because it is in a two-place group bound to use port `sql_write` and in a four-place group bound to use port `sql_read`, meaning that these two ports must be `provided` before a token can be put on one of its places. In this case, provide ports `sql_write` and `sql_read` of instance `db` must become active.

8. Skipping a few steps, transition `sw` of instance `db` is holding a token. Notice that `sw` is a particular type of transition with multiple endings called a `switch`. Switches are used to decide at run-time which path in it life-cycle a component instance should take. The decision is made by code provided by the module developer.

9. In this snapshot, the token was moved from `sw` to one of its endings, the one going to place `sw1`. Only one ending receives a token, which in practice is decided by the code written by the module developer.

10. Skipping a few steps, in this snapshot the transition ending of transition `run` is holding a token. In this case, both transition endings going to place `running` do not have to hold a token to put a token in the place. This is because this is not possible due to switch `sw`. This is symbolized by the notion of *station* explained in details later in this chapter, and graphically represented by a small black circle to which the transition ending is connected.

11. In this snapshot, the token was moved from the transition ending of transiting `run` to place `running`. Note that because `running` is in the single-place group connected to provide port `sql_write` and the two-place group connected to provide port `sql_read`, both ports are now active. Also, because transitions of the `install` behavior can no longer be triggered in instance `db` (place `running` is the only element holding a token and has no outgoing transitions associated to that behavior), `install` is removed from the queue of behaviors which is now empty.

12. In this snapshot, because both use ports of instance `proxy` are provided, the token was moved from the transition ending of transition `rw` to place `read-write`. Because no transition associated to behavior `install` can be executed, it is removed from the behavior queue of instance `proxy` which is now empty.

13. So far, the execution of a Concerto assembly has been similar to the one of a Madeus assembly because only deployment was performed. However, the fact that each component has multiple possible behaviors makes it so that Concerto models the whole life-cycle of the software modules, possibly including their deployment. Behaviors can be added to the behavior queue of component instances by using a dedicated language, which is presented after this example. For now, we suppose that behaviors `backup`, `change-config` and `install` on the one hand, and `read-only`, `no-service` and `install` on the other hand were added respectively to instances `db` and `proxy`. Behaviors `backup` and `read-only` (in blue) are now the active behaviors. In this situation, transition `bak` of instance `db` cannot start because this would mean removing the token from place `running`, which is the only place in the group bound to provide port `sql_write`, which is currently being used.

14. Skipping a few steps, because the active behavior of instance `proxy` is `read-only`, the token on place `read-write` has moved to transition `ro2`. Notice that now the group bound to use port `sql_write` does not contain tokens.

15. Because provide port `sql_write` is not used anymore, the token from place `running` of instance `db` can be moved to transition `bak`.

This example illustrates the semantics of Concerto with a given static assembly. However, Concerto's assemblies are dynamic and can be changed with a *reconfiguration program.*

### 5.1.3 Reconfiguration Program

Concerto comes with a language with six types of instructions which can be used to write reconfiguration programs. A reconfiguration program can be applied to an existing (possibly empty) assembly. The types of instructions are the following:

- add$(id, t)$ : adds a new component instance of type $t$ with identifier $id$ to the assembly;
- del$(id)$ : removes the component instance with identifier $id$ to the assembly;
- con$(idu, pu, idp, pp)$ : adds a connection in the assembly between the use port $pu$ of the component instance with identifier $idu$ and the provide port $pp$ of the component instance with identifier $idp$;
- dcon$(idu, pu, idp, pp)$ : similar to *con*, but removed the connection instead of adding it, if the connection is being used (i.e., if the use port is currently active), the execution of the reconfiguration program is paused until the deactivation of the use port;
- pushB$(id, b)$ : push a behavior $b$ to the queue of behaviors of the component instance with identifier $id$;
- wait$(id)$ : pauses the execution of the reconfiguration program until the component instance with identifier $id$ has no more behaviors in its behavior queue.

Notice that most of the instructions are not blocking (only the *wait* and *dcon*, in some scenarios, may be blocking). For example, *pushB* simply adds a behavior to the queue of a component, which is not the same as actively executing this behavior. It follows that even though the reconfiguration program is a sequence of instructions, two component instances can execute their behaviors in parallel.

Listing 5.1: Reconfiguration program leading to the deployment scenario presented in Figure 5.3 (snapshots 0 to 12)

```
add(db, Db)
add(proxy, Proxy)
con(proxy, sql_write, db, sql_write)
con(proxy, sql_read, db, sql_read)
con(db, backup_in, other_comp, other_port)  # Not shown in the figure
pushB(db, install)
pushB(proxy, install)
wait(proxy)
```

An example of program is presented in Listing 5.1. This program creates the assembly of which we have studied the execution presented in Figure 5.3. This program leads to snapshot 0, and the operational semantics of Concerto then leads to snapshot 12. To go from snapshot 12 to snapshot 13, one can execute the program presented in Listing 5.2.

Listing 5.2: Reconfiguration program leading to the config change scenario presented in Figure 5.3 (starting from snapshot 13)

```
pushB(db, backup)
pushB(db, change-config)
pushB(db, install)
```

```
4 pushB(proxy, read-only)
5 pushB(proxy, no-service)
6 pushB(proxy, install)
7 wait(proxy)
```

### 5.1.4 Changes from Madeus to Concerto

Madeus and Concerto share many concepts like internal-nets an ports, and have similarities in their execution semantics. However, some key points differentiate them so that, at the cost of more complexity (in the sense of featuring more concepts), Concerto is strictly more expressive than Madeus.

**Dynamic assemblies** In Madeus, assemblies are static and cannot evolve over time. This is not a problem when considering only the deployment part of the life-cycle of distributed systems. However, Concerto allows to model their whole life-cycle, which can include structural changes through reconfiguration. Therefore, Concerto provides a reconfiguration language to create assemblies of components or change existing ones. Adding new components at run-time implies that one has a library of "blueprints" which can be used as a model for the new component. This leads to the separation of two notions in Concerto: component type and component instance. A component type is a "blueprint", while a component instance is the actual model of the life-cycle of a distributed system module.

**Behaviors** In Madeus, the components' internal-nets are basically acyclic directed graphs of deployment tasks with a starting point: there is only one direction in which the execution can go. In Concerto, the internal-nets model the whole life-cycle of the modules, possibly including deployment. The transitions of each component must be differentiated depending on their objective (usually, the deployment of a module is one possible objective). This is represented by behaviors in Concerto, each behavior corresponding to one objective.

**Deactivation of ports** Usually, ports represent services or information that the software module modeled by the component is able to provide. While deployment usually only involves providing additional services or information as the process goes on (e.g., IP address, configuration information, API, etc.), reconfiguration often leads to services being suspended or information becoming invalid. Consequently, while Madeus' provide ports are merely signals that some data or services are now available, Concerto's provide ports can both be activated, signaling availability, but also deactivated, signaling unavailability. Concerto introduces the notion of *groups* to designate the part of the life-cycle of a component during which a provide port is active or a use port is actually used. In terms of semantics, this implies that in addition to waiting for a provide port to be active before entering a part of the life-cycle, the

opposite also exists: waiting for a use port to be inactive so that a provide port may be deactivated.

**Transitions** In Madeus, a transition is going from one place to another, and tokens can be in three locations of the transition, meaning either that the transition is ready to be executed, executing or has finished executing. In Concerto, the introduction of *switches* lead to a transition going from one place to one or more places (using *transition endings*). The destination of the transition is decided at run-time when its corresponding action is executed. Note that switches could also be integrated to Madeus without difficulties. Also, in Concerto, only two locations exist for tokens: the transition itself, meaning that it is executing, and transitions endings, meaning that it has finished executing. The other location used by Madeus (ready to be executed) is not needed in Concerto because transitions can always start their execution right away, as use ports are connected to groups (instead of transitions themselves in Madeus). This means that if a token has entered the origin place of the transition, then all the use ports it uses are provided.

## 5.2 Formal Model

While the previous section gave an overview of the concepts used in Concerto, this section presents them and in a formal way. Table 5.2 lists all the notations for easy reference. After that, the operational semantics of the model is also detailed.

### 5.2.1 Component Type

Recall that a *component type* is a template used to create component instances. It includes an *internal-net* that describes the life-cycle of the components and *ports* that corresponds to its external interface, along with its list of behaviors. Formally, a component type is a tuple $(\Pi, \pi_{\text{init}}, \Delta, Pl, \Theta, B, P_{\text{u}}, P_{\text{p}}, Gr)$.

$\Pi$ is the set of *places* in the internal-net, with a distinguished element $\pi_{\text{init}}$ which is the *initial place*. To handle synchronization of parallel transitions, places are equipped with *stations*, each attached to a place. $\Delta$ is the set stations, and the place to which a station $\delta$ is attached is denoted $Pl(\delta)$. $B$ is the set of behaviors of the component type. $\Theta$ is the set of *transitions* in the internal-net, where each element of $\Theta$ is a tuple $(\pi, b, D)$ with $\pi \in \Pi$ the source place from which the transition originates, $b \in B$ is its associated behavior and $D \subseteq \Delta$ 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*, i.e., a pair comprised of a transition and a station contained in its set of destination stations. $P_{\text{p}}$ is the set of *provide ports* and $P_{\text{u}}$ the set of *use*

| Notation | Description |
|---|---|
| **Component type** | |
| $\Pi$ | Set of places. |
| $\pi_{\text{init}} \in \Pi$ | Initial place. |
| $B$ | Set of behaviors. |
| $\Delta$ | Set of stations. |
| $Pl : \Delta \to \Pi$ | Function associating a place to each station. |
| $\Theta \subseteq \Pi \times B \times (\mathcal{P}(\Delta) \setminus \emptyset)$ | Set of transitions (switch if multiple stations). |
| $P_{\text{u}}$ | Set of use ports. |
| $P_{\text{p}}$ | Set of provide ports. |
| $Gr : (P_{\text{u}} \cup P_{\text{p}}) \to \mathcal{P}(\Pi)$ | Function associating a group to each port. |
| **Component instance** | |
| $id$ | Instance identifier. |
| $c$ | Component type of the instance. |
| $X^c$ | For a notation $X$: element corresponding to $X$ in the tuple $c$. |
| $Q \in \mathcal{L}(B^c)$ | FIFO queue of behaviors in $B^c$ ($\mathcal{L}$: list). |
| $\mathcal{M} \subseteq \Pi^c \cup \Theta^c \cup (\Theta^c \times \Delta)$ | Set of active places, transitions and transition endings. |
| **Assembly** | |
| $I$ | Finite set of component instances. |
| $X^i$ | For a notation $X$: element corresponding to $X$ in the tuple $c$ of instance $i \in I$. |
| $L \subseteq \bigcup_{\substack{i_1, i_2 \in I \\ i_1 \neq i_2}} \left({P_{\text{u}}}^{i_1} \times {P_{\text{p}}}^{i_2}\right)$ | Set of connections between use and provide ports of distinct instances in $I$. |

Table 5.2: Notations used in Concerto.

*ports* of the component type. Finally, $Gr : (P_{\text{u}} \cup P_{\text{p}}) \to \mathcal{P}(\Pi)$ is the function that associates each port (use or provide) to a group, represented as a set of places (i.e., a subset of $\Pi$).

The transitions in a single behavior are not allowed to 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.

**Notation** When we need to distinguish the elements of various component types, we use a superscript notation, e.g., $\Pi^c$ to refer to the places of the type $c$.

### 5.2.2 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).

### 5.2.3 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 $(i_1, p_\mathrm{u}, i_2, p_\mathrm{p})$, where $i_1 \in I$ is a component of type $c_1$, $i_1 \in I$ is a distinct component of type $c_2$, $p_\mathrm{u} \in {P_\mathrm{u}}^{c_1}$ and $p_\mathrm{p} \in {P_\mathrm{p}}^{c_2}$.

A *reconfiguration program* is a sequence of reconfiguration instructions targeting an assembly. 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 $\texttt{add}(id, c)$ and $\texttt{del}(id)$ (creation and deletion of component), $\texttt{con}(id_1, p_\mathrm{u}, id_2, p_\mathrm{p})$ and $\texttt{dcon}(id_1, p_\mathrm{u}, id_2, p_\mathrm{p})$ (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, $p_\mathrm{u}$ a use port, $p_\mathrm{p}$ a provide port and $b$ a behavior. The semantics of these instruction is detailed below.

### 5.2.4 Operational Semantics

We now define the rules that describe how a Concerto assembly evolves and how a reconfiguration program affects it. 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 relation $\rightsquigarrow$ over configurations.

#### 5.2.4.1 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) which it is bound to. A group is active if at least one of its places holds a token, or if at least one transition (or transition ending) located between two of its places holds a token. Formally, we

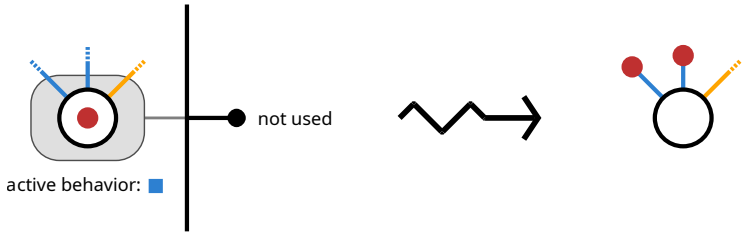



Figure 5.4: Illustration of the rule $\mathsf{Fire}^{b}_{\pi}$.

define the elements of a group $G$ to be

$$elements(G) \equiv \bigcup \begin{cases} G \in \Pi^c \\ \{(\pi_{\text{source}}, b, D) \in \Theta \mid \pi_{\text{source}} \in G \wedge \forall \delta \in D,\ Pl(\delta) \in G\} \\ \{((\pi_{\text{source}}, b, D), \delta) \in \Theta \times \Delta \mid \delta \in D \wedge s \in G \wedge \forall \delta' \in D,\ Pl(\delta') \in G\} \end{cases}$$

A use or provide port $p$ in an instance $i = (c, id, Q, \mathcal{M})$ is active if the group bound to it is active.

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

For an active provide port $p_{\text{p}}$ 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 de-activation 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:

$$refusing(i, p_{\text{p}}) \equiv \big(\forall e \in elements(Gr(p_{\text{p}})), e \in \mathcal{M} \implies exit(e, Gr(p_{\text{p}}))\big)$$

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

$$exit(e, G) \equiv e \in \Pi^c \wedge \big(\exists D, (e, b, D) \in \Theta\big) \wedge \big(\forall D \forall \delta, (e, b, D) \in \Theta \wedge \delta \in D \implies Pl(\delta) \notin G\big)$$

#### 5.2.4.2 Evolution of component instances

We now present the rules that describe the evolution of component instances, independent of any reconfiguration instruction. Each of 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.

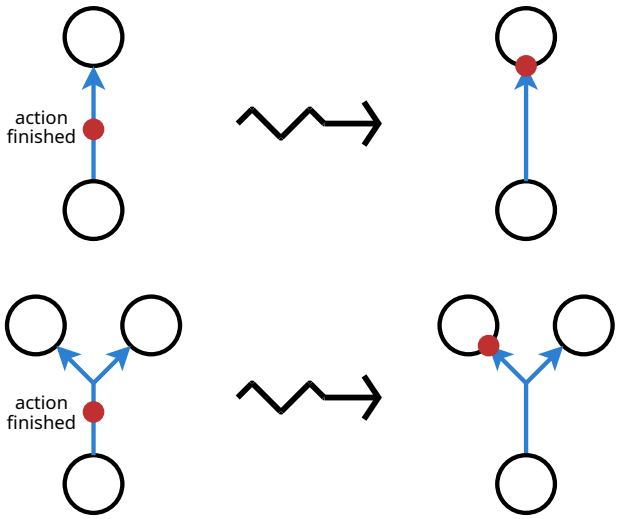



Figure 5.5: Illustration of the rule $\mathsf{End}^{\delta}_{\theta}$.

### Firing transitions

$$\frac{\pi \in \Pi^c \cap \mathcal{M} \qquad \forall (j, p_{\mathrm{u}}, i, p_{\mathrm{p}}) \in L,\ active(j, p_{\mathrm{u}}) \implies active(i', p_{\mathrm{p}})}{\langle (I \cup \{i\}, L), R\rangle \rightsquigarrow \langle (I \cup \{i'\}, L), R\rangle}\ \mathsf{Fire}^{b}_{\pi}$$

where

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

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 de-activate a provide port (in particular, one bound to a group containing $\pi$) that is being used by another component. This rule is illustrated by Figure 5.4.

### Ending transition

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

where

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

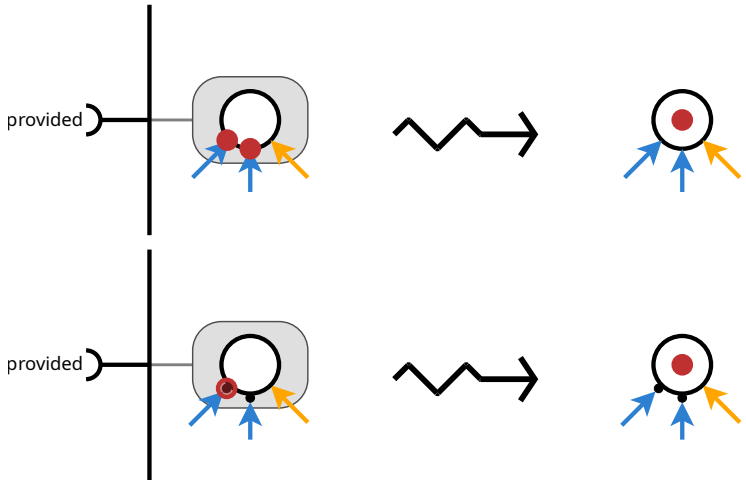



Figure 5.6: Illustration of the rule $\mathsf{Reach}_\delta$.

This rule represents the end of a transition by transferring the token from the transition to one of its ending. 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 outcome, depending on parameters (e.g., user input, time) that are not specified in the semantic model. This rule is illustrated by Figure 5.5.

**Entering place**

$$\frac{\delta \in \Delta \qquad E_\delta \subseteq \mathcal{M} \qquad \forall p_\mathrm{u},\ Pl(\delta) \in Gr(p_\mathrm{u}) \implies provided(i, p_\mathrm{u}) \wedge allowed(i, p_\mathrm{u})}{\langle (I \cup \{i\}, L), R\rangle \rightsquigarrow \langle (I \cup \{i'\}, L), R\rangle}\ \mathsf{Reach}_\delta$$

where

- $E_\delta = \{\big((\pi_\mathrm{source}, b, D), \delta\big) \mid (\pi_\mathrm{source}, b, D) \in \Theta^c \wedge \delta \in D\}$, i.e., the set of transitions ending that reach the station $\delta$
- $i = (id, c, Q, \mathcal{M})$
- $i' = (id, c, Q, \mathcal{M} \cup \{Pl(\delta)\} \setminus E_\delta)$

and

- $provided(i, p_\mathrm{u}) \equiv \exists (i, p_\mathrm{u}, j, p_\mathrm{p}) \in L,\ active(j, p_\mathrm{p})$, indicating that the requirement of the use port $p_\mathrm{u}$ is satisfied,
- $allowed(i, p_\mathrm{u}) \equiv \forall (i, p_\mathrm{u}, j, p_\mathrm{p}) \in L,\ refusing(j, p_\mathrm{p}) \wedge \neg\, active(i, p_\mathrm{u}) \implies \neg\, active(i', p_\mathrm{u})$, 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. This rule is illustrated by Figure 5.6.

**Finishing behavior**

$$\frac{\mathcal{M} \subseteq \Pi^c \qquad \forall (\pi, b', D) \in \Theta,\, b' = b : \pi \notin \mathcal{M}}{\langle (I \cup \{i\}, L), R\rangle \rightsquigarrow \langle (I \cup \{i'\}, L), R\rangle}\ \mathsf{Finish}_b$$

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.

#### 5.2.4.3 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**

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

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**

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

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**

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

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 $p_\text{u} \in {P_\text{u}}^{c_1}$ and $p_\text{p} \in {P_\text{p}}^{c_2}$.

The instruction $\texttt{con}(id_1, p_\text{u}, id_2, p_\text{p})$ adds a connection between the use port $p_\text{u}$ and provide port $p_\text{p}$ 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 $p_\text{u}$ is not already connected.

**Disconnect ports**

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

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 $p_\text{u} \in {P_\text{u}}^{c_1}$ and $p_\text{p} \in {P_\text{p}}^{c_2}$.

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

**Pushing behavior**

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

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**

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

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.

## 5.3 Performance Model

This section describes a performance model for Concerto reconfigurations. It relies on the formal semantics of Concerto presented in Section 5.2. 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 multi-set over $V \times \mathbb{R}_0^+ \times V$. 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 (in terms of weights) 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.

### 5.3.1 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 $\mathbb{R}_0^+$ 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. In the following, we refer to the entrance and exit places of a group $G$ respectively $in(G)$ and $out(G)$.

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, such as those presented in the evaluation of this work (Chapter 6).

### 5.3.2 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 $tokens_{\Pi}$ 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 $v^{\text{source}}$ that represents the beginning of the execution of the reconfiguration program.

**Data:** assembly $(I, L)$, reconfiguration $\iota_1 \cdot \iota_2 \cdot \ldots \cdot \iota_n$
**Result:** graph $(V, A)$

$V \leftarrow \{v^{\text{source}}\}$ ;
$A \leftarrow \emptyset$ ;
$tokens_{\Pi}, end_v \leftarrow f_{\perp}$ ;
**for** $(id, c, Q, \mathcal{M}) \in I$ **do**
  $tokens_{\Pi} \leftarrow tokens_{\Pi}[id := \mathcal{M}]$ ;
  $end_v \leftarrow end_v[id := v^{\text{source}}]$ ;
  **for** $p \in {P_u}^c \cup {P_p}^c$ **do**
    $V \leftarrow V \cup \{v^{\text{act}}_{id,p}, v^{\text{deact}}_{id,p}\}$ ;
  **end**
**end**
**for** $(i, p_u, j, p_p) \in L$ **do**
  $A \leftarrow A \cup \{(v^{\text{act}}_{id_j,p_{\text{p}}}, 0, v^{\text{act}}_{id_i,p_{\text{u}}}), (v^{\text{deact}}_{id_i,p_{\text{u}}}, 0, v^{\text{deact}}_{id_j,p_{\text{p}}})\}$ ;
**end**
$v^{\text{sync}} \leftarrow v^{\text{source}}$ ;
**for** $i$ **from** 1 **to** $n$ **do**
  **match** $\iota_i$ **with**
    **case** `wait`$(id)$ **do**
      $V \leftarrow V \cup \{v^{\text{wait}}_i\}$ ;
      $A \leftarrow A \cup \{(v^{\text{sync}}, 0, v^{\text{wait}}_i), (end_v(id), 0, v^{\text{wait}}_i)\}$ ;
      $v^{\text{sync}} \leftarrow v^{\text{wait}}_i$ ;
    **case** `con`$(id_1, p_u, id_2, p_p)$ **do**
      $A \leftarrow A \cup \{(v^{\text{sync}}, 0, v^{\text{act}}_{id_1,p_{\text{u}}}),(v^{\text{act}}_{id_2,p_{\text{p}}}, 0, v^{\text{act}}_{id_1,p_{\text{u}}}),(v^{\text{deact}}_{id_1,p_{\text{u}}}, 0, v^{\text{deact}}_{id_2,p_{\text{p}}})\}$ ;
    **case** `dcon`$(id_1, p_u, id_2, p_p)$ **do**
      $V \leftarrow V \cup \{v^{\text{dcon}}_i\}$ ;
      $A \leftarrow A \cup \{(v^{\text{sync}}, 0, v^{\text{dcon}}_i), (v^{\text{deact}}_{id_1,p_{\text{u}}}, 0, v^{\text{dcon}})\}$ ;
      $v^{\text{sync}} \leftarrow v^{\text{dcon}}_i$ ;
    **case** `pushB`$(id, b)$ **do**
      $extendGraph(id, b)$ ;
    **end**
  **end**
**end**
$V \leftarrow V \cup \{v^{\text{sink}}\}$ ;
**for** $(id, c, Q, \mathcal{M}) \in I$ **do**
  $A \leftarrow A \cup \{(end_v(id), 0, v^{\text{sink}})\}$ ;
**end**
$A \leftarrow A \cup \{(v^{\text{sync}}, 0, v^{\text{sink}})\}$ ;
**return** $(V, A)$ ;

**Algorithm 1:** The construction of the dependency graph.

The auxiliary function $end_v$ initially maps each component identifier to $v^{\text{source}}$, while the function $tokens_{\Pi}$ 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.

**Procedure** *extendGraph*$(id, b)$ **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** $\theta = (\pi, b', \{\delta\}) \in \Theta$ *such that* $b' = b$ **do**
    $V \leftarrow V \cup \{v^{\text{fire}}_{\theta}\}$ ;
    $A \leftarrow A \cup \{(v^{\text{leave}}_{\pi}, 0, v^{\text{fire}}_{\theta}), (v^{\text{fire}}_{\theta}, time(id, b), v^{\text{enter}}_{Pl(\delta)})\}$ ;
  **end**
  **for** $p_p \in {P_p}^c$ **do**
    $A \leftarrow A \cup \{(v^{\text{enter}}_{in(Gr(p_{\text{p}}),b)}, 0, v^{\text{act}}_{id,p_{\text{p}}}), (v^{\text{deact}}_{id,p_{\text{p}}}, 0, v^{\text{leave}}_{out(Gr(p_{\text{p}}),b)})\}$ ;
  **end**
  **for** $p_u \in {P_u}^c$ **do**
    $A \leftarrow A \cup \{(v^{\text{act}}_{id,p_{\text{u}}}, 0, v^{\text{enter}}_{in(Gr(p_{\text{u}}),b)}), (v^{\text{leave}}_{out(Gr(p_{\text{u}}),b)}, 0, v^{\text{deact}}_{id,p_{\text{u}}})\}$ ;
  **end**
  remove elements of $V$ and $A$ that are not reachable from $v^{\text{source}}_{id,b}$ ;
  $tokens_{\Pi} \leftarrow tokens_{\Pi}[id := \{\pi \mid v^{\text{enter}}_{\pi} \in V \wedge \neg\exists(\pi, b', D), b' = b\}]$ ;
  $end_v \leftarrow end_v[id := v^{\text{sink}}_{id,b}]$ ;
  **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 sub-graph for each instruction.

**Wait** Given an instruction `wait`$(id)$, the graph is extended with a unique vertex $v^{\text{wait}}$, representing the synchronization event. An arc is added from the vertex $v^{\text{sync}}$ to $v^{\text{wait}}$ (synchronization occurs after the previous synchronization barrier) and another from $end_v(id)$ to $v^{\text{sync}}$ (synchronization occurs after the component identified by $id$

has executed all its behavior requests). The auxiliary variable $v^{\text{sync}}$, which represents the last synchronization point, is updated.

**Connection** Given an instruction $\texttt{con}(id_1, p_{\text{u}}, id_2, p_{\text{p}})$, 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 $v^{\text{sync}}$.

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

**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 *extendGraph* (Algorithm 2). Given a component identifier $id$ and a behavior $b$, *extendGraph* extends the graph with vertices representing the events occurring during the execution of that behavior. The construction of this sub-graph 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 $v^{\text{source}}_{id,b}$ and $v^{\text{sink}}_{id,b}$ 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 $v^{\text{sync}}$ to ensure that the last synchronization point is taken into account.

For each place $\pi$ in the component type, a vertex $v^{\text{enter}}_{\pi}$ 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 $v^{\text{source}}_{id,b}$. Another vertex $v^{\text{leave}}_{\pi}$ 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 $\theta = (\pi, b', \{\delta\})$ such that $b' = b$, one vertex $v^{\text{fire}}_{\theta}$ is added, connected to $v^{\text{leave}}_{\pi}$ to encode the fact that the transition may only start after a token leaves its source place, and to $v^{\text{enter}}_{Pl(\delta)}$ 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 $\theta$.

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

Note that the sub-graph that was just constructed to describe the event 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 $v^{\mathrm{source}}_{id,b}$ 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 $v^{\mathrm{sink}}_{id,b}$, denoting the end of the behaviors when all final places are reached.

### 5.3.3 Example

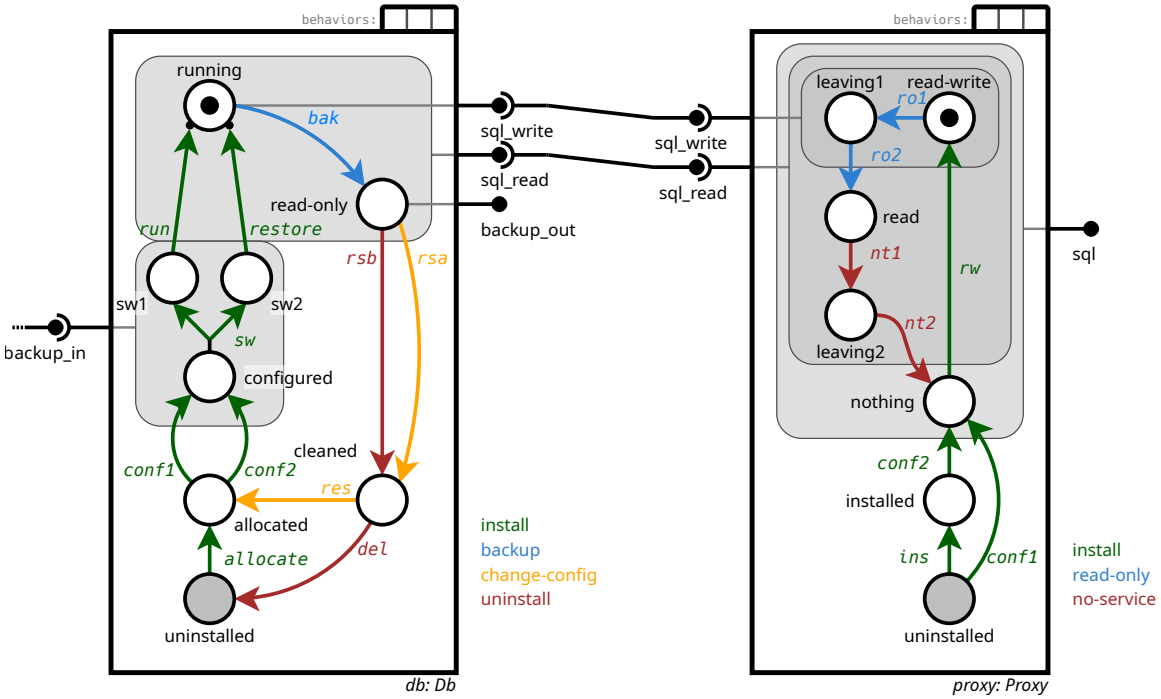



Figure 5.7: Concerto assembly composed of one instance of each component type of Figure 5.1. The two instances are in their deployed state.

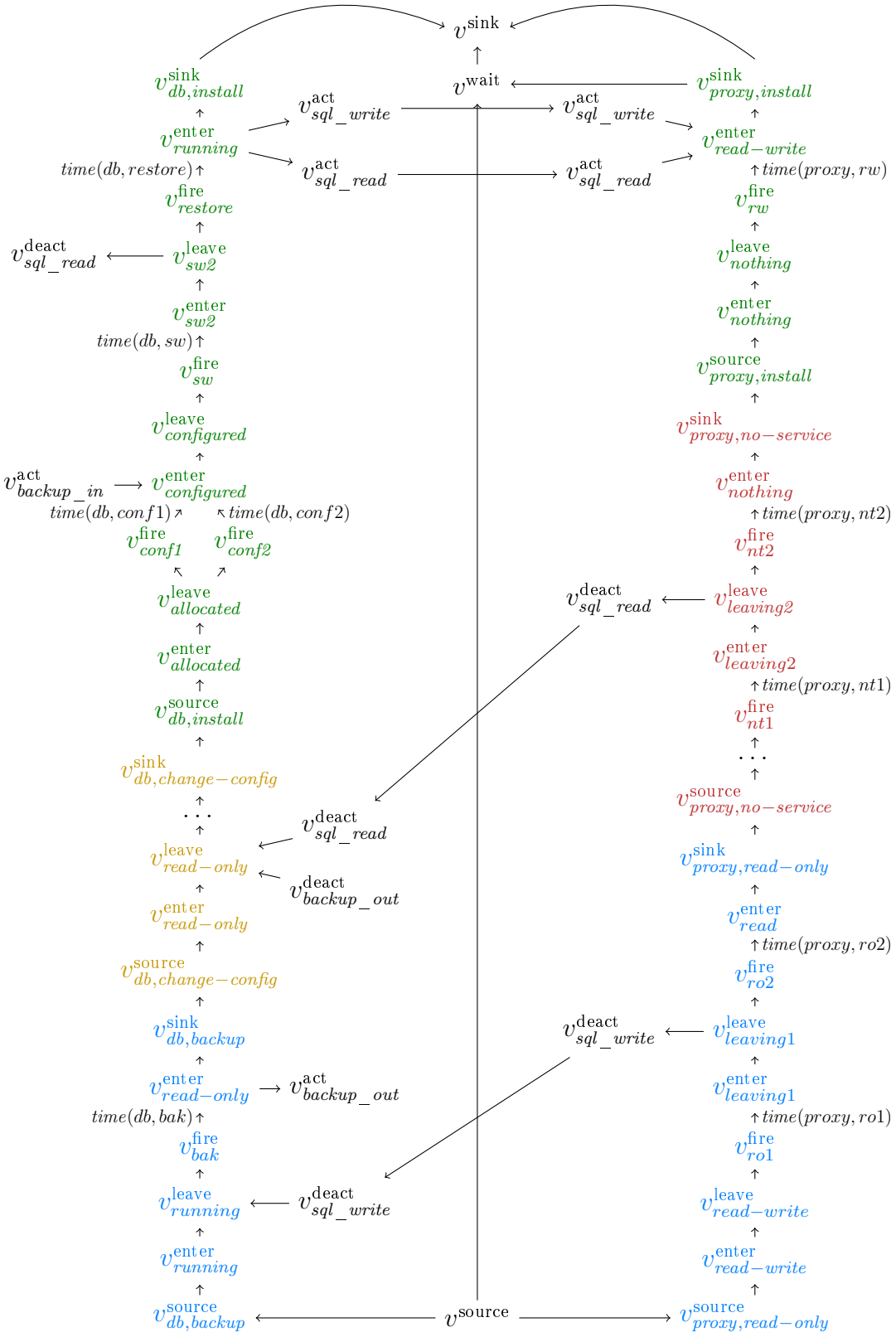



Figure 5.8: Dependency graph corresponding to the reconfiguration program in Listing 5.2 (on page 88) applied to the assembly in Figure 5.7. Each sub-graph is represented with the color corresponding to its behavior. Note that the use port `backup_in` of instance `db` is assumed to be provided.

Figure 5.8 shows the dependency graph corresponding to the reconfiguration presented in Listing 5.2 (on page 88) applied to the assembly in Figure 5.7. Each color of the colored vertices correspond to the sub-graph generated by a call to *extendGraph*(`pushB`($id, b$)) for some identifier $id$ and behavior $b$. The color of the sub-graphs matches the representation of the behavior $b$ in Figure 5.7. The graph also contains a vertex $v^{\text{wait}}$ generated by *extendGraph*(`wait`(*proxy*)), and vertices $v^{\text{source}}$ and $v^{\text{sink}}$ generated in the initial phase of the graph construction. For vertices corresponding to transitions, the weight is indicated. All other vertices have a weight of 0.

## 5.4 Behavioral Interfaces

Concerto component types expose detailed information about the life-cycle of a software module. However, from the perspective of the reconfiguration developer, a part of this information is irrelevant and is detrimental to good separation of concerns. For instance, parallelism within the component, or the detailed set of transitions executing is irrelevant. What is important however are the set of behaviors which can be executed at any given time, which use or provide ports are affected by the execution of a behavior (and in which order) and which use or provide ports are active when the component is stable, i.e., not executing any behavior.

Behavioral interfaces are a view generated for a Concerto component type which expose only this useful information to the reconfiguration developer. It can also be viewed as a contract/interface, which component types can implement (distinct component types can have the same behavioral interface). Two component types implementing the same interface are guaranteed to behave in the same way from the point of view of the reconfiguration developer.

### 5.4.1 Definition

Given a Concerto component type, its *behavioral interface* contains the following information: its ports, its behaviors, and a state-machine composed of a set of *stable states* as states and a set of *behavior executions* as transitions. Intuitively, stable states correspond to a state that can be reached by an instance of the component type by executing a given sequence of behaviors, starting from the initial place. They are characterized by which use or provide ports are active and which behaviors can be executed after reaching them. Behavior executions correspond to the execution of a given behavior, which leads from a stable state to another stable state. They are defined by the origin and destination stable states, the associated behavior and the set of partially ordered port-related events (activation or deactivation of a port) which occur during the execution of the behavior. Note that the order is partial because if there are parallel transitions within the component, the order in which port-related event occur might depend on the execution time of the transitions.

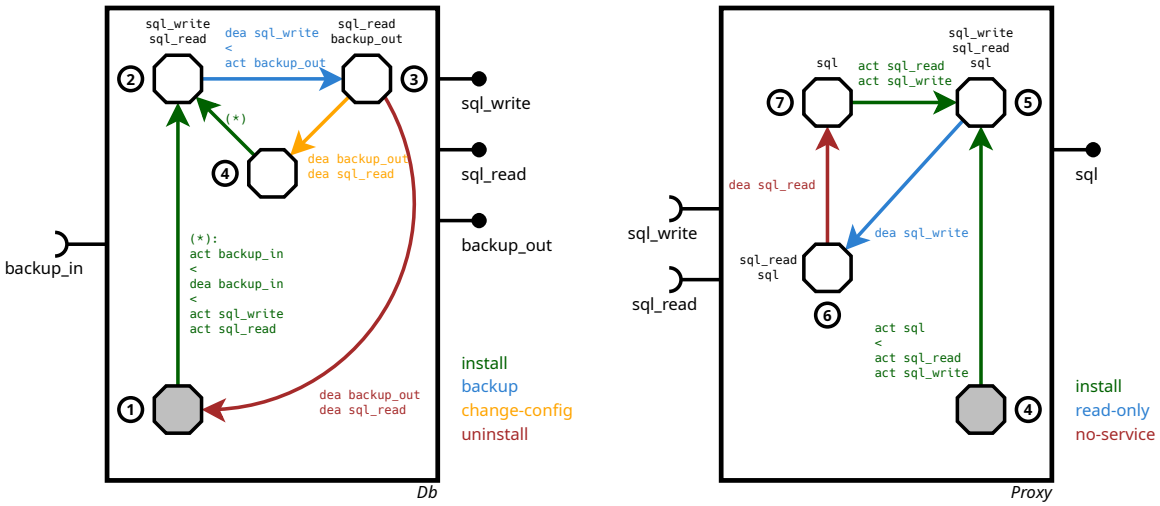



Figure 5.9: Behavioral interfaces corresponding to the component types presented in Figure 5.1 (on page 81). Octagons represent *stable states* while colored arrows represent *behavior executions*. Stable states are tagged with their active ports, while behavior executions are tagged with a partial order for their port events. Two events on consecutive lines are considered to happen at the same time, while a $<$ indicates that all events above happen before all events below. Note that in this example, the orders happens to be total. A *act* (resp. *dea*) event corresponds to the activation (resp. deactivation) of a use or provide port. Numbers in circles are only for referencing states in the text and do not bear any meaning.

For example, Figure 5.9 shows behavioral interfaces for the component types of Figure 5.1 (on page 81). In *Db*, stable state 1 is the initial state, which in practice corresponds to having one token in place *uninstalled*. Note that it is possible to go back to this state by executing behaviors *install*, *backup* and *uninstall* any number of times in this order. In this state, no ports are active. Stable state 2 is reached by executing *install* from the initial state. In this state, ports *sql_write* and *sql_read* are active. In practice, it corresponds to a token being in place *running*. A behavior execution of behavior *install* allows to go from stable state 1 to stable state 2. During this execution, four events happen: first, use port *backup_in* starts to be used, then it stops to be used, and then provide ports *sql_write* is activated at the same time as provide port *sql_read*.

**Formal description** Formally, a behavioral interface is defined by a tuple of 6 elements $(\Sigma, B, P_{\mathrm{u}}, P_{\mathrm{p}}, \rightleftharpoons, E)$. $\Sigma$ is a set of stable states, $B$ is a set of behaviors, $P_{\mathrm{u}}$ is a set of use ports and $P_{\mathrm{p}}$ is a set of provide ports. Note that $B$, $P_{\mathrm{u}}$ and $P_{\mathrm{p}}$ are

exactly identical to the elements of the same name in the tuple of a component type. $\rightleftharpoons \subseteq \Sigma \times (P_{\mathrm{u}} \cup P_{\mathrm{p}})$ is a binding relation between stable states and ports, defining which ports are active in each state. Finally, $E \subseteq \Sigma \times B \times Ev \times \Sigma$ is a set of behavior executions corresponding of a source state, a behavior, a set of ordered events and a destination state. A set of ordered events in $Ev$ is defined as a tuple $(\Omega_\epsilon, \preccurlyeq_\epsilon)$ where $\Omega_\epsilon \subseteq \{\mathrm{act}, \mathrm{dea}\} \times (P_{\mathrm{u}} \cup P_{\mathrm{p}})$ is a set of events and $\preccurlyeq_\epsilon \subseteq \Omega_\epsilon \times \Omega_\epsilon$ is a partial order over these events.

### 5.4.2 Generating a behavioral interface

In this section, we present an algorithm to generate the behavioral interface of a Concerto component type. This algorithm is a proof of concept and it is conceived with clarity in mind, as opposed to optimization in terms of complexity. It is also restricted to component types which do not have parallel exit or entry points in groups (i.e.,, for each group and each behavior, there is only one place which when receiving a token causes the group to become active and only one place which when losing a token causes the group to become inactive).

Algorithm 3 contains the main function `GetInterface` as well as two auxiliary functions, `ExploreState` and `ExploreBehavior`. Other auxiliary functions are given in Algorithms 4 and 5. We describe the algorithm in a top-down fashion, starting with function `GetInterface`.

**Notation** In the following, tuples usually denoted $(e_1, \ldots, e_n)$ in mathematics are denoted $\langle e_1, \ldots, e_n \rangle$ to disambiguate with function calls.

#### 5.4.2.1 Description of the algorithm

**GetInterface** The main function, `GetInterface`, takes as input a component type $c$ and returns a tuple $\langle \Sigma, E \rangle$ where $\Sigma$ is the set of stable states of its behavioral interface and $E$ is its set of behavior executions. The main idea of the algorithm is to explore the possible behavior executions starting from known stable states to discover new stable states, until no more stable states can be discovered. We start with the one stable state common to all component types: the one corresponding to a token being in the initial place of the internal-net. Line 2 initializes the explored states to an empty set, the discovered behavior executions to an empty set, and the set of states to explore as containing the single set $\{\pi_{\mathrm{init}}\}$. Remind that stable states correspond to places holding tokens in the original component type. In this algorithm, stable states are encoded by the set of places holding a token they correspond to. Line 3 is the main loop of the algorithm and consists in exploring stable states (i.e., finding the behavior executions starting from that state) as long as the set of states to explore is not empty. Function `ExploreState` does that exploration of a state. `ExploreState` $(\sigma, c)$ returns a tuple containing the set of newly discovered stable states and the set of newly

```
1  Function GetInterface(c = ⟨Π, π_init, Δ, Pl, Θ, B, P_u, P_p, Gr⟩):
2      Σ ← ∅ ; E ← ∅ ; Σ_to_explore ← {{π_init}}
3      while Σ_to_explore is not empty do
4          extract σ from Σ_to_explore
5          ⟨Σ_discovered, E_discovered⟩ ← ExploreState(σ, c)
6          Σ ← Σ ∪ {σ}
7          Σ_to_explore ← Σ_to_explore ∪ Σ_discovered \ Σ
8          E ← E ∪ E_discovered
9      end
10     return ⟨Σ, E⟩
11 AuxFunction ExploreState(σ,c = ⟨Π, π_init, Δ, Pl, Θ, B, P_u, P_p, Gr⟩):
12     Σ_discovered ← ∅ ; E_discovered ← ∅
13     explored_behaviors ← ∅
14     for ⟨π_source, b, D⟩ in Θ do
15         if π_source ∈ σ and b is not in explored_behaviors then
16             results ← ExploreBehavior(b, σ, c)
17             for ⟨final_places, events⟩ in results do
18                 Σ_discovered ← Σ_discovered ∪ {final_places}
19                 E_discovered ← E_discovered ∪ {⟨σ, b, events, final_places⟩}
20             end
21             explored_behaviors ← explored_behaviors ∪ {b}
22         end
23     end
24     return ⟨Σ_discovered, E_discovered⟩
25 AuxFunction ExploreBehavior(b,σ,c = ⟨Π, π_init, Δ, Pl, Θ, B, P_u, P_p, Gr⟩):
26     choices ← InitSwitchChoices(b, c)
27     results ← ∅
28     for choice in choices do
29         results ← results ∪ {ExploreBehaviorChoice(choice, b, σ, c)}
30     end
31     return results
```

**Algorithm 3:** Main function `GetInterface` to get a behavioral interface from a component type along with auxiliary functions `ExploreState` which finds possible behavior executions from a given state, and `ExploreBehavior` which computes the execution of a given behavior from a given state.

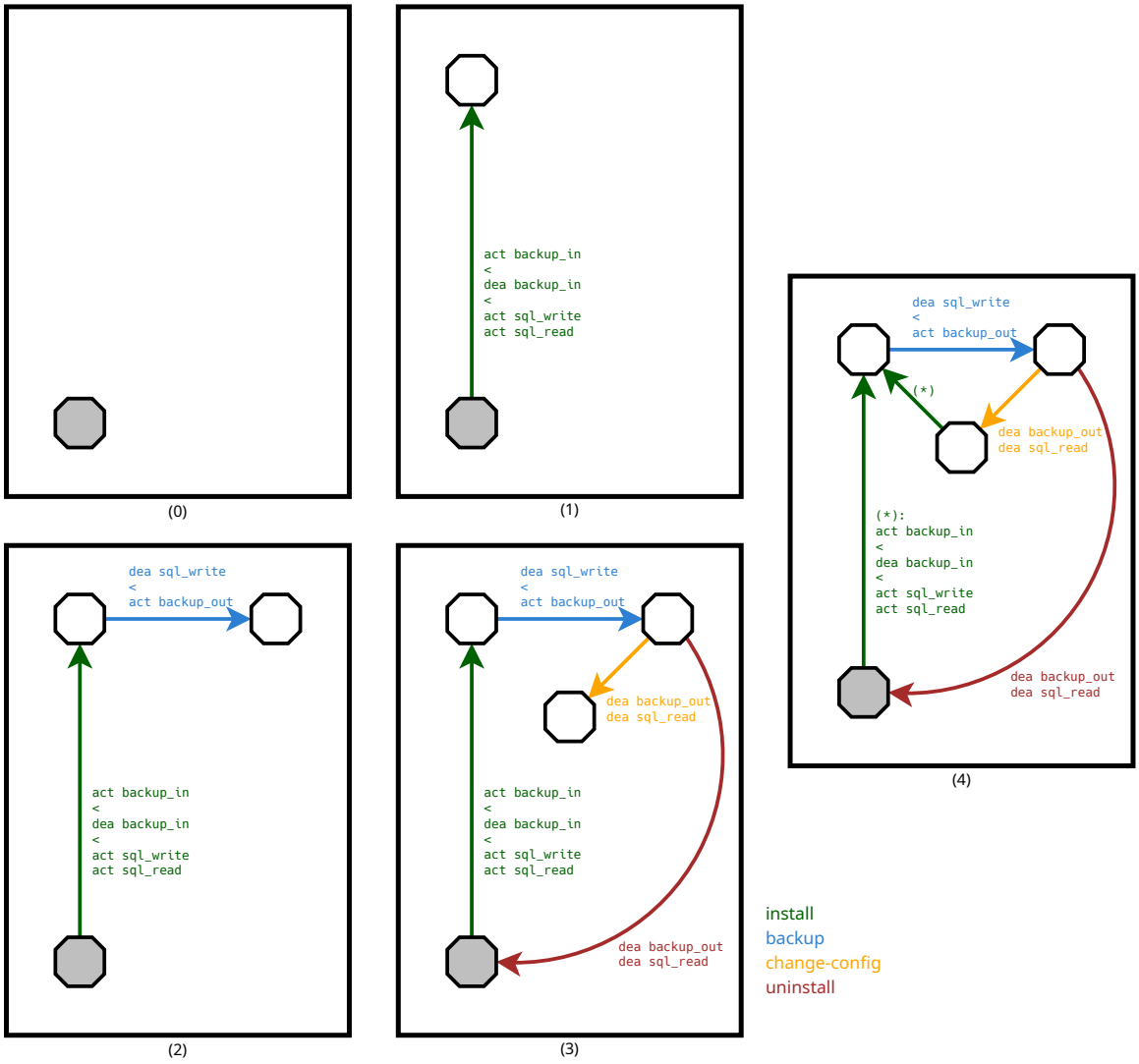



Figure 5.10: Execution of the main loop of function `GetInterface` of Algorithm 3 on component type `Db` presented in Figure 5.1 (on page 81), eventually obtaining the behavioral interface shown in Figure 5.9. The number below each sub-figure corresponds to the number of iterations of the loop which have been performed.

discovered behavior executions while exploring state $\sigma$. In the main loop, a state to explore is extracted (line 4), explored (line 5) and marked as explored (line 6). The set of states to explore is populated with the newly discovered states, minus those already explored (line 7) and the set of discovered behavior executions is populated by the newly discovered ones (line 8).

Figure 5.10 illustrates how the behavioral interface of component type `Db` presented in Figure 5.1 (on page 81) is constructed by showing the state machine de-

scribed by $\Sigma \cup \Sigma_{\text{to\_explore}}$ and $E$ at the beginning of each iteration of the loop. Notice that this algorithm returns the set of stable states and the set of behavior executions, but, for the sake of conciseness, not the binding relation between stable states and ports. Because stable states are encoded by the set of places holding a token they represent, this is easy to deduce from $\Sigma$ itself and is left as exercise to the reader.

**ExploreState** Recall that `ExploreState` is a function that takes as input a stable state $\sigma$ and the description of a component type $c$ and returns a tuple containing the set of newly discovered stable states and the set of newly discovered behavior executions while exploring state $\sigma$. This is done by exploring each behavior which has at least one transition outgoing from one place in stable state $\sigma$. To do this, line 14 loops over the transitions in $\Theta$ and if its source place is in $\sigma$ and the behavior has not yet been explored (line 15), a call is made to function `ExploreBehavior` (line 16) which, given a behavior, a starting state and a component type returns the set of possible executions of this behavior (there can be multiple executions when there are switches: the events caused by the execution of a behavior might not be the same depending on which branch of a switch is taken). Each behavior execution in the returned set is encoded as a tuple containing the stable state reached after the execution (encoded as a set of places) and a partially ordered set of events. The loop from line 17 to line 20 updates the sets of discovered stable states and behavior executions and the behavior just explored is marked as such line 21.

**ExploreBehavior** Recall that `ExploreBehavior` takes as input a behavior, a starting state and a component type and returns the set of possible executions of this behavior encoded as a tuple containing the stable state reached after the execution and a partially ordered set of events. This function only iterates through the possible combination of choices that can be made in switches (i.e., which output station the token goes to when leaving the transition). An example of choice mapping is illustrated by Figure 5.11. The list of possible choice mappings is computed by function `InitSwitchChoices` (defined in Algorithm 5) on line 26, and the loop from line 28 to line 30 goes through each of them. For each choice possibility, function `ExploreBehaviorChoice` (defined in Algorithm 4) is used to compute the behavior execution (encoded as a tuple containing the stable state reached after the execution and a partially ordered set of events). This behavior is added to the list *results* which is returned at the end of the loop.

**Partially ordered event sets** Before defining the `ExploreBehaviorChoice` function, we define a few primitives to be used to construct partially ordered event sets. We define an event as an element of $\{\text{act}, \text{dea}\} \times \langle P_{\text{u}} \cup P_{\text{p}} \rangle$ (where *act* corresponds to the activation of a port and *dea* to the deactivation of a port).

Recall that a set of ordered events is defined as a tuple $\langle \Omega, \preccurlyeq \rangle$ where $\Omega \subseteq \{\text{act}, \text{dea}\} \times \langle P_{\text{u}} \cup P_{\text{p}} \rangle$ is a set of events and $\preccurlyeq \subseteq \Omega \times \Omega$ is a partial order over these

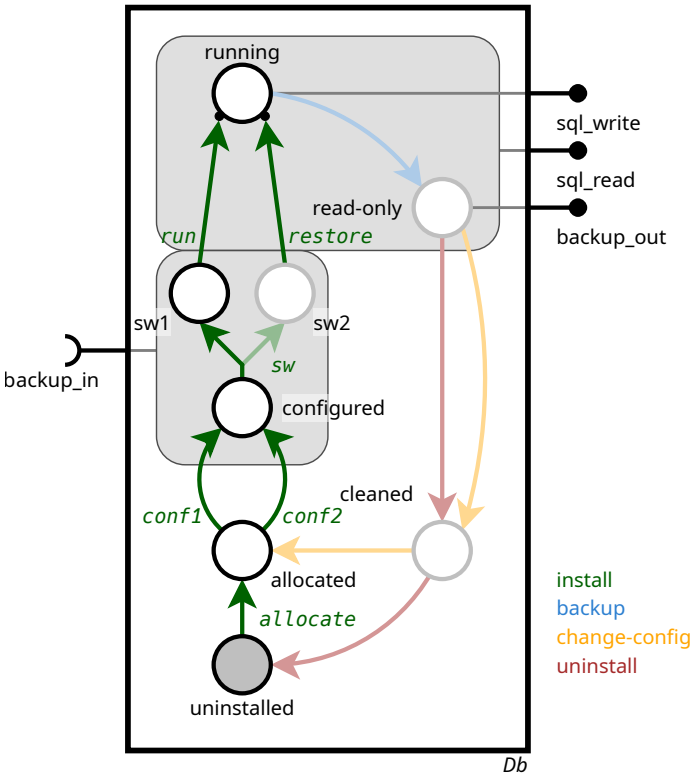



Figure 5.11: Illustration of a choice mapping as produced by function `InitSwitchChoices` of Algorithm 4 on component type `Db` presented in Figure 5.1 (on page 81), with initial stable state $\{uninstalled\}$ and behavior `install`. In this mapping, transition `allocate` maps to the station of its only transition ending. The same goes for transitions `conf1`, `conf2`, `run` and `restore`. In the case of transition `sw`, among two possible stations, it is mapped to the one of place `sw1`.

events.

We define:

$$\texttt{Nothing} \coloneqq \langle \emptyset, \emptyset \rangle$$

as the empty partially ordered events set.

Given an event $\omega \in \Omega$, we define:

$$\texttt{Ev}\,(\omega) \coloneqq \langle \{\omega\}, \emptyset \rangle$$

which gives a partially ordered events set containing this single event.

Given $n$ events $\omega_1 \ldots \omega_n$ in $\Omega$, we define:

$$\texttt{Sim}\,\{\omega_1, \ldots, \omega_n\} \coloneqq \left\langle \{\omega_1, \ldots, \omega_n\}, \bigcup_{i=1}^{n} \bigcup_{j=i+1}^{n} \{(\omega_i, \omega_j), (\omega_j, \omega_i)\} \right\rangle$$

which gives a partially ordered events set containing events $\omega_1 \dots \omega_n$ with all of them considered as happening at the same time (they are all equal for $\preccurlyeq$).

Given two partially ordered sets $s_1 = \langle \Omega_1, \preccurlyeq_1 \rangle$ and $s_2 = \langle \Omega_2, \preccurlyeq_2 \rangle$, we define:

$$\texttt{Seq}\,(s_1, s_2) \coloneqq \left\langle \Omega_1 \cup \Omega_2, \preccurlyeq_1 \cup \preccurlyeq_2 \cup \bigcup_{\langle \omega_1, \omega_2 \rangle \in \Omega_1 \times \Omega_2} \langle \omega_1, \omega_2 \rangle \right\rangle$$

which gives a partially ordered events set corresponding to the union of $s_1$ and $s_2$ with, in addition, all events of $s_1$ being ordered before all events of $s_2$.

Finally, given $n$ partially ordered sets $s_1 = \langle \Omega_1, \preccurlyeq_1 \rangle, \dots, s_n = \langle \Omega_n, \preccurlyeq_n \rangle$, we define:

$$\texttt{Par}\,\{s_1, \dots, s_n\} \coloneqq \langle \Omega_1 \cup \Omega_2, \preccurlyeq_1 \cup \preccurlyeq_2 \rangle$$

which gives a partially ordered events set corresponding to the union of $s_1$ and $s_2$ with no additional ordering (allowing events of $s_1$ and $s_n$ to occur in parallel).

**ExploreBehaviorChoice** The `ExploreBehaviorChoice` function is defined in Algorithm 4. It takes as input a choice map *choice* (associating each transition to one output station of this transition), a behavior $b$, a starting stable state $\sigma$ and a component type $c$, and returns the behavior execution (encoded as a tuple containing the stable state reached after the execution and a partially ordered set of events). This function simulates the execution of the behavior in a component instance of type $c$ with tokens starting in each place contained in $\sigma$ while remembering at each step when use and provide ports were activated or deactivated. However, unlike an actual execution of Concerto's semantics, it does not make any assumption on the order in which semantic rules are applied. In order to keep track of the events during the execution, when a place is reached during the simulation, the partially ordered set of events that happened during the execution so far is associated to the place.

Lines 2 to 10 consist in initializing variables. *act_places* is the set of active places in the simulation and is initialized to contain the places contained in $\sigma$. *act_stations* is the set of active stations, corresponding to when there is a token in the transition ending associated to this station of all the incoming transitions of the station for a given behavior. *final_places* is the set of place holding a token which have no outgoing transitions of the given behavior. At the end of the simulation, this set corresponds to the destination stable state of the behavior execution. *station_sources* is a dictionary associating to each station the set of origin places of the transitions leading to this station. Originally, each station is associated to an empty set (lines 5 to 7) and the mapping is updated during the simulation. *event_logs* is a dictionary associating each place to the partially ordered set of events that happened before arriving to that place. Originally, each place is associated to `Nothing`, i.e., the empty partially ordered events set (lines 8 to 10), and the mapping is updated during the simulation. *station_counts* is a dictionary used to keep track of how many more tokens a station needs to receive before tokens can be moved from transition endings

```
1  AuxFunction ExploreBehaviorChoice(choice,b,σ,c = ⟨Π, π_init, Δ, Pl, Θ, B, P_u, P_p, Gr⟩):
2      act_places ← σ ; act_stations ← ∅ ; final_places ← ∅
3      station_sources ← EmptyDictionary ; event_logs ← EmptyDictionary
4      station_counts ← InitStationCounts (b, c)
5      for δ in Δ do
6          station_sources [δ] ← ∅
7      end
8      for π in σ do
9          event_logs [π] ← Nothing
10     end
11     while act_places is not empty or act_stations is not empty do
12         if act_places is not empty then
13             π ← extract one from act_places
14             out_trans ← {⟨π_source, b′, D⟩ | b′ = b ∧ π_source = π}
15             if out_trans is empty then
16                 final_places ← final_places ∪ {π}
17             else
18                 deact_ports ← PlacePorts (π, c)
19                 for θ in out_trans do
20                     δ ← choice [θ]
21                     deact_ports ← deact_ports \ PlacePorts (Pl (δ), c)
22                     station_counts [δ] ← station_counts [δ] − 1
23                     station_sources ← station_sources ∪ {π}
24                     if station_counts [δ] = 0 then
25                         act_stations ← act_stations ∪ {δ}
26                     end
27                 end
28                 event_logs [π] ← Suc ⟨event_logs [π], Sim {⟨dea, p⟩ | p ∈ deact_ports}⟩
29             end
30         end
31         if act_stations is not empty then
32             δ ← extract one from act_stations
33             π ← Pl (δ)
34             act_ports ← PlacePorts (π, c)
35             for π_source in station_sources [δ] do
36                 act_ports ← act_ports \ PlacePorts (π_source, c)
37             end
38             trans_events ← Par {event_logs [π_source] | π_source ∈ station_sources [δ]}
39             act_events ← Sim {⟨act, p⟩ | p ∈ act_ports}
40             event_logs [π] ← Suc (trans_events, act_events)
41         end
42     end
43     return ⟨final_places, Par {event_logs [π] | π ∈ final_places}⟩
```

**Algorithm 4:** Auxiliary function `ExploreBehaviorChoice` which computes the execution of a behavior given a starting state and a choice of output station for each transition (used in the case of a switch).

```
1  AuxFunction InitStationCounts(b,c = ⟨Π, π_init, Δ, Pl, Θ, B, P_u, P_p, Gr⟩):
2      station_counts ← EmptyDictionary
3      for δ in Δ do
4          station_counts[δ] ← 0
5      end
6      for ⟨π_source, b', D⟩ in Θ do
7          if b' = b then
8              for δ in D do
9                  station_counts[δ] ← station_counts[δ] + 1
10             end
11         end
12     end
13     return station_counts
14 AuxFunction InitSwitchChoices(b,c = ⟨Π, π_init, Δ, Pl, Θ, B, P_u, P_p, Gr⟩):
15     choices ← [EmptyDictionary]
16     for θ = ⟨π_source, b', D⟩ in Θ do
17         if b' = b then
18             new_choices ← []
19             for δ in D do
20                 for choice in choices do
21                     new_choice ← choice
22                     new_choice[θ] ← δ
23                     new_choices ← new_choices + [new_choice]
24                 end
25             end
26             choices ← new_choices
27         end
28     end
29     return choices
30 AuxFunction PlacePorts(π,c = ⟨Π, π_init, Δ, Pl, Θ, B, P_u, P_p, Gr⟩):
31     ports ← ∅
32     for p in P_u ∪ P_p do
33         if π is in Gr(p) then
34             ports ← ports ∪ {p}
35         end
36     end
37     return ports
```

**Algorithm 5:** Auxiliary functions which are used by functions in Algorithms 3 and 4 to initialize variables. `InitStationCounts` returns a dictionary associating to each station the number of incoming transitions of a given behavior, i.e., the number of tokens needed to enter a place through this station. `InitSwitchChoices` returns a list of all possible choices for switches of a given behavior, encoded by dictionaries which each associate one output station to each transition of a given behavior. `PlacePorts` returns the list of ports associated to a group containing a given place.

to a place. It is initialized by using function `InitStationCounts` defined in Algorithm 5. Originally, each station is associated to its number of incoming transitions, and this number is decreased during the simulation when a token is put in a relevant transition ending. When it reaches 0, there are enough tokens.

Lines 11 to 42 consist of a loop simulating the semantic rules of Concerto in a simplified way. Lines 12 to 30 treat the case of active places, and lines 31 to 41 treat the case of active stations.

If an active place exists, it is extracted from the set of active places (line 13). If it has no outgoing transitions for the given behavior, it is added to the set of final places (lines 14 to 16). Otherwise, the execution of the semantic rules allowing the token to go from the place to the transition endings of its outgoing transitions is simulated. Line 18, variable *deact_ports* is initialized with the set of ports containing the current place in their group. This set is filtered in the following instructions to end up being the set of ports which are deactivated when a token leaves the place. Lines 19 to 27, for each transition, the ports which are still active in the destination place are removed from *deact_ports*, the current place is added to the set of station sources of the destination station (according to the *choice* mapping), the count of that station is decremented, and if it reaches 0 the station is added to the set of active stations. Finally, line 28, the set of deactivated ports is known, so the simultaneous deactivation of these ports is added to the partially ordered events set memorized for the current place.

If an active station exists, it is extracted from the set of active stations (line 32). Then, the set of activated ports when entering the place this station is attached to, *act_ports*, is determined by listing all the ports with the given place in their group (line 34) and removing the ports which had at least one of the source places of the station in their group (lines 35 to 37). Finally, the partially ordered events set of this place is defined as the sequence of two sets of events: the parallel execution of the sets of events of the origin places and the simultaneous execution of the activation of the ports in the *act_ports* set.

Finally, a tuple is returned, consisting of the set of final places (corresponding to the stable state reached by the simulated behavior execution) and of the partially ordered events set consisting of the events occurring while reach each of the places, in parallel.

#### 5.4.2.2 Discussion

This algorithm is a proof of concept conceived for clarity, and it could be improved. In terms of complexity for example, some computations are done multiple times like the calls to `PlacePorts`, which could be pre-computed, or the iteration over all edges to find the outgoing transitions of a place, which could be made more efficient by using appropriate data structures. However, this algorithm is polynomial in the size of all the sets composing component type $c$, with one exception: `ExploreBehavior`

makes an exponential number of calls to `ExploreBehaviorChoice` with respect to the number of switches in the considered behavior. However, we argue that the number of switches in a component type is meant to remain low.

## 5.5 Discussion

**Expressivity** Concerto is a strict super-set of Madeus, in the sense that any Madeus assembly can be created in Concerto. To that end, a Concerto component type is created for each Madeus component in the assembly. Each component has a single behavior *deploy*, and the internal-nets would be similar, with some additional dummy places or transitions to account for the fact that in Madeus, use ports are bound to transitions and not places or groups of places. Then, a reconfiguration program can be generated to create one instance of each component type, connect the ports like they are connected in the Madeus assembly and push the *deploy* behavior in each component instance.

However, Concerto has a much broader expressiveness compared to Madeus. First, a paradigm change allows not to "forget" the modules once deployed, but to keep tracking their current state in their life-cycle and reconfigure them. In each component, behaviors correspond to different local reconfigurations that can be applied to a given module, depending on its current state. In order to perform reconfiguration, a reconfiguration language is provided by Concerto to dynamically update the assembly of components, i.e., the architecture of the overall distributed system, as well as trigger the reconfiguration of individual software modules.

**Performance of reconfiguration** Concerto supports a high level of parallelism during reconfiguration, both at the module level with parallel transitions and at the assembly level with mostly asynchronous execution of behaviors. The performance model provided with Concerto allows, with some restrictions on the component types, to express the total execution time of a reconfiguration program as a function of the execution time of each transition, i.e., of each reconfiguration action they are associated to. In practice, the restrictions on the component types are not problematic because they either align with usage (e.g., restrictions on groups) or can be easily worked around if necessary (e.g., by replacing a switch with a single transition going to the intended place).

**Separation of concerns** In Concerto, component types are created by module developers. They are specialists who know the life-cycles of their modules and translating them to a Concerto component type is quite straightforward. Reconfiguration programs are created by reconfiguration developers. They need to understand how each module of a distributed system can be interacted with, and how modules interact with each other. This is made easier by behavioral interfaces, which expose only

the information necessary to use the component types in a reconfiguration (which behavior can be executed, in which state the component will be after its execution and which ports are affected). Finally, reconfiguration programs are executed by system administrators. Each reconfiguration program can be simulated (performing the changes on the assembly without actually executing the actions associated to the transitions) to ensure that the resulting assembly corresponds to the expectations prior to the execution of the program. Also, the expected execution time of a reconfiguration program can be broken down to the execution time of individual reconfiguration actions. This can help estimating or bounding it, helping the system administrator or the autonomic tools to decide whether to execute the reconfiguration or not.

**Limitations** In terms of expressivity, Concerto is designed as a low-level framework which can be used by reconfiguration planning systems to write and execute a reconfiguration plan. As such, specific functionalities such as error handling or cardinality are not addressed with specific features. However, solutions dedicated to these issues can be built on top of Concerto by generating component types and reconfiguration programs, making use of Concerto features such as switches to model alternative reconfiguration actions in case of error for example.

In terms of performance, the main limitation that we can identify is the *wait* instruction of the reconfiguration language, which is global to the whole reconfiguration program. In the case of very large assemblies, one might need for optimal performance to keep executing reconfiguration instructions on a part of the assembly, while the other part keeps on executing reconfiguration instructions. This could be overcome by changing the language to support multiple independent *threads* of execution, or allowing the execution of concurrent reconfigurations.

In terms of separation of concerns, a Concerto component type cannot be provided as-is without documentation. For example, ports are not typed, and no guarantee is provided that all dependencies are represented by ports of the component type. This design allows greater flexibility, but can be detrimental to good separation of concerns. However, tools with higher-level concepts like typed ports can be made to generate Concerto component types.

## 5.6 Conclusion

In this chapter we have presented Concerto, a model for the reconfiguration of distributed systems. Component types are used to model the life-cycles of each module of distributed systems. Each component type has a set of behaviors, which correspond to local reconfigurations which can be performed on instances of this component. Using a reconfiguration language, one can write a reconfiguration program which creates or changes an existing assembly by triggering local reconfiguration actions for the modules asynchronously (through their behaviors) while ensuring the coordination

between the life-cycles of inter-dependent modules.

A full formalization of the model is provided, covering the definitions of component types, assemblies, reconfiguration programs as well as the complete operational semantics. A performance model is also provided to express the total execution time of a reconfiguration program as a function of the execution time of individual transitions, i.e., reconfiguration actions local to modules. This will be used in Chapter 6 to bound and estimate the total running time thanks to historical data.

Finally, Concerto provides behavioral interfaces, i.e., views of the component types exposing only the information required by reconfiguration developers. They fulfill two objectives: first, they increase separation of concerns by relieving the reconfiguration developers from having to understand the internals of a component type, and second they act as a sort of contract, ensuring that replacing a component type with another sharing the same behavioral interface will not change how it must be handled by reconfiguration programs.

# Chapter 6

# Evaluation

## Contents

In the previous chapters, the Madeus deployment model and the Concerto reconfiguration model were presented. Both models provide mechanisms for parallelism of deployment and reconfiguration actions, while providing performance models to define the expected level of parallelism precisely. They also target their concepts to different actors: module developers, reconfiguration developers and system administrators.

In Chapter 3 we have identified that one challenge in reconfiguration is to reconcile separation of concerns and performance, in particular because a high parallelism expressivity is usually detrimental to separation of concerns. This evaluation therefore focuses on these two points, parallelism and separation of concerns. Two solutions from the literature are used as comparison points. First, the Aeolus model, for its state-of-the-art performance and general similarity in terms of component-based approach. No other solution with the same level of parallelism expressivity offers better separation of concerns in the literature. Second, the Ansible software configuration management tool for its wide-spread use in the industry and its unique approach to parallelism.

Because Concerto is a super-set of Madeus and both models offer the same levels of parallelism expressivity, in this chapter we focus on evaluating Concerto. The examples used throughout the chapter include deployment cases, so everything that is stated about Concerto also apply to Madeus in these cases. When Madeus has to be analyzed specifically, we mention it explicitly.

In Section 6.1, we introduce an implementation of Madeus and Concerto which have been used for the experiments which this chapter is based on. In Section 6.2, we present the synthetic and production use-cases that are used in the rest of the chapter to perform the evaluation. Then, in Section 6.3, we provide a performance model for our comparison points, Ansible and Aeolus, and evaluate the performance model of Madeus and Concerto. Then, we evaluate Madeus and Concerto in terms of parallelism expressivity (Section 6.4) and separation of concerns (Section 6.5), based on the previously introduced use-cases. Precise comparisons are drawn with Aeolus and Ansible. All the experiments presented here are reproducible and links to access the code used are provided.

## 6.1 Implementation

In this section, we present the Python implementation that we developed in order to evaluate Madeus and Concerto. In practice, this is an implementation of Concerto, on top of which was developed a Madeus abstraction layer. Consequently, in the following we focus on the Concerto implementation itself. First, we detail the architecture of the implementation, outlining some design choices. Then, we showcase how it can be used from the point of view of a developer. Note that the full source code is available

online[1] under the GNU GPLv3 license.

### 6.1.1 Architecture of the implementation and design choices

#### 6.1.1.1 Programming language

Concerto reconfigurations, from a system administrator's perspective, essentially consists in coordinating the execution of commands on multiple hosts. This means that Concerto itself is never doing heavy computation, for example. Therefore, there is no need for a highly optimized language with a compiler. On the other hand, it is very important for the language used to describe Concerto actions (what is executed by the transitions of a component) to accessible, if not familiar, to the actors involved (module developers, reconfiguration developers, system administrators). Also, because Concerto may be used in multiple environments, and in particular on multiple OSs, the portability of the code is important.

With these considerations in mind, Python was chosen as the language for our implementation. In addition to fulfilling the requirements listed before, Python is widely used in the field of system configuration and reconfiguration. This ensures that extensive tooling and libraries are available to be used by module developers. The "battery included" approach of Python also makes it one of the most accessible general-purpose programming languages.

#### 6.1.1.2 General architecture

Object-oriented programming was used extensively to expose the major Concerto concepts to the users (component types, assemblies, reconfiguration programs). Figure 6.1 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 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.

An instance of the class `Assembly` corresponds to an environment in which to execute reconfiguration programs. 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

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

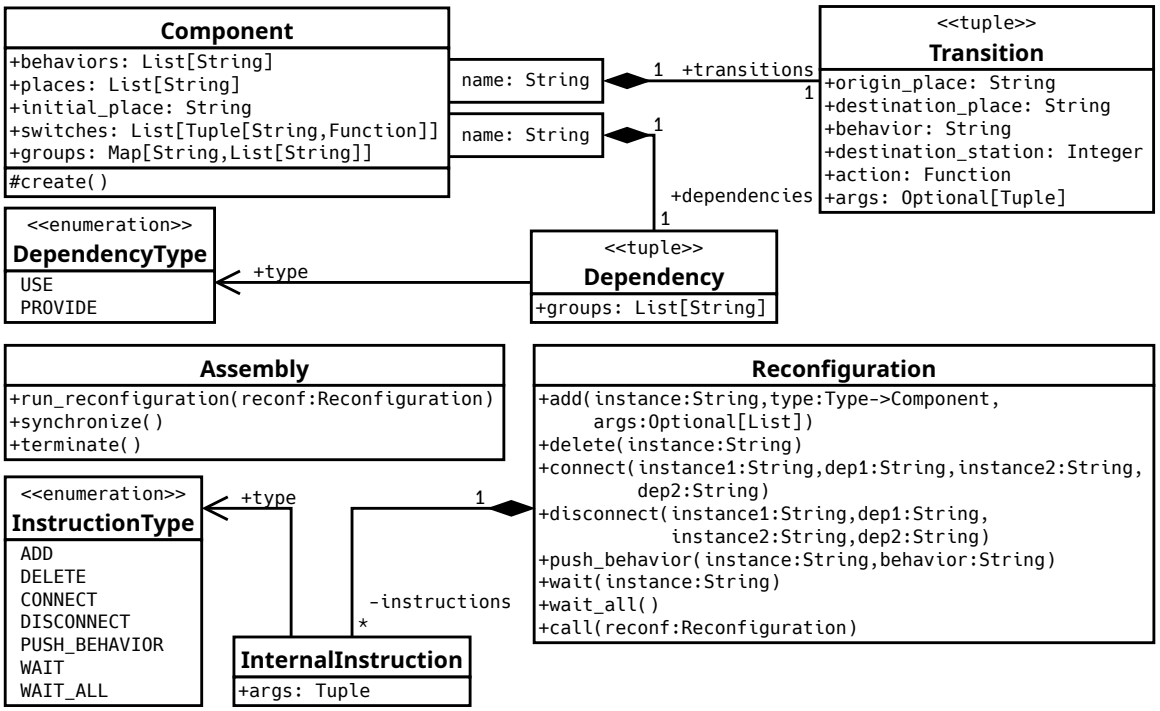



Figure 6.1: UML class diagram of our implementation of Concerto.

thread until the reconfiguration has been fully executed, then the method `terminate` can be used to destroy the thread.

A `Reconfiguration` object stores a list of reconfiguration instructions (instances of `InternalInstruction`). Instructions can be appended to the list with dedicated methods, one for each instruction type. Note two additional instructions compared to the formal model: `wait_all` prevents further instructions to be executed until all existing component instances have finished executing their behaviors (this is 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.

#### 6.1.1.3 Execution

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.

This loop effectively attempts to execute the semantic rules defined in the formal model of Concerto in a particular order. While other orders could be chosen, this one has the advantage of ensuring that the reconfiguration program and each component instance have a chance to move forward in their execution at regular intervals of time. Also, because the actions associated to transitions (i.e., Python functions) are executed in other threads, the semantic rules are applied independently of these user-defined actions. 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.

### 6.1.2 Describing component types

In this section, we see how a component developer can define a component type by looking at the implementation (given in Listing 6.1) of the component type `Db` from Figure 5.1 on page 81. Line 1, we see that a `Db` class is declared and extends class *Component*. Lines 2 to 56, the `create` method is defined as required by the implementation, defining the behaviors, the places, the initial place, the switches, the transitions, the groups and the dependencies (i.e., ports) of the component type. Lines 59 to 63 hint at a possible implementation of a reconfiguration action for transition `allocate`.

The declaration of the behaviors, places and initial place of the component type (lines 3 to 21) are straightforward.

Lines 26 to 28, the switch of the internal-net of the component type is declared. It is called `sw_choice` and references a function declared just before, `switch_choice_f` (lines 23 and 24). This function is a user-defined choice function for which branch of the switch to choose, which corresponds in the formal model to which transition ending to choose. The implementation allows connected components to communicate Python values through their ports. This is used line 24 to choose a transition ending depending on which value is provided by another component to the input port `backup_in`.

Lines 30 to 44, a list of transitions is given in the form of a dictionary associating a name to a tuple containing: the name of the source place, the name of the destination place, the name of the behavior associated to the transition, a station identifier for the destination place and the Python function to execute as transition action. Most of the time, the station identifier is 0, but when two transitions associated with the same behavior must be connected to distinct stations, each one must have a distinct

Listing 6.1: Declaration of the *Db* component type from Figure 5.1 in our implementation

```
class Db(Component):
  def create(self):
    self.behaviors = [
      'install',
      'backup',
      'change-config',
      'uninstall'
    ]

    self.places = [
      'uninstalled',
      'allocated',
      'configured',
      'sw1',
      'sw2',
      'running',
      'read-only',
      'cleaned'
    ]

    self.initial_place = 'uninstalled'

    def switch_choice_f(component_instance, current_behavior):
      return [0] if component_instance.read('backup_in') is None else [1]

    self.switches = [
      ('sw_choice', switch_choice_f)
    ]

    self.transitions = {
      'allocate': ('uninstalled', 'allocated'  , 'install'       , 0, self.allocate),
      'conf1'   : ('allocated'  , 'configured' , 'install'       , 0, self.conf1   ),
      'conf2'   : ('allocated'  , 'configured' , 'install'       , 0, self.conf2   ),
      'sw'      : ('configured' , 'sw_choice'  , 'install'       , 0, self.sw      ),
      'sw1t'    : ('sw_choice'  , 'sw1'        , 'install'       , 0, empty        ),
      'sw2t'    : ('sw_choice'  , 'sw2'        , 'install'       , 0, empty        ),
      'run'     : ('sw1'        , 'running'    , 'install'       , 0, self.run     ),
      'restore' : ('sw2'        , 'running'    , 'install'       , 1, self.restore ),
      'bak'     : ('running'    , 'read-only'  , 'backup'        , 0, self.bak     ),
      'rsa'     : ('read-only'  , 'cleaned'    , 'change-config', 0, self.rs      ),
      'res'     : ('cleaned'    , 'allocated'  , 'change-config', 0, self.res     ),
      'rsb'     : ('read-only'  , 'cleaned'    , 'uninstall'    , 0, self.rs      ),
      'del'     : ('cleaned'    , 'uninstalled', 'uninstall'    , 0, self.delete  )
    }

    self.groups = {
      'using_backup_in':    ['configured', 'sw1', 'sw2'],
      'providing_sql_read': ['running', 'read-only']
    }

    self.dependencies = {
      'backup_in' : (DepType.USE    , 'using_backup_in'   ),
      'sql_read'  : (DepType.PROVIDE, 'providing_sql_read'),
      'sql_write' : (DepType.PROVIDE, 'running'           ),
      'backup_out': (DepType.PROVIDE, 'read-only'         )
    }

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

Listing 6.2: Declaration of the reconfiguration program from Listing 5.1 in our implementation

```
1  rprog = Reconfiguration()
2  rprog.add("db", Db)
3  rprog.add("proxy", Proxy)
4  rprog.con("proxy", "sql_write", "db", "sql_write")
5  rprog.con("proxy", "sql_read", "db", "sql_read")
6  rprog.con("db", "backup_in", "other_comp", "other_port")
7  rprog.pushB("db", "install")
8  rprog.pushB("proxy", "install")
9  rprog.wait("proxy")
10
11 assembly = Assembly()
12 assembly.run_reconfiguration(rprog)
13 print("Reconfiguration in progress")
14 assembly.synchronize()
15 print("Reconfiguration completed")
```

number. This is the case for the `run` and `restore` transitions (lines 37 and 38), which each are in a different branch of the switch, and therefore must be connected to distinct stations of place `running`.

Lines 51 to 56, the dependencies of the component type (its ports in the formal model) are defined as a dictionary associating each name of dependency to a tuple containing the type of dependency (use or provide) and the name of the group it is associated to. The groups reference the dictionary of groups (here declared lines 46 to 49 which associates each name of group to its list of places), or names of places directly (in which case it corresponds to a group containing only this place).

### 6.1.3 Describing reconfiguration programs

In this section, we see how a reconfiguration developer can define a reconfiguration program by looking at the implementation (given in Listing 6.2) of the reconfiguration program defined in Listing 5.1 on page 88.

On line 1, an instance `rprog` of class `Reconfiguration` is created. It is an empty reconfiguration program which can be completed by calling its methods, which each correspond to one of the Concerto instruction types. Note that calling these methods only expands the reconfiguration program itself and does not execute anything. For example, on line 2, the instruction to add an instance with identifier `db` of component type `Db` is added to `rprog` (assuming that class `Db` such as defined in Listing 6.2 is in scope). Note that in our implementation, the `add` method supports passing additional arguments which will be passed to the constructor of the class corresponding to the component type when the *add* instruction is executed, which effectively allows for parametric component types. Lines 3 to 9 are straightforward and add the other reconfiguration instructions to `rprog`.

On line 11, an instance `assembly` of class `Assembly` is created. It acts as a medium

for reconfiguration to be executed. For example, on line 12, the `rprog` reconfiguration is executed. This execution occurs in another thread, so the `print` instruction on line 13 is executed immediately (i.e., the call to `run_reconfiguration` is not blocking). On line 14, the call to `synchronize` effectively waits actively for the execution of all the reconfiguration instructions queued in the assembly to be finished. In this case, the last instruction being a *wait* to ensure the completion of the reconfiguration, the `print` instruction on line 15 is executed after the completion of `rprog`.

### 6.1.4 Madeus abstraction layer

Concerto comes with many additional concepts compared to Madeus in order to support reconfiguration. In our Python implementation, when only the deployment part of the life-cycles need to be modeled, this leads to unnecessarily long component type descriptions and to the declaration of a reconfiguration which is essentially always the same in this case: create component instances, connect them and push their only behavior to their queue.

Because Concerto is a super-set of Madeus, we were able to provide an abstraction layer on top of Concerto which allows users to manipulate Madeus concepts directly. This layer consists in two classes exposed to the user, `MadeusComponent` and `MadeusAssembly`. For practical reasons, even if component types and component instances are the same in Madeus, we still consider these to be distinct in the implementation, similarly to Concerto.

Listing 6.3 demonstrates how the example of Figure 4.1 on page 60 can be implemented with this Madeus abstraction layer.

Lines 1 to 26 are the declaration of the `Db` component type for the *db* Madeus component of Figure 4.1. Notice that unlike in a Concerto component type, there are no behaviors, no groups and no switches declared. Also, notice that the declaration of the transitions (lines 11 to 14) only associates a starting place, a destination place and an action function to each transition name. Finally, the declaration of dependencies (i.e., ports) does not associate names to groups but names to places, as is expected in Madeus.

Lines 31 to 43 are the declaration of a type of assembly `MyAssembly`. It must implement the `create` function. This function defines its set of components (lines 33 to 36) with a dictionary associating component identifiers with one component instance each. It also defines its set of dependencies (i.e., connections between ports) between component instances (lines 38 to 43).

Finally, on line 45, an instance of this assembly is created, and run on line 46 (effectively starting the deployment process). The `run` function is non-blocking, so the `print` instruction on line 47 is executed immediately. Similarly to Concerto, the execution of the assembly can be synchronized (line 48), effectively blocking until the deployment has completed. The `print` instruction on line 49 therefore executed after the completion of the deployment.

Listing 6.3: Declaration of the components and assembly of Figure 4.1 in our implementation

```
class Db(MadeusComponent):
  def create(self):
    self.places = [
      'undeployed',
      'allocated',
      'running'
    ]

    self.initial_place = 'undeployed'

    self.transitions = {
      'allocate': ('undeployed', 'allocated', self.allocate),
      'run'     : ('allocated' , 'running'  , self.run     )
    }

    self.dependencies = {
      'ip'      : (DepType.PROVIDE, ['using_backup_in']),
      'service': (DepType.PROVIDE, ['service']        )
    }

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

class Server(MadeusComponent):
  ...

class MyAssembly(MadeusAssembly):
  def create(self):
    self.components = {
      'db'    : Db(),
      'server': Server()
    }

    self.dependencies = [
      ('server', 'database',
       'db'    , 'service'),
      ('server', 'database_ip',
       'db'    , 'ip')
    ]

assembly = MyAssembly()
assembly.run()
print("Deployment in progress")
assembly.synchronize()
print("Deployment completed")
```

## 6.2 Use-cases

To evaluate Concerto, we use two main scenarios, each composed of multiple reconfigurations. One of these scenarios is a real production use-case, while the other is a synthetic use-case. In this section, we introduce both of these use-cases and how they were implemented in a reproducible fashion.

### 6.2.1 Production use-case

Our production use use-case is used to ensure that Madeus and Concerto are functional, evaluate to which extent the performance model allows to predict the execution time of a reconfiguration program and separation of concerns.

This use-case is a restriction of a scenario which was presented in the context of a multi-region deployment of OpenStack at the 2018 Vancouver OpenStack summit[2]. 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.

#### 6.2.1.1 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.

Figure 6.2 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).

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

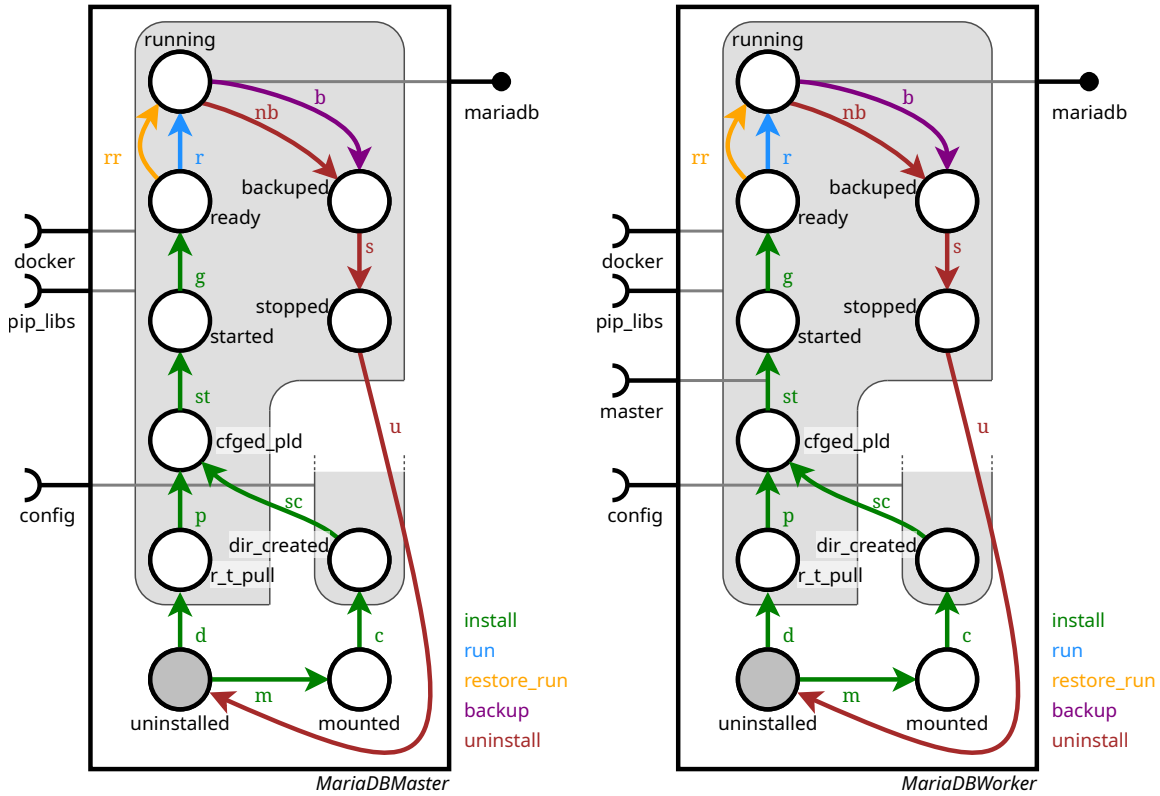



Figure 6.2: Two possible Concerto component type for MariaDB, one for the *initial host* (Master) and one for the *additional hosts* (Worker). The only difference is the presence of a `master` use port in the Worker implementation, allowing for proper coordination when joining a Galera cluster. In both component types, the group containing place `dir_created` also contain all the places represented above them (`cfged_pld`, `started`, `ready`, `running`, `backuped`, and `stopped`).

Its life-cycle is as follows. To deploy MariaDB, a dedicated directory must be mounted (transition `m` and place `mounted` of the green behavior `install`), in which a hierarchy of directories must be created (transition `c` and place `dir_created`). The configuration files, which state among other things in which mode MariaDB will be run, must be placed at the appropriate location (transition `sc`). In parallel to this, the MariaDB docker image may be downloaded once Docker is available on the host (transition `p` and place `r_t_pull`). Then, the Docker image may be started (transition `st` and place `started`). Once the database service is operational, a previously created backup present on the host may be restored, hence populating the database (transition `rr` of the yellow behavior `restore_run`). After possibly restoring this backup, the database can be used by other modules (`mariadb` provide port). When running, the MariaDB service may be stopped, after possibly saving a backup of the content of the database on the host (transition `b` of the purple behavior

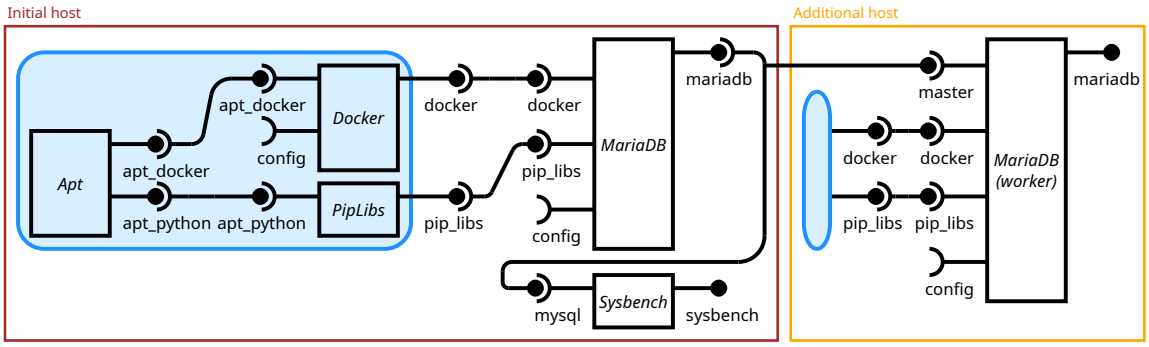



Figure 6.3: Overview of the Concerto assembly of a Galera distributed database with one initial and one additional host. The content of the blue rectangle represented for the initial host is hidden in the additional host for the sake of readability.

`backup`). The directories containing the configuration of MariaDB and its data can then be unmounted (transition `u` of the red behavior `uninstall`), effectively resetting the database. Note that many other operations could be performed on this module, but these are the ones which we use in this chapter. A database developer may build a much more complex and complete control component for MariaDB.

Figure 6.3 shows an overview of a Concerto assembly corresponding to the deployed state of two instances of a MariaDB database in a Galera cluster, one the *initial host* and one on an *additional host*.

Other modules are used to support directly or indirectly the database. Docker must be installed on each host running the database (component `Docker` of Figure 6.3), as well as appropriate Python libraries (component `PipLibs`). In turn, these require software packages to be installed through the package manager of the host OS (component `Apt`). The life-cycle of these modules is entirely sequential. Finally, the *Sysbench* benchmarking software (`Sysbench` component) 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.

#### 6.2.1.2 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 6.4 gives a possible implementation in Concerto to reach this first

Listing 6.4: 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 6.5: 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 6.6: 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)
```

configuration. This Concerto program will not be evaluated as being considered as an initial configuration in our scenario.

The first reconfiguration, namely *Decentralization*, start from the initial configuration obtained after Listing 6.4. 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 6.5 gives a Concerto program for this reconfiguration. Component

types `ClusterInitConfig` and `ClusterJoinConfig` are used to provide configuration information to put MariaDB respectively in *cluster-init mode* and *cluster-join mode*.

The second reconfiguration, namely *Scaling*, scales up the distributed database by increasing the number of additional hosts (from $n$ to $m$). Listing 6.6 gives one possible Concerto program for this reconfiguration.

#### 6.2.1.3 Implementation details

This scenario was coded in a reproducible fashion using Concerto and Ansible. The code is accessible on a public repository[3].

In Concerto, the reconfiguration programs presented in Listings 6.4 to 6.6 are used. The `MariaDBMaster` and `MariaDBWorker` component types correspond to what is shown in Figure 6.2.

In Ansible, each reconfiguration is coded as a *playbook* to execute. Some reconfiguration actions can be gathered under the same Ansible task, in particular all the deployment actions of the modules other than the database that are deployed on multiple hosts. When it comes to the database, when there are multiple database modules (migration and scaling reconfigurations), the deployment actions of the databases of the additional DB nodes can respectively be gathered under the same Ansible task. However, the non-deployment actions (backup, stopping the container, restoring backup) and the action to start the Docker container of the database of the initial DB node have to execute in a separate task.

### 6.2.2 Synthetic use-cases

The goal of our synthetic use-cases is to test the different possible situations in terms of parallelism that can be encountered during reconfiguration. This diversity allows us to ensure that the implementation behaves in conformity with what the performance model predicts and to analyze how performance is affected by scale or different types of parallelism.

Despite their synthetic nature, these use-cases are inspired by common problems: modules having dependencies to several other modules. While no system is actually reconfigured, the component types and reconfigurations have been implemented and the only difference between this and a real use-case is that the transition actions wait for a given amount of time to simulate something happening instead of actually executing commands on remote hosts. In the following, we therefore explain the use-cases using the same vocabulary as we would if the scenario were real.

[3] `https://gitlab.inria.fr/VeRDi-project/galera-experiment`

#### 6.2.2.1 Modules

We consider a *server* module, the goal of which is to provide a service, e.g., a web service, to users external to the assembly. This server relies on number $n$ of other modules called *dependencies*, which may be databases, other web services, libraries, etc. Each module runs on a distinct host machine.

Figure 6.4 shows a Concerto assembly implementing this scenario in the deployed state. 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. The assembly of Figure 6.4 is composed of three instances of three different component types: $Server_2$, $Dependency_1$ and $Dependency_2$.

For the sake of simplicity, we assume that all the dependencies have similar life-cycles. Their deployment is done in two steps. First install the dependency, second start the service. After installation (green behavior), we consider that any configuration information required by the server to use this dependency is available (through `config` and `service` ports). After starting the service, it can be used by the server through its `service` port. When it is running, each dependency may be updated, and the service can not be provided during this process. It is done in two phases: first perform the update itself, and then restart the service. The update process is distinct among the dependencies.

The server's life-cycle is as follows. Its deployment (green behavior of Figure 6.4) starts by allocating some resources to be used (transition `sa` of the green behavior `deploy`). Then, multiple configuration actions are performed in parallel, one for each dependency (transitions `sc1` and `sc2`). Each of these actions can only be performed once the configuration information of the corresponding dependency is available. Finally, once each configuration action has been performed, the server may start its own service. When it comes to reconfiguration, the server may be suspended (red behavior of Figure 6.4), which causes it to stop using the services provided by the dependencies. The actions performed to stop using these services are done in parallel.

#### 6.2.2.2 Reconfigurations

We consider four reconfiguration use cases, each featuring different kinds of parallelism. Concerto implementations of these reconfigurations are given in Listings 6.7 to 6.10. 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-module parallelism when all reconfiguration actions are identical on all components. The second, namely *UpdateNoServer*, updates these $n$ dependencies and features inter-module parallelism when all reconfiguration actions are not all identical on all components. The third, namely *DeployServer*, deploys the server (with its dependencies already deployed) and features intra-module parallelism. Finally,

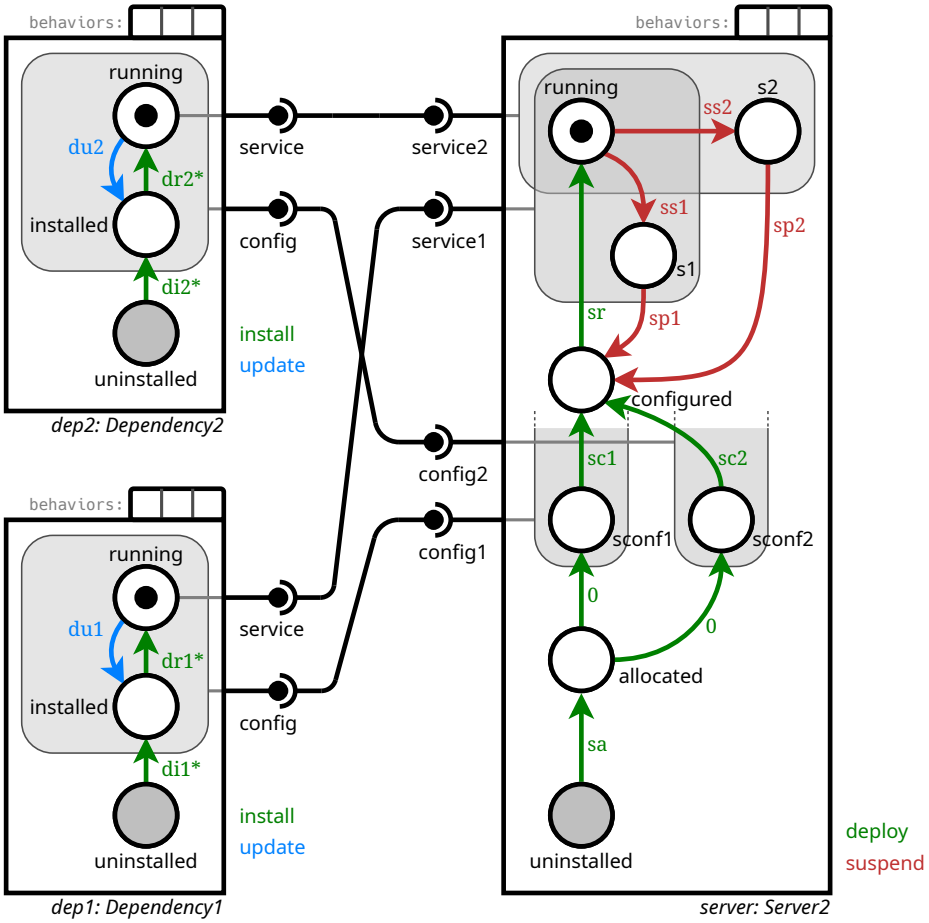



Figure 6.4: A Concerto assembly with two components $\text{Dependency}_i$ and one component $\text{Server}_2$. In the `server` component, the two groups containing either place `sconf1` or place `sconf2` also contain all the places represented above them (`configured`, `running`, `s1` and `s2`). In the dependency components, the transitions $di_i$ and $di_r$ (marked with *) are considered to be associated respectively to the same actions.

the fourth, namely *UpdateWithServer*, updates the $n$ dependencies, which requires the suspension of the server. It features both inter-module and intra-module parallelisms.

Listing 6.7: (1) DeployDeps

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

Listing 6.8: (2) UpdateNoServer

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

Listing 6.9: (3) DeployServer

```
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 6.10: (4) UpdateWithServer

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

#### 6.2.2.3 Implementation details

This scenario was coded in a reproducible fashion using Concerto and Ansible. The code is accessible on a public repository[4].

Because it is a synthetic use-case, the reconfiguration actions only consist in waiting for a given time, which is a parameter of the experiment (for each action). The component types of Figure 6.4 are implemented as is, with all transition actions executing Python's *time.sleep* function locally (on the machine executing Concerto) to wait for the time given as parameter. The reconfiguration programs presented in Listings 6.7 to 6.10 are used.

In Ansible, each reconfiguration is coded as a *playbook* to execute, i.e., a sequence of tasks. Each task is composed of a reconfiguration action and metadata, in particular on which set of hosts this action must be executed. These tasks are executed sequentially, but when a task's action is executed on multiple hosts, it is done in parallel, as explained by the case *Inter-host action-based* of Figure 3.2 of Chapter 3. The *install* and *start service* actions of the dependencies are respectively considered to be executable by the same command, allowing us to write it as a single task to be executed on each of the dependencies' hosts. The *update* action is however different on each dependency and they have to be encoded by distinct tasks. Each reconfiguration action is actually a bash *sleep* command to be executed on the remote host. The time to wait is determined by text files sent to the remote host prior to execution, which are generated from the parameters of the experiment.

One could notice that the Concerto implementation uses a local function to wait for a given amount of time in the actions, while the Ansible one uses a remote one (over SSH). This difference is due to design differences between the two systems and is not problematic here as they will not be compared directly for performance. Instead,

[4] https://gitlab.inria.fr/VeRDi-project/concerto-evaluation/-/tree/master/synthetic

they will be used separately to check performance models for each solution.

## 6.3 Performance models

In this section, we present performance models for our main comparison points, Ansible and Aeolus. These, in addition to the performance model of Concerto (and Madeus), are meant to allow us to analyze how much parallelism is achieved by each solution on a given reconfiguration scenario, to predict the execution time of each of them as a function of the durations of the reconfiguration actions, and to compare these results.

First, we present the performance model for Ansible and validate it. Second, we explain how a performance model for Aeolus can be obtained by a simple transformation from the one of Concerto. Then, we explain why the analysis made using Concerto's performance model also apply to Madeus deployments. Finally, we validate the performance model of Concerto using the synthetic use-cases introduced in the previous section.

### 6.3.1 A performance model for Ansible

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 also. 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 measurements had a difference of more than 1.6s. One can note from these experiments that a small overhead is observed when executing Ansible.

This shows that our execution and performance modeling for Ansible matches what happens in reality, and is even a bit optimistic, not taking into account the slight overhead. The code to reproduce these experiments is available online[5].

### 6.3.2 A 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 inside a component. For this reason, we emulate the execution of Aeolus by replacing the Concerto component types by 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.

### 6.3.3 Validation of Concerto's performance model

For Madeus and Concerto, we use their respective performance models presented in the previous chapters. Note that when it comes to deployment, Madeus and Concerto have the same level of parallelism expressivity: parallel transitions in the internal-nets of the components and port-based parallelism between components. For example, Figure 6.5 shows the Madeus assembly equivalent to deploying the Concerto assembly shown in Figure 6.4. Therefore, in the following, we only evaluate Concerto while everything which does not have to do with the dynamicity of the assembly also applies to Madeus.

In order to ensure that Concerto's performance model matches the experimental results, the execution time of our four synthetic reconfigurations presented in Section 6.2.2 were measured using our Python implementation of Concerto. Recall that the Concerto reconfiguration programs that we analyze are given in Listings 6.7 to 6.10.

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.

We ran this experiment 250 times for each reconfiguration, for 1, 5 and 10 dependencies, for a total of 3000 executions. Table 6.1 summarizes the results obtained. The full results as well as the code to reproduce these experiments are available online [6].

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

| Number of dependencies | Median | Average | Max relative difference (%) | Max absolute difference (s) |
|---|---|---|---|---|
| **DeployDeps** | | | | |
| 1 | 10.09s | 10.15s | 5.7% (of 0.37s) | 0.04s (0.2%) |
| 5 | 15.01s | 14.84s | 0.4% (of 7.76s) | 0.04s (0.2%) |
| 10 | 16.16s | 16.01s | 0.3% (of 10.94s) | 0.05s (0.3%) |
| **UpdateNoServer** | | | | |
| 1 | 10.41s | 10.35s | 2.5% (of 0.67s) | 0.04s (0.2%) |
| 5 | 15.01s | 14.84s | 0.3% (of 7.27s) | 0.04s (0.2%) |
| 10 | 16.52s | 16.36s | 0.3% (of 10.46s) | 0.04s (0.2%) |
| **DeployServer** | | | | |
| 1 | 14.59s | 14.58s | 2.5% (of 0.90s) | 0.05s (0.2%) |
| 5 | 18.34s | 17.95s | 0.4% (of 8.49s) | 0.05s (0.2%) |
| 10 | 19.19s | 19.13s | 0.3% (of 9.64s) | 0.05s (0.2%) |
| **UpdateWithServer** | | | | |
| 1 | 17.64s | 17.49s | 0.5% (of 5.57s) | 0.05s (0.2%) |
| 5 | 21.96s | 22.04s | 0.3% (of 11.57s) | 0.05s (0.2%) |
| 10 | 24.21s | 23.89s | 0.2% (of 16.80s) | 0.05s (0.2%) |

Table 6.1: 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 one, the median and average over all the runs are given. Then, the maximum relative time difference between the time predicted by the performance model and the actual measured time in both percentage and seconds is given. Similarly, the maximum absolute time difference is given.

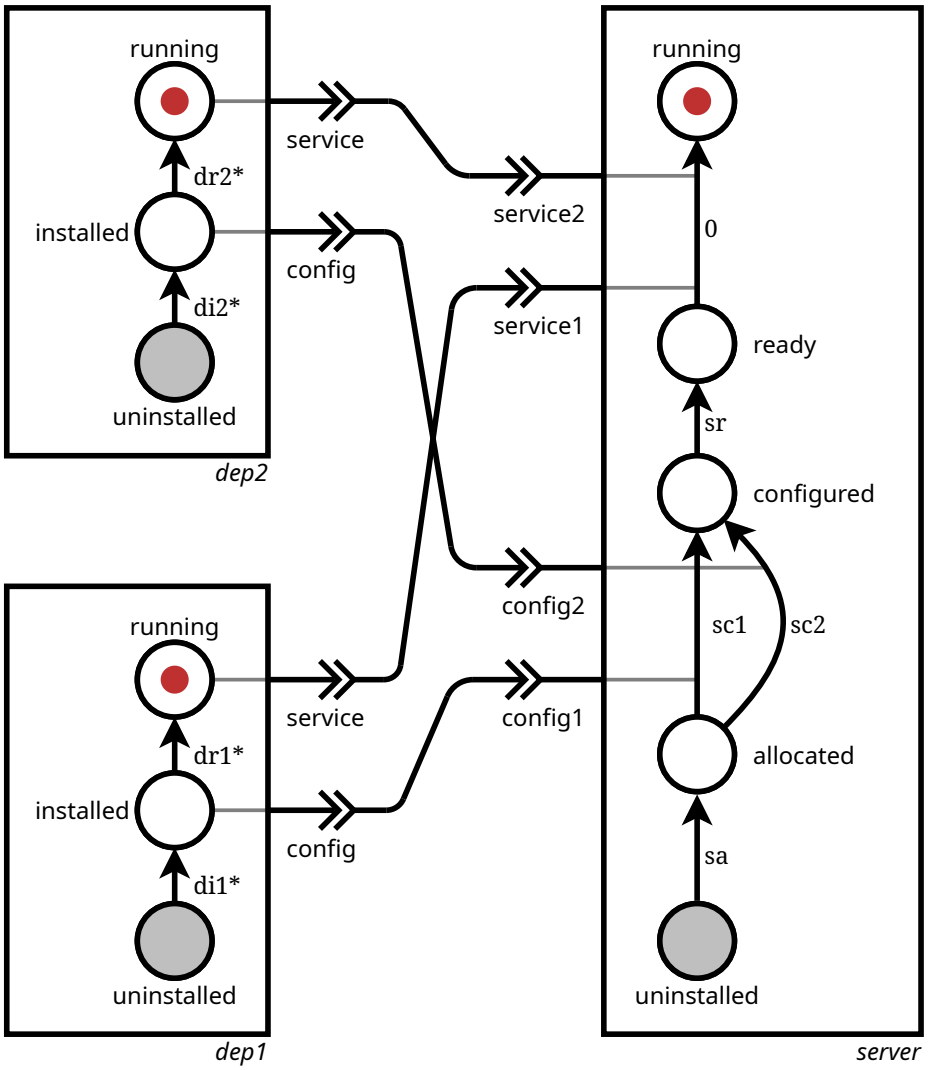



Figure 6.5: Madeus assembly corresponding to deploying the Concerto assembly shown in Figure 6.4.

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 median execution times were, depending on the use cases, included between 10.9 seconds and 24.21 seconds, while the average execution times were included between 10.15 seconds and 23.89 seconds, which is large compared to the maximum difference between the estimated time and the measured time. Note that the measured time was always slightly larger than the estimated time. This is explained by the small overhead introduced by the Concerto implementation. The performance model therefore matches what is

observed in reality.

## 6.4 Parallelism

In this section, we evaluate the parallelism introduced by Madeus and Concerto, which corresponds to their capacity to execute deployment and reconfiguration actions in parallel while respecting the dependencies between these actions. The expected consequence of increasing such parallelism is the reduction of the total execution time of a given deployment or reconfiguration.

We first use the production use-case presented in Section 6.2.1 to show that the predictions in terms of parallelism expressivity provided by the performance models match what happens in reality when using our implementation of Concerto.

Then, we use the performance models presented in Section 6.3 to analyze how much parallelism Concerto, Aeolus and Ansible express in the synthetic use-case presented in Section 6.2.2 and compare the resulting total execution times.

### 6.4.1 Accuracy of the performance model and execution times on a production use case

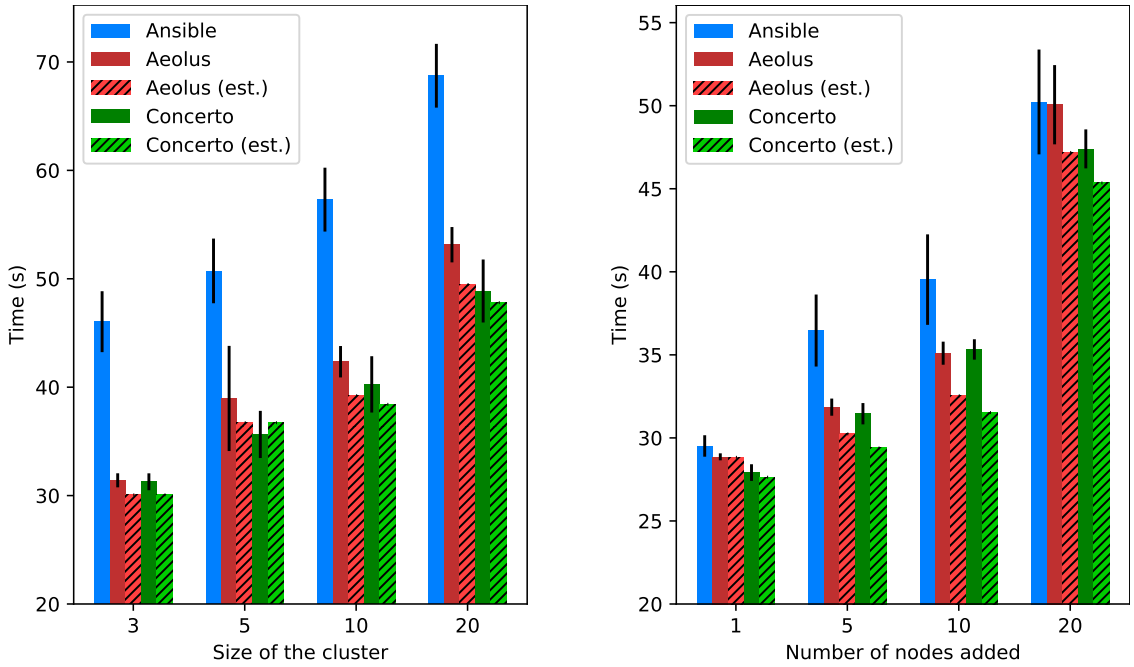
In this section, we make use of the production use case presented in Section 6.2 to evaluate: first, to which extent the performance models allow one to predict the total reconfiguration time for Madeus and Concerto; second, the gains of Concerto compared to Ansible and Aeolus.

We consider the two reconfigurations previously presented in Listings 6.5 and 6.6. First, the *decentralization* from a MariaDB instance to a Galera cluster of size 3, 5, 10 and 20. Second, the *scaling* of a cluster of size 3, with the number of nodes added equal to 1, 5, 10 and 20. This can for example be part of a multi-region deployment of OpenStack, as discussed at the 2018 Vancouver OpenStack summit[7]: originally all regions use the same centralized database; then OpenStack is reconfigured so that multiple regions have a local instance of the database synced in a Galera cluster; and nodes are added when increasing the number of regions having their own local instances.

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

The use case has been implemented in Concerto and Ansible. Moreover, Aeolus is emulated by transforming the Concerto components so that there is no *intra-module*

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




(a) Decentralization (b) Scaling

Figure 6.6: Measured running times for Ansible, Aeolus and Concerto and estimated times of Aeolus and Concerto for execution of the *decentralization* and *scaling* reconfigurations (error bars: standard deviation).

parallelism. The Concerto and Aeolus implementations use *SSH* calls to execute *Bash* scripts on the remote nodes, while the *Ansible* one uses the corresponding commands provided by Ansible (which also uses SSH as a back-end).

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 does not allow to measure this for each individual host). The transition durations measured for Aeolus were used as input of the performance models to estimate the running time of the reconfigurations for Aeolus and Concerto. Figure 6.6 shows these estimations as well as the measured running times.

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

Overall, the performance gain is, as expected very high compared to Ansible which is one of the most used production tool to handle reconfigurations. However, the performance gain appears to be low compared to Aeolus in this real use-case. This result was however predicted by the performance model and is due to a lack of exploitable intra-module parallelism. We believe that inserting this use-case within the complete OpenStack multi-region reconfiguration case would offer a much more convincing gain as introducing more components to reconfigure. We did not get enough time to perform this integration, but we have conducted experiments on the deployment of OpenStack with Madeus (under minor revision to the Journal of Systems and Software (JSS)[8]) that have shown a performance gain up to 30% compared to Aeolus. That work used the implementation of Madeus presented in Section 6.1.4, however we do not detail this experiment in this document as the author did not take part in this work other than providing the Madeus implementation.

### 6.4.2 Analysis of parallelism expressivity

In order to evaluate the potential gain of Concerto, we use its performance model, along with those of Aeolus and Ansible to compare their parallelism in the synthetic use cases presented in Section 6.2. We do this by using the performance models presented before to express the total execution time of each of the four reconfigurations as a function of the duration of their individual reconfiguration actions. The resulting formulas are listed in Table 6.2 and allow us to understand the consequences of each model's parallelism expressivity. We do this by discussing each reconfiguration one by one, discussing what happens in Ansible, Aeolus and Concerto depending on the number of dependencies (scalability) and on the relative duration of the different reconfiguration actions.

**(1) DeployDeps:** We consider the deployment of $n$ instances of $n$ component types $Dependency_1$, ..., $Dependency_n$. The reconfiguration is given in Listing 6.7 and the formulas given by the performance models of Concerto, Aeolus and Ansible are given in Table 6.2 In this use-case the behavior `install` is the only one considered. As all $Dependency_i$ component types are supposed to perform the same actions within this behavior (denoted with a star on Figure 6.4), 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-module parallelism. The gain for Concerto and Aeolus compared to Ansible depends on the difference of duration of similar transitions across instances. One the one hand, Figure 6.7 illustrates what happens

[8] https://hal.inria.fr/hal-02737859

| Framework | Formula |
|---|---|
| **(1) *DeployDeps*** | |
| 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 6.2: Theoretical total execution time for each reconfiguration of the synthetic use-case in Concerto, Aeolus and Ansible as a function of the duration of each reconfiguration action.

when this difference is large, which may happen because of hosts having different hardware, differences in 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.

On the other hand, when running the same actions on similar hardware, we expect a normal distribution $\mathcal{N}(\mu, \sigma^2)$ (of mean $\mu$ and standard deviation $\sigma$) for the durations of the transitions $di_i$ and $dr_i$ respectively. As a result, Table 6.3 shows the distribution of the total execution time for different values of the number of dependencies $n$ and $\sigma$. 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-module 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

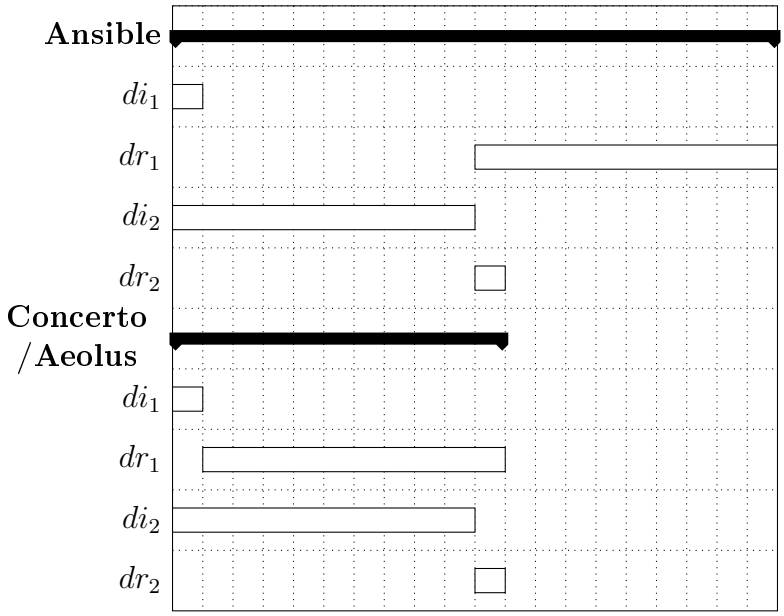



Figure 6.7: Gantt chart representing the difference in parallelism between Ansible on the one hand and Concerto and Aeolus in the other hand.

$n = 2000$, 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 considered standard deviation of transitions durations $\sigma$.

**(2) UpdateNoServer:** Given $n$ dependencies in place `running`, we consider the update of these dependencies. The reconfiguration is given in Listing 6.8 and the formulas are given in Table 6.2 (2). Recall here that unlike the previous case, the transitions $du_i$ 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 the better scalability resulting from inter-module 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 6.9 and the formulas are given in Table 6.2 (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 a suspension of the server.

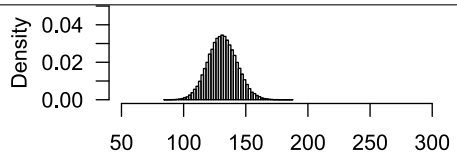
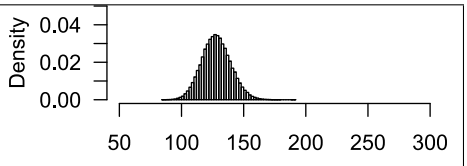
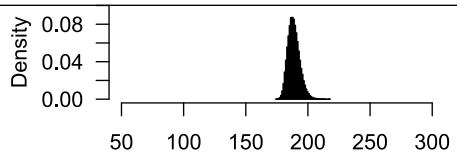
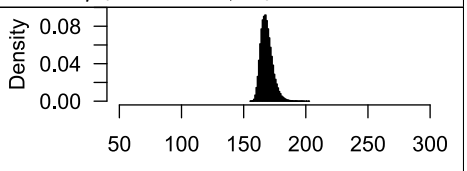
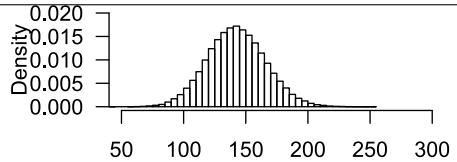
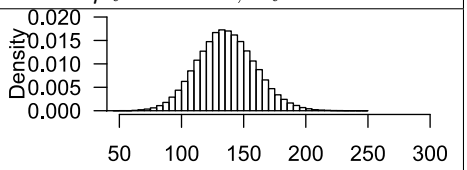
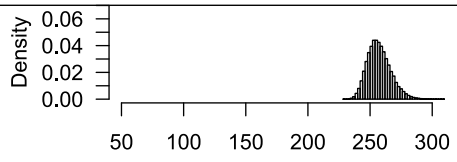
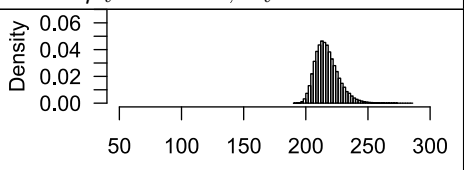
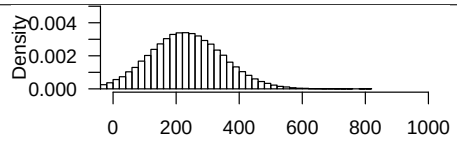
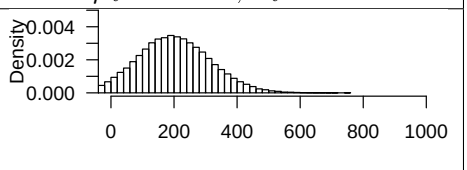
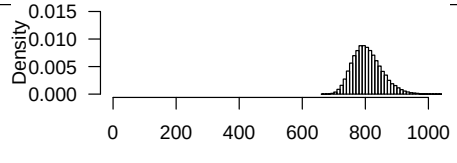
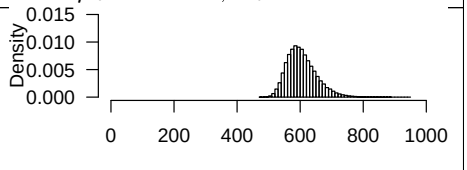
| $\sigma$ | n | Ansible | Concerto/Aeolus |
|---|---|---|---|
| 10 | 2 |   $\mu_t = 131.3$, $\sigma_t = 11.7$ |   $\mu_t = 128.0$, $\sigma_t = 11.6$ |
| | 2000 |   $\mu_t = 188.7$, $\sigma_t = 4.7$ |   $\mu_t = 168.6$, $\sigma_t = 4.7$ |
| 20 | 2 |   $\mu_t = 142.6$, $\sigma_t = 23.4$ |   $\mu_t = 135.9$,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

Table 6.3: 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 $\sigma$ (with $\mu = 60$). Each histogram was obtained by simulation over 100,000 samples. Histograms were omitted for $\sigma = 100$ for the sake of readability.

The reconfiguration is given in Listing 6.10 and the formulas are given in Table 6.2 (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-module 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 instance $dep_i$ is $du_i + dr_i$ plus the time required before the port `service` may be de-activated. There is no intra-module parallelism in Aeolus, so the $ss_i$ transitions execute sequentially. Therefore, the time for use port $service_i$ to be deactivated is $\sum_{j \leq i} \{d_{ss_j}\}$, which hence is the time it takes for the port `service` to be able to be deactivated. The second part of the outer *max* corresponds to the execution of the server in which all the transitions have been sequentialized, hence giving the sum of all the transition durations. In Concerto, the 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 $i^{\text{th}}$ dependency module, 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 remaining at 15 seconds for Concerto.

Overall, we saw that inter-module parallelism (which Aeolus, Madeus and Concerto have) improves performance as the number of components having to perform actions at the same time increases, which improves the scalability in terms of performance (*DeployDeps*, *UpdateNoServer*, *UpdateWithServer*). Even when Ansible can execute reconfiguration actions in parallel on multiple hosts, the difference in duration of reconfiguration actions on each host generates a loss of performance (*DeployDeps*). We also saw that intra-module parallelism (which only Madeus and Concerto have) improves performance as the number of actions that can be done in parallel in a component increases (*DeployServer* and *UpdateWithServer*). Finally, we saw that a combination of inter-module and intra-module parallelism as offered by Concerto (and Madeus) can have a very high impact on the overall reconfiguration execution time. A good example of this is the use-case *UpdateWithServer*, as the number of `Dependency_i` components increases.

## 6.5 Separation of concerns

In this section, we evaluate the separation of concerns of Madeus and Concerto. As presented in Chapter 3, we consider three types of actors which are commonly involved in distributed software during their existence: *module developers*, *reconfiguration developers* and *system administrators*. Recall that *module developers* are experts in a piece of software, i.e., in what it does, how it works, its life-cycle and its dependencies

(e.g., expert in databases); *reconfiguration developers* are experts at assembling components into a functional application (e.g., expert in web applications) and know how to properly reconfigure a given application to ensure continued functionality, integrity of data, minimal downtime, etc.; and *system administrators* are in charge of a system, usually the infrastructure (physical or virtual machines, network, etc.) as well as the software running on it. They are the ones who trigger manual reconfigurations or set up autonomic reconfiguration rules.

To evaluate separation of concerns, we first list for each actor what they need to be provided in order to properly do their job. For each element, we check whether this piece of information is provided directly to them (by other actors or by the reconfiguration framework), or if they need to obtain it by looking at information (code, documentation, etc.) which is not addressed to them, possibly requiring skills which are not part of their area of expertise.

### 6.5.1 Module developers

Module developers' main purpose is to create and maintain a piece of software's source code. Additionally, they need to provide documentation explaining how to use their piece of software.

Because we focus on modular development of distributed systems, the specificity of module developers is that they do not need information from other actors as they are at the start of the chain. However, they need to specify the requirements of their modules, what they provide and how to operate them. For example, they at least need to provide a way to deploy, start and/or run them.

If the reconfiguration solution is capable of handling non-trivial life-cycles, module developers also have to provide ways to control this life-cycle. For instance, in Aeolus, Madeus and Concerto, they need to express their life-cycle as an object close to state machine, indicate the set of services used and provided by the piece of software during its life-cycle, and associate them to the corresponding part of their life-cycle state-machines.

In solutions that support more than deployment, such as Aeolus and Concerto, but also TOSCA-based approaches among others, they need to define additional parts of their life-cycles such as removal, update, suspended states, etc.

In Madeus and Concerto, they also have to precisely define the dependencies between deployment/reconfiguration actions so that intra-module parallelism can occur. While technically representing extra work, we argue that this is similar to writing precise documentation or good deployment scripts, which is good practice, and is in the area of expertise of the module developer.

Finally, Concerto requires the definition of behaviors for the component, essentially mapping high-level actions to be performed by the component (e.g., deployment, update) to a set of reconfiguration actions, i.e., transitions of the internal-net. Again, this is similar to writing documentation and is in the module developer's area of

expertise.

### 6.5.2 Reconfiguration developers

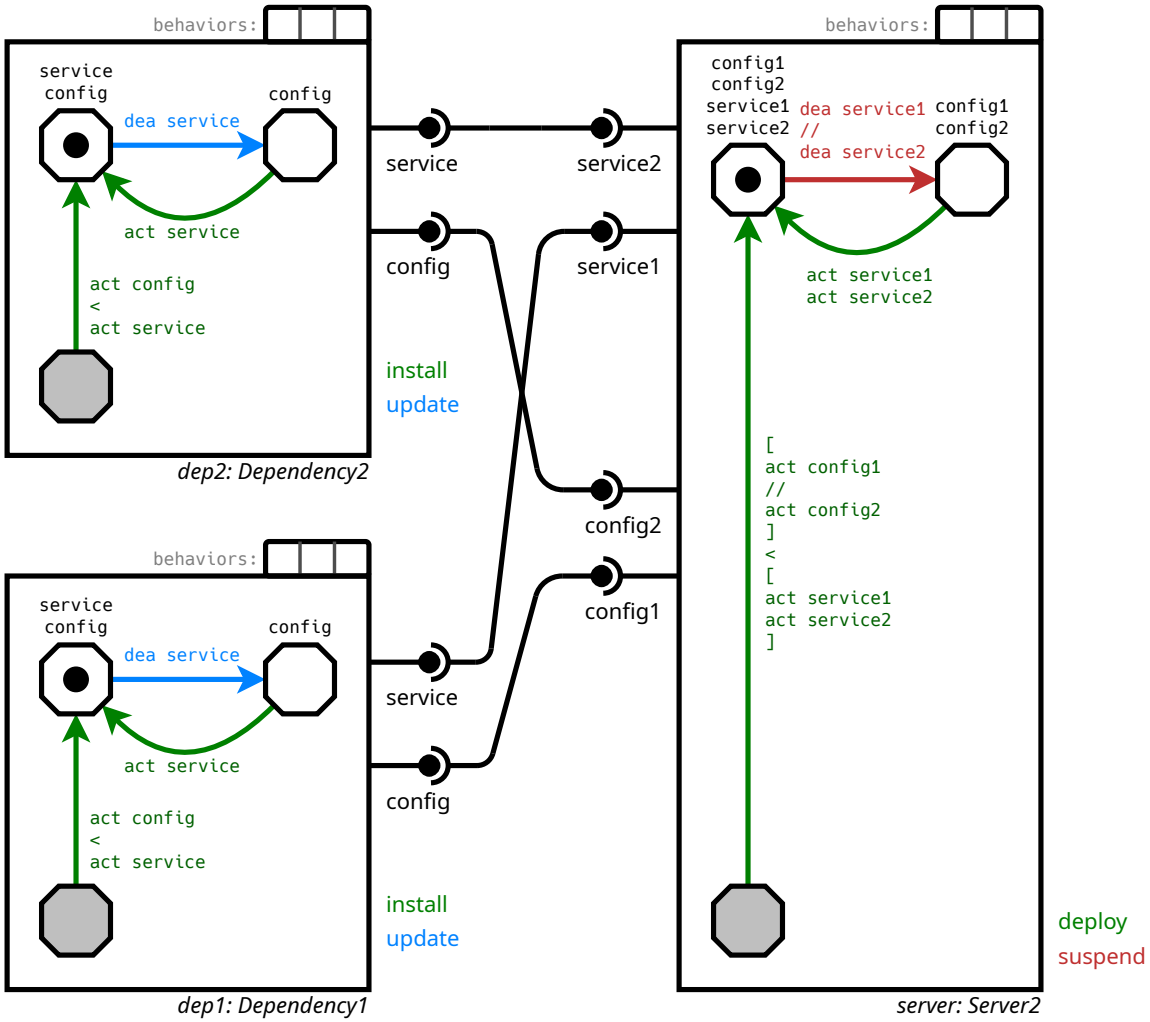



Figure 6.8: Concerto assembly with behavioral interfaces corresponding to the Concerto assembly shown in Figure 6.4 (on page 134).

The main purpose of reconfiguration developers is to list and configure individual pieces of software (already existing or yet to be deployed) so that they can work together as a functional application. They also define operations that can be applied to this application (e.g., changes in scale, updates, changes in functionality, etc.).

They need to know the requirements of each of the modules they want to manipulate and what these provide. They also need to have information about the life-cycles

of the modules, their different possible configurations and how to make them go from one configuration to the other.

In Ansible, unless extensive documentation is provided, the requirements of the roles (which are the most appropriate unit in Ansible to define a module), what they provide and how to change their configuration is not explicitly provided to the reconfiguration developer. The order in which roles have to be executed in a playbook is therefore not straightforward to determine. To do so, they need to check the code written by the module developers. For instance, in the production use-case presented in Section 6.2, if the deployment playbook was modified to deploy the database (which uses a Docker container) before Docker, the error would only be detected at run-time, and the error message would not clearly state that a dependency was not respected. Instead, the application reconfiguration would have to explore the database role, realize that the task responsible for the error tries to boot a Docker container and deduce that Docker must be deployed before this happens.

Concerto's behavioral interfaces hide the complexity of the life-cycle defined by the module developer, while providing for each possible configuration of the module (stable states) information about how to go to the other configurations of the module (behavior executions), as well as information necessary to coordinate its life-cycle with other life-cycles (partial order in which use and provide ports are activated and deactivated). For example, Figure 6.8 shows the assembly of Figure 6.4 with behavioral interfaces, which corresponds to the information that a reconfiguration developer needs.

For deployment, the equivalent of reconfiguration developers are deployment process developers. In Madeus, while behavioral interfaces could also be applied, we argue that because there is only one action to be performed on each module (i.e., deployment), the deployment process developer already knows the two possible configurations and how to go from one to the other. When it comes to the orchestration with other life-cycles, we argue that, because the transitions in Madeus only go in one direction, the partial order can be easily deduced from the *internal-net*, which is much more simple than in Concerto.

### 6.5.3 System administrators

The main purpose of system administrators is to host distributed software and use deployment and reconfiguration programs or mechanisms developed by deployment/reconfiguration developers to deploy them, manage them and apply changes to them.

System administrators should not have to deeply understand these programs or mechanisms or understand how they work, but rather should only have to focus on the result they want to achieve on their system.

Some reconfiguration solutions define their reconfigurations by the expected state of the system (in this case the reconfiguration operations are deduced by comput-

ing the difference with its current state). In this case the separation of concerns is guaranteed by design.

In other cases, including Madeus and Concerto, a reconfiguration is defined by a set of operations to perform. When no abstraction is provided over the current structure of the application, such as in Ansible, it is impossible to automatically get a representation of the resulting system if a reconfiguration were to be performed. However, graph-based and component-based solutions (e.g., TOSCA-based approaches, Aeolus, Madeus, Concerto) represent the state of the system using a graph or an assembly, and their reconfiguration language acts on this graph or assembly. It is therefore possible to know what the graph or assembly will be like after executing a dry-run of a reconfiguration.

## 6.6 Conclusion

In this chapter we have evaluated Madeus and Concerto in terms of parallelism expressivity and separation of concerns. To this end, we presented both a production scenario and a synthetic one, the latter allowing us to test different scenarios of parallelism in isolated and combined fashion. We then have shown that these performance models conform with real-life scenarios using a production use-case. We have also shown that the parallelism expressivity of Madeus and Concerto is superior to the state of the art, by first realizing that these of Madeus and Concerto are equivalent, and then comparing Concerto to Aeolus, which has the best level of parallelism expressivity in the state of the art, and Ansible which is widely used in the industry. This comparison was done thanks to performance models for each solution.

In terms of separation of concerns, we have shown that despite the increased parallelism expressivity, Madeus and Concerto both maintain a high level of separation of concerns between three types of actors: module developers, reconfiguration developers and system administrators. This is achieved thanks to the various kinds of interfaces that they provide (components, behavioral interfaces, assemblies).

# Chapter 7

# Conclusion and Perspectives

## 7.1 Conclusion

Distributed computer systems are now commonplace and, for some of them, have become critical. Deployment and reconfiguration of distributed systems are complex tasks because of the complexity of the software and infrastructures involved, especially with the advent of infrastructures such as fog and edge computing. For this reason, many solutions assist in the deployment and reconfiguration of distributed systems, and in particular in the execution step of the MAPE-K loop.

In this thesis, we have first presented existing solutions to assist in the execution of reconfiguration. The analysis of these solutions have led us to notice that they all fall short of providing at the same time genericity, parallelism expressivity and separation of concerns. More precisely, generic reconfiguration frameworks make a trade-off between parallelism expressivity and separation of concerns using abstractions over the life-cycle of the modules of distributed systems.

Our first contribution, Madeus, addresses this issue in the specific case of deployment. It is a formal component model in which each module of a distributed system is represented as a component. The interface of these components, i.e., their requirements and what it provides, are clearly defined using ports. At the same time, the deployment life-cycle of each module can be defined with a high degree of parallelism. Parallelism between distinct modules is also possible thanks to fine-grained dependencies between them made possible by the fact that the ports are linked to the life-cycles. A formal semantics is provided, as well as a performance model which allows to define the total execution time of a deployment as a function of the atomic deployment actions that it executes. Overall, parallelism expressivity is obtained thanks to the high level of parallelism inside each component and among components, while separation of concerns is obtained thanks to the clear interface made of ports.

Our second contribution, Concerto, extends Madeus for general reconfiguration. It does so in two ways. First, by introducing the concept of *behaviors*, which correspond to high-level actions one might want to perform on each module. Second,

by providing a reconfiguration language which allows to add or delete components, change the connections between their ports and triggering the execution of behaviors inside components asynchronously. A formal semantics is also provided, as well as a performance model which allows to define the total execution time of a reconfiguration program as a function of the atomic reconfiguration actions that it executes. Finally, behavioral interfaces are views of Concerto components which contain all and only the information (contained in the model) required to perform reconfiguration with a given component. An algorithm is presented to generate the behavioral interface of any Concerto component. Overall, parallelism expressivity is obtained thanks to the high level of parallelism inside each component and among components and to the asynchronicity of the reconfiguration language, while separation of concerns is obtained thanks to the clear interface made of ports and behavioral interfaces, which abstract away details which are of no use to the reconfiguration developer.

We introduced implementations of Madeus and Concerto, which we used to evaluate our work. We then presented two use-cases. First, a production use-case consisting in deploying a centralized database, reconfiguring it to become a decentralized database, and then reconfiguring it again to change its number of nodes. Second, synthetic use-cases showcasing different types of parallelism. We used the production use-case to prove the feasibility of our solution and evaluate the precision of the performance model in a real scenario. We then used the synthetic use-cases to determine precisely the gain in parallelism expressivity. Finally, we have discussed the separation of concerns of Madeus and Concerto. Overall, we have shown that Madeus and Concerto have a higher level of parallelism expressivity than their counterparts while being generic and conserving a good level of separation of concerns between module developers, reconfiguration developers and system administrators.

## 7.2 Perspectives

This work can be extended in multiple directions. First, Madeus and Concerto are formally defined, which can be used with formal methods to provide additional guarantees, perform automatic inference of reconfiguration programs, etc. Second, Madeus and Concerto were conceived with parallelism in mind, which could be used to allows concurrent reconfigurations to occur or decentralize their execution. Third, behavioral interfaces are a big step for separation of concerns of Concerto. This separation of concerns could be extended further by adding more abstractions on top of both Madeus and Concerto, such as composite components. Finally, we do not address fault-tolerance in this work, and even though the users can implement fault-tolerance mechanisms using Concerto's switches for example, dedicated mechanisms would make fault-tolerance in Madeus and Concerto more practical.

**Formal guarantees** Both Madeus and Concerto are formally defined, and in particular their operational semantics are formally defined. This makes it possible to perform static analysis on the components and assemblies. This could be used to make sure that models are consistent, and to make sure that for a given assembly or a given reconfiguration program deadlocks are not introduced. In the case of Madeus, some work has already been done in this direction [93].

Given a reconfiguration program, a valuable guarantee to have is that it will terminate. Also, it could be interesting in some situations to ensure that two events (for example reaching a place in two different components) always happen in the same order to make sure that a reconfiguration designer did not introduce unwilling errors. It could also be interesting to ensure that a given property is an invariant during the execution of the reconfiguration, in particular when executing a reconfiguration program designed by someone else. For instance, one could check that some component will not be affected by the reconfiguration, or that some transitions will never be executed etc.

These checks could be performed at multiple stages: when designing components, when designing Madeus assemblies or Concerto reconfiguration programs and right before executing one (at run-time). This would increase the safety of these procedures that are not risk-free when they affect critical systems or services. Furthermore risks are greater by introducing separation of concerns, thus needing to ensure that third-party assemblies or programs comply with our expectations.

**Automatic correct-by-design inference of reconfiguration programs** In Concerto, reconfiguration programs are imperative in the sense that they are a sequence of operations to perform on an existing assembly. While this approach makes sense, this is often not how reconfiguration developers approach the design of a reconfiguration program. When an assembly needs to be changed by adding or removing components, one intuitively thinks of the desired result, not of the sequence of steps to get there. In other cases, one might want to perform a given operation on an existing component by executing a sequence of behaviors. However, executing these behaviors may only be possible by executing other behaviors in other components (for example when a provide port currently in use would become inactive).

Because Concerto is formally defined, it could be possible to generate reconfiguration programs which satisfy a set of requirements, such as producing an assembly B starting from an assembly A, or coming back to the original assembly after having executed a given behavior. A selection among the possible programs could then be done either automatically (e.g., those which execute less transitions) and/or manually. This would greatly simplify the process of designing complex reconfiguration programs, while ensuring that they are correct by construction.

**Concurrent execution of reconfiguration programs** In large systems which use autonomic computing to adapt to their environment, the need to perform a given

change is not always anticipated. In a traditional autonomic loop (modeled as a MAPE-K loop), only one reconfiguration can be executed at any given time, and if any even occurs after the analysis phase, it will only be taken into consideration after the current cycle has ended. However, alternative approaches exist (e.g., multiple parallel MAPE-K loops, event-based methods) in which multiple reconfigurations may occur at the same time.

Concerto currently does not support this approach. Its semantics only allows the execution of a single reconfiguration program, and once one is executing it cannot be canceled. There are multiple ways in which this could be overcome. First, Concerto could support the execution of multiple reconfigurations out of the box, which should not be difficult to do given the asynchronous nature of its reconfiguration language. However, it would be quite hard to provide safety guarantees in this case. A second, perhaps more practical approach, would be to provide the ability to automatically merge two reconfiguration programs, while handling potential conflicts automatically if possible. Because splitting a Concerto reconfiguration program in two gives two valid programs, when a new reconfiguration should occur, it would be possible to suspend the execution of the current program and try to merge the new reconfiguration program with the remaining instructions that have not yet been executed. Formal methods could then be used, as explained previously, to ensure that the newly obtained program terminated and complies with what is expected.

**Decentralization of the execution** In large distributed systems, it is well-known that single points of failure are to be avoided. Currently, the execution of Madeus and Concerto are centralized, which creates a single point of failure. This could be overcome by replicating the current state of the system on multiple machines, with one of them active (actually executing deployment or reconfiguration) and the other ones passive, ready to take over in case of failure of the first machine. However, this is not the only problem introduced by a centralized execution. First, in large system if all actions must be triggered from one node this could lead to network congestion, in particular when managing thousands or millions of nodes. Second, in emerging infrastructures such as fog or edge computing, the inability for two nodes to communicate is not necessarily considered as an exceptional error but is rather to be expected.

To overcome these problems, Madeus and Concerto would need to drop the central and exact representation of the whole system, and instead rely on a partial representation. This would allow to have multiple Madeus or Concerto execution nodes, each with an exact representation of a part of the system and a partial representation of the rest of the system. By ensuring that one execution node exists in each area of the network that might become isolated, local reconfigurations could always be performed. When executing reconfigurations which span over multiple areas, the execution nodes would have to collaborate.

We believe that this approach is feasible because of the port-based interactions

between components. If each execution node is responsible for a part of the assembly, the only information required to know if another execution node is affected would be the list of connections between local ports and ports from other parts of the assembly (existing or to be created by a reconfiguration program). When this is the case, the two execution nodes could collaborate and would not need to inform other nodes.

**Additional abstractions** In Madeus and Concerto, there exist two levels of reasoning: component and assembly. Component developers reason about the life-cycles of the components they develop, while reconfiguration developers design assemblies or reconfiguration programs to generate or modify assemblies. However, one could argue that intermediate levels should exist. For example, if we consider a Map-Reduce system, it can be part of a larger assembly while still being composed of multiple components. The Map-Reduce expert is neither a component developer nor a reconfiguration developer for the distributed system that uses a Map-Reduce architecture among other elements.

A way to address this problem would be to provide the ability to define composite components in Madeus and Concerto. Composite components are a well-known feature of some components models and designate components which are made of other components (we can see them as sub-assemblies). In our case, the Map-Reduce expert could assemble components into a map-reduce composite component, which could then be used directly by reconfiguration developers.

Composite components could also be parametric. While Concerto's reconfiguration language can already be parametric by adding simple conditional and loop constructs, having the ability to directly use parametric components would increase separation of concerns.

Life-cycle patterns specific to common use-cases would also increase separation of concerns and ease the use of Concerto. For example, life-cycle patterns could be provided for virtual machine provisioning or containers commissioning.

**Fault-tolerance** Finally, Madeus and Concerto do not provide dedicated mechanisms for fault-tolerance. If the action associated with a transition fails and the component developer did not plan for this to happen, the execution will either be stuck (if the action does not terminate) or attempt to continue as if nothing happened (if the action does terminate), most likely producing an inconsistent state for the system being deployed or reconfigured.

Concerto has the concept of switch, which allows to choose one path or another for a behavior depending on what happened during the execution of the action (in particular the occurrence of errors). However, ensuring that the application can recover or goes into an error state, as well as reporting that an error occurred is the responsibility of the component developer.

This could be overcome by having a dedicated part of the life-cycle of each component recognized as an error state, and take this into account at the assembly level

(either by automatically performing recovery actions, such as replacing the component with another one, or by reporting the error). Other (possibly complementary) approaches include requiring components to have inverse transitions for each transition, which would allow to roll back the reconfiguration program in case of error.

# Bibliography


[1] P. M. Mell, T. Grance, Sp 800-145. the nist definition of cloud computing, Tech. rep., Gaithersburg, MD, USA (2011).

[2] B. P. Rimal, E. Choi, I. Lumb, A taxonomy and survey of cloud computing systems, in: 2009 Fifth International Joint Conference on INC, IMS and IDC, 2009, pp. 44–51.

[3] AWS CloudFormation, https://aws.amazon.com/cloudformation/.

[4] OpenStack Heat documentation, https://docs.openstack.org/heat/latest/.

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

[6] 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.

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

[8] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, X. Yang, A survey on the edge computing for the internet of things, IEEE Access 6 (2018) 6900–6919.

[9] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, S. Sinha, Real-time video analytics: The killer app for edge computing, Computer 50 (10) (2017) 58–67.

[10] J. R. Thomson, E. Greer, J. E. Cooke, Proxy servers and databases for managing web-based information, University of Alberta (1997).

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

[12] K. Lau, Z. Wang, Software component models, IEEE Transactions on Software Engineering 33 (10) (2007) 709–724.

[13] S. C. Kleene, Representation of events in nerve nets and finite automata, Tech. rep., RAND PROJECT AIR FORCE SANTA MONICA CA (1951).

[14] M. O. Rabin, D. Scott, Finite automata and their decision problems, IBM Journal of Research and Development 3 (2) (1959) 114–125.

[15] C. A. Petri, Kommunikation mit automaten (1962).

[16] T. Murata, Petri nets: Properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580.

[17] D. Varró, A formal semantics of uml statecharts by model transition systems, in: A. Corradini, H. Ehrig, H. J. Kreowski, G. Rozenberg (Eds.), Graph Transformation, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 378–392.

[18] M. von der Beeck, A structured operational semantics for uml-statecharts, Software and Systems Modeling 1 (2) (2002) 130–141.

[19] T. C. Chieu, A. Mohindra, A. A. Karve, A. Segal, Dynamic scaling of web applications in a virtualized cloud computing environment, in: 2009 IEEE International Conference on e-Business Engineering, 2009, pp. 281–286.

[20] K. A. Nuaimi, N. Mohamed, M. A. Nuaimi, J. Al-Jaroodi, A survey of load balancing in cloud computing: Challenges and algorithms, in: 2012 Second Symposium on Network Cloud Computing and Applications, 2012, pp. 137–142.

[21] S. Sotiriadis, N. Bessis, C. Amza, R. Buyya, Elastic load balancing for dynamic virtual machine reconfiguration based on vertical and horizontal scaling, IEEE Transactions on Services Computing 12 (2) (2019) 319–334.

[22] M. A. Wermelinger, Specification of software architecture reconfiguration (1999).

[23] G. Taentzer, M. Goedicke, T. Meyer, Dynamic change management by distributed graph transformation: Towards configurable distributed systems, in: H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Theory and Application of Graph Transformations, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 179–193.

[24] H. Gomaa, M. Hussein, Dynamic software reconfiguration in software product families, in: F. J. van der Linden (Ed.), Software Product-Family Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 435–444.

[25] F. Bannò, D. Marletta, G. Pappalardo, E. Tramontana, Handling consistent dynamic updates on distributed systems, in: The IEEE symposium on Computers and Communications, 2010, pp. 471–476.

[26] F. Bannò, D. Marletta, G. Pappalardo, E. Tramontana, Tackling consistency issues for runtime updating distributed systems, in: 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010, pp. 1–8.

[27] V. Panzica La Manna, Local dynamic update for component-based distributed systems, in: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 167–176. doi:10.1145/2304736.2304764.

[28] K. Ye, X. Jiang, D. Huang, J. Chen, B. Wang, Live migration of multiple virtual machines with resource reservation in cloud computing environments, in: 2011 IEEE 4th International Conference on Cloud Computing, 2011, pp. 267–274.

[29] J. Carrasco, F. Durán, E. Pimentel, Live migration of trans-cloud applications, Computer Standards & Interfaces 69 (2020) 103392. doi:https://doi.org/10.1016/j.csi.2019.103392.

[30] A. Meier, R. Dippold, J. Mercerat, A. Muriset, J. . Untersinger, R. Eckerlin, F. Ferrara, Hierarchical to relational database migration, IEEE Software 11 (3) (1994) 21–27.

[31] A. Maatuk, A. Ali, N. Rossiter, Relational database migration: A perspective, in: S. S. Bhowmick, J. Küng, R. Wagner (Eds.), Database and Expert Systems Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 676–683.

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

[33] R. Di Cosmo, A. Eiche, J. Mauro, S. Zacchiroli, G. Zavattaro, J. Zwolakowski, Automatic deployment of services in the cloud with aeolus blender, in: A. Barros, D. Grigori, N. C. Narendra, H. K. Dam (Eds.), Service-Oriented Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 397–411.

[34] E. Dolstra, E. Visser, The nix build farm: A declarative approach to continuous integration, in: 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1), Citeseer, 2008.

[35] E. Dolstra, A. Löh, Nixos: A purely functional linux distribution, SIGPLAN Not. 43 (9) (2008) 367–378. doi:10.1145/1411203.1411255.

[36] L. Bass, I. Weber, L. Zhu, DevOps: A software architect's perspective, Addison-Wesley Professional, 2015.

[37] I. Neamtiu, M. Hicks, G. Stoyle, M. Oriol, Practical dynamic software updating for c, in: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06, Association for Computing Machinery, New York, NY, USA, 2006, p. 72–83. doi:10.1145/1133981.1133991.

[38] M. Hicks, J. T. Moore, S. Nettles, Dynamic software updating, in: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI '01, Association for Computing Machinery, New York, NY, USA, 2001, p. 13–23. doi:10.1145/378795.378798.

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

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

[41] S. Crosby, R. Doyle, M. Gering, M. Gionfriddo, S. Grarup, S. Hand, M. Hapner, D. Hiltgen, et al., Open virtualization format specification, Standards and Technology, no. DSP0243 in DMTF Specifications, Distributed Management Task Force (2009).

[42] R. Boujbel, S. Rottenberg, S. Leriche, C. Taconet, J. Arcangeli, C. Lecocq, Muscadel: A deployment dsl based on a multiscale characterization framework, in: 2014 IEEE 38th International Computer Software and Applications Conference Workshops, 2014, pp. 708–715. doi:10.1109/COMPSACW.2014.120.

[43] P. Persson, O. Angelsmark, Calvin – merging cloud and iot, Procedia Computer Science 52 (2015) 210 – 217, the 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015). doi:https://doi.org/10.1016/j.procs.2015.05.059.

[44] A. Mehta, R. Baddour, F. Svensson, H. Gustafsson, E. Elmroth, Calvin constrained — a framework for iot applications in heterogeneous environments, in: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), 2017, pp. 1063–1073.

[45] F. Alvares, E. Rutten, L. Seinturier, High-level language support for reconfiguration control in component-based architectures, in: D. Weyns, R. Mirandola, I. Crnkovic (Eds.), Software Architecture, Springer International Publishing, Cham, 2015, pp. 3–19.

[46] F. Alvares, É. Rutten, L. Seinturier, A Domain-specific Language for The Control of Self-adaptive Component-based Architecture, Journal of Systems and Software (Jan. 2017).

[47] J. A. Hewson, P. Anderson, A. D. Gordon, A declarative approach to automated configuration, in: Proceedings of the 26th International Conference on Large Installation System Administration: Strategies, Tools, and Techniques, lisa'12, USENIX Association, USA, 2012, p. 51–66.

[48] A. Beugnard, S. Chabridon, D. Conan, C. Taconet, F. Dagnat, E. Kaboré, Towards context-aware components, in: Proceedings of the First International Workshop on Context-Aware Software Technology and Applications, CASTA '09, Association for Computing Machinery, New York, NY, USA, 2009, p. 1–4. doi:10.1145/1595768.1595770.

[49] C. Costa-Soria, J. Pérez, J. A. Carsí, Handling the dynamic reconfiguration of software architectures using aspects, in: 2009 13th European Conference on Software Maintenance and Reengineering, 2009, pp. 263–266.

[50] 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.

[51] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. Mclnnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, S. Zhou, A component architecture for high-performance scientific computing, The International Journal of High Performance Computing Applications 20 (2) (2006) 163–202. arXiv:https://doi.org/10.1177/1094342006064488, doi:10.1177/1094342006064488.

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

[53] 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.

[54] M. Beisiegel, H. Blohm, D. Booz, J.-J. Dubray, A. C. Interface21, M. Edwards, D. Ferguson, J. Mischkinsky, M. Nally, G. Pavlik, Service component architecture, Building systems using a Service Oriented Architecture. BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase, white paper, version 9 (2007).

[55] 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.

[56] 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.

[57] H. Ossher, P. Tarr, Using multidimensional separation of concerns to (re)shape evolving software, Commun. ACM 44 (10) (2001) 43–50. doi:10.1145/383845.383856.

[58] Ansible, https://www.ansible.com/.

[59] chef, https://www.chef.io/.

[60] 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).

[61] 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.

[62] Cloudify, https://cloudify.co/ (2012).

[63] H. Yoshida, K. Ogata, K. Futatsugi, Formalization and verification of declarative cloud orchestration, in: M. Butler, S. Conchon, F. Zaïdi (Eds.), Formal Methods and Software Engineering, Springer International Publishing, Cham, 2015, pp. 33–49.

[64] W. Chareonsuk, W. Vatanawood, Formal verification of cloud orchestration design with tosca and bpel, in: 2016 13th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2016, pp. 1–5.

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

[66] C. Butler, Automating orchestration in the cloud with ubuntu juju, USENIX Association, Philadelphia, PA, 2014.

[67] Juju, https://jujucharms.com/.

[68] Terraform by HashiCorp, https://www.terraform.io/.

[69] A. Rossini, Cloud application modelling and execution language (camel) and the paasage workflow, in: Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC, Vol. 567, 2015, pp. 437–439.

[70] PaaSage, https://paasage.ercim.eu/.

[71] A. Rossini, K. Kritikos, N. Nikolov, J. Domaschka, F. Griesinger, D. Seybold, D. Romero, M. Orzechowski, G. Kapitsaki, A. Achilleos, The cloud application modelling and execution language (camel), Universität Ulm (2017). doi:10.18725/OPARU-4339.

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

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

[74] Y. Al-Dhuraibi, F. Zalila, N. Djarallah, P. Merle, Model-driven elasticity management with occi, IEEE Transactions on Cloud Computing (2019) 1–1.

[75] Puppet, https://puppet.com/.

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

[77] M. Gabbrielli, S. Giallorenzo, C. Guidi, J. Mauro, F. Montesi, Self-Reconfiguring Microservices, in: E. Ábrahám, M. Bonsangue, E. B. Johnsen (Eds.), Theory and Practice of Formal Methods, no. 9660 in Lecture Notes in Computer Science, Springer, 2016, pp. 194–210. doi:10.1007/978-3-319-30734-3_14.

[78] F. Montesi, C. Guidi, G. Zavattaro, Composing services with jolie, in: Fifth European Conference on Web Services (ECOWS'07), 2007, pp. 13–22.

[79] F. Montesi, C. Guidi, G. Zavattaro, Service-Oriented Programming with Jolie, Springer New York, New York, NY, 2014, pp. 81–107. doi:10.1007/978-1-4614-7518-7_4.

[80] 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.

[81] P. Merle, J.-B. Stefani, A formal specification of the Fractal component model in Alloy, Research Report RR-6721, INRIA (2008).

[82] P.-C. David, T. Ledoux, Safe Dynamic Reconfigurations of Fractal Architectures with FScript, in: 5th Fractal Workshop at ECOOP 2006, Nantes, France, 2006.

[83] 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.

[84] 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.

[85] S. Lacour, C. Perez, T. Priol, Generic application description model: toward automatic deployment of applications on computational grids, in: The 6th IEEE/ACM International Workshop on Grid Computing, 2005., 2005, pp. 4 pp.–.

[86] Adage, http://adage.gforge.inria.fr/.

[87] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, P. Toft, Smartfrog: Configuration and automatic ignition of distributed applications, HP OVUA (2003).

[88] 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.

[89] SmartFrog, http://www.smartfrog.org/.

[90] P. Anderson, H. Herry, A formal semantics for the smartfrog configuration language, Journal of Network and Systems Management 24 (2) (2016) 309–345. doi:10.1007/s10922-015-9351-y.

[91] 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.

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

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

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Concilier expressivité du parallélisme et séparation des préoccupations lors de la reconfiguration de systèmes distribués

**Mots clés :** déploiement ; reconfiguration ; modèles à composants ; coordination ; parallélisme ; systèmes distribués

**Résumé :** Les systèmes informatiques distribués, qui fonctionnent sur plusieurs ordinateurs, sont désormais courants et même utilisés dans des services critiques. Cependant, ces systèmes deviennent de plus en plus complexes, en termes d'échelle, de dynamicité et de qualité de service attendue.

La reconfiguration de systèmes distribués consiste à modifier leur état durant leur exécution. Les systèmes distribués peuvent être reconfigurés pour plusieurs raison, parmi lesquelles leur déploiement, leur mise à jour, leur adaptation pour obéir à de nouvelles contraintes (en termes de capacité utilisateurs, d'efficacité énergétique, de fiabilité, de coûts, etc.) ou même le changement de leurs fonctionnalités.

Les systèmes de reconfiguration existants ne parviennent pas à fournir en même temps une bonne expressivité du parallélisme dans les actions de reconfiguration et la séparation des préoccupations entre les différents acteurs qui interagissent avec le système.

L'objectif de cette thèse est de prouver que ces propriétés peuvent être conciliées en modélisant précisément le cycle de vie de chaque module des systèmes distribués, tout en fournissant des interfaces appropriées entre différents niveaux de conception. Deux modèles formels implantant cette idée sont fournis, un pour le cas particulier du déploiement et un pour la reconfiguration. Une évaluation est réalisée à la fois sur des cas d'usage synthétiques et réels et montre que ces modèles ont un plus haut niveau d'expressivité du parallélisme que leurs homologues tout en conservant un bon niveau de séparation des préoccupations.

Title : Reconciling Parallelism Expressivity and Separation of Concerns in Reconfiguration of Distributed Systems

**Keywords :** deployment; reconfiguration; component models; coordination; parallelism; distributed systems

**Abstract :** Distributed computer systems, which run on multiple computers, are now commonplace and used even in critical systems. However, these systems are becoming more and more complex, in terms of scale, dynamicity and expected quality of service.

Reconfiguration of distributed systems consists in changing their state at runtime. Distributed systems may be reconfigured for many reasons, including deploying them, updating them, adapting them to fulfill new requirements (in terms of user capacity, energy efficiency, reliability, costs, etc.) or even changing their capabilities.

Existing reconfiguration frameworks fall short of providing at the same time parallelism expressivity and separation of concerns between the different actors interacting with the system.

The focus of this thesis is to prove that these properties can be reconciled by modelling precisely the life-cycle of each module of distributed systems, while providing appropriate interfaces between the different levels of conception. Two formal models implementing this idea are provided, one for the specific case of deployment and one for reconfiguration. Evaluation is performed on both synthetic and real use-cases and show that these models have a higher level of parallelism expressivity than their counterparts while conserving a good level of separation of concerns.