Automatic Application Deployment in the Cloud: from Practice to Theory and Back*

Roberto Di Cosmo[2], Michael Lienhardt[1], Jacopo Mauro[1], Stefano Zacchiroli[2], Gianluigi Zavattaro[1], and Jakub Zwolakowski[2]

1 University of Bologna/INRIA, Italy
2 Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, F-75205 Paris, France

Abstract

The problem of deploying a complex software application has been formally investigated in previous work by means of the abstract component model named Aeolus. As the problem turned out to be undecidable, simplified versions of the model were investigated in which decidability was restored by introducing limitations on the ways components are described.

In this paper, we take an opposite approach, and investigate the possibility to address a relaxed version of the deployment problem without limiting the expressiveness of the component model. We identify three problems to be solved in sequence: (i) the verification of the existence of a final configuration in which all the constraints imposed by the single components are satisfied, (ii) the generation of a concrete configuration satisfying such constraints, and (iii) the synthesis of a plan to reach such a configuration possibly going through intermediary configurations that violate the non-functional constraints.

1998 ACM Subject Classification D.2.9 Management, F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Automatic deployment, Planning, DevOps, Constraint Programming

1 Introduction

Modern software systems are based on a large number of interconnected software components (e.g., packages or services) that must be deployed on (possibly virtual) machines that can be created and connected on-the-fly by exploiting currently available cloud computing technologies. The configuration and management of such applications is a challenging task, and several tools and technologies are under development to support application architects and managers in this complex activity. The mainstream approach is to exploit pre-configured virtual machines images, which contain all the needed software packages and services, and that just need to be run on the target cloud system (e.g., Bento Boxes [10], Cloud Blueprints [2], or AWS CloudFormation [1], to name just a few options). The main drawback of this approach is that pre-configured images do not support customization and often force users to run their applications on specific cloud providers, inducing an undesirable vendor lock-in effect. More advanced techniques, on the contrary, would allow application architects to design their own software architectures by using high level description languages, like the graphical drag-and-drop approach of Juju [12] or the declarative deployment languages ConfSolve [11] and Engage [9].

* Partly funded by the EU project FP7-610582 ENVISAGE. Work partially performed at IRILL, center for Free Software Research and Innovation in Paris, France, http://www.irill.org. Unless noted otherwise, all URLs in the text have been retrieved on July 1st, 2015.

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In previous work [7, 6, 5] we have investigated this deployment problem from a foundational point of view, trying to capture its most relevant aspects, and identifying among them those that contribute to the difficulty of the problem. We have defined a formal model called *Aeolus*, in which the classical notion of component, seen as a black-box that exposes *provide* and *require*-ports, is extended with a finite state automaton describing the component life-cycle. The automaton states correspond to different configuration modalities, like *uninstalled*, *installed*, *running*, *stopped*, etc, and the transitions represent configuration actions like install, run, stop, etc. Depending on the internal state, the ports on the interface can be either active or inactive. For instance, an *uninstalled* component usually does not activate any require-port, while it can activate require-ports when it is in the *installed* state, and finally activate some provide-port when it actually enters the *running* state. Another specific feature of the *Aeolus* model is that capacity constraints can be associated to the ports: a provide-port could have a maximal number of connected require-ports, a require-port can ask for multiple providers offering a given functionality (used to model replication requirements) or even impose that no other component can provide a given functionality (used to model the notion of conflict among components).

The deployment problem is then formalized as the problem of verifying the possibility to configure at least one component of a given type in a given target state, by performing a sequence of actions like component creation/deletion, component binding/unbinding, and component internal state change.

In [7] we have proved that the deployment problem is, in general, undecidable. To overcome this negative result, we have considered in [6, 5] various simplifications of the Aeolus model for which deployment turns out to be decidable. In particular, we have proved that deployment is polynomial when capacity constraints and conflicts are not considered, while it is Ackermann-hard for the fragment without capacity constraints but with conflicts.

In this paper we take a different and more pragmatic approach by relaxing the deployment problem. Conceptually, we break the deployment problem in two independent subtasks to be executed one after the other. The first subtask abstracts away from component states, and considers the problem of computing a final correct configuration in which the target component is present and all the capacity and conflict constraints are satisfied. The second subtask abstracts away from the capacity constraints and the conflicts, and verifies the possibility to reach in this simplified scenario that desired target configuration.

We formalize these subtasks and discuss their complexity. In particular, for the first subtask, we consider two subproblems: the *Configuration* problem, consisting of checking whether it is possible to satisfy all the constraints directly or indirectly imposed by the target component on the final configuration, and the *Generation* problem addressing the concrete production of a configuration that actually satisfies these constraints. The *Configuration* problem is proved to be NP-complete, while the *Generation* problem is EXP-time. It is important to highlight that from a pragmatic point of view, the Configuration problem is much more challenging because the Generation problem has a more standard solution and has a higher theoretical complexity simply because there could be cases (that we consider rare in practice) in which the configuration to be generated is of exponential size. Finally, in a third phase called *Planning* we synthesize, if there exists, a sequence of deployment action to reach the desired final configuration. This problem is poly-time but, but as explained above the intermediary configurations traversed during the execution of the plan could not satisfy capacity constraints or conflicts because we have decided to loose correctness to favor tractability.

The split of the deployment problem in two subtasks is reflected also by two tools that

we have implemented. On the one hand, *Zephyrus* [4] addresses the problem of generating a final configuration that besides satisfying all the constraints that can be expressed in the Aeolus model, also considers the problem of the optimal distribution of the components to be deployed on available virtual machines. The second tool is called *Metis* [13] and it solves the deployment problem, but only for the simplified Aeolus model without capacity constraints and conflicts.

2 The Aeolus model

In this section we give a recap of the *Aeolus component model* following [14]. This formalization differs from other definitions of the *Aeolus model* reported in [7, 5] due to the absence of the so-called *multiple state change* actions. These actions are of interest when components are used to represent mutually dependent packages that must be contemporaneously installed, but are less relevant when components are used to model service deployment and configuration. We have opted for the formalization without multiple state changes as our focus in this paper is mainly on services and not on packages.

In the *Aeolus model*, components are represented as grey boxes offering services via provide-ports and requiring service through require-ports. They are grey boxes because the component configuration life-cycle is exposed in terms of a finite state automata indicating the different internal configuration states, and the corresponding configuration actions changing such states. When a component changes state, the sets of ports it requires or provides might also change; in other words, the component interface depends on its internal configuration state.

As an example, let us consider Figure 1 depicting the installation of a WordPress blog service according to the *Aeolus model*. Three services are modeled, viz. *wordpress*, *mysql*, and the HTTP daemon service *apache2*. Each service can be in the *uninstalled*, *installed*, or *running* state. Depending on the current state, the require and provide-ports could be active or inactive. For instance, the wordpress component in the installed state activates the require-port httpd, but does not activate the require-port mysql_up and the provide-port wordpress. The httpd require-ports activated by wordpress can be connected to the provide-port offered by apache2 when it is installed or running. Similarly, to be running, it also requires an active mysql service. This is represented by the requirement of the port mysql_up provided by mysql in its running state. The wordpress component in its running state is finally able to provide the wordpress functionality.

We now move to the formal definition of the *Aeolus component model*. Let us assume given the following disjoint sets: $\mathcal{I}$ for interfaces and $\mathcal{Z}$ for components. We use $\mathbb{N}$ to denote natural numbers and $\mathbb{N}^+_\infty$ for $\mathbb{N} \setminus \{0\} \cup \{\infty\}$.

▶ **Definition 1** (Component type). The set Γ of *component types* of the Aeolus model, ranged over by $\mathcal{T}_1, \mathcal{T}_2, \ldots$ contains 5-ple $\langle Q, q_0, T, P, D\rangle$ where:

- Q is a finite set of states;
- $q_0 \in Q$ is the initial state and $T \subseteq Q \times Q$ is the set of *transitions*;
- $P = \langle \mathsf{P}, \mathsf{R}\rangle$, with $\mathsf{P}, \mathsf{R} \subseteq \mathcal{I}$, is a pair composed of the set of *provide* and the set of *require*-ports, respectively;
- D is a function from Q to 2-ple in $(\mathsf{P} \nrightarrow \mathbb{N}^+_\infty) \times (\mathsf{R} \nrightarrow \mathbb{N})$.

Given a state $q \in Q$, $D(q)$ returns two partial functions $(\mathsf{P} \nrightarrow \mathbb{N}^+_\infty)$ and $(\mathsf{R} \nrightarrow \mathbb{N})$ that indicate respectively the provide and require-ports that q activates. The functions associate to the activate ports a numerical constraint indicating:

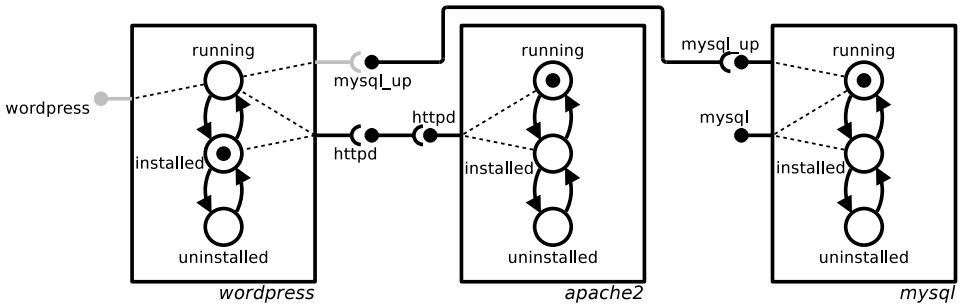


Figure 1 WordPress installation in Aeolus.

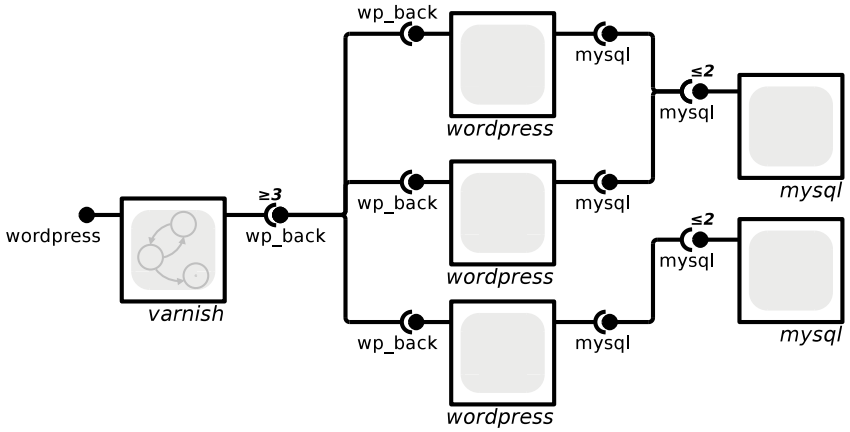


Figure 2 Redundancy and capacity constraints for a complex WordPress installation (internal sate machines are omitted for simplicity).

- for provide-ports, the *maximum* number of bindings the port can satisfy,
- for require-ports, the *minimum* number of required bindings to *distinct* components,
  - as a special case: if the number is 0 this indicates a conflict, meaning that there should be no other active port, in any other component, with the same name.

When the numerical constraint is not explicitly indicated, we assume as default value ∞ for provide-ports (i.e., they can satisfy an unlimited amount of requires) and 1 for require (i.e., one provide is enough to satisfy the requirement). We also assume that the initial state q_0 makes no demands (i.e., the second function of $D(q_0)$ has an empty domain).

As an example of the use of numerical constraints, Figure 2 shows the modeling of a WordPress hosting scenario where we want to offer high availability by putting the Varnish reverse proxy/load balancer in front of several WordPress instances, all connected to a cluster of MySQL databases.[1] For a configuration to be correct, the model requires that Varnish is connected to at least 3 (active and distinct) WordPress back-ends, and that each MySQL instance does not serve more than 2 clients.

We now define configurations that describe systems composed by component instances and bindings that interconnect them. A configuration, ranged over by $\mathcal{C}_1, \mathcal{C}_2, \ldots$, is given

[1] All WordPress instances run within distinct Apache instances, which have been omitted for simplicity.

by a set of component types, a set of deployed components with a type and an actual state, and a set of bindings. Formally:

▶ **Definition 2** (Configuration). A *configuration* $\mathcal{C}$ is a quadruple $\langle U, Z, S, B\rangle$ where:

- $U \subseteq \Gamma$ is the finite *universe* of all available component types;
- $Z \subseteq \mathcal{Z}$ is the set of the currently deployed *components*;
- S is the component *state description*, i.e., a function that associates to components in Z a pair $\langle \mathcal{T}, q\rangle$ where $\mathcal{T} \in U$ is a component type $\langle Q, q_0, T, P, D\rangle$, and $q \in Q$ is the current component state;
- $B \subseteq \mathcal{I} \times Z \times Z$ is the set of *bindings*, namely 3-ples composed by an interface, the component that requires that interface, and the component that provides it; we assume that the two components are distinct.

In the following we will use a notion of configuration equivalence that relate configurations having the same instances up to renaming. This is used to abstract away from component identifiers and bindings.

▶ **Definition 3** (Configuration equivalence). Two configurations $\langle U, Z, S, B\rangle$ and $\langle U, Z', S', B'\rangle$ are equivalent, noted $\langle U, Z, S, B\rangle \equiv \langle U, Z', S', B'\rangle$, iff there exists a bijective function ρ from Z to Z' s.t.:

1. $S(z) = S'(\rho(z))$ for every $z \in Z$; and
2. $\langle r, z_1, z_2\rangle \in B$ iff $\langle r, \rho(z_1), \rho(z_2)\rangle \in B'$.

Notation: we write $\mathcal{C}[z]$ as a lookup operation that retrieves the pair $\langle \mathcal{T}, q\rangle = S(z)$, where $\mathcal{C} = \langle U, Z, S, B\rangle$. On such a pair we then use the postfix projection operators `.type` and `.state` to retrieve $\mathcal{T}$ and q, respectively. Similarly, given a component type $\langle Q, q_0, T, \langle \mathsf{P}, \mathsf{R}\rangle, D\rangle$, we use projections to (recursively) decompose it: `.states`, `.init`, and `.trans` return the first three elements; `.prov`, `.req` return P and R; $.\mathsf{P}(q)$ and $.\mathsf{R}(q)$ return the two elements of the $D(q)$ tuple. When there is no ambiguity we take the liberty to apply the component type projections to $\langle \mathcal{T}, q\rangle$ pairs. For example, $\mathcal{C}[z].\mathsf{R}(q)$ stands for the partial function indicating the active require-ports (and their arities) of component z in configuration $\mathcal{C}$ when it is in state q.

We are now ready to formalize the notion of configuration correctness:

▶ **Definition 4** (Configuration correctness). Let us consider the configuration $\mathcal{C} = \langle U, Z, S, B\rangle$.

We write $\mathcal{C} \models_{req} (z, r, n)$ to indicate that the require-port of component z, with interface r, and associated number n is satisfied. Formally, if $n = 0$ all components other than z cannot have an active provide-port with interface r, namely for each $z' \in Z \setminus \{z\}$ such that $\mathcal{C}[z'] = \langle \mathcal{T}', q'\rangle$ we have that r is not in the domain of $\mathcal{T}'.\mathsf{P}(q')$. If $n > 0$ then the port is bound to at least n active ports, i.e., there exist n distinct components $z_1, \ldots, z_n \in Z \setminus \{z\}$ such that for every $1 \leq i \leq n$ we have that $\langle r, z, z_i\rangle \in B$, $\mathcal{C}[z_i] = \langle \mathcal{T}^i, q^i\rangle$ and r is in the domain of $\mathcal{T}^i.\mathsf{P}(q^i)$.

Similarly for provides, we write $\mathcal{C} \models_{prov} (z, p, n)$ to indicate that the provide-port of component z, with interface p, and associated number n is not bound to more than n active ports. Formally, there exist no m distinct components $z_1, \ldots, z_m \in Z \setminus \{z\}$, with $m > n$, such that for every $1 \leq i \leq m$ we have that $\langle p, z_i, z\rangle \in B$, $S(z_i) = \langle \mathcal{T}^i, q^i\rangle$ and p is in the domain of $\mathcal{T}^i.\mathsf{R}(q^i)$.

The configuration $\mathcal{C}$ is *correct* if for each component $z \in Z$, given $S(z) = \langle \mathcal{T}, q\rangle$ with $\mathcal{T} = \langle Q, q_0, T, P, D\rangle$ and $D(q) = \langle \mathcal{P}, \mathcal{R}\rangle$, we have that $(p \mapsto n_p) \in \mathcal{P}$ implies $\mathcal{C} \models_{prov} (z, p, n_p)$, and $(r \mapsto n_r) \in \mathcal{R}$ implies $\mathcal{C} \models_{req} (z, r, n_r)$.

We now formalize how configurations evolve from one state to another, by means of atomic actions:

▶ **Definition 5** (Actions). The set $\mathcal{A}$ contains the following actions:

- $stateChange(z, q_1, q_2)$ where $z \in \mathcal{Z}$: change the state of the component z from q_1 to q_2;
- $bind(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$: add a binding between z_1 and z_2 on port r;
- $unbind(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$: remove the specified binding;
- $new(z : \mathcal{T})$ where $z \in \mathcal{Z}$ and $\mathcal{T}$ is a component type: add a new component z of type $\mathcal{T}$;
- $del(z)$ where $z \in \mathcal{Z}$: remove the component z from the configuration.

The execution of actions can now be formalized using a labeled transition systems on configurations, which uses actions as labels.

▶ **Definition 6** (Reconfigurations). Reconfigurations are denoted by transitions $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ meaning that the execution of $\alpha \in \mathcal{A}$ on the configuration $\mathcal{C}$ produces a new configuration $\mathcal{C}'$. The transitions from a configuration $\mathcal{C} = \langle U, Z, S, B \rangle$ are defined as follows:

$$\mathcal{C} \xrightarrow{stateChange(z,q_1,q_2)} \langle U, Z, S', B \rangle$$
$$\text{if } \mathcal{C}[z].\texttt{state} = q_1$$
$$\text{and } (q_1, q_2) \in \mathcal{C}[z].\texttt{trans}$$
$$\text{and } S'(z') = \begin{cases} \langle \mathcal{C}[z].\texttt{type}, q_2 \rangle & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases}$$

$$\mathcal{C} \xrightarrow{bind(r,z_1,z_2)} \langle U, Z, S, B \cup \langle r, z_1, z_2 \rangle \rangle$$
$$\text{if } \langle r, z_1, z_2 \rangle \notin B$$
$$\text{and } r \in \mathcal{C}[z_1].\texttt{req} \cap \mathcal{C}[z_2].\texttt{prov}$$

$$\mathcal{C} \xrightarrow{unbind(r,z_1,z_2)} \langle U, Z, S, B \setminus \langle r, z_1, z_2 \rangle \rangle \quad \text{if } \langle r, z_1, z_2 \rangle \in B$$

$$\mathcal{C} \xrightarrow{new(z:\mathcal{T})} \langle U, Z \cup \{z\}, S', B \rangle$$
$$\text{if } z \notin Z,\ \mathcal{T} \in U$$
$$\text{and } S'(z') = \begin{cases} \langle \mathcal{T}, \mathcal{T}.\texttt{init} \rangle & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases}$$

$$\mathcal{C} \xrightarrow{del(z)} \langle U, Z \setminus \{z\}, S', B' \rangle$$
$$\text{if } S'(z') = \begin{cases} \bot & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases}$$
$$\text{and } B' = \{\langle r, z_1, z_2 \rangle \in B \mid z \notin \{z_1, z_2\}\}$$

We can now define a *deployment run*, which is a sequence of actions that transform an initial configuration into a final correct one without violating correctness along the way. A deployment run is the output we expect from a planner, when it is asked how to reach a desired target configuration.

▶ **Definition 7** (Deployment run). A *deployment run* is a sequence $\alpha_1 \dots \alpha_m$ of actions such that there exist $\mathcal{C}_i$ correct configurations such that $\mathcal{C} = \mathcal{C}_0$, $\mathcal{C}_{j-1} \xrightarrow{\alpha_j} \mathcal{C}_j$ for every $j \in \{1, \dots, m\}$.

As an example, the following is a deployment run allowing to reach the configuration depicted in Figure 1 starting from an empty configuration (we use z_1, z_2, and z_3 to identify the wordpress, apache2 and mysql components, respectively):

$new(z_1 : \mathsf{wordpress}), new(z_2 : \mathsf{apache2}), stateChange(z_2, \mathsf{uninstalled}, \mathsf{installed}),$
$bind(\mathsf{httpd}, z_1, z_2), stateChange(z_1, \mathsf{uninstalled}, \mathsf{installed}),$
$new(z_3 : \mathsf{mysql}), stateChange(z_3, \mathsf{uninstalled}, \mathsf{installed}), stateChange(z_3, \mathsf{installed}, \mathsf{running}),$
$bind(\mathsf{mysql\_up}, z_1, z_3)$

We now have all the ingredients to define the notion of *achievability*, that is our main concern: given a universe of component types, we want to know whether it is possible to deploy at least one component of a given component type $\mathcal{T}$ in a given state q.

▶ **Definition 8** (Achievability problem). The *achievability problem* has as input a universe U of component types, a component type $\mathcal{T} \in U$, and a target state q. It returns as output `true` if there exists a deployment run $\alpha_1 \dots \alpha_m$ such that $\langle U, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathcal{C}_m$ and $\mathcal{C}_m[z] = \langle \mathcal{T}, q \rangle$, for some component z in $\mathcal{C}_m$. Otherwise, it returns `false`.

In our running example, if we consider the wordpress component in the active state as target, we have that such target is achievable by the deployment run obtained by adding the action $stateChange(z_1, \mathsf{installed}, \mathsf{running})$ to the deployment run described above.

Notice that the restriction in this decision problem to one target component in a given state is not limiting. One can easily encode any given final configuration by adding a dummy provide-port enabled only by the required final states and a dummy component with requirements on all such provides.

In [5] we have proved that the achievability problem is undecidable. In this paper we present a way to relax the achievability problem in order to restore decidability. In particular, we require correctness only for the final configuration, while numerical constraints and conflicts are ignored for the intermediary states traversed during the deployment run. In many cases dealing with service deployment, violating capacity constraints during installation and configuration is not problematic because the services become publicly available only at the end. More precisely, we split the achievability problem in three separate phases: the verification of the existence of a final correct configuration that includes the target component (*Configuration* problem), the synthesis of such a configuration (*Generation* problem), and the computation of a deployment run reaching such a configuration (*Planning* problem). In this last phase, we exploit the efficient poly-time algorithm developed for the simplified *Aeolus* model without numerical constraints and conflicts. For this reason, it could indeed happen that such constraints are violated during the execution of the deployment run.

3 Configuration problem

In this section we deal with the *Configuration* problem, consisting in checking the existence of a correct component configuration including at least one instance of the target component.

The Configuration problem can be viewed as a *Constraint Satisfaction Problem* (CSP). A CSP consists of a finite set of variables, each of which associated with a domain of possible values that it could take, and a set of constraints that defines all the admissible assignments of values to the variables [15]. Given a CSP the goal is normally to find a solution—that is an assignment to the variables that satisfies all the constraints of the problem—through *one* suitable constraint solver.

For the encoding of the Configuration problem into a CSP, we can focus on an abstract representation of a configuration where components of the same type and state are grouped together. Also the specific port bindings are abstracted away: we only capture how many bindings connect the provide-ports with interface p of the components of type $\mathcal{T}$ in state q to require-ports of components of type $\mathcal{T}'$ in state q'. Formally:

▶ **Definition 9** (Abstract Configuration)**.** An abstract configuration $\mathcal{B}$ is a pair of mappings $\langle \texttt{comp}, \texttt{bind} \rangle$ such that:

- $\texttt{comp} : (\Gamma \times Q) \to \mathbb{N}$ associates to every component type $\mathcal{T}$ and one state in $\mathcal{T}.\texttt{states}$ a natural number;
- $\texttt{bind} : \mathcal{I} \times (\Gamma \times Q) \times (\Gamma \times Q) \to \mathbb{N}$ that associates to every port p and couple of component-state pairs a natural number.

▶ **Definition 10** (Concretization)**.** Given an abstract configuration $\mathcal{B} = \langle \texttt{comp}, \texttt{bind} \rangle$ we say that a correct configuration $\mathcal{C} = \langle U, Z, S, B \rangle$ is one concretization of $\mathcal{B}$ if

- the number of components of type $\mathcal{T}$ in state q in the configuration $\mathcal{C}$ is equal to $\texttt{comp}(\langle \mathcal{T}, q \rangle)$ for every type-state pair $\langle \mathcal{T}, q \rangle$
- the number of bindings between the port p provided by components of type $\mathcal{T}$ in state q and required by components of type $\mathcal{T}'$ in state q' is $\texttt{bind}(p, \langle \mathcal{T}, q \rangle, \langle \mathcal{T}', q' \rangle)$.

We write $\gamma(\mathcal{B})$ for the set of concretizations of $\mathcal{B}$.

An abstract configuration $\mathcal{B}$ is *correct* if it has at least one concretization (formally $\gamma(\mathcal{B}) \neq \emptyset$).

Not all possible abstract configurations have concretizations since the number of bindings may violate the capacity constraints and the number of components may violate the conflicts. It is however possible to characterize abstract configurations that always admit a concretization. This can be done by means of the following set of constraints.

$$\bigwedge_{p \in I} \bigwedge_{\mathcal{T},q} \mathcal{T}.\mathbf{R}(q)(p) \times \texttt{comp}(\langle \mathcal{T}, q \rangle) \leq \sum_{\mathcal{T}',q'} \texttt{bind}(p, \langle \mathcal{T}, q \rangle, \langle \mathcal{T}', q' \rangle) \tag{1a}$$

$$\bigwedge_{p \in I} \bigwedge_{\mathcal{T},q \,.\, \mathcal{T}.\mathbf{P}(q)(p) < \infty} \mathcal{T}.\mathbf{P}(q)(p) \times \texttt{comp}(\langle \mathcal{T}, q \rangle) \geq \sum_{\mathcal{T}',q'} \texttt{bind}(p, \langle \mathcal{T}, q \rangle, \langle \mathcal{T}', q' \rangle) \tag{1b}$$

$$\bigwedge_{p \in I} \bigwedge_{\mathcal{T},q \,.\, \mathcal{T}.\mathbf{P}(q)(p) = \infty} \texttt{comp}(\langle \mathcal{T}, q \rangle) = 0 \;\Rightarrow\; \sum_{\mathcal{T}',q'} \texttt{bind}(p, \langle \mathcal{T}, q \rangle, \langle \mathcal{T}', q' \rangle) = 0 \tag{1c}$$

$$\bigwedge_{p \in I} \bigwedge_{\substack{\mathcal{T},q \,.\, \mathcal{T}.\mathbf{R}(q)(p)=0 \\ \mathcal{T}.\mathbf{P}(q)(p)>0}} \texttt{comp}(\langle \mathcal{T}, q \rangle) \leq 1 \tag{1d}$$

$$\bigwedge_{p \in I} \bigwedge_{\substack{\mathcal{T},q \,.\, \\ \mathcal{T}.\mathbf{R}(q)(p)=0}} \bigwedge_{\substack{\mathcal{T}',q' = \mathcal{T},q \,.\, \\ \mathcal{T}'.\mathbf{P}(q')(p)>0}} \texttt{comp}(\langle \mathcal{T}, q \rangle) > 0 \;\Rightarrow\; \texttt{comp}(\langle \mathcal{T}, q \rangle) = 0 \tag{1e}$$

$$\bigwedge_{p \in I} \bigwedge_{\mathcal{T},q} \bigwedge_{\mathcal{T}',q' = \mathcal{T},q} \texttt{bind}(p, \langle \mathcal{T}, q \rangle, \langle \mathcal{T}, q \rangle) \leq \texttt{comp}(\langle \mathcal{T}, q \rangle) \times \texttt{comp}(\langle \mathcal{T}, q \rangle) \tag{1f}$$

$$\bigwedge_{p \in I} \bigwedge_{\mathcal{T},q} \texttt{bind}(p, \langle \mathcal{T}, q \rangle, \langle \mathcal{T}, q \rangle) \leq \texttt{comp}(\langle \mathcal{T}, q \rangle) \times (\texttt{comp}(\langle \mathcal{T}, q \rangle) - 1) \tag{1g}$$

Constraint 1a enforces the fact that the number of bindings connected to the require-ports p of components of type-state pair $\langle \mathcal{T}, q \rangle$ cannot be smaller than the total requirements computed as the sum of the single requirements of each instance of type-state $\langle \mathcal{T}, q \rangle$. Symmetrically, constraint 1b guarantees that the number of bindings connected to the provide-ports

p of components of type-state pair $\langle \mathcal{T}, q \rangle$ cannot be greater than the total available capacity computed as the sum of the single capacities of each instance of type-state $\langle \mathcal{T}, q \rangle$. In case the port capacity is unbounded (i.e., ∞), it is sufficient to have at least one instance that activates such port to support any possible requirement (see constraint 1c). Constraints 1d and 1e deal with conflicts. In particular the constraint 1d limits to at most one the instances of component type-state pairs that can simultaneously provide and being in conflict with a given port. Constraint 1e enforces instead that when a conflict is active, then no other instance providing the same port exists. Finally, the constraints 1f and 1g guarantee that there are enough pairs of distinct instances to establish all the necessary bindings. Two distinct constraints are used: the first one deals with bindings between components of two different type-state pair, the second one considers those bindings that are established between two components of the same type-state pair.

▶ **Lemma 11.** *An abstract configuration $\mathcal{B}$ satisfies the constraints 1 iff it is correct.*

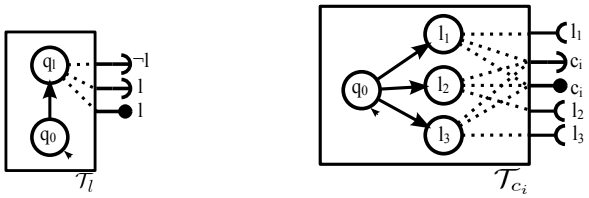
Proof. Suppose that $\mathcal{C}$ is a concretization of the abstract configuration $\mathcal{B}$. Now suppose that one among constraints 1 is violated.

- If 1a is violated, in $\mathcal{C}$ there exists a component of type $\mathcal{T}$ in state q requiring port p and having less than $\mathcal{T}.\mathsf{R}(q)(p)$ bindings. By the correctness of $\mathcal{C}$ this is impossible.
- If 1b is violated, in $\mathcal{C}$ there exists a component of type $\mathcal{T}$ in state q that has more than $\mathcal{T}.\mathsf{P}(q)(p)$ bindings to port p. This is impossible since by definition $\mathcal{C}$ is correct.
- If 1c is violated, in $\mathcal{C}$ there should be bindings connected to components of type-state pairs that do not appear in the configuration. By the correctness of $\mathcal{C}$ this is impossible.
- If 1d is violated, then there are two instances of a component of type $\mathcal{T}$ in state q that simultaneously provide and are in conflict with the port p. By the correctness of $\mathcal{C}$ this is impossible.
- If 1e is violated, then there is an instance of a component that is in conflict with a port p and a different instance that provides p. By the correctness of $\mathcal{C}$ this is impossible.
- If 1f is violated, then there exist a port p and two components of different type-state pair such that there exists two bindings connecting the port p between them. By the correctness of $\mathcal{C}$ this is impossible.
- If 1g is violated, then there exist a port p and either a component activating a provide and a require-port p which are connected, or two components of the same type-state pair with two bindings connecting the port p between them. By the correctness of $\mathcal{C}$ this is impossible.

Therefore, if $\mathcal{B}$ is correct then all of constraints 1 are satisfied.

Now let us suppose that there exists an abstract configuration $\mathcal{B}\langle \texttt{comp}, \texttt{bind} \rangle$ that satisfies the constraints 1. We show the existence of a concretization $\mathcal{C}$, such that for every component type-state pair $\langle \mathcal{T}, q \rangle$ it has $\texttt{comp}(\langle \mathcal{T}, q \rangle)$ different instances of type $\mathcal{T}$ in state q. Conflicts cannot happen between the components, otherwise constraints 1d or 1e would be violated. We now discuss the bindings in $\mathcal{C}$. From constraint 1f we have that it is possible to have a number of bindings $\texttt{bind}(p, \langle \mathcal{T}, q \rangle, \langle \mathcal{T}', q' \rangle)$ between distinct pairs of instances of type-state $\langle \mathcal{T}, q \rangle$ and $\langle \mathcal{T}', q' \rangle$ respectively providing and requiring port p; moreover, from constraint 1g we have that it is possible to have a number of bindings $\texttt{bind}(p, \langle \mathcal{T}, q \rangle, \langle \mathcal{T}, q \rangle)$ between distinct instances of type-state $\langle \mathcal{T}, q \rangle$ providing and requiring port p. It remains to show that there exists at least one distribution of these bindings that satisfies the capacity constraints on all the provide and require-ports.

Assume, by contraposition, that there exists no distribution of bindings that satisfies the capacity constraints. This means that for every possible distribution there are always



(a) Literal encoding. **(b)** Clause encoding.

Figure 3 3-SAT encoding into Aeolus.

provide-ports with a number of bindings greater than the capacity, or require-ports with a number of bindings smaller than those that are required. We call *discrepancy* the overall number of excessive bindings connected to provide-ports plus the number of missing bindings in require-ports. We consider one of the binding distributions with minimal discrepancy. It is not restrictive to assume that there is a component r_1 of type-state $\langle \mathcal{T}, q \rangle$ with a provide-port p having an excessive number of bindings. This assumption is not restrictive because if this is not the case, there is at least one require-port with an insufficient number of bindings, and the following reasoning can be symmetrically applied. By constraint 1b there is another instance r_2 of type-state $\langle \mathcal{T}, q \rangle$ that can host at least one additional binding on its provide-port p without exceeding its capacity. The idea is to rebind one of the bindings on the provide-port p of r_1 to the provide-port p of r_2. This can be done because there exist at least two components connected to the port p of r_1 which are not already connected to r_2 (the port of r_1 strictly exceeds its capacity while that of r_2 can host at least one additional binding). Among these two components, at least one component (let us call it r) is different from r_2. It is safe to rebind the require-port p of r from r_1 to r_2 because $r \neq r_2$ and r is not already connected to r_2. Upon rebinding, in the new configuration, discrepancy is strictly reduced thus contradicting minimality. ◀

In the light of Lemma 11, in order to check if there is a correct configuration containing the target component type-state pair $\langle \mathcal{T}_t, q_t \rangle$, we can simply check if there is a correct abstract configuration where $\texttt{comp}(\langle \mathcal{T}_t, q_t \rangle) > 0$. This can be done by solving a CSP problem where the functions `comp` and `bind` are defined by means of a set of variables having as domain $\mathbb{N}$ and by enforcing the constraints 1 (besides the constraint $\texttt{comp}(\langle \mathcal{T}_t, q_t \rangle) > 0$).

Thanks to this encoding it is possible to prove that the Configuration problem is NP-complete.

▶ **Theorem 12.** *The Configuration problem is NP-complete.*

Proof. To prove the NP-hardness we reduce the 3-SAT Problem into Configuration.

As depicted in Figure 3 a literal l of a 3-SAT formula φ is encoded into a component type $\mathcal{T}_l$ having, beyond an initial state, a final state q_l that provides a port l, is in conflict with the same port l and with its negation $\neg l$. A clause $c_i = l_1 \vee l_2 \vee l_3$ is encoded into a component type $\mathcal{T}_{c_i}$ having, beyond an initial state, three states l_j that require the corresponding port l_j, provide the port c_i and are in conflict with it. The target component type $\mathcal{T}_t$ is a two state component where the final state q_t requires for every clause c_i the port c_i.

The conflict ports impose that every correct configuration has at most one instance of type $\mathcal{T}_l$ or $\mathcal{T}_{\neg l}$. Similarly, at most one instance of $\mathcal{T}_{c_i}$ could be present. It is easy to see that a formula φ is satisfiable iff there exists a configuration where an instance of $\mathcal{T}_t$ is present in the state q_t. Indeed, if φ is satisfiable then it is possible to consider a configuration having

an instance of $\mathcal{T}_l$ for every literal l assigned to true. By validity of φ under the considered literal assignments, it is also possible to bind all the clause components $\mathcal{T}_{c_i}$ to at least one corresponding literal, and then all the $\mathcal{T}_{c_i}$ instances to the target component $\mathcal{T}_t$ in state q_t. Similarly, if there exists a configuration where $\mathcal{T}_t$ in state q_t is present we can assign to true every literal l_i such that an instance of $\mathcal{T}_l$ is present in the configuration and the remaining literals to false. This assignment satisfies the formula φ.

We now have to prove that Configuration is in NP. To this aim we encode Configuration into an Integer Linear Programming (ILP) problem, a well-known problem in NP [17].

The first problem to resort to ILP is that the constraints 1f and 1g are not linear since there are multiplications between two variables: $\mathtt{comp}(\langle \mathcal{T}, q\rangle) \times \mathtt{comp}(\langle \mathcal{T}', q'\rangle)$ in constraint 1f and $\mathtt{comp}(\langle \mathcal{T}, q\rangle) \times (\mathtt{comp}(\langle \mathcal{T}, q\rangle) - 1)$ in constraint 1g. Luckily, these constraints can be reformulated into a polynomial number of linear constraints. We descrive in details how to reformulate constraint 1f; constraint 1g can be translated similarly.

The basic observation is that in a correct configuration there is no need to have more bindings than those strictly needed to satisfy the require-ports. In the light of this observation, it is safe to assume that the number of bindings $\mathtt{bind}(p, \langle \mathcal{T}, q\rangle, \langle \mathcal{T}', q'\rangle)$ is less than $\mathtt{comp}(\langle \mathcal{T}', q'\rangle) \times \mathcal{T}'.\mathsf{R}(q')(p)$ which is the number of bindings required to satisfy the require-ports p of the instances of type-state $\langle \mathcal{T}', q'\rangle$. Hence we can consider the disequation $\mathtt{bind}(p, \langle \mathcal{T}, q\rangle, \langle \mathcal{T}', q'\rangle) \leq \mathtt{comp}(\langle \mathcal{T}, q\rangle) \times \mathtt{comp}(\langle \mathcal{T}', q'\rangle)$ in 1f only in those cases in which $\mathtt{comp}(\langle \mathcal{T}, q\rangle) < \mathcal{T}'.\mathsf{R}(q')(p)$, and for all the other cases consider the disequation $\mathtt{bind}(p, \langle \mathcal{T}, q\rangle, \langle \mathcal{T}', q'\rangle) \leq \mathcal{T}'.\mathsf{R}(q')(p) \times \mathtt{comp}(\langle \mathcal{T}', q'\rangle)$ without variable multiplication.

We now discuss how to transform $\mathtt{bind}(p, \langle \mathcal{T}, q\rangle, \langle \mathcal{T}', q'\rangle) \leq \mathtt{comp}(\langle \mathcal{T}, q\rangle) \times \mathtt{comp}(\langle \mathcal{T}', q'\rangle)$ under the assumption that $\mathtt{comp}(\langle \mathcal{T}, q\rangle) < \mathcal{T}'.\mathsf{R}(q')(p)$. Assume that $n = \lfloor \log(\mathcal{T}'.\mathsf{R}(q')(p))\rfloor$. We can consider $2n$ variables: $x_1, \cdots, x_n$ which are the n digits of the binary representation of $\mathtt{comp}(\langle \mathcal{T}, q\rangle)$, and $y_1, \cdots, y_n$ which coincides with 0 for those indexes i for which x_i is 0, and $2^i \times \mathtt{comp}(\langle \mathcal{T}', q'\rangle)$ for those indexes i for which x_i is 1. In this way, the sum of all the variables $y_1, \cdots, y_n$ coincides with $\mathtt{comp}(\langle \mathcal{T}, q\rangle) \times \mathtt{comp}(\langle \mathcal{T}', q'\rangle)$, and this value is obtained without exploiting variable multiplication, at the price of adding only a polynomial number of variables.

Summarizing, the new constraints replacing the constraint 1f are as follows:

$$\mathtt{bind}(p, \langle \mathcal{T}, q\rangle, \langle \mathcal{T}, q\rangle) \leq \mathcal{T}.\mathsf{R}(q)(p) \times \mathtt{comp}(\langle \mathcal{T}, q\rangle) \tag{2a}$$

$$\forall i \in [1, n] \,.\, 0 \leq x_i \leq 1 \qquad \forall i \in [1, n] \,.\, 0 \leq y_i \leq 1 \tag{2b}$$

$$\mathtt{comp}(\langle \mathcal{T}, q\rangle) < \mathcal{T}.\mathsf{R}(q)(p) \Rightarrow \sum_{i=0}^{n} 2^i x_i = \mathtt{comp}(\langle \mathcal{T}, q\rangle) \tag{2c}$$

$$\forall i \in [1, n] \,.\, x_i = 1 \Rightarrow y_i = 2^i \times \mathtt{comp}(\langle \mathcal{T}, q\rangle) \qquad \forall i \in [1, n] \,.\, x_i = 0 \Rightarrow y_i = 0 \tag{2d}$$

$$\mathtt{comp}(\langle \mathcal{T}, q\rangle) < \mathcal{T}.\mathsf{R}(q)(p) \Rightarrow \mathtt{bind}(p, \langle \mathcal{T}, q\rangle, \langle \mathcal{T}, q\rangle) \leq \sum_{i=0}^{n} y_i \tag{2e}$$

The first constraint binds the number of bindings to satisfy all the require-ports as explained before. The remaining constraints encode precisely the constraint 1f for the cases when $\mathtt{comp}(\langle \mathcal{T}, q\rangle)$ is smaller than $\mathcal{T}'.\mathsf{R}(q')(p)$. To do so, $5n + 2$ linear constraints are used introducing $2n$ fresh variables (i.e., x_i and y_i) which are used to compute $\mathtt{comp}(\langle \mathcal{T}, q\rangle) \times \mathtt{comp}(\langle \mathcal{T}', q'\rangle)$ without variable multiplication as described above.

Applying this transformation, the Configuration problem can be checked solving a polynomial number of linear constraints. However, some of these constraints are logical implication that have to be compiled into the linear (dis)equations of an ILP problem. This is possible exploiting a result from disjunctive programming [18] that indicates how to encode with the addition of a polynomial amount of variable and (dis)equations the requirement

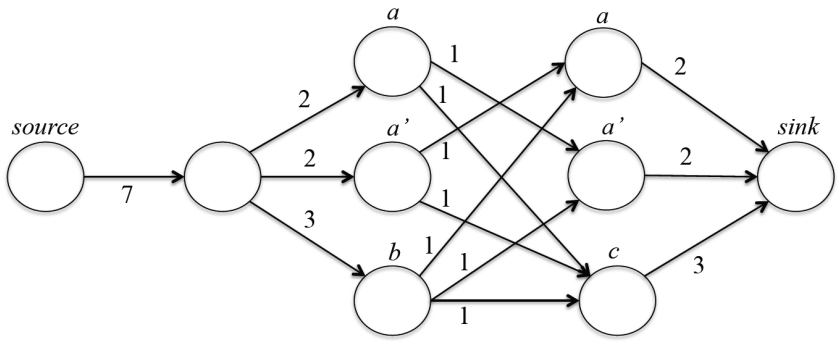


Figure 4 Example of max-flow graph.

that at least one in a set of (dis)equations holds. For instance, the logical implication in the constraint 2c holds if at least one of $\mathtt{comp}(\langle \mathcal{T}, q\rangle) \geq \mathcal{T}'.\mathsf{R}(q')(p)$ or $\sum_{i=0}^{n} 2^i x_i = \mathtt{comp}(\langle \mathcal{T}, q\rangle)$ holds. The Configuration problem can be therefore mapped into an ILP with a polynomial number of constraints and variables. Since ILP is in NP [17] this also holds for Configuration. ◀

4 Configuration generation

As previously shown, by using a constraint solver it is possible to compute an abstract configuration such that its concretizations have an instance of the target component type in the target state. Thanks to the fact that the Configuration problem is NP-complete, we have that the size of a representation of the abstract configuration, if any, is polynomially bounded. But, in the worst case, the generation of a concretization could require an exponential amount of time since the number of the components could be exponential w.r.t. the size of the abstract configuration representation. For instance, an abstract configuration of $O(\log(n))$ space could require the creation of n components but, to concretely represent the n instances, $O(n)$ space is needed.

In the proof of the following theorem we show how to generate, starting from the target abstract configuration, a correct configuration which is equivalent to one of its concretization. Hence, this configuration will contain the target component in the target state. If we denote with Generation the problem of computing a configuration equivalent to a concretization of the target abstract configuration, we have the following theorem.

▶ **Theorem 13.** *The Generation problem is EXP-time.*

Proof. The first step to solve the problem is to solve the Configuration problem to obtain an abstract configuration and then to compute one configuration that is equivalent to one of its concretizations. Solving a Configuration problem is NP-complete and therefore can be done in EXP-time.

Starting from the abstract configuration $\mathcal{B} = \langle \mathtt{comp}, \mathtt{bind}\rangle$ it is possible to generate for every component-type/state pair $\langle \mathcal{T}, q\rangle$ exactly $\mathtt{comp}(\langle \mathcal{T}, q\rangle)$ component of type $\mathcal{T}$ in state q. This can take an exponential time w.r.t. the size of the input since $\mathtt{comp}(\langle \mathcal{T}, q\rangle)$ can be stored in space $O(\log(\mathtt{comp}(\langle \mathcal{T}, q\rangle)))$.

The binding generation can be done solving a maximal flow problem. Given a port p it is possible to consider the graph where the nodes represent the component instances providing

p (provider) and the component instances requiring p (requirer), and an edge with capacity 1 exists from every provider to every distinct requirer. Every provider node representing an instance of type $\mathcal{T}$ in state q has an incoming edge from an auxiliary node with capacity equal to its provide numerical constraint $\mathcal{T}.\mathsf{P}(q)(p)$. This auxiliary node is connected to a source node with an edge that has a capacity equal to the sum of the bindings connected to the provide-port p on all the components of any type $\sum_{\langle\mathcal{T},q\rangle,\langle\mathcal{T}',q'\rangle} \mathtt{bind}(p,\langle\mathcal{T},q\rangle,\langle\mathcal{T}',q'\rangle))$. On the other hand, every requirer node is connected to a sink node with an edge having capacity equal to its require numerical constraint $\mathcal{T}.\mathsf{R}(q)(p)$.

As an example, Figure 4 shows the max-flow graph obtained when there are two kinds of provider of type-state $\langle\mathcal{T}_a,q_a\rangle$ and $\langle\mathcal{T}_b,q_b\rangle$, and two kinds of requirer of type-state $\langle\mathcal{T}_a,q_a\rangle$ and $\langle\mathcal{T}_c,q_c\rangle$. Assuming that two instances of type-state $\langle\mathcal{T}_a,q_a\rangle$ are in the configuration, and one instance for the other two type-states, three nodes a, a' and b are considered as provider and a, a' and c as requirer. Components $\mathcal{T}_a$ in state q_a provide the port p with a constraint of 2, while components of type $\mathcal{T}_b$ in state q_b provide p with a numerical constraint of 3. Components $\mathcal{T}_a$ in state q_a requires at least 2 bindings on its p port, while those of type $\mathcal{T}_c$ in state q_c requires 3 bindings. The solution depicted in the Figure shows the graph when it is required that 7 bindings on ports p should be established.

The maximal flow of this graph is $\sum_{\langle\mathcal{T},q\rangle,\langle\mathcal{T}',q'\rangle} \mathtt{bind}(p,\langle\mathcal{T}',q'\rangle,\langle\mathcal{T},q\rangle)$ and the edges between the provider and requirer selected by the maximal flow algorithm correspond to the bindings to create in the configuration. The configuration that can be obtained in this way is equivalent to any of the concretizations of the given abstract configuration.

Since the maximal flow can be computed by the Ford–Fulkerson algorithm [3] in $O(Ef)$, where E is the number of edges and f is the flow, and the number of components is exponential on the size of the input we have that the generation of the bindings is in EXP-time. ◀

5 Plan generation

In [14] we have presented an algorithm for solving the achievability problem under the assumption that the numerical constraints associated to require-ports are always equal to 1, and those associated to provide-ports are always ∞. This means that no conflicts can be expressed, as well as redundancy requirements on require-ports or maximal capacities on provide-ports. The algorithm runs in polynomial time, and is also capable of returning a corresponding deployment run when it exists (or an empty sequence when it does not). We have also realized a prototype called Metis [13] that implements this algorithm: in the following we call $\mathit{Metis}(U,\langle\mathcal{T}_t,q_t\rangle)$ the deployment run returned by our algorithm when executed on the universe of components U and target component type-state $\langle\mathcal{T}_t,q_t\rangle$.

Given a target configuration $\mathcal{C}$, we show how to exploit Metis in order to generate a sequence of action that reaches the final configuration $\mathcal{C}$. As Metis does not take into account the numerical constraints on component ports, it could happen that the intermediary configurations traversed during the execution of the returned sequence of action are not correct. Nevertheless, we guarantee that the finally reached configuration $\mathcal{C}$ is correct. We call Planning this specific problem of generating a sequence of actions that reaches the target configuration $\mathcal{C}$.

The idea is to start from the configuration $\mathcal{C}$ and generate a universe of component types $U_{\mathcal{C}}$, that extends the original universe U with new component types $\mathcal{T}_z$—one for each instance z in $\mathcal{C}$—plus a specific target type $\mathcal{T}_{\mathcal{C}}$ with a target state $q_{\mathcal{C}}$, such that the sequence of actions computed by $\mathit{Metis}(U_{\mathcal{C}},\langle\mathcal{T}_{\mathcal{C}},q_{\mathcal{C}}\rangle)$ can be post-processed to obtain the desired solution to the Planning problem.

We start by presenting the generation of $U_{\mathcal{C}}$. For every component z of type $\mathcal{T}$ in state q in the configuration $\mathcal{C}$, a new type of component $\mathcal{T}_z$ is added to the original universe. The new type $\mathcal{T}_z$ is equal to $\mathcal{T}$ with the following exceptions:

- every port p provided by $\mathcal{T}$ in state q is replaced by p_z;
- every port r required by $\mathcal{T}$ in state q is replaced with ports $\{r_{z'} \mid \langle r, z', z\rangle \in B\}$ where B is the set of bindings of $\mathcal{C}$ (i.e., every require-port is replaced by ports depending on the component that provides that port);
- the state q provides the port z.

The universe will include also the new target component type $\mathcal{T}_{\mathcal{C}}$: it has two states, the initial one and a connected state $q_{\mathcal{C}}$ that requires the port z for every component z in $\mathcal{C}$.

Given these definitions, it is immediate to see that $Metis(U_{\mathcal{C}}, \langle \mathcal{T}_{\mathcal{C}}, q_{\mathcal{C}}\rangle)$ will return either an empty sequence or a sequence of actions leading to a configuration that includes the components and bindings in $\mathcal{C}$ plus other components, like the target dummy components of type $\mathcal{T}_{\mathcal{C}}$ or other temporary components. By temporary components, we mean instances of components necessary during the execution of the deployment actions (e.g., to satisfy some requirements in intermediary states to be traversed to reach the final ones) but not present in the desired configuration $\mathcal{C}$. These temporary components are easily identifiable in the configuration reached by the synthesized plan because are not connected to the target component of type $\mathcal{T}_{\mathcal{C}}$. These additional components can be removed by adding to the plan the corresponding *del* actions.

We now discuss how to post-process the plan possibly generated by Metis. First of all, it is necessary to replace the bind actions $bind(r_x, z_1, z_2)$ with $bind(r, z_1, z_2)$ and deleting all the actions involving components of type $\mathcal{T}_{\mathcal{C}}$, to obtain a sequence of actions reaching a configuration equivalent to $\mathcal{C}$ up-to the additional temporary components (the target $\mathcal{T}_{\mathcal{C}}$ is not created). Then *del* actions are added to the plan for all the temporary components as described above.

▶ **Theorem 14.** *The Planning problem can be solved in polynomial time.*

Proof. The polynomiality of the algorithm described above derives directly from the polynomiality of Metis [14]. Indeed, the extended universe U' contains only ad additional linear number of component types w.r.t. the size of the input, and the post-processing of the sequence of actions performs a scan of the plan generated by Metis and add some *del* action. This scan as well as the addition of *del* actions can be done in polynomial time since the polynomiality of Metis bounds the plan and the reached configuration to be polynomial. ◀

As a consequence of this result we have that the chain of algorithms solving in sequence the Configuration, Generation, and Planning problems, compute a sequence of deployment actions to reach the desired target configuration in EXP-Time. From the practical point of view this complexity is however reached only when there are components that require a large number of other components to satisfy their needs. Indeed, in this case the space needed to store the port capacity may take $O(n)$ space requiring $O(2^n)$ deployment action to generate the providers.

6 Related work and Conclusion

With the current popularity of cloud computing, the problem of automating application deployment has recently attracted a lot of attention. As of today most industrial products offered by big companies, such as Amazon, HP and IBM, rely on holistic approaches where a complete model for the entire application is defined and the deployment plan is derived

in a top-down manner. In this context, one prominent work is represented by the TOSCA (Topology and Orchestration Specification for Cloud Applications) standard [16], promoted by the OASIS consortium for open standards. TOSCA proposes an XML-like (or YAML) rich language to describe an application. Deployment plans are usually BPEL notations, i.e., work-flow defined in the context of business process modeling.

Using these approaches the burden of specifying what components should be deployed and how to interconnect them is left to system administrators or cloud engineers. On the contrary, Zephyrus [4] and ConfSolve [11] automatize also this task starting from a high-level declarative specification of the desired configuration. As previously mentioned, Zephyrus is grounded on the Aeolus model and takes into account also packages, repositories, as well as the optimality of the proposed solution. ConfSolve relies on constraint solving techniques to propose an optimal allocation of virtual machines to servers, and of applications to virtual machines. An object-oriented declarative language is used to describe the entities (e.g., machines and services), the constraints, and the optimization criteria. Similarly to Zephyrus, but differently from Metis, ConfSolve does not consider the problem of synthesizing a low-level plan to reach the final configuration.

Two recent efforts, Feinerer's work on UML [8] and Engage [9], are more similar to our approach as they both rely on a solver to plan deployments. Feinerer's work is based on the UML component model, which includes conflicts and dependencies, but lacks the aspects concerning virtual machines and deployment. Engage, on the other hand, offers no support for conflicts in the specification language. Neither Feinerer's work nor Engage allows to find a deployment that uses resources in an optimal way, minimizing the number and cost of needed components as can be obtained by Zephyrus. Furthermore, no other tool that we are aware of allows to declare capacity or replication constraints, which are essential non functional constraints for any non-trivial, scalable application.

7 Conclusion

In this article we have studied the problem of reconfiguration using the abstract component model named Aeolus, with all its expressive power. This model allows to describe complex component systems with functional constraints, and non-functional constraints like redundancy, capacity and conflicts.

We have carefully decomposed the reconfiguration problem into three steps, Configuration, Generation and Planning, and we showed how to recover decidability by imposing restrictions only on the transient states of the Planning phase.

This restriction corresponds to the realistic deployment conditions of many current distributed applications, thus paving the way to a new generation of smarters deployment tools that will help automate a larger part of the work that is today requiring significant human intervention. A research prototype, described in [4], has been developed to show the viability of this approahc.

References

1 Amazon. AWS CloudFormation. http://aws.amazon.com/cloudformation/.

2 CenturyLink. Cloud Blueprints. https://www.centurylinkcloud.com/blueprints/.

3 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

4 Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. Automated synthesis and deployment of cloud applications. In *ASE*, 2014.

5 Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Aeolus: A component model for the cloud. *Inf. Comput.*, 239:100–121, 2014.

6 Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Component Reconfiguration in the Presence of Conflicts. In *ICALP 2013: 40th International Colloquium on Automata, Languages and Programming*, volume 7966 of *LNCS*, 2013.

7 Roberto Di Cosmo, Stefano Zacchiroli, and Gianluigi Zavattaro. Towards a Formal Component Model for the Cloud. In *SEFM 2012*, volume 7504 of *LNCS*, 2012.

8 Ingo Feinerer. Efficient large-scale configuration via integer linear programming. *AI EDAM*, 27(1):37–49, 2013.

9 Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. Engage: a deployment management system. In *PLDI*, 2012.

10 Flexiant. Bento Boxes. http://www.flexiant.com/2012/12/03/application-provisioning/.

11 John A. Hewson, Paul Anderson, and Andrew D. Gordon. A Declarative Approach to Automated Configuration. In *LISA*, 2012.

12 Juju, DevOps Distilled. https://jujucharms.com/.

13 Tudor A. Lascu, Jacopo Mauro, and Gianluigi Zavattaro. A Planning Tool Supporting the Deployment of Cloud Applications. In *ICTAI*, 2013.

14 Tudor A. Lascu, Jacopo Mauro, and Gianluigi Zavattaro. Automatic Component Deployment in the Presence of Circular Dependencies. In *FACS*, 2013.

15 Alan K. Mackworth. Consistency in Networks of Relations. *Artif. Intell.*, 8(1):99–118, 1977.

16 OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html.

17 Christos H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981.

18 Juan Pablo Vielma and George L. Nemhauser. Modeling disjunctive constraints with a logarithmic number of binary variables and constraints. *Math. Program.*, 128(1-2):49–72, 2011.