Towards a Formal Component Model for the Cloud *

Roberto Di Cosmo[1], Stefano Zacchiroli[1], and Gianluigi Zavattaro[2]

[1] Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France
roberto@dicosmo.org,zack@pps.univ-paris-diderot.fr
[2] Focus Team, Univ of Bologna/INRIA, Italy, zavattar@cs.unibo.it

Abstract. We consider the problem of deploying and (re)configuring resources in a "cloud" setting, where interconnected software components and services can be deployed on clusters of heterogeneous (virtual) machines that can be created and connected on-the-fly. We introduce the Aeolus component model to capture similar scenarii from realistic cloud deployments, and instrument automated planning of day-to-day activities such as software upgrade planning, service deployment, elastic scaling, etc. We formalize the model and characterize the feasibility and complexity of configuration achievability in Aeolus.

1 Introduction

The expression "*cloud computing*" is broadly used to refer to the possibility of building sophisticated distributed software systems that can be run, on-demand, on a virtualized infrastructure at a fraction of the cost which was necessary just a few years ago. Reaping all the benefits of cloud computing is not easy: while the infrastructure cost falls dramatically, writing a distributed software system that adapts to the demand is difficult, and maintaining and reconfiguring it is a serious challenge.

Several recent industry initiatives strive to address this challenge. CloudFoundry [6] provides tools that allow to select, connect, and push to a cloud well defined services (databases, message buses, ...), that are used as building blocks for writing applications using one of the supported frameworks. Canonical is developing Juju [8], that shares several of CloudFoundry concepts. In the academic world, the Fractal component model [4] focuses on expressivity and flexibility: it provides a general notion of component assembly that can be used to describe concisely, and independently of the programming language, a complex software system. Building on Fractal, FraSCAti [16] provides a middleware that can be used to deploy applications in the cloud.

In all these approaches, the goal is to allow the user to assemble a working system out of components that have been specifically designed or adapted to work together. Component selection and interconnection are the responsibility of the user, and if some reconfiguration needs to happen, it is either obtained by reassembling the system manually, or by writing specific code that is still the responsibility of the user.

While *expressivity* is certainly important, solving the cloud challenge also requires *automation*: when the number of components grows, or the need to reconfigure appears

* Work partially supported by Aeolus project, ANR-2010-SEGI-013-01, and performed at IRILL, center for Free Software Research and Innovation in Paris, France, http://www.irill.org

more frequently, it is essential to be able to specify at a certain level of abstraction a particular configuration of the distributed software system, and to develop tools that provide a set of possible evolution paths leading from the current system configuration to one that corresponds to a user request.

Automated approaches have been developed for the particular case of configuring *package-based* FOSS (Free and Open Source Software) distributions on a *single* system, and there are generic, solver-based component managers for this task [1].

The goal of this paper is to lay the foundations of such an automated approach for the much more complex situation that arises when one needs to: (re)configure not a single machine, but a variety of possibly "elastic" clusters of heterogeneous machines, living in different domains and offering interconnected services that need to be stopped, modified, and restarted in a specific order for the reconfiguration to be successful.

To this end, we propose (in Sections 2 and 3) a novel component model, called *Aeolus* and loosely inspired by Fractal, where components describe resources which provide and require different functionalities, and may be created or destroyed. As a major difference, though, Aeolus components are equipped with *state machines* that describe *declaratively* how required and provided functionalities are enacted. That way we can see Aeolus as an *abstraction* of Fractal, yet expressive enough to capture many common deployment scenarii in the cloud. The declarative information is essential to provide a planner with the input needed for exploring the possible evolution paths of the system, and propose a reconfiguration plan, which is the key automation enabler.

In Section 4 we study formally the complexity of finding a deployment plan in Aeolus, a property which we call *achievability*. We show that achievability is *decidable in polynomial time* if no capacity restriction is imposed on the provided and required functionalities. This simplified model, called Aeolus$^-$, corresponds to what current mainstream tools can handle, and our result explains why it is still possible, in simple cases, to manage such systems manually.

We show that achievability *becomes undecidable* as soon as one allows to impose restrictions on the number of connections between required and provided functionalities. This limiting result is particularly significant, as some industrial tools are starting to incorporate such restrictions to account for capacity limitations of services in the cloud. The model that we propose to deal with these aspects is called *Aeolus flat* to stress that we do not deal yet with hierarchically nested components or location boundaries, that we will address in future work on a comprehensive Aeolus model (Section 6).

2 Use cases

We introduce the key features of Aeolus by eliciting them, step-by-step, from the analysis of realistic scenarii. As a running example, we consider several deployment use cases for WordPress, a popular weblog solution that requires several software services to operate, the main ones being a Web server and a SQL database. We present the use cases in order of increasing complexity ranging from the simplest ones, where everything runs on a single physical machine, to more complex ones where the whole appliance runs on a cloud.

```
Package: wordpress
Version: 3.0.5+dfsg-0+squeeze1
Depends: httpd, mysql-client, php5, php5-mysql, libphp-phpmailer (>= 1.73-4), [...]

Package: mysql-server-5.5
Source: mysql-5.5
Version: 5.5.17-4
Provides: mysql-server, virtual-mysql-server
Depends: libc6 (>= 2.12), zlib1g (>= 1:1.1.4), debconf, [...]
Pre-Depends: mysql-common (>= 5.5.17-4), adduser (>= 3.40), debconf

Package: apache2
Version: 2.4.1-2
Maintainer: Debian Apache Maintainers <debian-apache@...>
Depends: lsb-base, procps, perl, mime-support, apache2-bin (= 2.4.1-2),
 apache2-data (= 2.4.1-2)
Conflicts: apache2.2-common
Provides: httpd
Description: Apache HTTP Server
```

Fig. 1. Debian package metadata for WordPress, Mysql and the Apache web server (excerpt)

Use case 1 — Package installation. Before considering the services that a machine is offering to others (locally or over the network), we need to model the *software installation* on the machine itself, so we will see how to model the three main components needed by WordPress, as far as their installation is concerned.

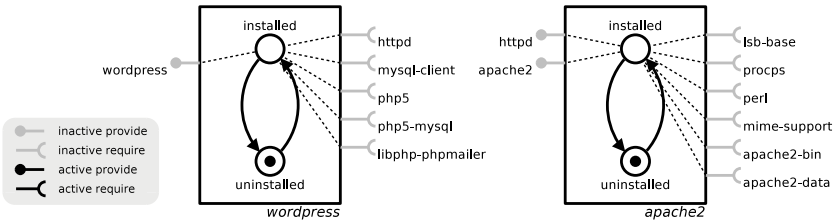
Software is often distributed according to the *package* paradigm [7], popularized by FOSS distributions, where software is shipped at the granularity of bundles called *packages*. Each package contains the actual software artifact, its default configuration, as well as a bunch of package metadata.

On a given machine, a software package may exists in different states (e.g. installed or uninstalled) and it should go through a complex sequence of states in different phases of unpacking and configuration to get there. In each of its states, similarly to what happens in most software component models [9], a package may have context *requirements* and offer some features, that we call *provides*. For instance in Debian, a popular FOSS distribution, requirements come in two flavors: *Depends* which must be satisfied before a package can be used, and *Pre-Depends* which must be satisfied before a package can be installed. This distinction is of general interest, as we will see later, so we will distinguish between *weak* requirements and *strong* requirements.

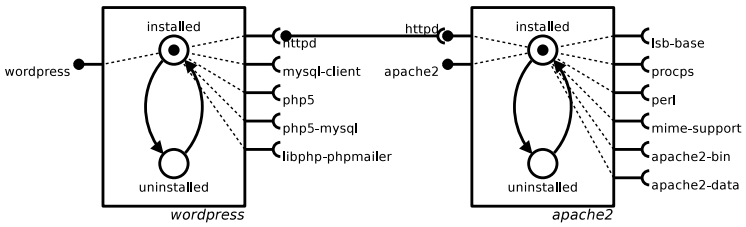
An excerpt of the concrete description of the packages present in Debian for WordPress, Apache2 and MySQL are shown in Fig. 1.

To model a software package at this level of abstraction, we may use a simple state machine, with requirements and provides associated to each state. The ingredients of this model are very simple: a set of states Q, an initial state q_0, a transition function T from states to states, a set R of requirements, a set P of provides, and a function that maps states to the requirements and provides that are *active* at that state, and tells whether requirements are weak or strong. We call *resource type* any such tuple $\langle Q, q_0, T, P, D \rangle$, which will be formalized in Definition 1.

A system *configuration* built out of a collection of resources types is given by an instance of each resource type, with its current state, and a set of connections between requirements and provides of the different resources, that indicate which provide is ful-



(a) available components, not installed



(b) installed components, bound together on the `httpd` port

Fig. 2. A simple graphical description of the basic model of a package

filling the need of each requirement. A configuration is *correct* if all the requires which are active are satisfied by active provides; this will be made precise in Definition 3.

A natural graphical notation captures all these pieces of information: Fig. 2 presents two correct configurations of a system built using the components from Fig. 1 (only modeling the dependency on `httpd` underlined in the metadata). In Fig. 2(b) the WordPress package is in the installed state, and activates the requirement on `httpd`; Apache2 is also in the installed state, so the `httpd` provide is active and is used to satisfy the requirement, fact which is visualized by the *binding* connecting the two ports.

Use case 2 — Services and packages. Installing the software on a single machine is a process that can already be automated using *package managers*: on Debian for instance, you only need to have an installed Apache server to be able to install WordPress. But bringing it *in production* requires to activate the associated service, which is more tricky and less automated: the system administrator will need to edit configuration files so that WordPress knows the network addresses of an accessible MySQL instance.

The ingredients we have seen up to now in our model are sufficient to capture the dependencies among services, as shown in Fig. 3. There we have added to each package an extra state corresponding to the activation of the associated service, and the strong requirement (graphically indicated by the double tip on the arrow) on `mysql_up` captures the fact that WordPress cannot be started before MySQL is running. In this case, the bindings really correspond to a piece of configuration information, i.e. where to find a suitable MySQL instance.

Notice how this model does not impose any particular way of modeling the relations between packages and services: instead of using a single resource with an installed and a running state, we can also model services and packages as different resources, and relate them through dependencies.

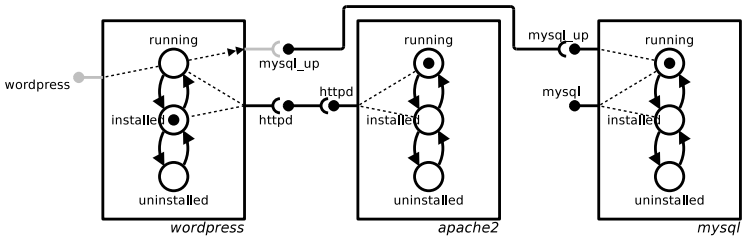


Fig. 3. A graphical description of the basic model of services and packages

Use case 3 — Redundancy, capacity planning, and conflicts. Services often need to be deployed on different machines to reduce the risk of failure or due to the limitations on the load they can bear. For example, system administrators might want to indicate that a MySQL instance can only support a certain number of WordPress instances. Symmetrically, a WordPress hosting service may want to expose a reverse web proxy / load balancer to the public and require to have a minimum number of *distinct* instances of WordPress available as its back-ends.

To model this kind of situations, we allow capacity information to be added on provides and requires of each resource in Aeolus: a number n on a provide port indicates that it can fulfill no more than n requirements, while a number n on a require port means that it needs to be connected to at least n provides from n *different* components. This information may then be used by a planner to find an optimal replication of the resources to satisfy a user requirement.

As an example, Fig. 4 shows the modeling of a WordPress hosting scenario where we want to offer high availability hosting by putting the Varnish reverse proxy / load balancer in front of several WordPress instances, all connected to a shared replicated MySQL database.[3] 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.

As a particular case, a 0 constraint on a require means that no provide with the same name can be active at the same time; this can be effectively used to model conflicts between components. For instance, we can use this to model the conflict between the `apache2` and `apache2.2-common` packages that has been omitted in Fig. 2.

Use case 4 — Creating and destroying resources. Use cases like WordPress hosting are commonplace in the cloud, to the point that they are often used to showcase the capabilities of state of the art cloud deployment technologies. The features of the model presented up to here are already expressive enough to encode these *static* deployment scenarii. If one takes Juju's (rather limiting) assumption that each service is hosted on a separate *machine*, Fig. 4 may then be the representation of the current set of virtual machines (VMs) that we have rented from a public cloud such as Amazon EC2.

To model faithfully deployment runs on the cloud, where an arbitrary number of instances of virtual machine images can be allocated and deallocated on the fly, we also

[3] All WordPress instances run within separate Apache-s, which have been omitted for simplicity.

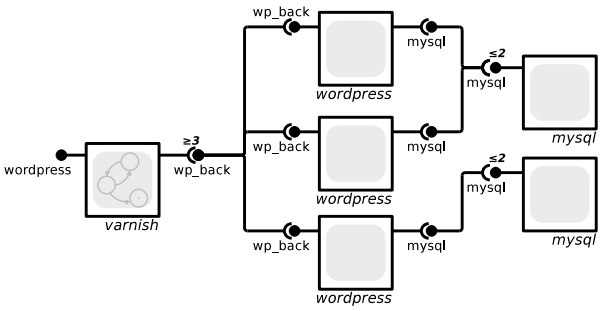


Fig. 4. A graphical description of the model with redundancy and capacity constraints (internal sate machines and activation arcs omitted for simplicity)

allow in our model creation and destruction of all kinds of resources, provided they belong to some existing resource type. This allows to compute reconfiguration plans that create new resources to avoid violating capacity constraints. For instance, in the configuration of Fig. 4, to respond to an increase in traffic load one will need to spawn 2 new WordPress instances, which in turn will require to create new MySQL instances, as the available MySQL-s are not enough to handle the load increase.

3 The Aeolus flat model

We now formalize the *Aeolus flat model*, that contains all the features elicited from the use cases of the previous section. It is "flat" in the sense that all components live in a single "global" context, are mutually visible, and can connect to each other as long as their ports are compatible.

Notation. We consider the following disjoint sets: $\mathscr{I}$ for interfaces and $\mathscr{Z}$ for resources. We use $\mathbb{N}$ to denote strictly positive natural numbers, $\mathbb{N}_\infty$ for $\mathbb{N}$ plus infinity, and $\mathbb{N}_0$ for $\mathbb{N}$ plus 0.

We model components as finite state automata indicating the current state and possible transitions. When a component changes its state, it can also change the ports that it requires from and provides to other components.

Definition 1 (Resource type). *The set $\mathscr{T}_{flat}$ of* resource types *of the Aeolus flat model, ranged over by $\mathscr{T}_1, \mathscr{T}_2, \ldots$ 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*;*
- *$P = \langle \mathbf{P}, \mathbf{R}\rangle$, with $\mathbf{P}, \mathbf{R} \subseteq \mathscr{I}$, is a pair composed of the set of* provide *and the set of* require *ports, respectively;*
- *D is a function from Q to 3-ple in $(\mathbf{P} \mapsto \mathbb{N}_\infty) \times (\mathbf{R} \mapsto \mathbb{N}_0) \times (\mathbf{R} \mapsto \mathbb{N}_0)$.*

Given a state $q \in Q$, the three partial *functions in $D(q)$ indicates respectively the provide, weak require, and strong require ports that q activates. The functions associate to the active ports a numerical constraint indicating:*

- *for provide ports, the maximum number of bindings the port can satisfy,*

– *for require ports, the minimum number of required bindings to distinct resources,*
  - *if the number is 0, that indicates a conflict, meaning that there should be no other active port with the same name.*

We assume as default constraints ∞ 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 has no strong demands (i.e. the third function of $D(q_0)$ is empty).

We now define configurations that describe systems composed by components and their bindings. A configuration, ranged over by $\mathscr{C}_1, \mathscr{C}_2, \ldots$, is given by a set of resource types, a set of deployed resources in some state, and a set of bindings. Formally:

Definition 2 (Configuration). *A* configuration *$\mathscr{C}$ is a 4-ple $\langle U, Z, S, B\rangle$ where:*

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

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

We are now ready to formalize the notion of configuration correctness. We consider two distinct notions of correctness: *weak* and *strong*. According to the former, only weak requirements are considered, while the latter also considers strong ones. Intuitively, weak correctness can be temporarily violated during the deployment of a new component configuration, but needs to be fulfilled at the end; strong correctness, on the other hand, shall never be violated.

Definition 3 (Correctness). *Let us consider the configuration $\mathscr{C} = \langle U, Z, S, B\rangle$.*

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

Similarly for provides, we write $\mathscr{C} \models_{prov} (z, p, n)$ to indicate that the provide port of resource z, with interface p, and associated number n is not bound to more than

n active ports. Formally, there exist no m distinct resources $z_1, \ldots, z_m \in Z \setminus \{z\}$*, with* $m > n$*, such that for every* $1 \leq i \leq m$ *we have that* $\langle p, z_i, z\rangle \in B$*,* $S(z_i) = \langle \mathscr{T}^i, q^i\rangle$ *and* p *is in the domain of* $\mathscr{T}^i.\mathbf{R}_w\texttt{map}(q^i)$ *or* $\mathscr{T}^i.\mathbf{R}_s\texttt{map}(q^i)$*.*

The configuration $\mathscr{C}$ *is* correct *if for each resource* z *in* Z*, given* $S(z) = \langle \mathscr{T}, q\rangle$ *with* $\mathscr{T} = \langle Q, q_0, T, P, D\rangle$ *and* $D(q) = \langle \mathscr{P}, \mathscr{R}_w, \mathscr{R}_s\rangle$*, we have that* $(p \mapsto n_p) \in \mathscr{P}$ *implies* $\mathscr{C} \models_{prov} (z, p, n_p)$*, and* $(r \mapsto n_r) \in \mathscr{R}_w$ *implies* $\mathscr{C} \models_{req} (z, r, n_r)$*, and* $(r \mapsto n'_r) \in \mathscr{R}_s$ *implies* $\mathscr{C} \models_{req} (z, r, n'_r)$*.*

Analogously we say that it is strong correct *if only the strong requirements are considered: namely, we require* $(p \mapsto n_p) \in \mathscr{P}$ *implies* $\mathscr{C} \models_{prov} (z, p, n_p)$ *and* $(r \mapsto n_r) \in \mathscr{R}_s$ *implies* $\mathscr{C} \models_{req} (z, r, n_r)$*.*

As our main interest is planning, we now formalize how configurations evolve from one state to another, by the means of atomic actions.

Definition 4 (Actions). *The set* $\mathscr{A}$ *contains the following actions:*

- $stateChange(z, q_1, q_2)$ *where* $z \in \mathscr{Z}$*;*
- $bind(r, z_1, z_2)$ *where* $z_1, z_2 \in \mathscr{Z}$ *and* $r \in \mathscr{I}$*;*
- $unbind(r, z_1, z_2)$ *where* $z_1, z_2 \in \mathscr{Z}$ *and* $r \in \mathscr{I}$*;*
- $newRsrc(z : \mathscr{T})$ *where* $z \in \mathscr{Z}$ *and* $\mathscr{T}$ *is a*
- $delRsrc(z)$ *where* $z \in \mathscr{Z}$*.*

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

Definition 5 (Reconfigurations). *Reconfigurations are denoted by transitions* $\mathscr{C} \xrightarrow{\alpha} \mathscr{C}'$ *meaning that the execution of* $\alpha \in \mathscr{A}$ *on the configuration* $\mathscr{C}$ *produces a new configuration* $\mathscr{C}'$*. The transitions from a configuration* $\mathscr{C} = \langle U, Z, S, B\rangle$ *are defined as follows:*

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

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

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

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

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

Notice that in the definition of the transitions there is no requirement on the reached configuration: the correctness of these configurations will be considered at the level of a deployment run.

Also, we observe that there are configurations that cannot be reached through sequences of the actions we have introduced. In Fig. 5, for instance, there is no way for

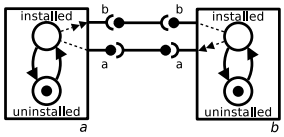


Fig. 5. On the need of a *multiple state change* action: how to install a and b?

package a and b to reach the installed state, as each package require the other to be installed *first*. In practice, when confronted with such situations—that can be found for example in FOSS distributions in the presence of *Pre-Depend* loops—current tools either perform all the state changes atomically, or abort deployment.

We want our planners to be able to propose reconfigurations containing such atomic transitions, as long as that is the only way to reach a requested configuration. To this end, we introduce the notion of *multiple state change*.

Definition 6 (Multiple state change).
A multiple state change $\mathscr{M} = \{stateChange(z^1, q_1^1, q_2^1), \cdots, stateChange(z^l, q_1^l, q_2^l)\}$ *is a set of state change actions on different resources (i.e. $z^i \neq z^j$ for every $1 \leq i < j \leq l$). We use $\langle U, Z, S, B\rangle \xrightarrow{\mathscr{M}} \langle U, Z, S', B\rangle$ to denote the effect of the simultaneous execution of the state changes in $\mathscr{M}$: formally, $\langle U, Z, S, B\rangle \xrightarrow{stateChange(z^1, q_1^1, q_2^1)} \ldots \xrightarrow{stateChange(z^l, q_1^l, q_2^l)} \langle U, Z, S', B\rangle$.*

Notice that the order of execution of the state change actions does not matter as all the actions are executed on different resources.

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 strong 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 $\alpha_1 \ldots \alpha_m$ of actions and multiple state changes such that there exist $\mathscr{C}_i$ such that $\mathscr{C} = \mathscr{C}_0$, $\mathscr{C}_{j-1} \xrightarrow{\alpha_j} \mathscr{C}_j$ for every $j \in \{1, \ldots, m\}$, and the following conditions hold:*

configuration correctness *$\mathscr{C}_0$ and $\mathscr{C}_m$ are correct while, for every $i \in \{1, \ldots, m-1\}$, $\mathscr{C}_i$ is strong correct;*

multi state change minimality *if α_j is a multiple state change then there exists no proper subset $\mathscr{M} \subset \alpha_j$, or state change action $\alpha \in \alpha_j$, and correct configuration $\mathscr{C}'$ such that $\mathscr{C}_{j-1} \xrightarrow{\mathscr{M}} \mathscr{C}'$, or $\mathscr{C}_{j-1} \xrightarrow{\alpha} \mathscr{C}'$.*

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

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

Notice that the restriction in this decision problem to one resource 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 weak requirements on all such provides.

4 Achievability

In this section, we establish our main results concerning the difficulty of the achievability problem. The results change significantly depending on the restrictions imposed on the numerical constraints that are allowed as co-domains of the three $D(q)$ partial functions. We consider here two extreme cases, which are detailed in the table below:

model	*co-domain*(.$\mathbf{P}$map())	*co-domain*(.$\mathbf{R}_w$map())	*co-domain*(.$\mathbf{R}_s$map())
Aeolus$^-$	$\{\infty\}$	$\{1\}$	$\{1\}$
Aeolus flat	$\mathbb{N}_\infty$	$\mathbb{N}_0$	$\mathbb{N}_0$

Aeolus flat is the same model of Def. 1, while Aeolus$^-$ is a restriction of it where only the default numerical constraints can be used: provide ports can always serve an unlimited amount of bindings, and require ports cannot conflict with other active ports, nor require a minimum number of bindings strictly higher than 1. In the following we will show that achievability is decidable in Aeolus$^-$, but undecidable in Aeolus flat.

Achievability is decidable in Aeolus$^-$. We start by presenting a decision algorithm for the achievability problem in Aeolus$^-$. The idea is to perform an abstract forward exploration of all reachable configurations. Before presenting the algorithm, we list the properties of Aeolus$^-$ we exploit:

- as in Aeolus$^-$ the value 0 on require ports is forbidden, the addition to a configuration of new components cannot forbid the execution of formerly possible actions;
- as in Aeolus$^-$ provide ports have capacity ∞ and require ports have numerical constraint 1, the correctness of a configuration can be checked simply by verifying that the set of active require ports is a subset of the set of active provide ports.

In the light of the second observation, and knowing that the sets of active require and provide ports are functions of the internal state of the components, we abstractly represent configurations simply as sets of pairs $\langle \mathscr{T}, q \rangle$ indicating the type and the state of the components in the configuration. This way, symbolic configurations abstract away from the exact number of instances of each kind of component, and from their current bindings.

We consider symbolic runs representing the evolutions of abstract configurations. Moreover, thanks to the first observation, we can restrict ourselves to consider only evolutions where the set of available pairs $\langle \mathscr{T}, q \rangle$ does not decrease. Namely, we perform a symbolic forward exploration starting from an abstract configuration containing all the pairs $\langle \mathscr{T}', \mathscr{T}'.\texttt{init} \rangle$ representing components in their initial state. Then we extend the abstract configuration by adding step-by-step new pairs $\langle \mathscr{T}', q' \rangle$.

Algorithm 1 checks achievability by relying on two auxiliary data structures: *absConf* is the set of pairs $\langle \mathscr{T}', q' \rangle$ indicating the type and state of the components in the current

Algorithm 1 Checking achievability in the Aeolus$^-$ model

function ACHIEVABILITY(universe of resources U, resource type $\mathscr{T}$, state q)
  $absConf := \{\langle \mathscr{T}', \mathscr{T}'.\texttt{init}\rangle \mid \mathscr{T}' \in U\}$
  $provPort := \bigcup_{\langle \mathscr{T}', q'\rangle \in absConf} \{dom(\mathscr{T}'.\mathbf{P}\texttt{map}(q'))\}$
  repeat
    $new := \{\langle \mathscr{T}', q'\rangle \mid \langle \mathscr{T}', q''\rangle \in absConf, (q'', q') \in \mathscr{T}'.\texttt{trans}\} \setminus absConf$
    $newPort := \bigoplus_{\langle \mathscr{T}', q'\rangle \in new} \{\!\{dom(\mathscr{T}'.\mathbf{P}\texttt{map}(q'))\}\!\}$
    while $\exists \langle \mathscr{T}', q'\rangle \in new$ s.t. $dom(\mathscr{T}'.\mathbf{R}_s\texttt{map}(q')) \not\subseteq provPort \cup newPort$ **do**
      $new := new \setminus \{\langle \mathscr{T}', q'\rangle\}$
      $newPort := newPort \ominus \{\!\{dom(\mathscr{T}'.\mathbf{P}\texttt{map}(q'))\}\!\}$
    end while
    $absConf := absConf \cup new$
    $provPort := provPort \cup newPort$
  until $new = \emptyset$
  if $\langle \mathscr{T}, q\rangle \in absConf$ and $dom(\mathscr{T}.\mathbf{R}_w\texttt{map}(q)) \subseteq provPort$ **then return true**
  else return false
  end if
end function

abstract configuration, and *provPort* is the set of provide ports active in such a configuration. The algorithm incrementally extends *absConf* until it is no longer possible to add new pairs.

At each iteration, the potential new pairs are initially computed by checking the automata transitions, and stored in the set *new*. Not all those states could be actually reached as one needs to check whether their strongly require ports are included in the available provide ports *provPort* or in the ports opened by the new states. This is done by a one-by-one elimination of pairs $\langle \mathscr{T}', q'\rangle$ from *new* when their strong requirements are unsatisfiable. During elimination, we use *newPort*, a *multiset* of the provide ports which are activated by the component states currently in *new*. We use double curly braces for multisets, and $\oplus$ and $\ominus$ for multiset union and difference.

When the final sets *absConf* and *provPort* are computed, achievability for the resource type $\mathscr{T}$ and state q can be simply checked by verifying the presence of $\langle \mathscr{T}, q\rangle$ in *absConf*, and by controlling whether its (weak) requirements are satisfied by the active provide ports *provPort*. Strong requirements are satisfied by construction.

We are now ready to prove our decidability result for the Aeolus$^-$ model.

Theorem 1. *Let U be a set of resource types of the Aeolus$^-$ model. Given the resource type $\mathscr{T}$ and the state q, the achievability problem for U, $\mathscr{T}$, and q can be checked in polynomial time (with respect to the size of the descriptions of the resources in U).*

Proof. The symbolic representation of the initial configuration $\langle U, \emptyset, \emptyset, \emptyset\rangle$ is included in the initial set *absConf*. It is easy to see that given the transition $\mathscr{C} \xrightarrow{\alpha} \mathscr{C}'$ of a deployment run, if the symbolic representation of $\mathscr{C}$ is included in *absConf* at the beginning of an iteration of the **repeat**, then the symbolic representation of $\mathscr{C}'$ will be surely included in *absConf* at the end of such iteration. Therefore, if there exists a deployment run able to achieve a component of type $\mathscr{T}$ in the state q, then the Algorithm 1 will detect achievability. This proves that the algorithm is complete.

The soundness of the algorithm follows from the following argument. The symbolic forward exploration performed by the algorithm corresponds to an actual deployment run that initially creates sufficiently many components in order to guarantee that all the state changes considered by the symbolic exploration can be actually executed, and every time an action changes the state of one component of type $\mathscr{T}'$ from q'' to q', there exists at least another component in the concrete system of type $\mathscr{T}'$ which remains in state q''.

The polynomial complexity of the algorithm follows from the fact that both the **repeat** and the **while** cycles perform a number of iterations smaller than the number of different pairs $\langle \mathscr{T}', q' \rangle$ in the universe of resource types U. □

Achievability is undecidable in Aeolus flat. We now show that the decision procedure for achievability of the previous section cannot be extended to deal with the Aeolus flat model. In fact, for this last model achievability turns out to be undecidable. The proof is by reduction from the reachability problem in 2 Counter Machines (2CMs) [11], a well-known Turing-complete computational model. A 2CM is a machine with *two registers* R_1 and R_2 holding arbitrary large natural numbers and a *program* P consisting of a finite sequence of numbered instructions of the following type:

- $j : \mathsf{Inc}(R_i)$: increments R_i and goes to the instruction $j+1$;
- $j : \mathsf{DecJump}(R_i, l)$: if the content of R_i is not zero, then decreases it by 1 and goes to the instruction $j+1$, otherwise jumps to the instruction l.

A state of the machine is given by a tuple (i, v_1, v_2) where i indicates the next instruction to execute (the program counter) and v_1 and v_2 are the contents of the two registers. It is not restrictive to assume that the registers are initially set to zero.

We model a 2CM as follows. We use a component to simulate the execution of the program instructions. The contents v_i of the register R_i is modeled by v_i components in a particular state q_i. Increment instructions add one component in this state q_i, while decrement instructions move one component in state q_i to a different state. The state q_i activates a provide port one_i, so the simulation of a jump has simply to check the absence in the environment of active one_i ports.

The resource types of the components that we use to model 2CMs are depicted in Fig. 6. Namely, we consider four resource types: $\mathscr{T}_P$ to simulate the execution of the program instructions, $\mathscr{T}_{R_1}$ and $\mathscr{T}_{R_2}$ for the two registers and $\mathscr{T}_B$ used to guarantee that components of type $\mathscr{T}_{R_i}$ involved in the simulation cannot be deleted. In $\mathscr{T}_P$ we assume one state q_j for each instruction j. If the j-th instruction is $j : \mathsf{Inc}(R_i)$, a protocol with three intermediary states is executed. The first one will activate a port on_i allowing a resource of type $\mathscr{T}_{R_i}$ to start a complementary protocol. The second state of the protocol activates a strong requirement on the provide port inc_i while the last state activates a conflict on the same port inc_i. The complementary protocol of the resource type $\mathscr{T}_{R_i}$ includes three states as well. The first one activates a strong requirement on the port on_i: in this way, the protocol can start only if the complementary protocol already started. The second state of the protocol activates the port inc_i in order to allow the protocol to progress. Finally, the protocol completes by entering the q_i state. Note that the first state of the protocol opens a provide port a, and the first two states activates a strong requirement on the absence in the environment of such an active port. This guarantees

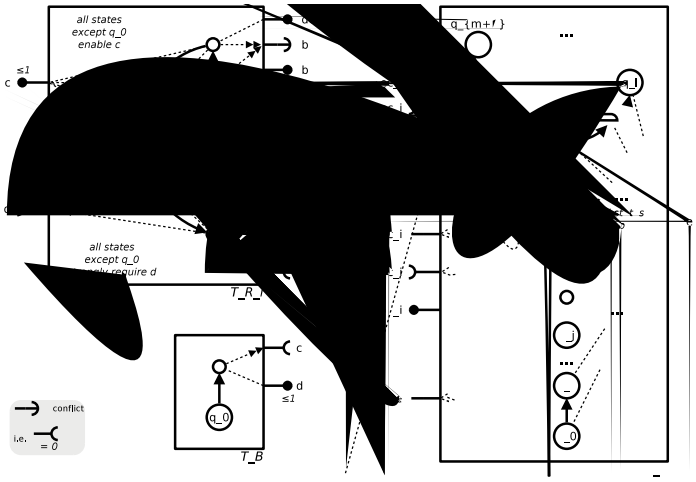


Fig. 6. Modeling 2 counter machines (2CMs) in the Aeolus flat model (sketch)

that exactly one resource of type $\mathscr{T}_{R_i}$ will execute the protocol, in other terms, the register R_i is incremented exactly by 1.

Fig. 6 also depicts the modeling of an instruction $m : \mathsf{DecJump}(R_i, l)$. The decrement branch executes a protocol similar to the previous one, whose effect here is to decrement R_i by 1. The jump branch simply checks the absence of components of type $\mathscr{T}_{R_i}$ in state q_i by activating a strong requirement on the absence of an active port one_i (note that such a port is indeed activated by components of type $\mathscr{T}_{R_i}$ in state q_i).

In our component model, when a resource z is not used to satisfy strong requirements, it could be removed by executing the $delRsrc(z)$ action. The cancellation of a components of type $\mathscr{T}_{R_i}$ could then erroneously change register contents during simulation. To avoid that we force the connection of each resource of type $\mathscr{T}_{R_i}$ with a corresponding instance of a component of type $\mathscr{T}_B$. These types of resources reciprocally "strongly" connect through the ports c and d as soon as they move from their initial state q_0. Such connections remain active during the entire simulation, ensuring components will not be deleted by mistake. Notice that it is necessary to add the capacity constraint 1 to the provide ports c and d, in order to have an exact one-to-one correspondence between the components of type $\mathscr{T}_{R_i}$ and those of type $\mathscr{T}_B$.

As a final remark, notice that the first state q_1 of the resource type $\mathscr{T}_P$ has a strong requirement on the absence in the environment of an active provide port e, port which is activated by all the states in $\mathscr{T}_P$. This guarantees that at most one component of type $\mathscr{T}_P$ will simulate program instructions. Moreover, we also have to avoid that such component is removed by a *delRsrc* action during the simulation: this can be guaranteed by using the same pairing technique with a component of type $\mathscr{T}_B$ described above. It is sufficient to impose that all the states of $\mathscr{T}_P$, but q_0, activate a provide port on c with numerical constraint 1, and a strongly require port on d. For simplicity, this part of the specification of $\mathscr{T}_P$ is not shown in Fig. 6.

We are now ready to formally state our undecidability result.

Theorem 2. *The achievability problem is undecidable in the Aeolus flat model.*

Proof. Let M be a 2CM and let $U = \{\mathscr{T}_P, \mathscr{T}_{R_1}, \mathscr{T}_{R_2}, \mathscr{T}_B\}$ be the set of the corresponding resource types defined as in Fig. 6. In the light of the discussion above, we have that achievability is satisfied for the universe U, the resource type $\mathscr{T}_P$ and the state q_j if and only if the j-th instruction is reachable in M. The undecidability of achievability thus follows from the undecidability of reachability for 2CMs. □

5 Related work

To the best of our knowledge this is the first paper that formally addresses the problem of component deployment in the cloud. In this section we compare the approach we have adopted to related formal models considered in slightly different contexts.

Automata have been adopted long ago in the context of component-oriented development frameworks. One of the most influential model are *interface automata* [3], where automata are used to represent the component behavior in terms of input, output, and internal actions. Interface automata support automatic compatibility check and refinement verification: a component refines another if its interface has weaker input assumptions and stronger output guarantees. Differently from that approach, we are not interested in component compatibility or refinement, and we do not require complementary behavior of components: we simply check in the current configuration whether all required functionalities are provided by currently deployed components. The automata in Aeolus do not represent the internal behavior of components, but the effect on the component of an external deployment or reconfiguration actions.

Aeolus reconfiguration actions show interesting similarities with transitions in Petri nets [13], a very popular model born from the attempt to extend automata with concurrency. At first sight, one might encode our model in Petri net, representing our component states as places, each deployed component as a token in the corresponding place, and reconfiguration actions as transitions that cancel and produce tokens. Achievability in Aeolus would then correspond to *coverability* in Petri nets. But there are several important differences. Multiple state change actions can atomically change the state of an unbounded number of components, while in Petri net each transition consumes a predefined number of tokens. More importantly, we have proved that achievability can be solved in polynomial time for the Aeolus$^-$ fragment and that it is undecidable for the Aeolus flat model, while in Petri nets coverability is an ExpSpace problem [14].

Several process calculi extend/modify the π-calculus [10] in order to deal with software components. The Piccola calculus [2] extends the asynchronous π-calculus [10] with *forms*, first-class extensible namespaces, useful to model component interfaces and bindings. Calculi like KELL [15] and HOMER [5] extends a core π-calculus with hierarchical locations, local actions, higher-order communication, programmable membranes, and dynamic binding. More recently, MECo [12] has extended this approach by proposing also explicit component interfaces and channels to realize tunneling effects traversing the hierarchical location boundaries. On the one hand, all these proposals differ from Aeolus model because they focus on the modeling of component interactions and communication, while we focus on their interdependencies during system deployment and reconfiguration. On the other hand, we plan to take inspiration from these calculi in order to extend our model with boundaries and administrative domains.

6 Conclusions and future work

We have presented *Aeolus flat*, a component model expressive enough to capture most common deployment scenarii for distributed software applications in the cloud. We have shown that it is possible to generate a deployment plan in polynomial time for the fragment *Aeolus*$^-$ of the model, corresponding to the industrial tools currently in use, while it is not possible to generate a deployment plan for *Aeolus flat*, that captures more faithfully the constraints imposed by real world applications.

Several interesting models between *Aeolus*$^-$ and *Aeolus flat* can now be considered, to reconcile expressivity and decidability: one can impose in *Aeolus flat* an upper limit on the number of resources that can be allocated during a deployment run; or one can extend *Aeolus*$^-$ with only conflict constraints.

The *Aeolus flat* model can be extended to a hierarchical component model to take into account administrative domains and components that are built by grouping together other components. We will also experiment with different planning systems to explore the issues related to plan generation, and use the results as additional guidance in the search for the best compromise between expressivity and decidability.

References

1. Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S.: MPM: a modular package manager. In: CBSE'11: 14th symposium on component based software eng. pp. 179–188. ACM (2011)
2. Achermann, F., Nierstrasz, O.: A calculus for reasoning about software composition. Theor. Comput. Sci. 331(2-3), 367–396 (2005)
3. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC / SIGSOFT FSE (2001)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The fractal component model and its support in java. Softw., Pract. Exper. 36(11-12), 1257–1284 (2006)
5. Bundgaard, M., Hildebrandt, T.T., Godskesen, J.C.: A cps encoding of name-passing in higher-order mobile embedded resources. Theor. Comput. Sci. 356(3), 422–439 (2006)
6. Cloud Foundry, deploy & scale your applications in seconds. http://www.cloudfoundry.com/, retrieved April 2012
7. Di Cosmo, R., Trezentos, P., Zacchiroli, S.: Package upgrades in FOSS distributions: Details and challenges. In: HotSWup'08 (2008)
8. Juju, devops distilled. https://juju.ubuntu.com/, retrieved April 2012
9. Lau, K.K., Wang, Z.: Software component models. IEEE Trans. Software Eng. 33(10), 709–724 (2007)
10. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, i/ii. Inf. Comput. 100(1), 1–77 (1992)
11. Minsky, M.: Computation: finite and infinite machines. Prentice Hall (1967)
12. Montesi, F., Sangiorgi, D.: A model of evolvable components. In: TGC. Lecture Notes in Computer Science, vol. 6084, pp. 153–171. Springer (2010)
13. Petri, C.A.: Kommunikation mit Automaten, PhD thesis. Institut für Instrumentelle Mathematik, Bonn, Germany (1962)
14. Rackoff, C.: The covering and boundedness problems for vector addition systems. Theoret. Comp. Sci. 6, 223–231 (1978)
15. Schmitt, A., Stefani, J.B.: The kell calculus: A family of higher-order distributed process calculi. In: Global Computing. LNCS, vol. 3267, pp. 146–178. Springer (2004)
16. Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., Stefani, J.B.: Reconfigurable SCA applications with the FraSCAti platform. In: IEEE SCC. pp. 268–275. IEEE (2009)