Information and Computation 239 (2014) 100–121

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Aeolus: A component model for the cloud

Roberto Di Cosmo [a,b], Jacopo Mauro [b,c], Stefano Zacchiroli [a], Gianluigi Zavattaro [b,c,*]

[a] Univ. Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, F-75205 Paris, France
[b] INRIA—Institut National de Recherche en Informatique et en Automatique, France
[c] Department of Computer Science and Engineering, University of Bologna, Via Mura Anteo Zamboni 7, 40127 Bologna, Italy

ARTICLE INFO

Article history:
Received 27 June 2014
Received in revised form 16 September 2014
Available online 6 November 2014

Keywords:
Software component
Model
Cloud computing
Distributed systems

ABSTRACT

We introduce the Aeolus component model, which is specifically designed to capture realistic scenarios arising when configuring and deploying distributed applications in the so-called *cloud* environments, where interconnected components can be deployed on clusters of heterogeneous virtual machines, which can be in turn created, destroyed, and connected on-the-fly.

The full Aeolus model is able to describe several component characteristics such as dependencies, conflicts, non-functional requirements (replication requests and load limits), as well as the fact that component interfaces to the world might vary depending on the internal component state.

When the number of components needed to build an application grows, it becomes important to be able to *automate* activities such as deployment and reconfiguration. This corresponds, at the level of the model, to the ability to decide whether a desired target system configuration is reachable, which we call the *achievability* problem, and producing a path to reach it.

In this work we show that the achievability problem is undecidable for the full Aeolus model, a strong limiting result for automated configuration in the cloud. We also show that the problem becomes decidable, but Ackermann-hard, as soon as one drops non-functional requirements. Finally, we provide a polynomial time algorithm for the further restriction of the model where support for inter-component conflicts is also removed.

1. Introduction

The expression "*cloud computing*" is broadly used to refer to the possibility of building sophisticated distributed software applications that can be run, on-demand, on virtualized hardware infrastructure at a fraction of the cost which was necessary just a few years ago. Reaping all the benefits of cloud computing is not an easy task: even when the infrastructure costs fall dramatically, the complexity of designing and maintaining distributed scalable software systems is a serious challenge.

Attempts are being made both in industry and in the research world to model and tame such complexity. On the industry side, a wealth of initiatives offer different kinds of solutions for isolated aspects of the problem. Tools like Puppet [1] or Chef [2] allow to automate the configuration of software components, based on a set of descriptions stored in a cen-

* Corresponding author at: Department of Computer Science and Engineering, University of Bologna, Via Mura Anteo Zamboni 7, 40127 Bologna, Italy.
E-mail addresses: roberto@dicosmo.org (R. Di Cosmo), jmauro@cs.unibo.it (J. Mauro), zack@pps.univ-paris-diderot.fr (S. Zacchiroli), zavattar@cs.unibo.it (G. Zavattaro).

http://dx.doi.org/10.1016/j.ic.2014.11.002
0890-5401/

tral server. CloudFoundry [3] allows to select, connect, and push to a cloud some predefined services (databases, message buses, proxies, ...), that can be used as building blocks for writing applications using one of the supported frameworks. Finally, Juju [4] tries to extend the basic concepts of package managers—used by software distributions to automate software upgrades.

On the academic side, several teams are working, with different approaches, on the problems posed by the complexity of designing cloud applications. The Fractal component model [5], which itself pre-dates the popularization of the "cloud computing" expression, 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, complex software systems. Building on Fractal, FraSCAti [6] provides a middleware that can be used to deploy applications in the cloud. ConfSolve [7] on the other hand aims at helping the application designer with some of the decisions to be made, and more specifically to optimally allocate virtual machines to concrete servers.

In all the above mentioned approaches, the goal is to allow the user (i.e., the application designer) to assemble a working system out of components that have been specifically designed or adapted to work together. The actual component selection (which web server should I use? which SQL database? which load balancer?) and interconnection (which front-end should I connect to which back-end, in order to avoid bottlenecks?) 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 left for the user to write.

We believe that to make further progress in taming the complexity of sophisticated cloud applications, two major concerns must be taken into account.

Expressivity We need component models that are expressive enough to capture all the component characteristics that are relevant for designing distributed, scalable applications which are typical in the cloud. Some of those characteristics (see Section 2 for a more in-depth discussion) are:

dependencies e.g., which other components should be deployed in order to be able to install, activate, upgrade, etc. a given component?

conflicts e.g., which other components, if any, would inhibit the deployment of a given component?

non-functional requirements e.g., if a component depends on others, how many of those would be needed to guarantee the desired level of fault-tolerance and/or load-balancing? Similarly, if a component offers functionalities to other, how many of them it can reasonably satisfy before needing to be replicated?

statefulness distributed/cloud-components have complex activation protocols, making their contextual requirements (dependencies, conflicts, etc.) vary over time, e.g., it might be enough to install a given component to be able to install another one, but the requirements to activate them might be different

Automation While *expressivity* is certainly important, solving the challenge of designing and maintaining a cloud also requires automation. When the number of components grows, or the need to reconfigure appears more frequently, it is essential to be able to specify at a certain level of abstraction a particular target configuration of the distributed software system we want to realize, and to develop tools that provide a set of possible evolution paths leading from the current system configuration to one that corresponds to such a user request.

Automated approaches have been developed already, but thus far mostly 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 [8]. Similar approaches have been developed in the context of Software Product Lines where a correct instance of a product needs to be composed of a consistent set of features [9].

The goal of this paper is to lay the formal 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.

Contributions We first elicit the expressivity requirements of a component model that is suitable for the cloud from specific use cases presented in Section 2. We then detail a formal component model for the cloud, called *Aeolus*, where components describe resources which provide and require different functionalities, and may be created or destroyed. As a major improvement over state-of-the-art component models, Aeolus components are equipped with *state machines* that *declaratively* describe how required and provided functionalities are enacted. 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 checking the existence of a deployment plan in Aeolus, a property which we call *achievability*. We study achievability in the full Aeolus model, as well as in more limited variants of it that exhibit different decidability and complexity characteristics.

We show that achievability is *undecidable* if one allows to impose capacity constraints—i.e., restrictions on the number of connections between required and provided functionalities—as it happens in the complete version of our model. This

```
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, [...]

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).

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.

If we remove the possibility of constraining the number of provided and required functionalities, we show that achievability becomes *decidable but Ackermann-hard*. Thus even in this simplified model, that we call *Aeolus core*, finding a plan can be extremely costly and infeasible from the computational point of view.

For this reason we consider a further restricted model, called Aeolus$^-$, where we drop the ability of stating capacity constraints on the provided and required functionalities, and declaring conflicts between resources. We prove that in Aeolus$^-$ achievability is *decidable in polynomial time*. This is interesting since Aeolus$^-$ corresponds to what mainstream industry tools can handle at present. Our result explains why it is still possible, in simple cases, to manage such systems manually.

2. A gentle introduction to Aeolus

We introduce the key features of Aeolus by eliciting them, step-by-step, from the analysis of realistic scenarios. 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.

Use case 1—Package installation

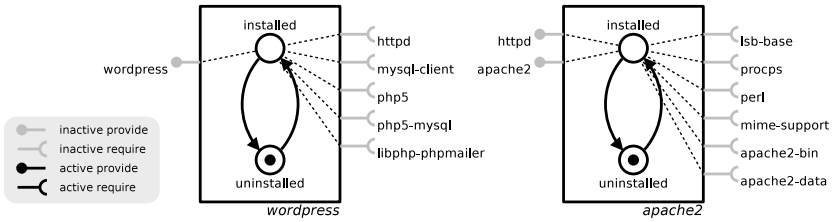
Before considering the services that a machine is offering to others (locally or over the network), we need to model the *software installation* on the machine itself, so we will see how to model the three main components needed by WordPress, as far as their installation is concerned.

Software is often distributed according to the *package* paradigm [12], 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.

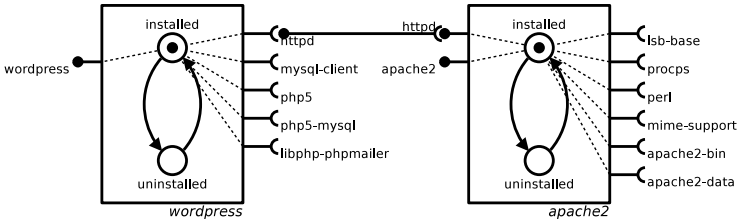
On a given machine, a software package may exists in different states (e.g., installed or uninstalled) and it should go through a complex sequence of states in different phases of unpacking and configuration to get there. In each of its states, similarly to what happens in most software component models [13], a package may have contextual *requirements* and offer some features, that we call *provides*. For instance in Debian, a popular FOSS distribution, there are packages for WordPress, Apache2 and MySQL equipped with metadata (reported in Fig. 1) including a list of requirements (the *Depends* field) and of functionalities that are offered (the *Provides* field).

To model a software package, at this level of abstraction, we may use a simple state machine to capture its life cycle, 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 $\mathbf{R}$ of requirements, a set $\mathbf{P}$ of provides, and a function D that maps states to the requirements and provides that are *active* at that state. We call *component type* any such tuple $\langle Q, q_0, T, \langle \mathbf{P}, \mathbf{R} \rangle, D \rangle$, which will be formalized in Definition 1.

A system *configuration* is then built out of a collection of components that are instances of component types, with its current state, and a set of connections between requirements and provides of the different components. Connections indicate which provide is fulfilling 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 4.



(a) Available components, not installed.



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

Fig. 2. Modelling package installation and dependencies.

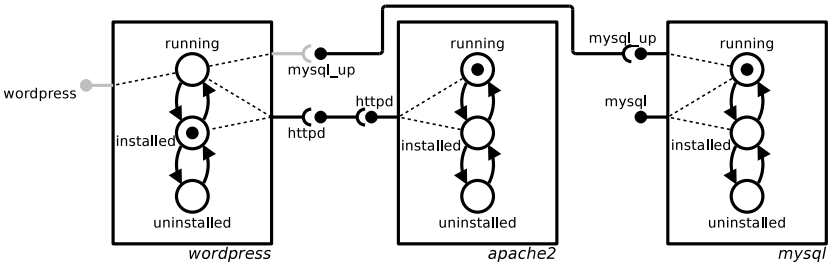


Fig. 3. A graphical description of the basic model of services and packages.

A straightforward graphical notation can capture all these pieces of information together: Fig. 2 presents systems built using the components from Fig. 1 (only modelling the dependency on `httpd` underlined in the metadata, for the sake of conciseness). In Fig. 2a the packages are available but not installed yet. In Fig. 2b 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 together the two *ports* named `httpd`.

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 tune and 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 requirement on `mysql_up` of the `running` state of WordPress 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 modelling the relations between packages and services. Instead of using a single component with an installed and a running state, we can simply model services and packages as different components, and relate them through dependencies.

Use case 3—Redundancy, capacity planning, and conflicts

Services often need to be deployed on different machines to reduce the risk of failure or to increase the load they can withstand by the means of load-balancing. To properly design such scalable architectures system administrators might want, for instance, to indicate that a MySQL instance can only support a certain number of connected WordPress instances.

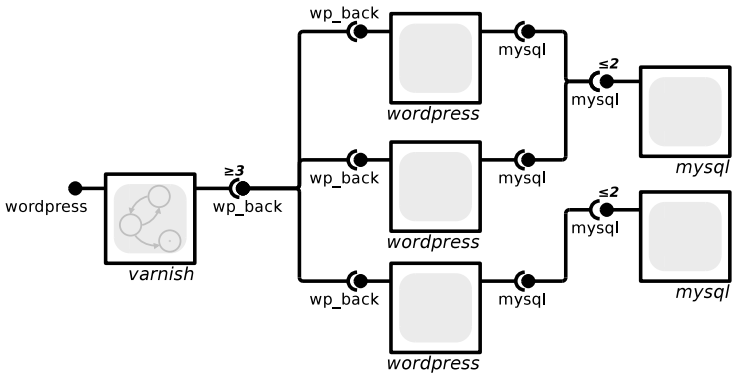


Fig. 4. A graphical description of the model with redundancy and capacity constraints (internal sate machines are omitted for simplicity).

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 component in Aeolus: a number n on a provide port indicates that it can fulfil 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.

As an example, Fig. 4 shows the modelling 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.

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 *global conflicts* between components. For instance, we can use this feature to model the conflict between the `apache2` and `apache2.2-common` packages that had been omitted in Fig. 2.

Use case 4—Creating and destroying components

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 scenarios, where the system architecture does not evolve over time in reaction to load changes.

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 allow in our model creation and destruction of all kinds of components, provided they belong to some existing component type. 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 no longer enough to handle the load increase.

3. The Aeolus model

We now formalize the *Aeolus model*, implementing all the features elicited from the use cases discussed in the previous section.

Notation. We assume given the following disjoint sets: $\mathcal{I}$ for interfaces and $\mathcal{Z}$ for components. We use $\mathbb{N}$ to denote strictly positive natural numbers, $\mathbb{N}_\infty$ for $\mathbb{N} \cup \{\infty\}$, and $\mathbb{N}_0$ for $\mathbb{N} \cup \{0\}$.

We model components as finite state automata indicating all possible component states and state transitions. When a component changes state, the sets of ports it requires from/provide to other components will also change: intuitively, the component interface with the external world varies with its state. A provide port represents the possibility of furnishing a functionality having a given interface. Similarly, a require port represent the need of a functionality with a given interface.

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*;

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

- $P = \langle \mathbf{P}, \mathbf{R} \rangle$, with $\mathbf{P}, \mathbf{R} \subseteq \mathcal{I}$, 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 $(\mathbf{P} \nrightarrow \mathbb{N}_\infty) \times (\mathbf{R} \nrightarrow \mathbb{N}_0)$.

Given a state $q \in Q$, $D(q)$ returns two partial functions $(\mathbf{P} \nrightarrow \mathbb{N}_\infty)$ and $(\mathbf{R} \nrightarrow \mathbb{N}_0)$ that indicate respectively the provide and require ports that q activates. The functions associate to the activate 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* 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 has no demands (i.e., the second function of $D(q_0)$ has an empty domain).

Example 1. Fig. 2a depicts two component types: wordpress and apache2. In particular wordpress is formally defined as the 5-ple $\langle Q, q_0, T, P, D \rangle$ with:

- $Q = \{\text{uninstalled}, \text{installed}\}$,
- $q_0 = \text{uninstalled}$,
- $T = \{(\text{uninstalled} \mapsto \text{installed}), (\text{installed} \mapsto \text{uninstalled})\}$,
- $P = \langle \{\text{wordpress}\}, \{\text{httpd}, \text{mysql-client}, \text{php5}, \text{php5-mysql}, \text{libphp-phpmailer}\} \rangle$,
- $D = \{(\text{uninstalled} \mapsto \langle \emptyset, \emptyset \rangle), (\text{installed} \mapsto \langle \{(\text{wordpress} \mapsto \infty)\}, f \rangle)\}$ where f is a function that associates 1 to all require ports.

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 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 4-ple $\langle U, Z, S, B \rangle$ 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.

Example 2. Fig. 2b depicts a configuration with two components and one binding. Formally, it corresponds to the 4-ple $\langle U, Z, S, B \rangle$ where:

- U is a set of component types including wordpress and apache2,
- $Z = \{z_1, z_2\}$,
- $S = \{(z_1 \mapsto \langle \text{wordpress}, \text{installed} \rangle), (z_2 \mapsto \langle \text{apache2}, \text{installed} \rangle)\}$,
- $B = \langle \text{httpd}, z_1, z_2 \rangle$.

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 $\langle Q, q_0, T, \langle \mathbf{P}, \mathbf{R} \rangle, D \rangle$, we use projections to (recursively) decompose it: `.states`, `.init`, and `.trans` return the first three elements; `.prov`, `.req` return $\mathbf{P}$ and $\mathbf{R}$; $.\mathbf{P}(q)$ and $.\mathbf{R}(q)$ 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, $\mathcal{C}[z].\mathbf{R}(q)$ 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 $\mathcal{T}'.\mathbf{P}(q')$. 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 $1 \le i \le n$ 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 $\mathcal{T}^i.\mathbf{P}(q^i)$.

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 $1 \le i \le m$ 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 $\mathcal{T}^i.\mathbf{R}(q^i)$.

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 $\mathcal{C} \models_{req} (z, r, n_r)$.

Example 3. Figs. 3 and 4 report examples of correct configurations. In Fig. 3 it is easy to see that all active require ports are bound to an active provide port: this condition is enough when the numerical constraints has the default values.

In Fig. 4 there are two kinds of non-default numerical constraints: the constraint 3 on the require port wp_back of the component of type varnish which is satisfied because there are at least three bindings connecting it to three distinct components (we assume that the wp_back provide ports of these three components are active) and the constraint 2 on the provide port mysql of the components of type mysql which are satisfied because those ports are connected to less than two bindings.

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 $z \in \mathcal{Z}$;
- $bind(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$;
- $unbind(r, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$ and $r \in \mathcal{I}$;
- $new(z : \mathcal{T})$ where $z \in \mathcal{Z}$ and $\mathcal{T}$ is a component type;
- $del(z)$ where $z \in \mathcal{Z}$.

The execution of actions can now be formalized using a labelled 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:

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

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

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

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

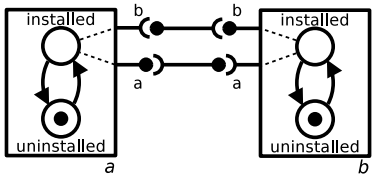


Fig. 5. On the need of a *multiple state change*: how to install a and b?

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

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 deployment runs.

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 package `a` and `b` to reach the installed state, as each package requires 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 loops of *Pre-Depends* that impose an order in the installation of two depending packages—current tools either perform all the state changes atomically, or abort deployment.

We want our planners to be able to propose deployment runs containing such atomic transitions. To this end, we introduce the notion of *multiple state change*:

Definition 7 *(Multiple state change).* A *multiple state change* $\mathcal{M} = \{stateChange(z^1, q_1^1, q_2^1), \cdots, stateChange(z^l, q_1^l, q_2^l)\}$ is a set of state change actions on different components (i.e., $z^i \neq z^j$ for every $1 \leq i < j \leq l$). We use $\langle U, Z, S, B \rangle \xrightarrow{\mathcal{M}} \langle U, Z, S', B \rangle$ to denote the effect of the simultaneous execution of the state changes in $\mathcal{M}$: formally,

$$\langle U, Z, S, B \rangle \xrightarrow{stateChange(z^1, q_1^1, q_2^1)} \cdots \xrightarrow{stateChange(z^l, q_1^l, q_2^l)} \langle U, Z, S', B \rangle$$

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

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 8 *(Deployment run).* A *deployment run* is a sequence $\alpha_1 \ldots \alpha_m$ of actions and multiple state changes such that there exist $\mathcal{C}_i$ such that $\mathcal{C} = \mathcal{C}_0$, $\mathcal{C}_{j-1} \xrightarrow{\alpha_j} \mathcal{C}_j$ for every $j \in \{1, \ldots, m\}$, and the following conditions hold:

configuration correctness for every $i \in \{0, \ldots, m\}$, $\mathcal{C}_i$ is correct;
multi state change minimality if α_j is a multiple state change then there exists no proper subset $\mathcal{M} \subset \alpha_j$, or state change action $\alpha \in \alpha_j$, and correct configuration $\mathcal{C}'$ such that $\mathcal{C}_{j-1} \xrightarrow{\mathcal{M}} \mathcal{C}'$, or $\mathcal{C}_{j-1} \xrightarrow{\alpha} \mathcal{C}'$.

Example 4. Consider the configuration reported in Fig. 3. Starting from an empty configuration. Such configuration can be reached upon execution of the following deployment run:

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

This sequence of actions is a deployment run because it guarantees the correctness of all the traversed configurations. Notice that this sequence of actions continues to be a deployment run even if $stateChange(z_1, \mathsf{uninstalled}, \mathsf{installed})$ is postponed.

On the contrary, it is no longer a deployment run if such action is anticipated because the requirement on the httpd port is not yet fulfilled. It is no longer a deployment run even if such action is joined with other state changes to form a multiple state change action (like, e.g., $\{stateChange(z_1, \mathsf{uninstalled}, \mathsf{installed}), stateChange(z_2, \mathsf{installed}, \mathsf{running})\}$) because this violates minimality.

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 9 *(Achievability problem).* The *achievability problem* has as input a universe U of component types, a component type $\mathcal{T}$, 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**.

Example 5. Consider the achievability problem for the universe of component types wordpress, apache2, and mysql in Fig. 3, and the target expressed by wordpress in its running state. In this case the problem returns **true** because there exists, for instance, the deployment run obtained by adding $stateChange(z_1, \mathsf{installed}, \mathsf{running})$ at the end of the sequence of actions in Example 4.

Notice that the restriction in this decision problem to one 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.

4. Decidability and complexity of achievability

In this section, we establish our main results concerning the decidability and complexity 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 two $D(q)$ partial functions. We consider here three cases, which are detailed in the table below:

model	*co-domain*(.**P**())	*co-domain*(.**R**())
Aeolus$^-$	$\{\infty\}$	$\{1\}$
Aeolus core	$\{\infty\}$	$\{1, 0\}$
Aeolus	$\mathbb{N}_\infty$	$\mathbb{N}_0$

Aeolus (last row) is the same model of Definition 1, while Aeolus$^-$ is a restriction of it where only the default numerical constraints can be used: provide ports 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. Aeolus core, instead, is similar to Aeolus$^-$ but with the added possibility of expressing conflicts.

In the following we will show that: achievability is undecidable in Aeolus; it is decidable, but not primitive recursive (i.e., Ackermann-hard) in Aeolus core; it is decidable and polynomial in Aeolus$^-$.

4.1. Achievability is undecidable in Aeolus

The proof that achievability is undecidable is by reduction from the reachability problem in 2 Counter Machines (2CMs) [14], 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 two following types:

- $j : \mathsf{Inc}(R_i)$: 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 l instruction.

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 values contained in the two registers, respectively.

Notation. In the following we use the notation $(i, v_1, v_2) \to (i', v_1', v_2')$ to say that the state of the machine changes from (i, v_1, v_2) to (i', v_1', v_2') as effect of the execution of the i-th instruction.

It is not restrictive to assume that the initial configuration of the machine is $(1, 0, 0)$. In 2CMs, the problem of checking whether a given l-th instruction is reachable from the initial configuration is undecidable.

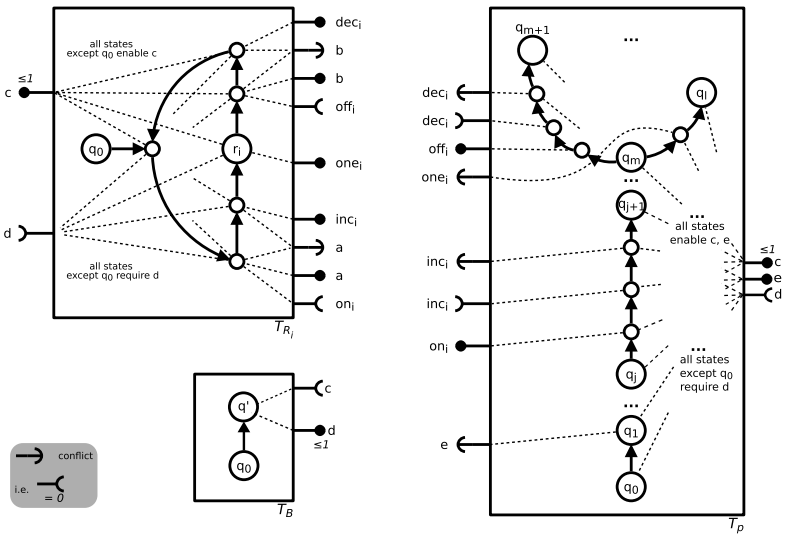


Fig. 6. Modelling 2 counter machines (2CMs) in the Aeolus model.

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 modelled by v_i components in a particular state r_i. Increment instructions add one component in this state r_i, while decrement instructions move one component in state r_i to a different state. The state r_i activates a provide port one_i, so the simulation of a test for zero has simply to check the absence in the environment of active one_i ports.

The component types used to model 2CMs in Aeolus are depicted in Fig. 6. Namely, we consider four component types: $\mathcal{T}_P$ to simulate the execution of the program instructions, $\mathcal{T}_{R_1}$ and $\mathcal{T}_{R_2}$ for the two registers and $\mathcal{T}_B$ used to guarantee that the components involved in the simulation cannot be deleted.

In $\mathcal{T}_P$ we assume one state q_j for each instruction j. If the j-th instruction is $j : \mathsf{Inc}(R_i)$ (see the state q_j in Fig. 6), a protocol with three intermediary states is executed that completes by entering the state q_{j+1}, representing the next instruction to execute. This protocol has the effect to force a component of type $\mathcal{T}_{R_i}$ to execute a complementary protocol that completes by entering the state r_i, thus representing the increment by one of the register R_i. A description of this protocol is reported in the proof of Proposition 1. If the m-th instruction is $m : \mathsf{DecJump}(R_i, l)$ (see the state q_m in Fig. 6), two state changes are possible from the state q_m. The first one starts a protocol similar to the previous one, whose effect here is to force one component of type $\mathcal{T}_{R_i}$ to exit from the state r_i, thus representing the decrement by one of the register R_i. The second one traverses a state that requires the absence in the configuration of active one_i provide port, thus checking that the content of R_i is zero, and then enters state q_l.

In our model, when a component z is not used to satisfy requirements, it could be removed by executing the $del(z)$ action. The cancellation of a component of type $\mathcal{T}_{R_i}$ could then erroneously change register contents during the simulation. To avoid that, we force the connection of each component of type $\mathcal{T}_{R_i}$ with a corresponding instance of a component of type $\mathcal{T}_B$. These types of components reciprocally 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 that 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 $\mathcal{T}_{R_i}$ and those of type $\mathcal{T}_B$.

As a final remark, notice that the first state q_1 of the component type $\mathcal{T}_P$ has a requirement on the absence in the environment of an active provide port e, port which is activated by all the states in $\mathcal{T}_P$. This guarantees that at most one component of type $\mathcal{T}_P$ can be in a state different from q_0. Moreover, we also have to avoid that such component is removed by a *del* action: this can be guaranteed by using the same pairing technique with a component of type $\mathcal{T}_B$ described above. It is sufficient to impose that all the states of $\mathcal{T}_P$, but q_0, activate a provide port on c with numerical constraint 1, and a require port on d, as shown in Fig. 6.

We are now ready to formally prove our undecidability result. In the following we assume given a 2CM program P and use $\mathcal{C}^{\#}_{\langle \mathcal{T}, q\rangle}$ to denote the number of components of type $\mathcal{T}$ in state q in the configuration $\mathcal{C}$.

Definition 10. Let (i, v_1, v_2) be a state of a 2CM. We define

$$\mathcal{C}_0 = \left\langle \{\mathcal{T}_P, \mathcal{T}_{R_1}, \mathcal{T}_{R_2}, \mathcal{T}_B\}, \emptyset, \emptyset, \emptyset \right\rangle$$

$$[\![(i, v_1, v_2)]\!] = \left\{ \mathcal{C} \;\middle|\; \mathcal{C} \text{ is a correct conf. with universe } \{\mathcal{T}_P, \mathcal{T}_{R_1}, \mathcal{T}_{R_2}, \mathcal{T}_B\},\ \mathcal{C}^{\#}_{\langle \mathcal{T}_P, q_i\rangle} = 1,\right.$$
$$\left. \mathcal{C}^{\#}_{\langle \mathcal{T}_{R_1}, r_1\rangle} = v_1, \text{ and } \mathcal{C}^{\#}_{\langle \mathcal{T}_{R_2}, r_2\rangle} = v_2 \right\}$$

In the following we call *program step* a sequence of reconfigurations that, beyond other actions, includes state changes of the component $\mathcal{T}_P$ until entering a state q_j (corresponding to an instruction of the program P). Formally, it is a non empty sequence of reconfigurations $\mathcal{C}_1 \xrightarrow{\alpha_1} \mathcal{C}_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{m-1}} \mathcal{C}_m$ such that:

- there exists an index j of a program instruction[2] for which ${\mathcal{C}_m}^{\#}_{\langle \mathcal{T}_P, q_j \rangle} = 1$ while ${\mathcal{C}_{m-1}}^{\#}_{\langle \mathcal{T}_P, q_j \rangle} = 0$;
- for every $1 < i < m$ there exists no index j of a program instruction for which ${\mathcal{C}_i}^{\#}_{\langle \mathcal{T}_P, q_j \rangle} = 1$ while ${\mathcal{C}_{i-1}}^{\#}_{\langle \mathcal{T}_P, q_j \rangle} = 0$.

Notice that in our modelling of 2CMs there exists also infinite sequences of reconfigurations that do not contain program steps: in these cases they include infinitely many actions that are irrelevant for the simulation (like creation or destruction of components, or bindings and unbindings) and only finitely many state changes of components of type $\mathcal{T}_P$ that are not sufficient to reach a new q_j state.

We first observe that the deployment run composed by the actions $new(z_1 : \mathcal{T}_P)$, $new(z_2 : \mathcal{T}_B)$, $bind(c, z_2, z_1)$, $bind(d, z_1, z_2)$, and the multi stage change action $\{stateChange(z_1, q_0, q_1), stateChange(z_2, q_0, q')\}$ guarantees the possibility to reach, from the initial empty configuration $\mathcal{C}_0$, a configuration corresponding to the initial state of the 2CM, i.e., a configuration in $[\![(1, 0, 0)]\!]$. Moreover, every *program step* from $\mathcal{C}_0$ reaches a configuration in $[\![(1, 0, 0)]\!]$. In fact, it is not possible for components of type $\mathcal{T}_{R_i}$ to enter their state r_i if components of type $\mathcal{T}_P$ perform only the state change action from q_0 to q_1.

Fact 1. *There exists a deployment run from $\mathcal{C}_0$ to a configuration in $[\![(1, 0, 0)]\!]$. Moreover, for every* program step *from $\mathcal{C}_0$ to a configuration $\mathcal{C}'$, we have that $\mathcal{C}' \in [\![(1, 0, 0)]\!]$.*

The proof of undecidability is based on two distinct propositions, a first one about *completeness* of the simulation (i.e., each computational step of the 2CM can be mimicked by a deployment run), and a second one about *soundness* (i.e., each program step of a configuration $\mathcal{C} \in [\![(j, v_1, v_2)]\!]$ corresponds to a step $(j, v_1, v_2) \to (j', v_1', v_2')$ of the 2CM).

Proposition 1. *Let (j, v_1, v_2) be a state of the 2CM and let $\mathcal{C} \in [\![(j, v_1, v_2)]\!]$. If $(j, v_1, v_2) \to (j', v_1', v_2')$ then there exists a deployment run from $\mathcal{C}$ to a configuration $\mathcal{C}' \in [\![(j', v_1', v_2')]\!]$.*

Proof. It is sufficient to perform an analysis of the three possible computational steps of the 2CM: increment, decrement and test for zero. We detail only the increment case (the other cases are treated similarly). If the j-th instruction is an increment on R_i then in $\mathcal{C}$ the component of type $\mathcal{T}_P$ is in state q_j. This means that an action can be executed to move it in the state that activates the on_i provide port (see Fig. 6). This permits to create a new pair of components of type $\mathcal{T}_{R_i}$ and $\mathcal{T}_B$, bind them on their ports c and d, and then move the former in the state requiring on_i (notice that a multiple state change is needed to satisfy the mutual requirements between the two new components). The deployment run can then be extended by moving the new component of type $\mathcal{T}_{R_i}$ in the state that activates the provide port inc_i, moving the component of type $\mathcal{T}_P$ in state q_{j+1} (with two state changes) and finally the new component of type $\mathcal{T}_{R_i}$ in its state r_i. The reached configuration $\mathcal{C}'$ belongs to $[\![(j', v_1', v_2')]\!]$ because ${\mathcal{C}'}^{\#}_{\langle \mathcal{T}_{R_i}, r_i \rangle} = {\mathcal{C}}^{\#}_{\langle \mathcal{T}_{R_i}, r_i \rangle} + 1$ and in this case $j' = j + 1$. □

We now move to the proof of the soundness result.

Proposition 2. *Let (j, v_1, v_2) be a state of the 2CM and let $\mathcal{C} \in [\![(j, v_1, v_2)]\!]$. If there exists a program step from $\mathcal{C}$ that reaches a configuration $\mathcal{C}'$ then $\mathcal{C}' \in [\![(j', v_1', v_2')]\!]$ and $(j, v_1, v_2) \to (j', v_1', v_2')$.*

Proof. We perform an analysis of the reconfiguration actions executed during the program step. There are three kinds of actions: state changes of the component of type $\mathcal{T}_P$ moving from state q_j to $q_{j'}$, state changes inside one of the components $\mathcal{T}_{R_i}$ and other actions. The other actions can be creation or destruction of resources, creation or deletion of bindings (that do not alter the configuration correctness), and multi state changes of new pairs of components of type $\mathcal{T}_{R_i}$ and $\mathcal{T}_B$. All these actions are irrelevant as their modifications on the configuration have no impact on the properties checked by the definition of $[\![(j', v_1', v_2')]\!]$. It is now sufficient to perform a case analysis on the three possible kinds of state changes from state q_j to $q_{j'}$ in the component of type $\mathcal{T}_P$: increment, decrement, and test for zero.

In the "test for zero" case, we have that the j-instruction is of the kind $\mathsf{DecJump}(R_i, j')$. Moreover, in the configuration (j, v_1, v_2) we have $v_i = 0$ because during the program step no component of type $\mathcal{T}_{R_1}$ or $\mathcal{T}_{R_2}$ can perform state changes (this would require the activation of either the port on_i or the port off_i) and the component of type $\mathcal{T}_P$ traverses a state that checks the absence of active one_i ports (this implies ${\mathcal{C}}^{\#}_{\langle \mathcal{T}_{R_i}, r_i \rangle} = 0$). Hence, we have $(j, v_1, v_2) \to (j', v_1, v_2)$ and $\mathcal{C}' \in [\![(j', v_1, v_2)]\!]$.

[2] Notice that 0 is not a correct index as we have assumed that the program P starts from instruction 1.

In the other two cases, it is sufficient to check that the execution of a protocol like the one described in the proof of Proposition 1 is executed by the component of type $\mathcal{T}_P$ and one component of type $\mathcal{T}_{R_i}$. □

We can finally state the main undecidability result:

Theorem 1. *The achievability problem is undecidable in the Aeolus model.*

Proof. Let M be a 2CM with program P, and let $U = \{\mathcal{T}_P, \mathcal{T}_{R_1}, \mathcal{T}_{R_2}, \mathcal{T}_B\}$ be the set of the corresponding component types defined as in Fig. 6.

We have that $(1,0,0) \rightarrow^* (j, v_1, v_2)$ *if and only if* there exists a deployment run from $\mathcal{C}_0$ to a configuration $\mathcal{C} \in [\![(j, v_1, v_2)]\!]$. The *only if* part follows from Fact 1 and Proposition 1, while the *if* part follows from Fact 1 and Proposition 2. Hence we have that the j-instruction is reachable in M *if and only if* the achievability problem is satisfied for the universe U, the component type $\mathcal{T}_P$ and the state q_j.

The undecidability of achievability thus follows from the undecidability of reachability for 2CMs. □

4.2. Achievability is decidable in Aeolus core

We demonstrate decidability of the achievability problem by resorting to the theory of Well-Structured Transition Systems (WSTS) [15,16].

A reflexive and transitive relation is called *quasi-ordering*. A *well-quasi-ordering* (wqo) is a quasi-ordering $(X, \leq)$ such that, for every infinite sequence $x_1, x_2, x_3, \cdots$, there exist $i < j$ with $x_i \leq x_j$. Given a quasi-order $\leq$ over X, an *upward-closed set* is a subset $I \subseteq X$ such that the following holds: $\forall x, y \in X : (x \in I \wedge x \leq y) \Rightarrow y \in I$. Given $x \in X$, its upward closure is $\uparrow x = \{y \in X \mid x \leq y\}$. This notion can be extended to sets in the obvious way: given a set $Y \subseteq X$ we define its upward closure as $\uparrow Y = \bigcup_{y \in Y} \uparrow y$. A *finite basis* of an upward-closed set I is a finite set B such that $I = \bigcup_{x \in B} \uparrow x$.

Definition 11. A WSTS is a transition system $(\mathcal{S}, \rightarrow, \preceq)$ where $\preceq$ is a wqo on $\mathcal{S}$ which is *compatible* with $\rightarrow$, i.e., for every $s_1 \preceq s_1'$ such that $s_1 \rightarrow s_2$, there exists $s_1' \rightarrow^* s_2'$ such that $s_2 \preceq s_2'$ ($\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$). Given a state $s \in \mathcal{S}$, $Pred(s)$ is the set $\{s' \in \mathcal{S} \mid s' \rightarrow s\}$ of immediate predecessors of s. *Pred* is extended to sets in the obvious way: $Pred(S) = \bigcup_{s \in S} Pred(s)$. A WSTS has *effective pred-basis* if there exists an algorithm that, given $s \in \mathcal{S}$, returns a finite basis of $\uparrow Pred(\uparrow s)$.

The following proposition is a special case of Proposition 3.5 in [16].

Proposition 3. *Let $(\mathcal{S}, \rightarrow, \preceq)$ be a finitely branching WSTS with decidable $\preceq$ and effective pred-basis. Let I be any upward-closed subset of $\mathcal{S}$ and let $Pred^*(I)$ be the set $\{s' \in S \mid s' \rightarrow^* s\}$ of predecessors of states in I. A finite basis of $Pred^*(I)$ is computable.*

In the remainder of this section, we assume a given universe U of component types; so we can consider that the set of distinct component type and state pairs $\langle \mathcal{T}, q \rangle$ is finite. Let k be its cardinality. We will resort to the theory of WSTS by considering an abstract model of configurations in which bindings are not taken into account.

Definition 12 *(Abstract configuration).* An abstract configuration $\mathcal{B}$ is a finite multiset of pairs $\langle \mathcal{T}, q \rangle$ where $\mathcal{T}$ is a component type and q is a corresponding state. We use *Conf* to denote the set of abstract configurations.

A concretization of an abstract configuration is simply a correct configuration that for every component type and state pair $\langle \mathcal{T}, q \rangle$ has as many instances of component $\mathcal{T}$ in state q as pairs $\langle \mathcal{T}, q \rangle$ in the abstract configuration.

Definition 13 *(Concretization).* Given an abstract configuration $\mathcal{B}$ we say that a correct configuration $\mathcal{C} = \langle U, Z, S, B \rangle$ is one concretization of $\mathcal{B}$ if there exists a bijection f from the multiset $\mathcal{B}$ to Z s.t. $\forall \langle \mathcal{T}, q \rangle \in \mathcal{B}$ we have that $S(f(\langle \mathcal{T}, q \rangle)) = \langle \mathcal{T}, q \rangle$. We denote with $\gamma(\mathcal{B})$ the set of concretizations of $\mathcal{B}$. We say that an abstract configuration $\mathcal{B}$ is correct if it has at least one concretization (formally $\gamma(\mathcal{B}) \neq \emptyset$).

An interesting property of an abstract configuration is that from one of its concretizations it is possible to reach via bind and unbind actions all the other concretizations (up to instance renaming). This is because it is always possible to switch one binding from one provide port to another one by adding a binding to the new port and then removing the old binding.

Property 1. *Given an abstract configuration $\mathcal{B}$ and configurations $\mathcal{C}_1, \mathcal{C}_2 \in \gamma(\mathcal{B})$ there exists $\alpha_1, \cdots, \alpha_n$ sequence of binding and unbinding actions s.t. $\mathcal{C}_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \mathcal{C} \equiv \mathcal{C}_2$.*

We now move to the definition of our quasi-ordering on abstract configurations. In order to be able to exploit the WSTS techniques in our context, we need to consider a quasi-ordering which is compatible with the notion of correctness, i.e., given a correct abstract configuration, all the greater configurations must be correct as well. For this reason, we cannot adopt the usual multiset inclusion ordering. In fact, the addition of one component to a correct configuration could introduce a conflict. If the type-state pair of the added component was absent in the configuration, the conflict might be with an already present component of a different type-state. If the type-state pair was present in a single copy, the new conflict might be with that component if the considered type-state pair activates one provide and one conflict port on the same interface. This sort of self-conflict is revealed when there are at least two instances, as one component cannot be in conflict with itself (by definition of correctness). If the type-state pair was already present in at least two copies, no new conflicts can be added otherwise such conflicts were already present in the configuration (thus contradicting its correctness).

In the light of the above observation, we define an ordering on configurations that corresponds to the product of three orderings: the identity on the set of type-state pairs that are absent, the identity on the pairs that occurs in one instance, and the multiset inclusion for the projections on the remaining type-state pairs.

Definition 14 *($\leq$).* Given a pair $\langle \mathcal{T}, q\rangle$ and an abstract configuration $\mathcal{B}$, let $\#_{\mathcal{B}}(\langle \mathcal{T}, q\rangle)$ be the number of occurrences in $\mathcal{B}$ of the pair $\langle \mathcal{T}, q\rangle$. Given two abstract configurations $\mathcal{B}_1, \mathcal{B}_2$ we write $\mathcal{B}_1 \leq \mathcal{B}_2$ if for every component type $\mathcal{T}$ and state q we have that $\#_{\mathcal{B}_1}(\langle \mathcal{T}, q\rangle) = \#_{\mathcal{B}_2}(\langle \mathcal{T}, q\rangle)$ when $\#_{\mathcal{B}_1}(\langle \mathcal{T}, q\rangle) \in \{0, 1\}$ or $\#_{\mathcal{B}_2}(\langle \mathcal{T}, q\rangle) \in \{0, 1\}$, and $\#_{\mathcal{B}_1}(\langle \mathcal{T}, q\rangle) \leq \#_{\mathcal{B}_2}(\langle \mathcal{T}, q\rangle)$ otherwise.

As discussed above, this ordering is compatible with correctness.

Property 2. *If an abstract configuration $\mathcal{B}$ is correct than all the configurations $\mathcal{B}'$ such that $\mathcal{B} \leq \mathcal{B}'$ are also correct.*

Another interesting property of the $\leq$ quasi-ordering is that from one concretization of an abstract configuration, it is always possible to reconfigure it to reach a concretization of a smaller abstract configuration. In this case it is possible to first add from the starting configuration the bindings that are present in the final configuration. Then the extra components present in the starting configuration can be deleted because not needed to guarantee correctness (they are instances of components that remain available in at least two copies). Finally the remaining extra bindings can be removed.

Property 3. *Given two abstract configurations $\mathcal{B}_1, \mathcal{B}_2$ s.t. $\mathcal{B}_1 \leq \mathcal{B}_2$, $\mathcal{C}_1 \in \gamma(\mathcal{B}_1)$, and $\mathcal{C}_2 \in \gamma(\mathcal{B}_2)$ we have that there exists a deployment run $\mathcal{C}_2 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \mathcal{C} \equiv \mathcal{C}_1$.*

We have that $\leq$ is a wqo on *Conf* because, as we consider finitely many component type-state pairs, the three distinct orderings that compose $\leq$ are themselves wqo.

Lemma 1. *$\leq$ is a wqo over Conf.*

Proof. The proof is based on a representation of abstract configurations as 3-ples of tuples: namely, given $\mathcal{B} \in \mathit{Conf}$ we represent it as the triple $\langle a, b, c\rangle$ where a is used to represent the component type-state pairs with cardinality 0 in $\mathcal{B}$, b represents those with cardinality 1, and c describes all the other pairs. We assume a total ordering on the set (of cardinality k) of the possible type-state pairs. The three elements a, b and c are vectors of arity k such that $a[i] = 1$ (resp. $b[i] = 1$) if the i-th component type-state pair has cardinality 0 (resp. 1) in $\mathcal{B}$ and $a[i] = 0$ (resp. $b[i] = 0$) otherwise, while $c[i]$ contains the cardinality of the i-th pair in $\mathcal{B}$ if it is greater or equal to 2 and $c[i] = 0$ otherwise. Consider now two abstract configurations $\mathcal{B}_1, \mathcal{B}_2 \in \mathit{Conf}$ and the corresponding triple representations $\langle a_1, b_1, c_1\rangle$ and $\langle a_2, b_2, c_2\rangle$. We have that $\mathcal{B}_1 \leq \mathcal{B}_2$ iff $a_1 = a_2$, $b_1 = b_2$ and $c_1 \leq^k c_2$ (where $\leq^k$ is the extension of the standard ordering on natural numbers to vectors of length k).

The equality on bit vectors of length k (a and b are indeed of length k) is a wqo as there are only finitely many such vectors (namely, 2^k). Dickson's lemma [17] states that a product of wqo is a wqo, thus $\leq^k$ is a wqo too. We can conclude that the ordering on the triples is a wqo by applying again Dickson's lemma. □

We now define a transition system on abstract reconfigurations and prove it is a WSTS with respect to the ordering defined above.

Definition 15 *(Abstract reconfigurations).* We write $\mathcal{B} \to \mathcal{B}'$ if there exists $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ for some $\mathcal{C} \in \gamma(\mathcal{B})$ and $\mathcal{C}' \in \gamma(\mathcal{B}')$.

Lemma 2. *The transition system $(\mathit{Conf}, \to, \leq)$ is a WSTS.*

Proof. The $\leq$ is a wqo for *Conf* by Lemma 1. To prove the thesis we need to prove that $\leq$ is compatible with $\to$ (i.e., if $\mathcal{B}_1 \leq \mathcal{B}_2$ and $\mathcal{B}_1 \to \mathcal{B}'_1$ then $\mathcal{B}_2 \to^* \mathcal{B}'_2$ for some $\mathcal{B}'_2$ s.t. $\mathcal{B}'_1 \leq \mathcal{B}'_2$). This is straightforward since we have $\mathcal{B}_2 \to^* \mathcal{B}_1$ (by Property 3), $\mathcal{B}_1 \to \mathcal{B}'_1$ (by hypothesis), and $\mathcal{B}'_1 \leq \mathcal{B}'_1$ (by reflexivity of $\leq$). □

The following lemma is rather technical and it will be used to prove that $(Conf, \rightarrow, \leq)$ has effective pred-basis. Intuitively it will allow us to consider, in the computation of the predecessors, only finitely many different (multiple) state change actions.

Lemma 3. *Let k be the number of distinct component type-state pairs. If $\mathcal{B}_1 \rightarrow \mathcal{B}_2$ then there exists $\mathcal{B}'_1 \rightarrow \mathcal{B}'_2$ such that $\mathcal{B}'_1 \leq \mathcal{B}_1$, $\mathcal{B}'_2 \leq \mathcal{B}_2$ and $|\mathcal{B}'_2| \leq 2k + 2k^2$.*

Proof. If $|\mathcal{B}_2| \leq 2k + 2k^2$ the thesis trivially holds. Consider now $|\mathcal{B}_2| > 2k + 2k^2$ and a transition $\mathcal{C}_1 \xrightarrow{\alpha} \mathcal{C}_2$ such that $\mathcal{C}_1 \in \gamma(\mathcal{B}_1)$ and $\mathcal{C}_2 \in \gamma(\mathcal{B}_2)$. We now show that is possible to remove one component from $\mathcal{C}_1$ while keeping the possibility to perform an action leading to a configuration corresponding to $\mathcal{C}_2$ without the component removed form $\mathcal{C}_1$. We consider two subcases.
Case 1. There are three components z_1, z_2 and z_3 having the same component type and internal state that do not perform a state change in the action α. Without loss of generality we can assume that z_3 does not appear in α (this is not restrictive because at most two components that do not perform a state change can occur in an action). We can now consider the configuration $\mathcal{C}'_1$ obtained by $\mathcal{C}_1$ after removing z_3 (if there are bindings connected to provide ports of z_3, these can be rebound to ports of z_1 or z_2). Consider now $\mathcal{C}'_1 \xrightarrow{\alpha} \mathcal{C}'_2$ and the corresponding abstract configurations $\mathcal{B}'_1$ and $\mathcal{B}'_2$. We have that $\mathcal{B}'_1 \rightarrow \mathcal{B}'_2$, $\mathcal{B}'_1 \leq \mathcal{B}_1$, $\mathcal{B}'_2 \leq \mathcal{B}_2$ and $|\mathcal{B}'_2| < |\mathcal{B}_2|$. If $|\mathcal{B}'_2| \leq 2k + 2k^2$ the thesis is proved, otherwise we repeat this deletion of components.
Case 2. There are no three components of the same type-state that do not perform a state change. Since $|\mathcal{B}_2| > 2k + 2k^2$ we have that α is a multiple state change involving strictly more than $2k^2$ components (otherwise there are strictly more than $2k$ components that do not perform state changes, thus at least three of them are of the same type-state). This ensures the existence of three components z_1, z_2 and z_3 of the same type that perform the same state change from q to q'. As in the previous case we consider the configuration $\mathcal{C}'_1$ obtained by $\mathcal{C}_1$ after removing z_3 and α' the state change similar to α but without the state change of z_3. Consider now $\mathcal{C}'_1 \xrightarrow{\alpha'} \mathcal{C}'_2$ and the corresponding abstract configurations $\mathcal{B}'_1$ and $\mathcal{B}'_2$. As above, $\mathcal{B}'_1 \leq \mathcal{B}_1$, $\mathcal{B}'_2 \leq \mathcal{B}_2$ and $|\mathcal{B}'_2| < |\mathcal{B}_2|$. If $|\mathcal{B}'_2| \leq 2k + 2k^2$ the thesis is proved, otherwise we repeat the deletion of components. □

We are now in place to prove that $(Conf, \rightarrow, \leq)$ has effective pred-basis.

Lemma 4. *The transition system $(Conf, \rightarrow, \leq)$ has effective pred-basis.*

Proof. We first observe that given an abstract configuration the set of its concretizations up to configuration equivalence is finite, and that given a configuration $\mathcal{C}$ the set of preceding configurations $\mathcal{C}'$ such that $\mathcal{C}' \xrightarrow{\alpha} \mathcal{C}$ is also finite (and effectively computable). Consider now an abstract configuration $\mathcal{B}$. We now show how to compute a finite basis for $\uparrow Pred(\uparrow \mathcal{B})$ by considering the preceding configurations of a finite set of corresponding concrete configurations. First of all we consider the finite set of abstract configurations composed by $\mathcal{B}$, if $|\mathcal{B}| > 2k + 2k^2$, or all the configurations $\mathcal{B}'$ such that $\mathcal{B} \leq \mathcal{B}'$ and $|\mathcal{B}'| \leq 2k + 2k^2$, otherwise. Then we consider the (finite) set of concretizations of all such abstract configurations. Finally we compute the (finite) set of the preceding configurations of all such concretizations. The finite basis is obtained by taking the set of abstract configurations corresponding to the latter: this is finite and it is a basis for $\uparrow Pred(\uparrow \mathcal{B})$ as a consequence of Lemma 3. □

We are finally ready to prove our decidability result.

Theorem 2. *The achievability problem in Aeolus core is decidable.*

Proof. Let k be the number of distinct component type-state pairs according to the considered universe of component types. We first observe that if there exists a correct configuration containing a component of type $\mathcal{T}$ in state q then it is possible to obtain via some binding, unbinding, and delete actions another correct configuration with k or less components. Hence, given a component type $\mathcal{T}$ and a state q, the number of target configurations that need to be considered is finite. Moreover, given a configuration $\mathcal{C}' \in \gamma(\mathcal{B}')$ there exists a deployment run from $\mathcal{C} \in \gamma(\mathcal{B})$ to $\mathcal{C}'$ iff $\mathcal{B} \in Pred^*(\uparrow \mathcal{B}')$.

To solve the achievability problem it is therefore possible to consider only the (finite set of) abstractions of the target configurations. For each of them, say $\mathcal{B}'$, by Proposition 3, Lemma 2, and Lemma 4 we know that a finite basis for $Pred^*(\uparrow \mathcal{B}')$ can be computed. It is sufficient to check whether the initial empty configuration is in such basis. □

In this section we have considered just the problem of reaching a target configuration starting from an initial empty configuration. The proof presented holds however also for the more general problem of finding if the target configuration can be reached by an initial (possibly non empty) configuration. Indeed, in this case, it is sufficient to check whether at least one of the abstract configurations in $Pred^*(\uparrow \mathcal{B}')$ contains a configuration that is $\leq$ w.r.t. the abstraction of the initial configuration.

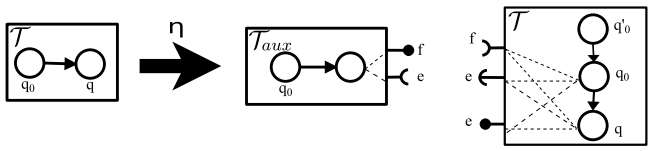
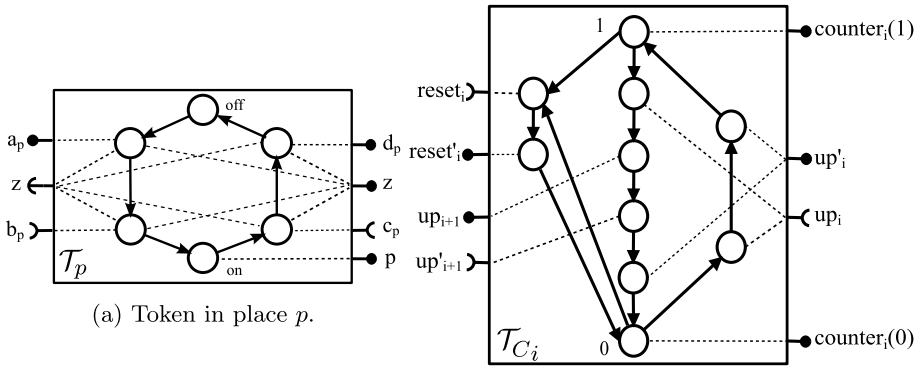


Fig. 7. Example of a component type transformation η.



(a) Token in place p.

(b) i-th bit counter.

Fig. 8. Token and counter component types.

4.3. Achievability is Ackermann-hard in Aeolus core

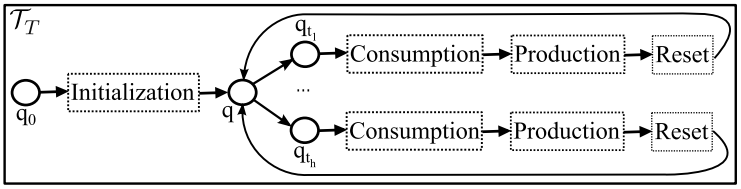
We now prove that the achievability problem in Aeolus core is Ackermann-hard by reduction from the coverability problem in reset Petri nets, a problem which is indeed known to be Ackermann-hard [18].

We start with some background on reset Petri nets.

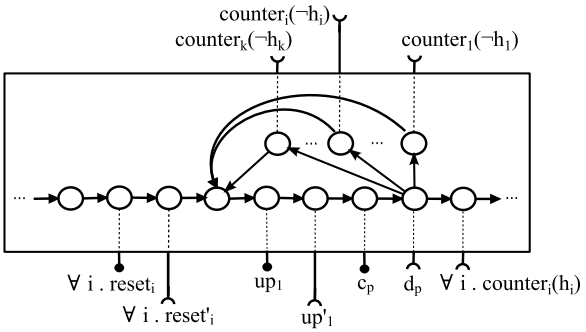
A *reset Petri net* RN is a tuple $\langle P, T, \vec{m}_0 \rangle$ such that P is a finite set of *places*, T is a finite set of *transitions*, and $\vec{m}_0$ is a marking, i.e., a mapping from P to $\mathbb{N}$ that defines the initial number of tokens in each place of the net. A transition $t \in T$ is defined by a mapping ${}^\bullet t$ (preset) from P to $\mathbb{N}$, a mapping $t^\bullet$ (postset), and by a set of reset arcs $t{\downarrow} \subseteq P$. A configuration is a marking $\vec{m}$. Transition t is enabled at marking $\vec{m}$ iff ${}^\bullet t(p) \leq \vec{m}(p)$ for each $p \in P$. Firing t at $\vec{m}$ leads to a new marking $\vec{m'}$ defined as $\vec{m'}(p) = \vec{m}(p) - {}^\bullet t(p) + t^\bullet(p)$ if $p \notin t{\downarrow}$, and $\vec{m'}(p) = 0$ otherwise; we denote this marking transformation with $\vec{m} \mapsto \vec{m'}$. A marking $\vec{m}$ is reachable from $\vec{m}_0$ if $\vec{m}_0 \mapsto^* \vec{m}$, i.e., it is possible to produce $\vec{m}$ after firing finitely many times transitions in T. Given a reset net $\langle P, T, \vec{m}_0 \rangle$ and a marking $\vec{m}$, the coverability problem consists in checking for the existence of a reachable marking $\vec{m'}$ such that $\vec{m} \leq \vec{m'}$, i.e. $\vec{m}(p) \leq \vec{m'}(p)$ for every $p \in P$. In [18] it is proved that the coverability problem for reset nets is Ackermann-hard.

Before entering into the details of our modelling of reset Petri nets, we observe that given a component type $\mathcal{T}$ it is always possible to modify it in such a way that its instances are persistent and unique. The uniqueness constraint can be enforced by allowing all the states of the component type to provide a new port with which they are in conflict. To avoid the component deletion it is sufficient to impose its reciprocal dependence with a new type of component. When this dependence is established the components cannot be deleted without violating it. In Fig. 7 we show an example of how a component type having two states can be modified in order to reach our goal. A new auxiliary initial state q'_0 is created. The new port e ensures that the instances of type $\mathcal{T}$ in a state different from q'_0 are unique. The require port f provided by a new component type $\mathcal{T}_{aux}$ forbids the deletion of the instances of type $\mathcal{T}$, if they are not in state q'_0. We assume that the ports e and f are fresh. We can therefore consider, without loss of generality, components that are unique and persistent. Given a component type $\mathcal{T}$ we denote this component type transformation with $\eta(\mathcal{T})$.

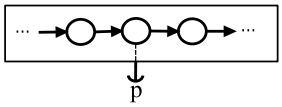
We now consider a given reset Petri net $RN = \langle P, T, \vec{m}_0 \rangle$ and discuss how to encode it in Aeolus core component types. We will use three types of components: one modelling the tokens, one for the transitions and one for defining a counter. The components for the transitions and the counter are unique and persistent, while those for the tokens cannot be unique because the number of tokens in a Petri net can be unbounded. The simplest component type, denoted with $\mathcal{T}_p$, is the one used to model a token in a given place $p \in P$. Namely, one token in a place p is encoded as one instance of $\mathcal{T}_p$ in the *on* state. There could be more than one of these components deployed simultaneously representing multiple tokens in a place. In Fig. 8a we represent the component type $\mathcal{T}_p$. The initial state is the *off* state. The token could be created following a protocol consisting of requiring the port a_p and then providing the port b_p. Symmetrically, a token can be removed by providing the port c_p and then requiring the port d_p. Even if multiple instances of the token component can be deployed simultaneously, only one of them at a time can initiate the protocol to change its state. This is guaranteed by the conflict on the port z, which is provided by all the states of the state change protocols. The component provides the port p when it is in the *on* state.



(a) Transitions component.



(b) Consumption phase of n tokens from place p for a transition t ($k = \lceil \log(n) \rceil$ and h_i is the i-th least significative bit of the binary representation of n).



(c) Encoding of a reset arc for the place p.

Fig. 9. Transitions component charts.

In order to model the transitions without having an exponential blow up of the size of the encoding we need a mechanism to count up to a fixed number. Indeed, a transition can consume and produce a given number of tokens from and to several places. To count a number up to n we will use instances of the component types $\mathcal{T}_{C_1}, \cdots, \mathcal{T}_{C_{\lceil \log(n) \rceil}}$; the type $\mathcal{T}_{C_i}$ will be used to represent the i-th less significant bit of the binary representation of the counter that, for our purposes, needs just to support the increment and reset operations. In Fig. 8b we represent one of the bits implementing the counter. The initial state is 0. To increment the bit it is necessary to provide and require in sequence the up_i and up'_i ports, while to reset it the $reset_i$ and $reset'_i$ ports. If the bit is in state 1 the increment will trigger the increment of the next bit (except for the component representing the most significant bit that will never need to do that). The instance of $\mathcal{T}_{C_i}$ can be used to count how many tokens are consumed or produced by checking if the right number is reached via the ports $counter_i(0)$ and $counter_i(1)$. We transform the component types $\mathcal{T}_{C_1}, \cdots, \mathcal{T}_{C_{\lceil \log(n) \rceil}}$ using the η transformation to ensure uniqueness and persistence of its instances.

The transitions in T can be represented with a single component interacting with token and counter components. This component, represented in Fig. 9a, during a so-called initialization phase, performs state changes until reaching a state q. The initialization phase is used to generate the representation of the initial marking $\vec{m_0}$. From the state q it can nondeterministically select one transition t to fire, by entering a corresponding q_t state. The subsequent state changes can be divided in three phases: consumption, production, and reset. These phases respectively model the consumption of tokens from the places in the preset of the transition t, the production of tokens in the places in the postset of t, and the complete elimination of the tokens in the reset places of t. Notice that, as the transition t to be fired is selected nondeterministically, the corresponding deployment run could block due to the unavailability of instances of token components required during the consumption phase. As we will discuss later, these blocking deployment runs are not problematic.

We now describe in details the three consumption, production and reset phases, and then comment the initialization phase. In the consumption phase, for every place p in the preset of the transition, the counter is first reset providing the $reset_i$ and requiring the $reset'_i$ ports for all the counter bits. Then a cycle starts incrementing the counter, by providing and requiring the ports up_1 and up'_1, and consuming a token, by providing and requiring the ports c_p and d_p. The cycle ends when all the bits of the counter represent in binary the right number of tokens that need to be consumed from p. If instead at least one bit is wrong the cycle restarts. In Fig. 9b we depict the part of the component type modelling the consumption of n tokens from the place p.

The production phase is similar to the consumption phase. For every kind of token that needs to be produced, the counter components are used to count the actual number of instances. The production of a single token follows a protocol similar to the one used for their consumption with the only difference that the ports a_p and b_p are required and provided in sequence, instead of providing and requiring c_p and d_p.

Reset arcs are instead modelled with a single state conflicting with the tokens in places that must be reset. For instance in Fig. 9c we depicted the part of the component modelling the reset of a place p: the conflict on the port p forces the

deletion of all the instances of component type $\mathcal{T}_p$ in the *on* state. At the end of the reset phase, the component has a transition to return in its state q.

The initialization phase is like a production phase in which the tokens of $\vec{m}_0$ are produced; at the end of the initialization phase the state q is entered.

We will denote with $\mathcal{T}_T$ the component type explained above.

Definition 16 *(Reset Petri net encoding in Aeolus core).* Given a reset Petri net $RN = (P, T, m_0)$ if n is the largest number of tokens that can be consumed or produced by a transition in T, the encoding of RN in Aeolus core is the set of component types:

$$\Gamma_{RN} = \{\mathcal{T}_p \mid p \in P\} \cup \left\{\eta(\mathcal{T}_{C_i}) \;\middle|\; i \in \left[1..\lceil \log(n) \rceil\right]\right\} \cup \left\{\eta(\mathcal{T}_T)\right\}$$

Notice that the components of type $\mathcal{T}_p$ are not guaranteed to be persistent, so they can be deleted even when they are in the *on* state. This corresponds to a nondeterministic elimination of one token from the place p. As we will discuss later, this token elimination in our net simulation is not problematic because it is not necessary to faithfully reproduce the net behaviour, but it is sufficient to preserve coverability (i.e., the possibility to generate at least a predefined number of tokens in some given places).

Before moving to the proof of the correspondence between the reset Petri net RN and its encoding Γ_{RN}, we observe that the size of the component types Γ_{RN} is polynomial w.r.t. the size of the reset Petri net. This is due to the fact that the counter and place components have a constant amount of states and ports while the component for the transitions has a number of states that grows linearly with respect to the number of places involved in the transitions.

We now introduce the notation $\mathcal{C}_0$ for denoting the empty initial configuration of our encoding, and $[\![\vec{m}]\!]$ to characterize configurations corresponding to the net marking $\vec{m}$.

Definition 17. Let $RN = (P, T, m_0)$ be a reset Petri net and $\vec{m}$ one of its markings. We define:

$$\mathcal{C}_0 = \langle \Gamma_{RN}, \emptyset, \emptyset, \emptyset \rangle$$

$$[\![\vec{m}]\!] = \left\{\mathcal{C} \;\middle|\; \mathcal{C} \text{ is a correct configuration with universe } \Gamma_{RN},\ \mathcal{C}^{\#}_{\langle \mathcal{T}_T, q\rangle} = 1,\ \forall p \in P. \mathcal{C}^{\#}_{\langle \mathcal{T}_p, on\rangle} = \vec{m}(p)\right\}$$

We call *net step* a sequence of reconfigurations on components instances of the universe Γ_{RN} that, beyond other actions, includes state changes of the component $\mathcal{T}_T$ until entering the state q. Formally, it is a non empty sequence of reconfigurations $\mathcal{C}_1 \xrightarrow{\alpha_1} \mathcal{C}_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{m-1}} \mathcal{C}_m$ such that $\mathcal{C}_m{}^{\#}_{\langle \mathcal{T}_T, q\rangle} = 1$, while $\mathcal{C}_i{}^{\#}_{\langle \mathcal{T}_T, q\rangle} = 0$, for every $1 < i < m$.

We first observe that the deployment run that creates an instance of $\mathcal{T}_T$ and then performs the state changes in the initialization phase until entering the state q guarantees the possibility to reach a configuration in $[\![\vec{m}_0]\!]$. Moreover, every *net step* from the initial empty configuration reaches a configuration in $[\![\vec{m}]\!]$ where $\vec{m} \leq \vec{m}_0$. Notice that we need to consider markings $[\![\vec{m}]\!]$ smaller than $[\![\vec{m}_0]\!]$ because token components moved in the *on* state during the initialization could be nondeterministically deleted.

Fact 2. *There exists a deployment run from $\mathcal{C}_0$ to a configuration in $[\![\vec{m}_0]\!]$. Moreover, for every* net step *from $\mathcal{C}_0$ to a configuration $\mathcal{C}'$, we have that $\mathcal{C}' \in [\![\vec{m}]\!]$ and $\vec{m} \leq \vec{m}_0$.*

The proof of correspondence between a reset Petri net RN and its encoding Γ_{RN} is based on two distinct propositions, a first one about *completeness* of the simulation (i.e., each firing of a net transition can be mimicked by a deployment run), and a second one about *soundness* (i.e., each net step of a configuration corresponds to the firing of a net transition).

Proposition 4. *Let $RN = (P, T, m_0)$ be a reset Petri net, $\vec{m}$ one of its markings, and $\mathcal{C}$ a configuration in $[\![\vec{m}]\!]$. If $\vec{m} \mapsto \vec{m}'$ then there exists a deployment run from $\mathcal{C}$ to a configuration $\mathcal{C}' \in [\![\vec{m}']\!]$.*

Proof. It is sufficient to observe that if $\vec{m} \mapsto \vec{m}'$ then there exists a transition $t \in T$ that, by consuming and producing tokens and resetting places, transforms $\vec{m}$ in $\vec{m}'$. This transition can be selected in a deployment from $\mathcal{C}$ that starts by changing the state of $\mathcal{T}_T$ form q to q_t. Then the corresponding consumption, production and reset phases can be executed to reach a configuration in $[\![\vec{m}']\!]$. □

We now move to the proof of the soundness result by showing that, if there exists a net step from a configuration in $[\![\vec{m}]\!]$ to a configuration in $[\![\vec{m}']\!]$, then there exists a marking $\vec{m}'' \geq \vec{m}'$ such that $\vec{m} \mapsto \vec{m}''$. We need to consider a greater marking in the net because, as already observed, token components could be deleted during the deployment run and a perfect correspondence between the reset Petri net and its simulation is not guaranteed.

Proposition 5. *Let $RN = (P, T, m_0)$ be a reset Petri net, $\vec{m}$ one of its markings, and $\mathcal{C}$ a configuration in $[\![\vec{m}]\!]$ having a net step to $\mathcal{C}'$. Then, there exists a marking $\vec{m'}$ such that $\mathcal{C}' \in [\![\vec{m'}]\!]$ and a marking $\vec{m''} \geq \vec{m'}$ such that $\vec{m} \mapsto \vec{m''}$.*

Proof. We first observe that the final configuration $\mathcal{C}'$ of a net step contains the $\mathcal{T}_T$ component in the q state; thus there exists $[\![\vec{m'}]\!]$ s.t. $\mathcal{C}' \in [\![\vec{m'}]\!]$.

The net step from $\mathcal{C} \in [\![\vec{m}]\!]$ to $\mathcal{C}' \in [\![\vec{m'}]\!]$ includes the state changes, on the instance of component type $\mathcal{T}_T$, corresponding to the consumption, production and reset phases for some transition t in T. The execution of the consumption phase guarantees that ${}^\bullet t \leq \vec{m}$ thus t can fire in $\vec{m}$: let $\vec{m} \mapsto \vec{m''}$ be the effect of firing such transition. We have that $\vec{m''} \geq \vec{m'}$ because $\vec{m''}$ is obtained from $\vec{m}$ by performing the same consumption, production and reset executed during the net step from $\mathcal{C} \in [\![\vec{m}]\!]$ to $\mathcal{C}' \in [\![\vec{m'}]\!]$. Notice that during this net step some token component in the *on* state could be nondeterministically deleted, but this has no impact on the property $\vec{m''} \geq \vec{m'}$ because its effect is simply to remove more instances of active token components w.r.t. those removed during the consumption phase. □

Notice that besides the net step from $\mathcal{C}$ to $\mathcal{C}'$ considered in the above Proposition, there are deployment runs starting from $\mathcal{C}$ that do not correspond to net steps. For instance, there could be an infinite sequence of creations and deletions of components or, more interesting, a non habilitated transition t could be tried. In this case the deployment run could block because the consumption phase cannot be completed. These additional deployment runs are not problematic as our encoding needs to preserve the possibility to reach specific target configurations (i.e., those that contain at least a given amount of instances of token components in the *on* state), and additional deployment runs that do not reach such configurations are irrelevant.

We are finally ready to prove Ackermann-hardness of the achievability problem for Aeolus core.

Theorem 3. *The achievability problem for Aeolus core is Ackermann-hard.*

Proof. Consider a reset Petri net $RN = \langle P, T, \vec{m_0}\rangle$ and a target marking $\vec{m}$. The problem of checking whether $\vec{m}$ can be covered in RN is Ackermann-hard [18]. We first construct a new reset Petri net $RN' = \langle P \uplus \{p'\}, T \uplus \{t'\}', \vec{m_0}\rangle$ with an additional place p' and a transition t' such that ${}^\bullet t' = m$ and $t'^\bullet = \{p'\}$. It is immediate to see that $\vec{m}$ can be covered in RN iff at least one token can be placed in p' in RN'. Moreover this transformation increases the size of the Petri net by a constant. We now consider the set of component types $\Gamma_{RN'}$, i.e., the encoding of RN' in Aeolus core. We have already observed that the size of $\Gamma_{RN'}$ is polynomial w.r.t. the size of the reset net RN'. We complete the proof by showing that a token can be placed in p' in RN' *iff* the component type-state pair $\langle \mathcal{T}_T, q_{p'}\rangle$ is achievable with the universe of component types $\Gamma_{RN'}$, where $q_{p'}$ is the state of $\mathcal{T}_T$ that provides the port $b_{p'}$, necessary to move in the *on* state a component of type $\mathcal{T}_{p'}$.

The "only if" part follows from Fact 2 and Proposition 4, that guarantee the existence of a deployment run reaching a configuration $\mathcal{C} \in [\![\vec{m}]\!]$ with $m(p') > 0$. As in $\mathcal{C}$ there is at least one instance of type $\mathcal{T}_{p'}$ in the *on* state, at least one configuration is traversed with the instance $\mathcal{T}_T$ in the $q_{p'}$ state.

The proof of the "if" part proceeds as follows. The achievability of the pair $\langle \mathcal{T}_T, q_{p'}\rangle$ guarantees the existence of a deployment run $\mathcal{C}_0 \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} \mathcal{C}_n$ where $\mathcal{C}_n$ contains one instance of $\mathcal{T}_T$ in the $q_{p'}$ state. Such computation can be extended in such a way that one instance of $\mathcal{T}_{p'}$ reaches the *on* state and the $\mathcal{T}_T$ component enters in the state q. Let $\mathcal{C}$ be the reached configuration. As the $\mathcal{T}_T$ component is in the state q, we have that $\mathcal{C} \in [\![\vec{m}]\!]$ for some marking $\vec{m}$. Moreover, $m(p') > 0$ because an instance of $\mathcal{T}_{p'}$ is in the *on* state. Fact 2 and Proposition 5 guarantees the reachability in the net RN' of a marking $\vec{m'}$ such that $\vec{m} \leq \vec{m'}$, thus guaranteeing $m'(p') > 0$. □

4.4. Achievability is polynomial in Aeolus$^-$

The achievability problem becomes polynomial in case no capacity constraints are specified on require and provide port, and no conflicts are allowed (i.e., the value 0 on require ports is forbidden). We prove this by presenting a decision algorithm for the achievability problem in Aeolus$^-$.

The underlying idea is to perform an abstract forward exploration of all reachable configurations. Since conflicts cannot be specified the addition to a configuration of new components cannot forbid the execution of previously possible actions. Moreover, since 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 $\langle \mathcal{T}, q\rangle$ 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. Thanks to the first observation we can restrict ourselves to consider only evolutions where the set of available pairs $\langle \mathcal{T}, q\rangle$ does not decrease. Namely, we perform

Algorithm 1 Verifying achievability in Aeolus$^-$.

function ACHIEVABILITY(U, $\mathcal{T}$, q)
  $absConf := \{\langle \mathcal{T}', \mathcal{T}'.\texttt{init}\rangle \mid \mathcal{T}' \in U\}$
  $provPort := \bigcup_{\langle \mathcal{T}', q'\rangle \in absConf} \{dom(\mathcal{T}'.\mathbf{P}(q'))\}$
  repeat
    $new := \{\langle \mathcal{T}', q'\rangle \mid \langle \mathcal{T}', q''\rangle \in absConf, (q'', q') \in \mathcal{T}'.\texttt{trans}\} \setminus absConf$
    $newPort := \bigcup_{\langle \mathcal{T}', q'\rangle \in new} \{dom(\mathcal{T}'.\mathbf{P}(q'))\}$
    while $\exists \langle \mathcal{T}', q'\rangle \in new.\ dom(\mathcal{T}'.\mathbf{R}(q')) \not\subseteq provPort \cup newPort$ **do**
      $new := new \setminus \{\langle \mathcal{T}', q'\rangle\}$
      $newPort := \bigcup_{\langle \mathcal{T}', q'\rangle \in new} \{dom(\mathcal{T}'.\mathbf{P}(q'))\}$
    end while
    $absConf := absConf \cup new$
    $provPort := provPort \cup newPort$
  until $new = \emptyset$
  if $\langle \mathcal{T}, q\rangle \in absConf$ **then return true**
  else return false
  end if
end function

a symbolic forward exploration starting from an abstract configuration containing all the pairs $\langle \mathcal{T}', \mathcal{T}'.\texttt{init}\rangle$ representing components in their initial state. Then we extend the abstract configuration by adding step-by-step new pairs $\langle \mathcal{T}', q'\rangle$.

Algorithm 1 checks achievability by relying on two auxiliary data structures: *absConf* is the set of pairs $\langle \mathcal{T}', q'\rangle$ indicating the type and state of the components in the current 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. Termination of the algorithm is guaranteed because there are only finitely many type-state pairs in a universe of component types.

At each iteration, the potential new pairs are initially computed by checking the automata transitions, and then they are stored in the set *new*. Not all those states could be actually reached as one needs to check whether their require ports are included in the available provide ports *provPort* or in the ports activated by the new states. This is done by a one-by-one elimination of pairs $\langle \mathcal{T}', q'\rangle$ from *new* when their requirements are unsatisfiable. During elimination, we use *newPort* to keep track of the provide ports which are activated by the component states currently in *new*.

When the final set *absConf* is computed, achievability for the component type $\mathcal{T}$ and state q can be simply checked by verifying whether $\langle \mathcal{T}, q\rangle$ is in *absConf*.

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

Theorem 4. *Let U be a set of component types of the Aeolus$^-$ model. Given the component type $\mathcal{T}$ and the state q, the achievability problem for U, $\mathcal{T}$, and q can be checked in polynomial time (with respect to the size of the descriptions of the components in U).*

Proof. We first prove completeness and soundness of Algorithm 1, i.e., if a pair $\langle \mathcal{T}, q\rangle$ is achievable then it will be included in *absConf* at the end of the algorithm, and if $\langle \mathcal{T}, q\rangle$ is added to *absConf* then there exists a deployment run to deploy one instance of $\mathcal{T}$ in the q state.

Completeness is proved as follows. The symbolic representation of the initial configuration $\langle U, \emptyset, \emptyset, \emptyset\rangle$ is included in the initial set *absConf*. Consider now a reconfiguration $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$. If the symbolic representation of $\mathcal{C}$ is included in *absConf* at the beginning of an iteration of the **repeat**, then the symbolic representation of $\mathcal{C}'$ will be surely included in *absConf* at the end of such iteration. This because the newly reached states in $\mathcal{C}'$ will be also in the *new* set at the end of the **while**. Therefore, if there exists a deployment run able to achieve a component of type $\mathcal{T}$ in the state q, then the algorithm will eventually include the pair $\langle \mathcal{T}, q\rangle$ in *absConf*.

The soundness is proved as follows. Let $absConf_i$ be the set *absConf* at the end of the i-th iteration of the **repeat**, and let $absConf_0$ be the set containing only the pairs $\langle \mathcal{T}', \mathcal{T}'.\texttt{init}\rangle$. By induction on i, we prove that there exists a deployment run that includes at least one instance for every type-state pair in $absConf_i$. For the base case $i = 0$, this trivially holds (it is sufficient to consider a deployment run that creates at least one instance for each component type). In the induction case we consider $absConf_i$, and by induction hypothesis we assume the existence of a deployment run $\mathcal{C}_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \mathcal{C}$ that generates at least one instance for the type-state pairs in $absConf_i$. We show the existence of a deployment run for the pairs in $absConf_{i+1}$. Consider the new pairs $\langle \mathcal{T}', q'\rangle$ added in $absConf_{i+1}$ and let $\mathcal{P}$ be the multiset of pairs necessary to generate such new pairs, i.e.,

$$\mathcal{P} = \left\{\!\!\left\{ \left\langle \mathcal{T}', q'' \right\rangle \;\middle|\; \left(q'', q'\right) \text{ is the transition used to add } \left\langle \mathcal{T}', q' \right\rangle \text{ to } absConf \right\}\!\!\right\}$$

Consider now a deployment run obtained by repeating the actions $\alpha_1, \cdots, \alpha_n$ of the deployment run $\mathcal{C}_0 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \mathcal{C}$ until the reached configuration contains at least one instance for the pairs in $absConf_i$ plus one additional instance for each occurrence of the pairs in the multiset $\mathcal{P}$. This deployment run can be extended with state changes of these additional instances to reach the new $\langle \mathcal{T}', q'\rangle$ pairs.

We now show that the complexity of the algorithm is polynomial w.r.t. the size of the description of the universe of component types U. Polynomiality is guaranteed by the fact that both the **repeat** and the **while** cycles perform a number of iterations smaller than the number of different pairs $\langle \mathcal{T}', q' \rangle$ in the universe U. □

5. Related work

To the best of our knowledge Aeolus is the first model that is designed on purpose to formally addresses the specific needs of software component deployment in the cloud. It was first introduced in [10] and further developed within the ANR project Aeolus "Mastering the Complexity of the Cloud" [19,20]. Differently from the definition of the language presented here, in [10] an additional kind of requirements—called *weak requirements*—was present. Whereas the requirements presented in this paper (formerly known as *strong requirements*) need to be enforced every step, weak requirements are verified only at the end of a deployment run and thus need to be satisfied only in the final configuration. We decided to drop the notion of weak requirements for two reasons: firstly, we noticed that they were not very used in practice; secondly, their behaviour can easily be simulated with strong requirements, so there was no real gain in terms of model expressivity.[3]

This paper improves the complexity result about achievability in the Aeolus core model, which was first proven decidable in [11]. In that paper, reconfigurability (the generalization of achievability with any initial configuration, also the non empty one) was proved to be ExpSpace-hard by reduction from the coverability problem in standard Petri nets; here we have considered reset Petri nets thus proving that the problem is furthermore Ackermann-hard.

5.1. Formal models

We now compare the Aeolus approach to related formal models that have been proposed 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* [21], where automata are used to represent the component behaviour 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 behaviour 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 behaviour 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 [22], a very popular model already presented in the previous sections and born from the attempt to extend automata with concurrency. At first sight one might encode our model in Petri nets, representing Aeolus 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 model, while in Petri nets coverability is an ExpSpace problem [23].

Several process calculi extend/modify the π-calculus [24] in order to deal with software components. The Piccola calculus [25] extends the asynchronous π-calculus [24] with *forms*, first-class extensible namespaces, useful to model component interfaces and bindings. Calculi like KELL [26] and HOMER [27] extends a core π-calculus with hierarchical locations, local actions, higher-order communication, programmable membranes, and dynamic binding. More recently, MECo [28] has extended this approach by proposing also explicit component interfaces and channels to realize tunnelling effects traversing the hierarchical location boundaries. On the one hand, all these proposals differ from Aeolus in that they focus on the modelling 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.

We have already briefly discussed in the introduction the Fractal component [5], and the related cloud middleware FraSCAti [6]. In terms of the underlying formal model, we observe that Fractal does provide an object-oriented API to manage the life-cycle of components which, in spirit, is close to what Aeolus aims to do with component automata. However, the OO API approach is more limited when it comes to the ability to reason on component activation: in Aeolus we can, within limits due to the problem complexity, reason on and automate component activation; in Fractal an external reasoner will have to stop at API invocation, without knowledge of what a specific method implementation will do. Also, in Aeolus component states are not limited to active/inactive, i.e., each component type can define its own life-cycle in detail.

[3] In order to impose requirements only on the final configuration one can duplicate every state q of the components by adding a new corresponding state q'. The state q' will then provide and require the same ports of q, and can be reached from q via a transition. Intuitively, q' states should be the ones that the components reach at the end of the plan. A weak requirement of state q in this case is modelled as a requirement of state q'. To impose that at the end of the deployment run only states q' are present it is enough to add to every q state a provide port that is in conflict with the target state of the desired component.

A declarative approach similar to Aeolus for modelling individual components of a system, together with their possible configuration states, is also introduced in the position paper [29]. However, the lack of a formal semantics for the approach makes impossible to analyse the complexity of the deployment problem in their setting. Moreover, as observed by the authors, their approach allows administrators to write erroneous models presenting deadlocks or livelocks that are difficult to detect and forbid the reaching of the desired target configuration.

5.2. Tools

The complementary tools Zephyrus and Metis, available at [30], are directly related to the Aeolus model. Zephyrus [31] tackles the problem of computing a valid system configuration (according to the Aeolus model), starting from an existing configuration, a universe of available component types, and a formal specification that captures user desiderata for the target system. Furthermore, Zephyrus also takes into account limited machine resources, such as CPU, memory, bandwidth, etc. The computation is done via translation to a set of integer constraints, plus a dedicated algorithm to compute bindings, and the approach makes it is possible to add an objective function that can be minimized or maximized to optimize the resulting configuration.

Metis [32,33] tackles instead the problem of quickly computing a plan that migrates a valid Aeolus configuration into a different one (possibly synthesized by Zephyrus). The authors consider a model similar to Aeolus$^-$ without the possibility of using multi-state changes and propose an algorithm to compute a deployment run to reach a given target state in a specific component. The algorithm has been proven to be sound, complete, and polynomial in time w.r.t. the size of the encoding of the components in the universe.

In the same area of Zephyrus and Metis we find various tools, coming from both industry and academia.

In [7] the ConfSolve tool is presented by showing how constraint solving techniques can be used 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. A final configuration is then computed by translating the declarative specification into a constraint optimization problem which is then solved by using a constraint solver. Similarly to Zephyrus, but differently from Metis, ConfSolve does not consider the problem of synthesizing a low-level plan to reach the final configuration.

Off-the shelf planning solvers are exploited in [34,35] to generate automatically the actions to reconfigure a system. To use these tools, however, all the deployment actions with theirs preconditions and effects need to be properly specified. This hinders the usability of these kinds of tools. Indeed, to use them, system administrators are forced to translate their requirements and specifications into a formalism similar to the one used by the Planning Domain Definition Language (i.e., the *de facto* standard language for classical planners). Aeolus does not relieve administrators from the need of expressing the dynamic behaviour of components, but allows to do so more succinctly, and in a formalism that is independent from low-level planning tools.

Two recent efforts, Juju and Engage, are similar to Zephyrus, but they both avoid the problem of dealing with conflicts. In Juju [4], each service is deployed on a single machine (or, more recently, in a virtual container). This avoids the issue of component incompatibilities, but does so at the price of potentially wasting resources. Engage [36] relies on a solver to plan deployments, but offers no support for conflicts in the specification language: one can only indicate that a service can be realized by exactly one out of a list of components. Furthermore, neither Juju nor Engage—or any 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.

6. Conclusions

The Aeolus formalism is a component model designed specifically to capture most common deployment scenarios for distributed software applications in the cloud. It allows to study formally the operations that are needed to deploy complex applications on modern computing infrastructure.

In this work, we have provided precise complexity results for several variants of the Aeolus component model. We have shown that it is possible to generate a deployment plan in polynomial time for the fragment *Aeolus*$^-$ that corresponds to the limited industrial tools currently in use.

Adding support for defining conflicts among components corresponds to using the fragment *Aeolus core*. We have shown that, within such fragment, it is still possible to automatically generate a plan, at the price of a very high worst case complexity, as the problem becomes Ackermann-hard.

As for the generation of deployment plans in the full generality of the *Aeolus* model, we have presented an undecidability result, which provides a clear limit to what automated tools can do.

We plan to investigate realistic restrictions on the *Aeolus* model for which efficient reconfigurability algorithms could still be devised. For instance, one can consider imposing an upper limit on the number of resources that can be allocated during a deployment run, or investigate the impact of restricting the shape or size of the internal state machine of the components.

It will also be interesting to extend the *Aeolus* model to take into account nested components or administrative domains and explore the impact of such extensions on the complexity of the generation of deployment plans.

From a practical point of view, we started to create a repository of software components with their *Aeolus* metadata. In particular, our industrial partner Mandriva is currently enriching the description of the packages used in their industrial products to automatically deploy them using Aeolus technologies.

Acknowledgments

This paper is an extended and revised version of [10,11]—a detailed comparison with these papers is reported in the related work Section 5. This work has been developed within the ANR-2010-SEGI-013-01 project Aeolus "Mastering the Complexity of the Cloud": we would like to thank all the project partners for the numerous discussions that contributed to the definition of the Aeolus model. This work has been partially performed at IRILL, center for Free Software Research and Innovation in Paris, France, http://www.irill.org.

References

[1] L. Kanies, Puppet: next-generation configuration management, ;login: 31 (1) (2006) 19–25.
[2] Opscode, Chef, http://www.opscode.com/chef/.
[3] Cloud Foundry, http://cloudfoundry.org/.
[4] Juju, devops distilled, https://juju.ubuntu.com/.
[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The FRACTAL component model and its support in Java, Softw. Pract. Exp. 36 (11–12) (2006) 1257–1284.
[6] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, J.-B. Stefani, Reconfigurable SCA applications with the FraSCAti platform, in: IEEE SCC, IEEE, 2009, pp. 268–275.
[7] J.A. Hewson, P. Anderson, A.D. Gordon, A declarative approach to automated configuration, in: Large Installation System Administration Conference, LISA'12, 2012, pp. 51–66.
[8] P. Abate, R. Di Cosmo, R. Treinen, S. Zacchiroli, MPM: a modular package manager, in: 14th Symposium on Component Based Software Engineering, CBSE'11, ACM, 2011, pp. 179–188.
[9] K. Schmid, A. Rummler, Cloud-based software product lines, in: SPLC, ACM, 2012, pp. 164–170.
[10] R. Di Cosmo, S. Zacchiroli, G. Zavattaro, Towards a formal component model for the cloud, in: SEFM 2012, in: Lect. Notes Comput. Sci., vol. 7504, Springer, 2012, pp. 156–171.
[11] R. Di Cosmo, J. Mauro, S. Zacchiroli, G. Zavattaro, Component reconfiguration in the presence of conflicts, in: The 40th International Colloquium on Automata, Languages and Programming, ICALP 2013, in: Lect. Notes Comput. Sci., vol. 7966, Springer-Verlag, 2013, pp. 187–198.
[12] R. Di Cosmo, P. Trezentos, S. Zacchiroli, Package upgrades in FOSS distributions: details and challenges, in: HotSWup'08, 2008.
[13] K.-K. Lau, Z. Wang, Software component models, IEEE Trans. Softw. Eng. 33 (10) (2007) 709–724.
[14] M. Minsky, Computation: Finite and Infinite Machines, Prentice Hall, 1967.
[15] P.A. Abdulla, K. Cerans, B. Jonsson, Y.-K. Tsay, General decidability theorems for infinite-state systems, in: LICS, IEEE, 1996, pp. 313–321.
[16] A. Finkel, P. Schnoebelen, Well-structured transition systems everywhere!, Theor. Comput. Sci. 256 (2001) 63–92.
[17] L.E. Dickson, Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors, Am. J. Math. 35 (4) (1913) 413–422.
[18] P. Schnoebelen, Revisiting Ackermann–Hardness for lossy counter machines and reset Petri nets, in: MFCS, in: Lect. Notes Comput. Sci., vol. 6281, Springer, 2010, pp. 616–628.
[19] M. Catan, R.D. Cosmo, A. Eiche, T.A. Lascu, M. Lienhardt, J. Mauro, R. Treinen, S. Zacchiroli, G. Zavattaro, J. Zwolakowski, Aeolus: mastering the complexity of cloud application deployment, in: ESOCC, in: Lect. Notes Comput. Sci., vol. 8135, Springer, 2013.
[20] Aeolus: mastering the complexity of the cloud, http://www.aeolus-project.org/.
[21] L. de Alfaro, T.A. Henzinger, Interface automata, in: ESEC/SIGSOFT FSE, 2001.
[22] C.A. Petri, Kommunikation mit Automaten, PhD thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.
[23] C. Rackoff, The covering and boundedness problems for vector addition systems, Theor. Comput. Sci. 6 (1978) 223–231.
[24] R. Milner, J. Parrow, D. Walker, A Calculus of Mobile Processes, I/II, Inf. Comput. 100 (1) (1992) 1–77.
[25] F. Achermann, O. Nierstrasz, A calculus for reasoning about software composition, Theor. Comput. Sci. 331 (2–3) (2005) 367–396.
[26] A. Schmitt, J.-B. Stefani, The Kell calculus: a family of higher-order distributed process calculi, in: Global Computing, in: Lect. Notes Comput. Sci., vol. 3267, Springer, 2004, pp. 146–178.
[27] M. Bundgaard, T.T. Hildebrandt, J.C. Godskesen, A CPS encoding of name-passing in higher-order mobile embedded resources, Theor. Comput. Sci. 356 (3) (2006) 422–439.
[28] F. Montesi, D. Sangiorgi, A model of evolvable components, in: TGC, in: Lect. Notes Comput. Sci., vol. 6084, Springer, 2010, pp. 153–171.
[29] P. Goldsack, P. Murray, A. Farrell, P. Toft, SmartFrog and data centre automation, Tech. rep., Microsoft Research, 2008.
[30] Aeolus tools, https://github.com/aeolus-project/.
[31] R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, Optimal provisioning in the cloud, Tech. rep., Aeolus project http://hal.archives-ouvertes.fr/hal-00831455, June 2013.
[32] T.A. Lascu, J. Mauro, G. Zavattaro, Automatic component deployment in the presence of circular dependencies, in: The 10th International Symposium on Formal Aspects of Component Software, FACS, 2013.
[33] T.A. Lascu, J. Mauro, G. Zavattaro, A planning tool supporting the deployment of cloud applications, in: ICTAI, IEEE, 2013, pp. 213–220.
[34] H. Herry, P. Anderson, G. Wickler, Automated planning for configuration changes, in: LISA, USENIX Association, 2011.
[35] H. Herry, P. Anderson, Planning with global constraints for computing infrastructure reconfiguration, in: CP4PS-12—The AAAI-12 Workshop on Problem Solving Using Classical Planners, 2012.
[36] J. Fischer, R. Majumdar, S. Esmaeilsabzali, Engage: a deployment management system, in: Programming Language Design and Implementation, PLDI'12, ACM, 2012, pp. 263–274.