On the Expressiveness of Synchronization in Component Deployment

Jacopo Mauro[1] and Gianluigi Zavattaro[2(✉)]

[1] Department of Informatics, University of Oslo, Oslo, Norway
[2] Department of Computer Science and Engineering, University of Bologna/INRIA FoCUS, Bologna, Italy
gianluigi.zavattaro@unibo.it

Abstract. The Aeolus component problem of automatic deployment of complex distributed component systems. In the general setting, the task of checking if a distributed application can be deployed is an undecidable problem. However, the current undecidability proof in Aeolus assumes the possibility to perform in a synchronized way atomic configuration actions on a set of interdependent components: this feature is usually not supported by deployment frameworks. In this paper we prove that even without synchronized configuration actions the Aeolus component model is still Turing complete. On the contrary, we show that other Aeolus features like capacity constraints and conflicts are necessary: if we remove the former the deployment problem becomes non-primitive recursive, while in the latter it becomes poly-time.

1 Introduction

Expressiveness of models for concurrent computation is one of those interest that accompanied Frank de Boer in his extremely productive and diversified research activity. For instance, in the early 90's he investigated the use of embedding as a tool for concurrent language comparison [6] and more recently he considered decidability/undecidability of termination problems to evaluate the expressiveness of basic features of the Actor concurrency model [5]. This paper falls in this line of research, by considering the Aeolus component model tailored to the analysis of automatic component deployment. The specific contribution of this paper is the study of the expressiveness of a specific mechanism for component configuration used to synchronously configure interdependent components. This is useful, for instance, when two components are mutually dependent and the easiest way to deploy them is to install them contemporaneously. Especially in a distributed component environment, such a synchronized installation is of difficult implementation. For this reason we have decided to investigate the impact of the elimination of this mechanism from the Aeolus model.

Supported by the EU projects FP7-610582 *Envisage: Engineering Virtualized Services* (http://www.envisage-project.eu) and FP7-644298 *HyVar: Scalable Hybrid Variability for Distributed, Evolving Software Systems* (http://www.hyvar-project.eu).

E. Ábrahám et al. (Eds.): de Boer Festschrift, LNCS 9660, pp. 344–359, 2016.
DOI: 10.1007/978-3-319-30734-3_23

The Aeolus component model has been proposed in [4,8] as a formal model to reason on the component deployment problem. Deployment and management of modern large scale component-based applications is a challenging task, and several tools and technologies are under development to support application architects and managers in these complex activities. According to the current mainstream approaches, such applications are either deployed by exploiting pre-configured virtual machines images, which already contain all the needed software components (see, e.g., Bento Boxes [9], Cloud Blueprints [3], or AWS CloudFormation [2]), or are designed by using drag-and-drop graphical tools like Juju [10] leaving the low-level component configuration to pre-programmed scripts or to automatic configuration tools like Puppet [16] or Chef [15].

Aeolus extends the classical notion of component, seen as a black-box that exposes *provide* and *require* ports, with a finite state automaton describing the component configuration life-cycle. The automaton states correspond to different configuration modalities, like *uninstalled*, *installed*, *running*, *stopped*, etc. and the transitions represent configuration actions like install, run, stop, etc. Depending on the internal state, the ports on the interface can be either active or inactive. For instance, an *uninstalled* component usually does not activate any require port, while it can activate require ports when it is in the *installed* state, and finally activate some provide port when it actually enters the *running* state. Another specific feature of the Aeolus model is that capacity constraints can be associated to the ports: a provide port could have a maximal number of connected require ports or a require port can ask for multiple providers offering a given functionality.[1] Additionally, in Aeolus it is also possible to express conflicts: components can activate special ports that forbids the activation of provide ports of a given type in the rest of the system.

In [4,8] we have investigated the expressiveness of the Aeolus component model, showing that it is Turing complete. From this expressiveness result we have, as a negative consequence, that in general the component deployment problem is not computable. More precisely, we proved the undecidability of the *achievability* problem. Given a finite universe of component types, a target component and a target state, the achievability problem consists of deciding if it possible to reach a final configuration containing at least one instance of the target component type in the target state by assuming the availability of an unbounded number of instances of the component types of the universe in their initial state.

The undecidability of achievability is proved by encoding counter machines, from which follows the Turing completeness of the Aeolus model. The proof relies on a specific feature of Aeolus called *multiple state change*: if there is a group of components that reciprocally depend one on another to advance in their internal configuration life-cycle, it is possible to synchronously and atomically change their state to allow all of them to progress. This specific feature of the model is clearly of non trivial implementation in distributed component systems

[1] This feature of the model is used to capture replication or fault tolerance requirements.

since it would require distributed synchronization and is usually not supported by deployment frameworks.

In this paper we investigate the impact of the removal of multiple state changes on the expressiveness of the Aeolus component model. The main result is that the model is still Turing complete, thus showing that the undecidability of achievability follows from its intrinsic complexity, and not from the expressive power of distributed synchronization.

As additional results, we show that this Turing completeness result relies on both capacity constraints and conflicts. In fact, if we remove at least one of these two features, achievability turns out to be decidable. In particular, if we remove conflicts it becomes poly-time, while if we remove capacity constraints (keeping conflicts), it turns out to be decidable but non-primitive recursive.

Comparison with Previous Work. The Aeolus component model was initially proposed in [8] where its Turing completeness was proved. In that paper, also the fragment of the Aeolus model without conflicts and capacity constraints was considered, showing that the achievability problem is poly-time for that fragment. In [7] the fragment without capacity constraints was studied, showing that the problem is decidable; its Ackerman-hardness was proved in [4]. A fragment of the Aeolus model without multiple state change (and without capacity constraints and conflicts) has been considered in [11,12] where a tool for automatic cloud application deployment was presented. In this paper we complete the analysis of the remaining relevant fragments without multiple state changes.

Structure of the Paper. In Sect. 2 we report the formal definition of the Aeolus component model following [4]. Turing completeness without multiple state change actions is proved in Sect. 3. In Sects. 4 and 5 we consider the two fragments obtained by removing, besides multiple state change actions, also capacity constraints or conflicts, respectively. Finally, in Sect. 6 we draw some concluding remarks.

2 The Aeolus Model

In this section we give a recap of the *Aeolus model* following [4].

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 component types 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 active ports changes depending on the internal state. A provide port represents the possibility of furnishing a functionality having a given interface. Similarly, a require port represents the need for 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*;*
- *$P = \langle \mathbf{P}, \mathbf{R}\rangle$, 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 interface.

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 ports (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).

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.*

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 denote with $\mathcal{C}^{\#}_{\langle \mathcal{T}, q \rangle}$ the number of components of type $\mathcal{T}$ in state q in the configuration $\mathcal{C}$.

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* $1 \leq i \leq n$ *we have that* $\langle r, z, z_i \rangle \in B$*,* $\mathcal{C}[z_i] = \langle \mathcal{T}^i, q^i \rangle$ *and* r *is in the domain of* $\mathcal{T}^i.\mathbf{P}(q^i)$.

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* $1 \leq i \leq m$ *we have that* $\langle p, z_i, z \rangle \in B$*,* $S(z_i) = \langle \mathcal{T}^i, q^i \rangle$ *and* p *is in the domain of* $\mathcal{T}^i.\mathbf{R}(q^i)$.

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

We now formalize how configurations evolve from one state to another one, 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 system 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:*

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

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

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

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

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

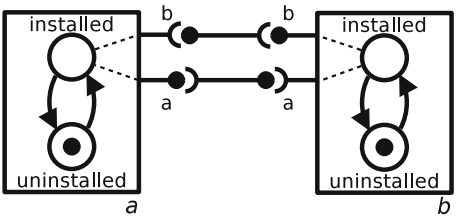

Fig. 1. On the need of a *multiple state change*: how to install a and b?

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 as later detailed. Also, we observe that there are configurations that cannot be reached through sequences of the actions we have introduced. In Fig. 1, for instance, there is no way for package `a` and `b` to reach the installed state, as each package requires the other one 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, more often, they abort deployment.

The Aeolus model allows for simultaneous installations by introducing the notion of *multiple state change*.

Definition 7 (Multiple State Change). *A* multiple state change *action* $\mathcal{M} = \{stateChange(z^1, q_1^1, q_2^1), \cdots, stateChange(z^l, q_1^l, q_2^l)\}$ *is a set of actions of type* state change *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,*

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

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

multiple 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* $\mathcal{C}_{j-1} \xrightarrow{\alpha} \mathcal{C}'$*.*

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* $\alpha_1 \ldots \alpha_m$ *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****.*

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.

3 Turing Completeness Without Multiple State Changes

In [4,8] it is proved that the *Aeolus* component model is Turing complete. More precisely, we show how to reduce termination for 2 Counter Machines [14], a well-known Turing-complete computational model, in the achievability problem for the *Aeolus* component model. The presented reduction makes use of the multiple state change actions. In this section, we revisit that proof, showing that multiple state changes are not strictly necessary.

Before entering into the details, 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. To avoid the component deletion it is sufficient to impose a reciprocal dependence with a new auxiliary type of components. When this dependence is established the components cannot be deleted without violating configuration correctness. In [4] this reciprocal dependence was established via a multiple state change. However, multiple state changes are not the only way to enforce the persistence of an instance since the reciprocal dependence can be established by creating one binding at the time following a precise protocol.

In Fig. 2 we show an example of how a component type can be modified in order to reach our goal. Three new auxiliary states q'_0, q'_a, and q'_b are created, with q'_0 becoming the new initial state. States q'_a and q'_b require and provide respectively ports a and b. Only one instance of $\mathcal{T}$ can be present at once in these two states. This is enforced by simultaneously providing and requiring the port e. The original states of $\mathcal{T}$ are modified to require the port a and provide the port b. Dually, the auxiliary component $\mathcal{T}_{aux}$ has an initial state q_0, a final one q_f, and two intermediate states q_a and q_b providing and requiring respectively ports a and b. Also in this case, at most one instance of $\mathcal{T}_{aux}$ can be in states q_a or q_b. We assume that the ports a, b, e and f are fresh, that is, they are not used by any other component type in the considered universe.

Given a component type $\mathcal{T}$ we denote this component type transformation with $\varphi(\mathcal{T})$. The φ transformation is defined to guarantee the establishment of

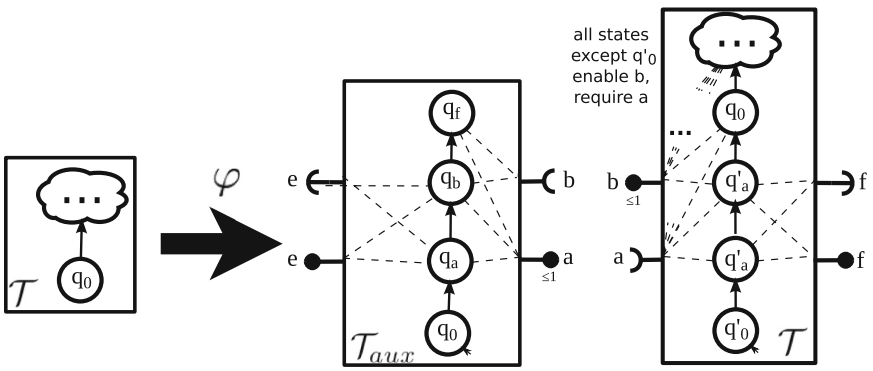

Fig. 2. Component type transformation φ.

two reciprocal bindings between pairs of $\mathcal{T}$ and $\mathcal{T}_{aux}$ instances that forbid their deletion. In particular, an instance of $\mathcal{T}$ cannot be deleted after the state q'_b was left while the instance of $\mathcal{T}_{aux}$ cannot be deleted after state q_a. This is due to the fact that after these states are left the instances are guaranteed to provide something required by the other instance. If we denote with $\mathcal{C}^{\#}_{\langle p \rangle}$ the number of instances providing the port p, this property is captured as follows.

Property 1 (φ-persistence). A configuration $\mathcal{C}$ is φ-persistent if $\mathcal{C}^{\#}_{\langle a \rangle} - \mathcal{C}^{\#}_{\langle \mathcal{T}_{aux}, q_a \rangle} = \mathcal{C}^{\#}_{\langle b \rangle} - \mathcal{C}^{\#}_{\langle \mathcal{T}, q'_b \rangle}$.

The encoding φ preserves the φ-persistence.

Lemma 1. *If $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ and $\mathcal{C}$ is φ-persistent than also $\mathcal{C}'$ is φ-persistent.*

Proof. The proof can be done considering the type of α actions. Since φ-persistence considers just the amount of active ports of type a and b we can restrict to consider only actions that can alter these quantities.

If $\alpha = \mathit{stateChange}(z, q_0, q_a)$ a new port a is provided but at the same time an instance of type $\mathcal{T}_{aux}$ leaving the quantity $\mathcal{C}^{\#}_{\langle a \rangle} - \mathcal{C}^{\#}_{\langle \mathcal{T}_{aux}, q_a \rangle}$ unaltered. The same happens with $\alpha = \mathit{stateChange}(z, q_a, q_b)$ and for port b when $\alpha = \mathit{stateChange}(z, q'_a, q'_b)$ or $\mathit{stateChange}(z, q'_b, q_0)$.

When a and b are provided, by construction, the only way to reduce their amount is by deleting a component. A deletion of an instance of type $\mathcal{T}_{aux}$ in state q_0, q_a or the deletion of type $\mathcal{T}$ in states q'_0, q'_a, q'_b does not alter the amount of a or b ports provided. A deletion of an instance z of $\mathcal{T}$ or $\mathcal{T}_{aux}$ in a different state q' is not possible. In fact, if z is of type $\mathcal{T}$ than its deletion reduces $\mathcal{C}^{\#}_{\langle b \rangle}$ by 1. But since there are exactly $\mathcal{C}^{\#}_{\langle a \rangle} - \mathcal{C}^{\#}_{\langle \mathcal{T}_{aux}, q_a \rangle} = \mathcal{C}^{\#}_{\langle b \rangle} - \mathcal{C}^{\#}_{\langle \mathcal{T}, q'_b \rangle}$ instances requiring a port b the deletion of z violates the configuration correctness. Similarly, the configuration correctness is violated also if z is of type $\mathcal{T}_{aux}$. □

We can therefore consider, without loss of generality, components that can be deployed in a persistent way. This can lead to a modification of the proof of the undecidability of achievability for Aeolus [4] that does not assume to use multiple state changes. The original prove was by reduction from the termination 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.

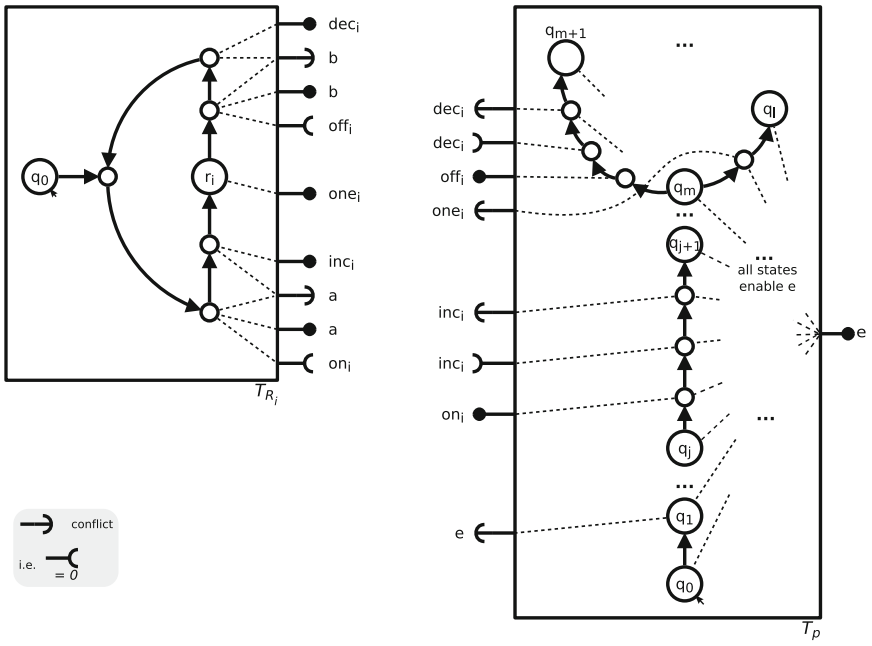

Fig. 3. Modeling 2 counter machines (2CMs) in the Aeolus model.

For modelling 2CMs a component to simulate the execution of the program instructions was used. The content 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. In particular, as depicted in Fig. 3, a component type $\mathcal{T}_P$ was used to simulate the execution of the program instructions while $\mathcal{T}_{R_1}$ and $\mathcal{T}_{R_2}$ were used for the two registers. All these components where made persistent by forcing the initial execution of multiple state changes creating reciprocal bindings with an additional component. However, the same can be obtained without the use of the multiple state change simply by applying the φ transformation.

We can therefore prove a stricter undecidability result.

Theorem 1. *The achievability problem is undecidable in the fragment of the Aeolus component model that does not support multiple state changes.*

Proof. The proof follows the same technique as the one used in [4] by considering the universe $U = \varphi(\mathcal{T}_P) \cup \varphi(\mathcal{T}_{R_1}) \cup \varphi(\mathcal{T}_{R_2})$. □

4 Ackermann-Hardness Without Capacity Constraints and Multiple State Changes

In [4] the achievability problem was proven to be decidable but Ackermann-hard in the fragment of Aeolus without capacity constraints. Even in this case the complexity does not decrease if we also remove multiple state changes.

The complexity result in [4] was obtained by reduction from the coverability problem in reset Petri nets, a problem which is indeed known to be Ackermann-hard [17].

We start with some background on reset Petri nets.

A *reset Petri net* RN is a tuple $\langle P, T, \boldsymbol{m_0}\rangle$ such that P is a finite set of *places*, T is a finite set of *transitions*, and $\boldsymbol{m_0}$ 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 $\boldsymbol{m}$. Transition t is enabled at marking $\boldsymbol{m}$ iff ${}^\bullet t(p) \leq \boldsymbol{m}(p)$ for each $p \in P$. Firing t at $\boldsymbol{m}$ leads to a new marking $\boldsymbol{m'}$ defined as $\boldsymbol{m'}(p) = \boldsymbol{m}(p) -^\bullet t(p) + t^\bullet(p)$ if $p \notin t\downarrow$, and $\boldsymbol{m'}(p) = 0$ otherwise; we denote this marking transformation with $\boldsymbol{m} \mapsto \boldsymbol{m'}$. A marking $\boldsymbol{m}$ is reachable from $\boldsymbol{m_0}$ if $\boldsymbol{m_0} \mapsto^* \boldsymbol{m}$, i.e., it is possible to produce $\boldsymbol{m}$ after firing finitely many times transitions in T. Given a reset net $\langle P, T, \boldsymbol{m_0}\rangle$ and a marking $\boldsymbol{m}$, the coverability problem consists in checking for the existence of a reachable marking $\boldsymbol{m'}$ such that $\boldsymbol{m} \leq \boldsymbol{m'}$, i.e. $\boldsymbol{m}(p) \leq \boldsymbol{m'}(p)$ for every $p \in P$. In [17] it is proved that the coverability problem for reset nets is Ackermann-hard.

The encoding of reset Petri nets into Aeolus presented in [4] relied on three types of component types: $\mathcal{T}_p$ for modelling the tokens, $\mathcal{T}_t$ for the transitions, and $log(n)$ component types $\mathcal{T}_{C_i}$ (for $1 \leq i \leq log(n)$) for modeling the bits in a binary counter used to count the tokens to be produced or consumed during the simulation of a transition firing. Here n is the maximal number of tokens that one transition can produce or consume. The proof technique requires that the components for the transitions and the counter bits are unique and persistent. This was ensured via a transformation that exploited conflicts but also multiple state changes. This however, following the example of the φ transformation defined in the previous section, can be obtained also without the use of multiple state changes. The key idea to avoid a multiple state change is to create a mutual dependency between pairs of components in two phases and use the conflicts to ensure that only one component at a time can be present.

Figure 4 depicts the transformation η that guarantees persistence and forbids the presence of two distinct instances of the same component. This is obtained by requiring that all the states except the initial ones activate contemporaneously a require and a conflict port on the same interface (in the two component types in the Figure we use the fresh interfaces e and f, respectively). The interdependencies among the two component types are similar to those used in the φ transformation to guarantee persistency. The unique difference is that no capacity constraint is considered; indeed, this is no longer needed because, thanks to the simultaneous requirement and conflict on the interfaces e and f, and the

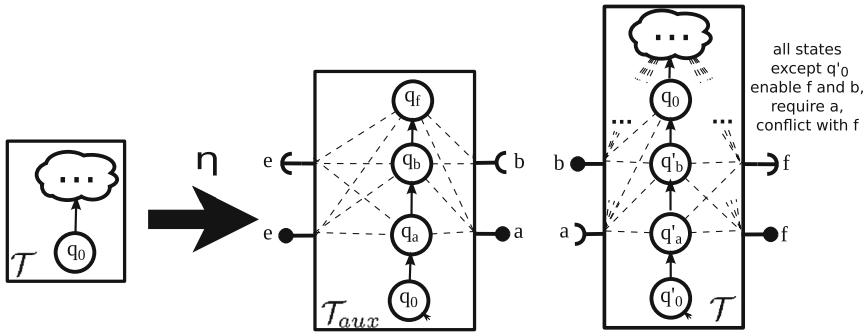

Fig. 4. Component type transformation η.

freshness of the ports a and b, a configuration has at most one a and one b provide port active.

Note that the size of $\eta(\mathcal{T})$ is polynomial w.r.t. the size of $\mathcal{T}$ because we are just introducing a new component type $\mathcal{T}_{aux}$ of constant size and we modify $\mathcal{T}$ by adding three new states and three ports.

We can now conclude the new version of the Ackermann-hardness result for the fragment of Aeolus without capacity constraints and multiple state changes.

Theorem 2. *The achievability problem is Ackermann-hard for the fragment of the Aeolus component model that does not support capacity constraints and multiple state changes.*

Proof. The proof follows the same technique as the one used in [4] with the difference that now the encoding of the Petri net $RN = (P, T, m_0)$ is $\Gamma_{RN} = \{\mathcal{T}_p \mid p \in P\} \cup \{\eta(\mathcal{T}_{C_i}) \mid i \in [1..\lceil \log(n) \rceil]\} \cup \{\eta(\mathcal{T}_T)\}$ where n is the largest number of tokens that can be consumed or produced by a transition in T and η is the transformation depicted in Fig. 4. □

5 Poly-time Without Conflicts and Multiple State Changes

In [12] it was proven that the achievability problem is poly-time considering the fragment of Aeolus where no capacity constraint, no multiple state changes, and no conflicts can be used.

This was done by means of an algorithm that builds a reachability graph used to check whether a given target component-state pair may be obtained. As detailed in Algorithm 1, the nodes of the reachability graph are organized in layers $Nodes_0$, $Nodes_1$, $\cdots$, $Nodes_n$ that are generated in subsequent phases. Initially, $Nodes_0$ contains all the pairs $\langle \mathcal{T}, \mathcal{T}.\texttt{init} \rangle$ corresponding to the initial states. Given $Nodes_j$, $Nodes_{j+1}$ is generated by copying the pairs already available in $Nodes_j$ and by adding those new pairs that can be reached by transitions

from states in $Nodes_j$, assuming the availability in the context of components of type and state $\langle \mathcal{T}, q \rangle$ already in $Nodes_j$. The reachability analysis terminates since there is a finite number of possible component type-state pairs.

Luckily, the same reachability technique can be used to decide achievability also when capacity constraints can be used. Indeed, we prove that at each layer $Nodes_j$ a sufficient number of components can be created to satisfy all the constraints that will be activated by the new components at layer $Nodes_{j+1}$. The reachability algorithm is therefore correct also in the presence of capacity constraints.

Algorithm 1. REACHABILITYANALYSIS()

1: $Nodes_0 = \{\langle \mathcal{T}, \mathcal{T}.\texttt{init} \rangle \mid \mathcal{T} \in U\}$; $provPort = \bigcup_{\langle \mathcal{T},q \rangle \in Nodes_0} \{\mathcal{T}.\mathbf{P}(q)\}$; $n = 0$
2: **repeat**
3: $\quad n = n + 1$
4: $\quad Arcs_n = \emptyset; Nodes_n = \emptyset$
5: $\quad$ **for all** $\langle \mathcal{T}, q \rangle \in Nodes_{n-1}$ **do**
6: $\quad\quad$ **for all** $(q, q') \in \mathcal{T}.\texttt{trans}$ **do**
7: $\quad\quad\quad$ **if** $\mathcal{T}.\mathbf{R}(q') \subseteq provPort$ **then**
8: $\quad\quad\quad\quad Nodes_n.\texttt{add}(\langle \mathcal{T}, q' \rangle)$
9: $\quad$ **for all** $\langle \mathcal{T}, q \rangle \in Nodes_n$ **do**
10: $\quad\quad provPort.\texttt{add}(\mathcal{T}.\mathbf{P}(q))$
11: $\quad Nodes_n = Nodes_{n-1} \cup Nodes_n$
12: $\quad$ **for all** $\langle \mathcal{T}, q \rangle \in Nodes_{n-1}, \langle \mathcal{T}, q' \rangle \in Nodes_n$ **do**
13: $\quad\quad$ **if** $(q, q') \in \mathcal{T}.\texttt{trans}$ **then**
14: $\quad\quad\quad Arcs_n.\texttt{add}(\langle \mathcal{T}, q' \rangle \longrightarrow \langle \mathcal{T}, q \rangle)$
15: $\quad\quad$ **if** $q == q'$ **then**
16: $\quad\quad\quad Arcs_n.\texttt{add}(\langle \mathcal{T}, q' \rangle \cdots\cdots \langle \mathcal{T}, q \rangle)$
17: **until** $Nodes_{n-1} == Nodes_n$

Lemma 2. *Given a universe of components U, a component type $\mathcal{T}_{target}$, and a state q_{target}, we have that $\langle \mathcal{T}, q \rangle$ belongs to the reachability graph computed by Algorithm 1 if and only if there exists a deployment plan that deploys at least one component of type $\mathcal{T}$ in state q.*

Proof. We first consider the *only if* part. We prove that given $h_{\langle \mathcal{T},q \rangle} > 0$ for every $\langle \mathcal{T}, q \rangle \in Nodes_n$, and given $h_p > 0$ for every provide port p activated by the component type-state pairs $\langle \mathcal{T}, q \rangle \in Nodes_n$, there exists a deployment plan from an empty configuration to a configuration containing at least $h_{\langle \mathcal{T},q \rangle}$ components of type $\mathcal{T}$ in state q, in which at least h_p provide ports with interface p have no incoming bindings (thus they are available to satisfy additional complementary require port with interface p). We proceed by induction on n.

The base case holds because $Nodes_0$ contains all the pairs with just initial states (Line 1) and components could always be created in their initial state because they have no requirements by definition.

In the inductive case we have that for every pair $\langle \mathcal{T}, q \rangle \in Nodes_{i+1} \setminus Nodes_i$ there exists a pair $\langle \mathcal{T}, q' \rangle \in Nodes_i$ where q' is a predecessor of q (Lines 5–8 of Algorithm 1). Moreover, for every require port r activated by the components of type-state $\langle \mathcal{T}, q \rangle \in Nodes_{i+1} \setminus Nodes_i$ there exists a pair $\langle \mathcal{T}'', q'' \rangle \in Nodes_i$ that activates a provide port r (Line 7).

By inductive hypothesis it is possible to obtain a configuration such that:

1. for every $\langle \mathcal{T}, q' \rangle \in Nodes_i$ it has at least

$$h_{\langle \mathcal{T}, q' \rangle} + \sum_{\langle \mathcal{T}, q \rangle \in Nodes_{i+1} \setminus Nodes_i} (h_{\langle \mathcal{T}, q \rangle} + max_p)$$

   components of type $\mathcal{T}$ in state q'. By max_p we mean the maximal value among all the h_p;
2. for every provide port p activated by a component type-state pair $\langle \mathcal{T}'', q'' \rangle \in Nodes_i$ in $Nodes_i$ there are at least

$$h_p + (maxRequire_p + 1) \times \sum_{\langle \mathcal{T}, q \rangle \in Nodes_{i+1} \setminus Nodes_i} (h_{\langle \mathcal{T}, q \rangle} + max_p)$$

   active provide port p that have no incoming binding. By $maxRequire_p$ we mean the maximal number of provide ports with interface p that are necessary to satisfy the requirements of one component instances of any possible type, in any possible state.

Thanks to point 1, starting from this configuration it is possible to perform for every pair $\langle \mathcal{T}, q \rangle$ of $Nodes_{i+1} \setminus Nodes_i$ exactly $h_{\langle \mathcal{T}, q \rangle} + max_p(h_p)$ state changes to obtain components of type $\mathcal{T}$ in state q. Indeed, point 2 guarantees there are enough free provide ports to satisfy the requirements of these components. Moreover, if the new state q activates new provide ports that were inactive in the previous state, we have the guarantee that it is possible to unbind these ports so that these components will have no incoming binding. Hence, for these interfaces p we will have at least h_p free provide ports.

Concerning pairs $\langle \mathcal{T}'', q'' \rangle \in Nodes_{i+1} \cap Nodes_i$, thanks to point 1 the reached configuration will contain at least $h_{\langle \mathcal{T}'', q'' \rangle}$ instances of type $\mathcal{T}''$ in state q''. By point 2, we also have the guarantee that for provide ports p activated by these pairs $\langle \mathcal{T}'', q'' \rangle$ at least h_p instances remain free in the new configuration. In fact, some of them (at most $maxRequire_p \times \sum_{\langle \mathcal{T}, q \rangle \in Nodes_{i+1} \setminus Nodes_i} (h_{\langle \mathcal{T}, q \rangle} + max_p)$) will be used to satisfy the requirements of the new component type-state pairs, and some other (at most $\sum_{\langle \mathcal{T}, q \rangle \in Nodes_{i+1} \setminus Nodes_i} (h_{\langle \mathcal{T}, q \rangle} + max_p)$) could become inactive due to a state change.

We now move to the *if* part. We proceed by contradiction. Let us suppose the existence of a deployment plan $\langle U, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_m} \mathcal{C}_m$ such that $\mathcal{C}_m$ contains a component of type $\mathcal{T}$ in state q while $\langle \mathcal{T}, q \rangle$ is not present in the reachability graph. It is not restrictive to assume that $\mathcal{C}_m$ is the first configuration of the plan having such property (i.e., all the pairs $\langle \mathcal{T}', q' \rangle$ of the components in $\mathcal{C}_1, \cdots, \mathcal{C}_{m-1}$ are present in the reachability graph).

Obviously q cannot be an initial state of $\mathcal{T}$ since all the component types with their initial states are added in $Nodes_0$ (Line 1). Therefore we have that the last transition of the plan is $\mathcal{C}_{m-1} \xrightarrow{stateChange(i,s,q)} \mathcal{C}_m$. This action can be executed only if all the require ports activated by q are fulfilled by components in $\mathcal{C}_{m-1}$. For the previous assumption, we have that $\langle \mathcal{T}, s \rangle$, as well as all the pairs $\langle \mathcal{T}', q' \rangle$ of types and states of components in $\mathcal{C}_{m-1}$, are part of the computed reachability graph. Let $Nodes_j$ be the first layer containing all such pairs: by construction (Lines 5–8) we will have that $\langle \mathcal{T}, q \rangle \in Nodes_{j+1}$, thus contradicting the hypothesis. □

As a consequence of Lemma 2 we have the following result.

Theorem 3. *The achievability problem is poly-time for the fragment of the Aeolus component model that does not support multiple state changes and conflicts.*

The proof immediately follows from Lemma 2 and the fact that Algorithm 1 is poly-time [12].

6 Conclusions

To the best of our knowledge Aeolus is the first formal model that is designed on purpose to address the specific problem of software component deployment in the cloud. It was first introduced in [8]. Differently from the definition of the language presented here, in [8] an additional kind of requirements—called *weak requirements*—was present. Differently from the requirements presented in this paper (formerly known as *strong requirements*) that needs to be enforced at every deployment step, weak requirements must be satisfied only at the end of a deployment run. The notion of weak requirement was removed from the model because we found out that their behavior could be simulated with normal requirements. In this paper we proved that also the notion of multiple state change can be removed because from the complexity point of view it does not have an impact. It is interesting to point out that this new foundational result reflects a recent technic adopted in the context of deployment tools for *package-based* software distributions [1] that replaces synchronous installation of circular dependent packages with multi-stage configuration protocol.

In this work we have considered the deployment of an application from scratch. If we assume the initial configuration is not empty, we move to a so-called *reconfiguration* problem. It is interesting to observe that this problem is harder than the achievability problem for the fragment without multiple state change and conflicts for which we prove in this paper that achievability is poly-time. In fact, in [13] we have recently proved that reconfiguration is PSpace complete already for the fragment without multiple state change, conflicts and capacity constraints.

References

1. Abate, P., Johannes, S.: Bootstrapping software distributions. In: CBSE 2013, pp. 131–142. ACM (2013)
2. Amazon. AWS CloudFormation. http://aws.amazon.com/cloudformation/
3. CenturyLink. Cloud Blueprints. http://www.centurylinkcloud.com/products/management/blueprints
4. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: a component model for the cloud. Inf. Comput. **239**, 100–121 (2014)
5. de Boer, F.S., Jaghoori, M.M., Laneve, C., Zavattaro, G.: Decidability problems for actor systems. Logical Meth. Comput. Sci. **10**(4:5), 1–29 (2014)
6. de Boer, F.S., Palamidessi, C.: Embedding as a tool for language comparison. Inf. Comput. **108**(1), 128–157 (1994)
7. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Component reconfiguration in the presence of conflicts. In: Fomin, F.V., Kwiatkowska, M., Peleg, D. (eds.) ICALP 2013, Part II. LNCS, vol. 7966, pp. 187–198. Springer, Heidelberg (2013)
8. Di Cosmo, R., Zacchiroli, S., Zavattaro, G.: Towards a formal component model for the cloud. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 156–171. Springer, Heidelberg (2012)
9. Flexiant. Bento Boxes. http://www.flexiant.com/2012/12/03/application-provisioning/
10. Juju, devops distilled. https://juju.ubuntu.com/
11. Lascu, T.A., Mauro, J., Zavattaro, G.: A planning tool supporting the deployment of cloud applications. In: ICTAI (2013)
12. Lascu, T.A., Mauro, J., Zavattaro, G.: Automatic component deployment in the presence of circular dependencies. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) FACS 2013. LNCS, vol. 8348, pp. 254–272. Springer, Heidelberg (2014)
13. Mauro, J., Zavattaro, G.: On the complexity of reconfiguration in systems with legacy components. In: Italiano, G.F., Pighizzini, G., Sannella, D.T. (eds.) MFCS 2015. LNCS, vol. 9234, pp. 382–393. Springer, Heidelberg (2015)
14. Minsky, M.: Computation: Finite and Infinite Machines. Prentice Hall, Englewood Cliffs (1967)
15. Opscode. Chef. http://www.opscode.com/chef/
16. Puppetlabs. Puppet. http://puppetlabs.com/
17. Schnoebelen, P.: Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets. In: Hliněný, P., Kučera, A. (eds.) MFCS 2010. LNCS, vol. 6281, pp. 616–628. Springer, Heidelberg (2010)