Automatic component deployment in the presence of circular dependencies [⋆]

Tudor A. Lascu, Jacopo Mauro, and Gianluigi Zavattaro

FOCUS team, Dept. of Computer Science/INRIA, University of Bologna.
{lascu,jmauro,zavattar}@cs.unibo.it

Abstract. In distributed systems like clouds or service oriented frameworks, applications are typically assembled by deploying and connecting a large number of heterogeneous software components, spanning from fine-grained packages to coarse-grained complex services. The complexity of such systems requires a rich set of techniques and tools to support the automation of their deployment process. By relying on a formal model of components, we describe a sound and complete algorithm for computing the sequence of actions allowing the deployment of a desired configuration. Moreover, differently from other proposals in the literature, our technique works even in the presence of circular dependencies among components. We give a proof for the polynomiality of the devised algorithm, thus guaranteeing efficiency and effectiveness of automatic tools for component deployment based on our algorithm.

1 Introduction

Deploying software component systems is becoming a critical challenge, especially due to the advent of Cloud Computing technologies that make it possible to quickly run complex distributed software systems on-demand on a virtualized infrastructure, at a fraction of the cost which was necessary just a few years ago. When the number of software components needed to run the application grows, and their interdependencies become too complex to be manually managed, it is necessary for the system administrator to use high-level languages for specifying the expected minimal system requirements, and then rely on tools that automatically synthesize the low-level deployment actions necessary to actually realize a correct and complete system configuration that satisfies such requests.

Recent works have introduced formalisms which focus on this automation aspect of the deployment process, like the Juju initiative within Ubuntu [15] or the Engage system [13]. According to the Juju approach, the system administrator decides which are the high-level services needed in the system and how they should be reciprocally connected, and then the actual deployment is realized by low-level scripts. Similarly, in Engage it is possible to indicate only the relevant services which are needed and their interdependencies, and then the entire system is automatically completed and the actual deployment is synthesized.

[⋆] Work partially supported by Aeolus project, ANR-2010-SEGI-013-01.

One of the limitations of the Engage system is that component interdependencies cannot be circular. This limitation follows from the fact that Engage synthesizes the deployment plan by performing a topological visit of a graph representing the component dependencies: the presence of cycles would forbid to complete such visit. Nevertheless, in many cases, the assumption on the absence of circular dependencies is not admissible. As a first example, we can mention package-based software distributions where circularities are frequent (see [9] for a list of circular dependencies among packages in Debian). Another example of circularity is between replicated database services. For instance, in order to realize a MySQL master-slave replication [4], the master needs from the slave some authentication information (like the IP address), while the slave needs to receive from the master a dump of the database.

In this paper, we address the problem of automatic synthesis of deployment plans in the presence of component circular dependencies. To study the problem we consider the Aeolus component model [12], that enriches traditional component models, based on require/provide ports, with an internal *state machine* that describes the component life-cycle. Each internal state can activate only some of the ports at the component interface. Automatizing a deployment plan consists in specifying a sequence of low-level actions like creation/deletion of components, port binding/unbinding, and internal state changes, that reaches a configuration with at least one component in a specific target internal state. The Aeolus model has been introduced to study the computational boundaries of deployment automatization. In the full Aeolus model it is possible to specify *conflicts* among components and also *capacity constraints*, i.e. for each provided port how many requirements it can satisfy, and for each require port how many different instances of a complementary provide port are needed. In [12] we have proved that the deployment problem is undecidable for the full Aeolus model. On the contrary, if capacity constraints are not considered, we have proved in [11] that the problem turns out to be decidable, but it is Exp-Space hard. In order to allow efficient algorithms for automatic deployment, in this paper we further simplify the Aeolus model by removing also conflicts. Juju and Engage also, abstract away from conflicts and this is useful, for instance in Engage, to complete partial configurations simply by adding new components without having to check whether these are incompatible with already present components.

Paper contribution. The novel solution for automatic component deployment that we propose in this paper is based on an algorithm divided in three distinct phases. In a first phase the existence of a plan is checked by performing a forward symbolic *reachability analysis* of all possible reachable states of the components. If the target state is reachable, a second phase of *abstract planning* generates a graph that indicates the kinds of internal state change actions that are necessary, and the causal dependencies among them. Causal dependencies reflect, for instance, the fact that a component should enter a state enacting a provide port before another component enters a state requiring that port. In the third phase of *plan generation* an adaptive topological sort of the abstract plan is performed. By adaptive, we mean that the abstract plan could be rearranged during the topological sort if component duplication is needed. Component duplication is used to deal with those cases in which more instances of the same kind of component must be contemporaneously deployed, in different states, in order to enact different ports at the same time.

The algorithm is described in detail, and its correctness and completeness is proved. By correctness we mean that in all the system configurations traversed during the execution of the deployment plan, each active require port is guaranteed to be connected to a corresponding active provide port. By completeness we mean that if it is possible to reach the required final configuration, our algorithm is guaranteed to return a corresponding deployment plan. Finally, we show the polynomial complexity of our algorithm.

In this paper we present the formalization of our algorithm, the correctness and completeness proof, and the complexity analysis; in a related paper [18] we present a proof of concept implementation.

Paper structure. In Sect. 2 we report the Aeolus component model and the formalization of the component deployment problem. In Sect. 3 we present our novel solution to this problem, and in Sect. 4 we provide the correctness, completeness and computational complexity results for the given algorithm. Finally, in Sect. 5 we discuss related work and draw some concluding remarks.

2 The Aeolus component model

In this section we introduce the fragment of the Aeolus model used to frame the problem addressed. The Aeolus model, defined in [12], is a formal model of components, specifically tailored to describe both fine grained software components, like packages to be installed on a single (virtual) machine, and coarse grained ones, like services, obtained as composition of distributed and properly connected sub-services. The problem that we address in this paper is finding a plan, i.e. a correct sequence of actions, that, given a universe of components, leads to a configuration where a target component is in a given state.

A component is a grey-box showing relevant internal states and the actions that can be acted on the component to change state during deployment. Each state activates provide and require ports representing resources that the component provides and needs. Active require ports must be bound to active provide ports of other components.

As an example consider, for instance, the task of configuring a *master-slave replication*, typically used to scale a MySQL deployment over two servers. The master node must be created, installed and configured, and put in running mode to start serving external requests. To activate the slave, an initial *dump* of the data stored in the master is needed. Moreover, the master has to authorize the slave. This is a circular dependency between master and slave, since the latter requires the dump of the former that, on its turn, requires the IP address of the slave to grant its authorization. The Aeolus model for the master and slave component is shown in Fig.1.

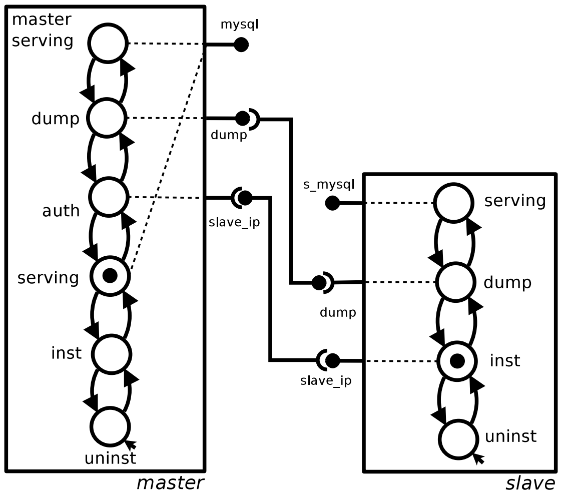

Fig. 1: MySQL master-slave components according to the Aeolus model

The master component has 5 states, an initial uninst state followed by inst and serving. In serving state, it activates the provide port mysql. When replication is needed, in order to enter the final master serving state, it first traverses the state auth that requires the IP address from the slave, and the state dump to provide the dump to the slave. The slave component has instead 4 states, an initial uninst state and 3 states which complement those of the master during the replication process.

We now move to the formal definition of the Aeolus component model. It is based on the notion of *component type*, used to specify the behaviour of a particular kind of component. In the following, $\mathcal{I}$ denotes the set of port names and $\mathcal{Z}$ the set of components.

Definition 1 (Component type). *The set $\mathcal{T}_{flat}$ of* component types *ranged over by $\mathcal{T}, \mathcal{T}_1, \mathcal{T}_2, \ldots$ contains 4-ples $\langle Q, q_0, T, D \rangle$ where:*

- *Q is a finite set of states containing the initial state q_0;*
- *$T \subseteq Q \times Q$ is the set of* transitions*;*
- *D is a function from Q to a pair $\langle \mathbf{P}, \mathbf{R} \rangle$ of port names (i.e. $\mathbf{P}, \mathbf{R} \subseteq \mathcal{I}$) indicating the provide and require ports that each state activates. We assume that the initial state q_0 has no requirements (i.e. $D(q_0) = \langle \mathbf{P}, \emptyset \rangle$).*

We now define configurations that describe systems composed by components and their bindings. Each component has a unique identifier, taken from the set $\mathcal{Z}$. A configuration, ranged over by $\mathcal{C}_1, \mathcal{C}_2, \ldots$, is given by a set of component types, a set of components in some state, and a set of bindings.

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

- *$U \subseteq \mathcal{T}_{flat}$ is the finite* universe *of the 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 $\langle Q, q_0, T, D \rangle$, and $q \in Q$ is the current component state;*
- *$B \subseteq \mathcal{I} \times Z \times Z$ is the set of* bindings*, namely 3-ple composed by a port, the component that provides that port, and the component that requires it; we assume that the two components are distinct.*

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 $\langle Q, q_0, T, D \rangle$, we use projections to decompose it: `.states`, `.init`, and `.trans` return the first three elements; $.\mathbf{P}(q)$ and $.\mathbf{R}(q)$ return the two elements of the $D(q)$ tuple. Moreover, we use `.prov` (resp. `.req`) to denote the union of all the provide ports (resp. require ports) of the states in Q. When there is no ambiguity we take the liberty to apply the component type projections to $\langle \mathcal{T}, q \rangle$ pairs. *Example*: $\mathcal{C}[z].\mathbf{R}(q)$ stands for the require ports of component z in configuration $\mathcal{C}$ when it is in state q.

A configuration is correct if all the active require ports are bound to active provide ports.

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

We write $\mathcal{C} \models_{req} (z, r)$ to indicate that the require port of component z, with port r, is bound to an active port providing r, i.e. there exists a component $z' \in Z \setminus \{z\}$ such that $\langle r, z', z \rangle \in B$, $\mathcal{C}[z'] = \langle \mathcal{T}', q' \rangle$ and r is in $\mathcal{T}'.\mathbf{P}(q')$.

The configuration $\mathcal{C}$ is correct *if for every component $z \in Z$ with $S(z) = \langle \mathcal{T}, q \rangle$ we have that $\mathcal{C} \models_{req} (z, r)$ for every $r \in \mathcal{T}.\mathbf{R}(q)$.*

We now formalize how configurations evolve by means of actions.

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

- *$stateChange(z, q, q')$ changes the state of the component $z \in \mathcal{Z}$ from q to q'*
- *$bind(r, z_1, z_2)$ creates a binding between the provide port $r \in \mathcal{I}$ of the component z_1 and the require port r of z_2 ($z_1, z_2 \in \mathcal{Z}$);*
- *$unbind(r, z_1, z_2)$ deletes the binding between the provide port $r \in \mathcal{I}$ of the component z_1 and the require port r of z_2 ($z_1, z_2 \in \mathcal{Z}$);*
- *$new(z : \mathcal{T})$ creates a new component of type $\mathcal{T}$ in its initial state. The new component is identified by a unique and fresh identifier $z \in \mathcal{Z}$;*
- *$del(z)$ deletes the component $z \in \mathcal{Z}$.*

The execution of actions is formalized by means of a labeled transition system on configurations, which uses actions as labels.

Definition 5 (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:*

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

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

$$\mathcal{C} \xrightarrow{unbind(r,z_1,z_2)} \langle U, Z, S, B \setminus \langle r, z_1, z_2\rangle\rangle$$
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*
$$S'(z') = \begin{cases} \langle \mathcal{T}, \mathcal{T}.\mathtt{init}\rangle & \textit{if } z' = z \\ \mathcal{C}[z'] & \textit{otherwise} \end{cases}$$

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

We can now define a *deployment plan* as a sequence of actions that transform a correct configuration without violating correctness along the way.

Definition 6 (Deployment plan). *A* deployment plan P *is a sequence of reconfigurations* $\mathcal{C}_0 \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathcal{C}_m$ *such that* $\mathcal{C}_i$ *is correct, for* $0 \le i \le m$.

As an example of a deployment plan let us consider the configuration depicted in Fig. 1. If we want to activate the slave, a possible deployment plan that allows to do so requires to perform two consecutive state changes in the master to reach the dump state. At this point, the slave component can reach the serving state performing first the state change into the dump state and then into the serving state. Note that every action in the deployment plan will correspond to one or more concrete instructions. For instance, the state change from the serving to the auth state in the master corresponds to issue the command `grant replication slave on *.* to user@'slave_ip'`.

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

Definition 7 (Deployment problem). *The* deployment problem *has as input an universe* U *of component types, a component type* $\mathcal{T}_t$*, and a target state* q_t*. The output is a deployment plan* $\mathsf{P} = \mathcal{C}_0 \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathcal{C}_m$ *such that* $\mathcal{C}_0 = \langle U, \emptyset, \emptyset, \emptyset\rangle$ *and* $\mathcal{C}_m[z] = \langle \mathcal{T}_t, q_t\rangle$*, for some component* z *in* $\mathcal{C}_m$*, if there exists one. Otherwise, it returns a negative answer, stating that no such a plan exists.*

Notice that the restriction to consider one component in a given state is not limiting: one can easily encode any given final configuration by adding dummy provide ports enabled only by the required final states and a dummy component that requires all such provides. For instance, Fig. 2 depicts the dummy target component that in inst state requires the presence of both an active master and an active slave.

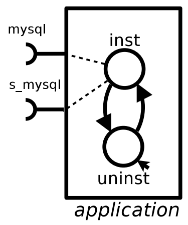

Fig. 2: Target

3 Solving the Deployment problem

The algorithm that we present for solving the deployment problem is a chain of three phases: *reachability analysis*, *abstract planning* and *plan generation*. Each phase works on a representation of the meaningful information output by the previous one. Namely, the *reachability analysis* produces a reachability graph where the component types to be used in the deployment plan are selected in combination with their internal states and the necessary bindings that will have to be established between the activated provide and require ports. If the target is reachable, the subsequent *abstract planning* phase produces a graph where nodes represent deployment actions and arcs denote precedence constraints among them. Finally, the *plan generation* phase synthesizes the deployment plan.

3.1 Reachability analysis

The aim of the first phase is to check if the target can be obtained starting from an initial empty configuration. This is achieved through a forward symbolic reachability analysis that relies on an abstract representation of components. For each component its individual identity as well as the number of its instances are ignored, keeping only its component type and its state $\langle\mathcal{T}, q\rangle$. Also, we abstract away from individual bindings without considering *delete* actions. The abstraction on the bindings is possible since we can safely assume that, given a set of components, all complementary ports on two distinct components are bound. Delete actions are superfluous since the presence of one component does not hinder the reachability of a state in another component.

Algorithm 1 Reachability graph construction

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

Alg. 1 creates a *reachability graph* that visually could be seen as a pyramid where the top level contains all the component types in their initial state and, at every step, a new level is produced by adding new component type-state pairs, reachable from the ones at the previous level (see the grey part of Fig. 3). $Nodes_i$ is the set of the type-state pairs at level i, while $Arcs_i$ represents the possible ways a type-state pair can be obtained; $x \longrightarrow y$ means that component state y, at level $i+1$, is obtained from x at level i by a state change, otherwise y is a copy of x (denoted as $x \cdots\cdots y$). $ProvPort$ is a set containing the ports provided by the components. Initially, it contains the ports provided by all components in their initial state (line 1) and then it is incrementally augmented with the ports provided by the newly added components (lines 9-10). The new type-state pairs to be added are computed by checking if all their requirements are satisfied by at least one component state at the previous level (lines 5-8). Finally, variable $Arcs_i$ is updated (lines 13-16), listing all the possible ways a type-state pair can be obtained. The generation of levels proceeds until a fix-point is reached (line 17). Termination is guaranteed by the fact that the number of possible type-state pairs is finite and at every cycle at least a new pair is added to the $Node_i$ set. When the fix-point is reached, if the last set does not contain the target component type-state pair, a plan to achieve the goal does not exist and we do not execute the subsequent phases of the algorithm.

Once all pairs have been generated, starting from the target pair at the bottom of the pyramid, a selection procedure is carried out in order to pick the pairs to be employed in the deployment plan. The selection is performed by means of a bottom-up visit of the reachability graph as described in Alg. 2.

Algorithm 2 Component Selection

```
1:  $SNodes_n = \{\langle \mathcal{T}_{target}, q_{target} \rangle\}$;
2:  for $i = n$ downto 1 do
3:      $SNodes_{i-1} = SArcs_{i-1} = \emptyset$;
4:      for all $\langle \mathcal{T}, q \rangle \in SNodes_i$ do
5:          $\langle \mathcal{T}', q' \rangle =$ heuristic_parent($\langle \mathcal{T}, q \rangle, i$);
6:          $SNodes_{i-1}$.add($\langle \mathcal{T}', q' \rangle$);
7:          $SArcs_{i-1}$.add($\langle \langle \mathcal{T}', q' \rangle, \langle \mathcal{T}, q \rangle \rangle$);
8:          for all $r \in \mathcal{T}.\mathbf{R}(q)$ do
9:              $\langle \mathcal{T}', q' \rangle =$ heuristic_prov($\langle \mathcal{T}, q \rangle, r, i$);
10:             $SNodes_{i-1}$.add($\langle \mathcal{T}', q' \rangle$);
11:             $SReq$.add($\langle \mathcal{T}', q' \rangle \overset{r}{\multimap} \langle \mathcal{T}, q \rangle$);
```

From the bottom level (that we denote with n) we proceed upward selecting the pairs used to deploy the pairs at the lower level. Variables $SNodes_i$ and $SArcs_i$ denote, respectively, the selected components state pairs at level i and how these pairs are obtained. From the last level only the target pair is selected (line 1). For every selected component at level $i+1$, we select at level i one of its predecessors and we store this choice in variables $SNodes_{i-1}$ and $SArcs_{i-1}$ (lines 5-7). Since there may be more than one possible choice, we rely for the decision on heuristics, here abstracted by function `heuristic_parent`. The decision at this point could affect the length of the deployment plan. A study of the best heuristics is out of the scope of this paper; we leave this task for future work. For an example of a possible heuristic we refer to [18].

For every require port needed by the selected pairs of level $i+1$ that are not copies, we select a pair at level i that is able to activate a complementary provide port. This choice is recorded in $SNodes_{i-1}$ and $SReq$ (lines 10-11). In particular, $SReq$ maintains the indication of the kinds of binding between provide and require ports of components that will be used in the plan to be subsequently synthesized; these dependencies are represented by arcs $\langle \mathcal{T}', q' \rangle \overset{r}{\multimap} \langle \mathcal{T}, q \rangle$ where $\langle \mathcal{T}', q' \rangle$ is the component type-state pair that activates the provide port r, while$\langle \mathcal{T}, q \rangle$ activates the complementary require port. Even in this case there is usually more than one possible alternative in the selection of the type-pair that can provide the requested port. As before, we rely on an heuristics, dubbed `heuristic_prov`, to decide which pair is used as a provider.

Fig. 3 depicts the output of this first phase for the MySQL master-slave example. The grey and black part is the reachability graph generated by Alg. 1, while the part only in black is a possible selection done by Alg. 2. For space reasons, master, slave and application are denoted by M, S and A respectively, and each state is referred by its initial upper-case letter: U for uninst, I for inst, S for serving, A for auth, D for dump and MS for master serving.

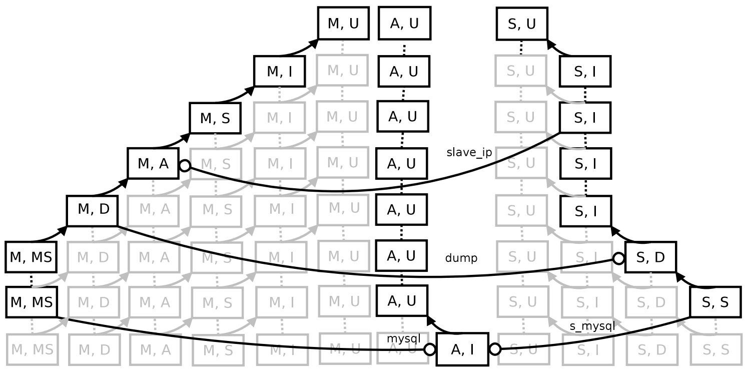

Fig. 3: Reachability graph and component selection for the running example.

The first level of Fig. 3 contains components M, S and A in their initial states. At the second level, two pairs are added: component M in I and component S in I, derived respectively from M in U and S in U. At level 3, pair $\langle \mathsf{M}, S \rangle$ is added. At next step, pair $\langle \mathsf{M}, A \rangle$ can also be added since it derives from $\langle \mathsf{M}, S \rangle$ and its requirement on the interface *slave_ip* is fulfilled by $\langle \mathsf{S}, I \rangle$, appearing at previous level. The generation of the reachability graph proceeds as depicted until the pair $\langle \mathsf{A}, I \rangle$ is added: this is the last level as no new type-state pairs can be generated.

The selection procedure starts from the target node, $\langle \mathsf{A}, I \rangle$ in the last level. There is only one possible derivation for $\langle \mathsf{A}, I \rangle$ and so $\langle \mathsf{A}, U \rangle$ is selected as its origin. Since $\langle \mathsf{A}, I \rangle$ requires two interfaces, *r_mysql* and *s_mysql*, provided by $\langle \mathsf{M}, RS \rangle$ and $\langle \mathsf{S}, S \rangle$, these providers are also selected. The selection process continues until components at the top level are selected.

3.2 Abstract planning

The abstract plan specifies the life-cycle of all component types employed in the deployment of the target state. It can be seen as a directed graph where nodes represent either a *new*, *del*, or *stateChange* action, and arcs represent action precedence constraints. Every node is tagged by a triple denoting an action: $\langle z, q, q' \rangle$ for a *stateChange* from state q to q' of instance z; $\langle z, \varepsilon, q_0 \rangle$ for a *new* action of instance z (in state q_0), and $\langle z, q, \varepsilon \rangle$ for *del* action on the instance z (in state q). Precedence arcs are of three kinds: (i) $\longrightarrow$: precedence of *stateChange* actions on the same instance; (ii) $\overset{r}{\twoheadrightarrow}$: precedence of instances that provide a resource r w.r.t instances requiring it; (iii) $\overset{r}{\dashrightarrow}$: precedence of an instance requiring a port r w.r.t. actions that deactivate it.

Algorithm 3 Abstract Plan Generation

```
1:  $Paths$ = getMaxPaths($Nodes_0, \ldots, Nodes_n$);
2:  $Act = \emptyset$; $InstMap = \{\,\}$;
3:  for all $(\langle \mathcal{T}, q_0 \rangle, \ldots, \langle \mathcal{T}, q_h \rangle) \in Paths$ do
4:      $inst$ = getFreshName();
5:      $InstMap[inst] = \mathcal{T}$;
6:      $Act$.add($\langle inst, \varepsilon, q_0 \rangle$); $Act$.add($\langle inst, q_h, \varepsilon \rangle$);
7:      for all $i \in [0..h-1]$ do
8:          $Act$.add($\langle inst, q_i, q_{i+1} \rangle$)
9:      $Prec$.add($\langle \langle inst, \varepsilon, q_0 \rangle \longrightarrow \langle inst, q_0, q_1 \rangle \rangle$);
10:     $Prec$.add($\langle \langle inst, q_{h-1} q_h \rangle \longrightarrow \langle inst, q_h, \varepsilon \rangle \rangle$);
11:     for all $i \in [0..h-2]$ do
12:         $Prec$.add($\langle \langle inst, q_i, q_{i+1} \rangle \longrightarrow \langle inst, q_{i+1}, q_{i+2} \rangle \rangle$);
13: for all $\langle \langle \mathcal{T}, q' \rangle \overset{r}{\multimap} \langle \mathcal{T}', s' \rangle \rangle \in SReq$ do
14:     for all $n_1 == \langle i_1, s, s' \rangle \in Act$ . $InstMap[i_1] == \mathcal{T}'$ do
15:         let $n_2 = \langle i_2, q, q' \rangle \in Act$ where $InstMap[i_2] == \mathcal{T}$ in
16:             $Prec$.add($n_2 \overset{r}{\twoheadrightarrow} n_1$)
17:             let $n_1'$ where $n_1 \longrightarrow n_1'$ in
18:                 repeat
19:                     let $n_2' = \langle i_2, q', q'' \rangle$ where $n_2 \longrightarrow n_2'$ in
20:                         if $q' \neq \varepsilon \ \wedge \ r \in \mathcal{T}.\mathbf{P}(q')$ then
21:                             $n_2 = n_2'$
22:                 until $q'' == \varepsilon \ \vee \ r \notin \mathcal{T}.\mathbf{P}(q')$
23:                 $Prec$.add($n_1' \overset{r}{\dashrightarrow} n_2$)
```

Alg. 3 is used to derive the abstract plan. To generate an abstract plan we consider an instance for every maximal path that starts from a type-state pair in the top level and reaches a type-state that is not a copy. For instance, as shown in Fig. 3, for the master-slave example there are three maximal paths: one for the master (starting from $\langle \mathsf{M}, U \rangle$ and ending in $\langle \mathsf{M}, MS \rangle$), one for the dummy component and one for the slave (starting from $\langle \mathsf{S}, U \rangle$ and ending in $\langle \mathsf{S}, S \rangle$). The computation of the maximal paths is performed by the function `getMaxPaths` (line 1). Variables *Act* and *Prec* are used to store the actions of the abstract plan and the precedence constraints, respectively.

The first loop (lines 3-12) is used to generate the nodes of the abstract plan and the precedence constraints $\longrightarrow$ among them. First of all, a new fresh name for the instance is generated (line 4) and is associated to the component type of the instance using the map $InstMap$ (line 5). After that, nodes corresponding to the creation and deletion of the instance are added (line 6), as well as nodes representing intermediate state changes (line 8). The last part of the loop (lines 9-12) is used to generate the precedence arcs $\longrightarrow$.

The second loop, starting at line 13, adds for every dependency arc, selected in the reachability graph, a pair of $\twoheadrightarrow$ and $\dashrightarrow$ arcs. In particular, lines 17-23 apply a relaxation of the $\overset{r}{\dashrightarrow}$ arc, since if a port r is provided also by successor states, then we can relax the constraint imposed by the $\overset{r}{\dashrightarrow}$ arc by setting its destination to the last successor node that still provides r.

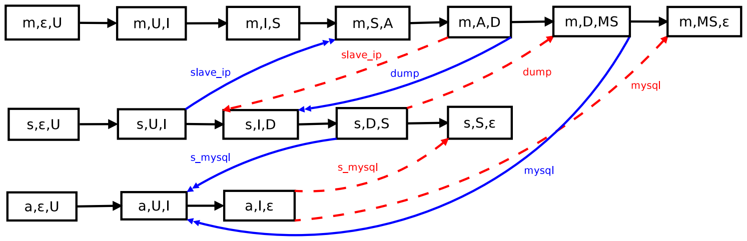

Fig. 4: Abstract plan for the running example.

Fig. 4 displays the abstract plan for the running example. The rows represent the life-cycles of master, slave and application, respectively. The $\overset{slave\_ip}{\twoheadrightarrow}$ from $\langle s, U, I\rangle$ to $\langle m, S, A\rangle$ expresses the fact that the $stateChange$ of slave from uninstalled to installed must precede the $stateChange$ of master from serving to auth because state auth of server requires interface $slave\_ip$, provided by slave in state installed. The twin $\dashrightarrow$ arc states that master must switch from auth to dump before slave switches from installed to dump, as this state ceases providing interface $slave\_ip$, otherwise its requirement would become unfulfilled. Following the same principle we can interpret the pair of arcs $\langle m, A, D\rangle \twoheadrightarrow \langle s, I, D\rangle$ and $\langle s, D, S\rangle \dashrightarrow \langle m, D, MS\rangle$ for interface $dump$. Finally, the target is represented by node $\langle a, U, I\rangle$, namely application entering state installed. This state requires two interfaces, $mysql$ and s_mysql provided respectively by master in state master serving and slave in state serving. Two $\twoheadrightarrow$ arcs (together with their $\dashrightarrow$ counterparts) are thus added with destination $\langle a, U, I\rangle$, one from $\langle s, D, S\rangle$ and the other one from $\langle m, D, MS\rangle$.

3.3 Plan generation

The main idea for the synthesis of a concrete deployment plan is to visit the nodes of the abstract plan in topological order until the target component is reached. Visiting a node consists of performing that action. Moreover, in order to properly satisfy component requirements, when an incoming $\twoheadrightarrow$ is encountered a, new binding should be created, and when an outgoing $\dashrightarrow$ is encountered, the corresponding binding should be deleted.

Algorithm 4 Plan synthesis

```
1:  $Plan = [\,];\ ToVisit = [\,];\ finished = \textbf{false};$
2:  for all $n = \langle i, x, y\rangle \in Act$ do
3:      if no_incoming_edges($n$) then
4:          $Plan$.append($new(i : InstMap[i])$);
5:          $ToVisit$.push($n$);
6:  repeat
7:      repeat
8:          $\langle i, x, y\rangle = ToVisit$.pop();
9:          for all $\langle i, x, y\rangle \overset{r}{\dashrightarrow} \langle i', x', y'\rangle \in Prec$ do
10:             $Plan$.append($unbind(r, i', i)$); $Prec$.remove($\langle i, x, y\rangle \overset{r}{\dashrightarrow} \langle i', x', y'\rangle$);
11:             if no_incoming_edges($\langle i', x', y'\rangle$) then $ToVisit$.push($\langle i', x', y'\rangle$);
12:         if $y == \varepsilon$ then $Plan$.append($del(i)$);
13:         else
14:             for all $\langle i, x, y\rangle \overset{r}{\twoheadrightarrow} \langle i', x', y'\rangle \in Prec$ do
15:                 $Plan$.append($bind(r, i, i')$); $Prec$.remove($\langle i, x, y\rangle \overset{r}{\twoheadrightarrow} \langle i', x', y'\rangle$);
16:                 if no_incoming_edges($\langle i', x', y'\rangle$) then $ToVisit$.push($\langle i', x', y'\rangle$);
17:             if $x \neq \varepsilon$ then $Plan$.append($stateChange(\langle i, x, y\rangle)$);
18:             let $n \in Act$ where $\langle i, x, y\rangle \longrightarrow n \in Prec$ in
19:                 $Prec$.remove($\langle i, x, y\rangle \longrightarrow n$);
20:                 if no_incoming_edges($n$) then $ToVisit$.push($n$);
21:         if $InstMap[i] == \mathcal{T}_{target} \wedge y == q_{target}$ then $finished = \textbf{true}$;
22:         $Act$.remove($\langle i, x, y\rangle$);
23:     until $ToVisit == [\,] \vee finished$
24:     if $\neg finished$ then
25:         $n$ = Duplicate();
26:         $ToVisit$.push($n$);
27: until $finished$
```

Alg. 4 builds the plan adding actions to the list $Plan$. Nodes can be visited if they do not have precedence constraints, i.e. incoming arcs. Function `no_incoming_edges` checks this and if it is true nodes are added to the stack $ToVisit$. At the beginning, all initial nodes are pushed on $ToVisit$ (lines 2-5) and a *new* action is added to the plan for every initial node (line 4). The algorithm then proceeds considering one action $a = \langle i, x, y\rangle$ in $ToVisit$ at a time until the target node is encountered or $ToVisit$ becomes empty. For every outgoing $\dashrightarrow$ arc of a, an *unbind* action is added to the plan and the arc is removed from $Prec$ (lines 9-11). If a is a delete action then the corresponding delete action is added to the plan (line 12). For every outgoing $\twoheadrightarrow$ arc of a, a *bind* action is added to the plan and the arc is removed from $Prec$ (lines 14-16). Finally, if a is an intermediate node, a *stateChange* action is added to the plan (line 17). In case of new or state changes the outgoing $\longrightarrow$ arcs of n are removed from $Prec$. Everytime an arc is removed, the target of the arc is pushed to $ToVisit$ if it has no incoming arcs.

Note that the topological visit could not reach the target if a cycle is present in the graph. This happens when an instance is required to perform a state change as well as provide a port that the state change deactivates. In these cases, it is necessary to duplicate the instance: one new copy remains in the state, thus keeping the provide port active, and in this way the original instance is allowed to perform the state change. Lines 24-26 deal with the duplication process, calling function Duplicate in Alg. 5.

Algorithm 5 Duplicate

```
1:  function Duplicate
2:      let $n = \langle i, x, y\rangle \in Act$ where $y \neq \varepsilon \wedge \nexists n' \in Act \,.\, (n' \longrightarrow n \in Prec \vee n' \overset{r}{\twoheadrightarrow} n \in Prec)$ in
3:          $i'$ = getFreshName(); $InstMap[i'] = InstMap[i]$; $Act$.add($\langle i', x, \varepsilon\rangle$);
4:          for all $n' \overset{r}{\dashrightarrow} \langle i, x, y\rangle \in Prec$ do
5:              $Prec$.remove($n' \overset{r}{\dashrightarrow} \langle i, x, y\rangle$); $Prec$.add($n' \overset{r}{\dashrightarrow} \langle i', x, \varepsilon\rangle$);
6:          for $(j = Plan.size() - 1; j \geq 0; j = j - 1)$ do
7:              if $Plan[j] == bind(r, i, z)$ then $Plan[j] = bind(r, i', z)$;
8:              else if $Plan[j] == bind(r, z, i)$ then $Plan$.insert($bind(r, z, i'), j$);
9:              else if $Plan[j] == unbind(r, i, z)$ then $Plan[j] = unbind(r, i', z)$;
10:             else if $Plan[j] == unbind(r, z, i)$ then $Plan$.insert($unbind(r, z, i'), j$);
11:             else if $Plan[j] == new(i : \mathcal{T})$ then $Plan$.insert($new(i : \mathcal{T}), j$);
12:             else if $Plan[j] == stateChange(\langle i, x, y\rangle)$ then
13:                 $Plan$.insert($stateChange(\langle i', x, y\rangle, j$);
14:         return $\langle i, x, y\rangle$;
```

The Duplicate function first identifies a state change node $\langle i, x, y\rangle$ with only incoming $\dashrightarrow$ arcs (line 2). i is the instance to duplicate until the node preceding $\langle i, x, y\rangle$. A fresh name i' is assigned to identify the new instance and the delete node of i' is added to the set of actions (line 3). All $\dashrightarrow$ arcs incoming into $\langle i, x, y\rangle$ are redirected towards the new node $\langle i', x, \varepsilon\rangle$ (lines 4-5). Then, the actions already performed on i are duplicated in order to perform them also on the new instance i' (lines 6-13). The actions *new* and *stateChange* of i' are added to the plan immediately after the *new* and *stateChange* actions of i (lines 11, 13). Similarly, *bind* and *unbind* actions where i requires something from another instance, are replicated (lines 8, 10). The *bind* and *unbind* actions where i instead provides something for other instances, are replaced with bind and unbind actions involving i' instead of i (lines 7, 9).

The Duplicate function returns the node $\langle i, x, y\rangle$; notice that this node is immediately added to the $ToVisit$ stack since, after the duplication procedure, it has no precedence constraints. Alg. 4 eventually terminates since the number of duplications needed to reach the target component is bound by the number of actions in the original abstract plan.

As an example, starting from the abstract plan of Fig.4, a possible deployment plan for the master slave scenario is the following one:

$$\begin{aligned}
&Plan = [\; new(m : \mathsf{master}); new(s : \mathsf{slave}); new(a : \mathsf{application});\\
&stateChange(m, \mathsf{uninst}, \mathsf{inst}); stateChange(m, \mathsf{inst}, \mathsf{serving}); bind(slave\_ip, s, m);\\
&stateChange(s, \mathsf{uninst}, \mathsf{inst}); stateChange(m, \mathsf{serving}, \mathsf{auth}); unbind(slave\_ip, s, m);\\
&bind(dump, m, s); stateChange(m, \mathsf{auth}, \mathsf{dump}); stateChange(s, \mathsf{inst}, \mathsf{dump});\\
&unbind(dump, m, s); bind(s\_mysql, s, a); stateChange(s, \mathsf{dump}, \mathsf{serving});\\
&bind(mysql, m, a); stateChange(m, \mathsf{dump}, \mathsf{master\ serving});\\
&stateChange(a, \mathsf{uninst}, \mathsf{inst}) \;]
\end{aligned}$$

Note that this plan is generated without applying instance duplication; the interested reader can refer to [18] for an example involving duplication.

4 Formal analysis of the algorithm

In this section we prove that the proposed algorithm, called *DeploymentPlanner* in the following, is sound and complete, i.e. it produces a correct deployment plan if and only if it exists. Moreover, we prove that it runs in polynomial time w.r.t. the size of the problem.

Theorem 1 (Soundness). *Given a universe of components U, a component type $\mathcal{T}_t$, and a target state q_t, if the DeploymentPlanner algorithm computes a sequence of actions $\alpha_1, \ldots, \alpha_m$, then $\langle U, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_m} \mathcal{C}_m$ is a deployment plan for $\mathcal{T}_t$ in state q_t.*

Proof. $\langle U, \emptyset, \emptyset, \emptyset \rangle$ is the empty configuration and therefore it is correct by definition. $\langle \mathcal{T}_t, q_t \rangle$ is contained in $\mathcal{C}_m$ since Alg. 4 terminates when the state change to obtain $\langle \mathcal{T}_t, q_t \rangle$ is added to the plan. To prove the thesis we have to show that every reconfiguration action preserves correctness. This can be proven by cases on the kind of α_j action. If α_j is a bind or new action, correctness is preserved since these two actions do not violate any requirement.

If $\alpha_j = \mathit{stateChange}(i, x, y)$ then α_j may invalidate correctness in two ways: either i stops providing a port p, needed by someone else or state y of i requires a port r, not provided in $\mathcal{C}_j$. In the first case, if i' is the component requiring p, by Alg. 3 there is an arc $\overset{p}{\twoheadrightarrow}$ from i to i', that goes from a predecessor of $\langle i, x, y \rangle$ to a node of i'. Together with it, a twin $\overset{p}{\dashrightarrow}$ arc, from i' to i, is added, that has $\langle i, x, y \rangle$ as destination. This guarantees that an *unbind* action is added to the plan before the $\mathit{stateChange}(i, x, y)$, thus i' does not require p any longer, and so correctness is ensured. For the second case, if i in y requires a port r, then, for the same reason as above, there exists an $\overset{r}{\dashrightarrow}$ arc from a successor of $\langle i, x, y \rangle$ in i, to one node of i'. Thus a twin $\overset{r}{\twoheadrightarrow}$ arc exists, from i' to a predecessor of $\langle i, x, y \rangle$ in i, meaning that the corresponding *bind* action is added to the plan before the $\mathit{stateChange}(i, x, y)$ action, and correctness is not violated.

Let us consider the case $\alpha_j = \mathit{unbind}(r, i', i)$. It does not violate correctness since by Alg. 4 we add an *unbind* action for every $\langle i, x, y \rangle \overset{r}{\dashrightarrow} \langle i', x', y' \rangle$ arc. This ensures that instance i, that required r, has already stopped requiring it.

Similarly, if $\alpha_j = \mathit{del}(i)$, it may violate correctness by deleting a component that still provides a needed port. This, however, is never the case because delete actions have just $\overset{r}{\dashrightarrow}$ incoming arcs. Therefore, by Alg. 4, this action is performed only after all instances requiring r have stopped requiring it. □

The second result shows that the algorithm is complete, i.e. if a deployment plan exists, then the algorithm will eventually find one. To prove completeness we rely on the following lemma, stating that all circularities in the abstract plan contain at least an $\dashrightarrow$ arc. This key property guarantees, in presence of circularities, the existence of a node that has as only $\dashrightarrow$ incoming arcs. This is the node chosen by Alg. 5 for duplication, to eliminate a cycle and proceed with the topological visit.

Lemma 1. *Every cycle in the abstract plan contains at least an* $\dashrightarrow$ *arc.*

Proof. (By contradiction). Assume that the cycle contains just $\twoheadrightarrow$ and $\longrightarrow$. If an $\longrightarrow$ arc belongs to the cycle this means that during the reachability analysis a component type-state pair of a higher level required a port from a type-state pair in a lower level. This is impossible by construction. Hence the cycle contains only $\twoheadrightarrow$ arrows. This means that the actions involved in the cycle are just state changes. Moreover the type-state pairs obtained with these state changes are mutually dependent, i.e. the component z_1 to reach a state q_1 needs something provided by z_2 in state q_2 and, vice versa, the component z_2 to reach q_2 needs something provided by z_1 in q_1. By Alg. 1 mutually dependent type-state cannot be obtained. $\square$

Theorem 2 (Completeness). *Given an universe of components U, a component type $\mathcal{T}_t$, and a target state q_t, if a solution exists to the deployment problem on input $I = (U, \mathcal{T}_t, q_t)$, then algorithm DeploymentPlanner returns a deployment plan for I.*

Proof. Since by hypothesis there is a sequence of create and state change actions that allow the deployment of $\mathcal{T}_t$ in state q_t, during the reachability analysis the component state pair $\langle \mathcal{T}_t, q_t \rangle$ is obtained. A correct plan will be produced (Thm. 1) assuming that the abstract plan and plan generation phases terminate. The former terminates because given the reachability graph, the maximal number of maximal paths is finite. The latter terminates because duplication will be needed at most k^2 times, where k is the number of component type-state pairs in the last level of the reachability graph. Indeed, to reach the target component state pair, potentially all the state changes and create actions of the abstract plan could be visited. When there is a cycle that forbids the visit of a state change action, as a direct consequence of Lemma 1, there is at least a state change action that has only $\dashrightarrow$ incoming arcs. The duplication procedure removes all the cycles involving that action without creating new ones. The topological visit can therefore proceed and it eventually terminates since at every duplication at least a state change could be performed and the number of state change actions in the abstract plan is finite and fixed. $\square$

As a final result we prove that *DeploymentPlanner* runs in polynomial time.

Theorem 3 (Complexity). *The DeploymentPlanner algorithm runs in polynomial time.*

Proof. Let us denote with k the total number of possible component type-state pairs, with b the maximal number of predecessors of a type-state pair, and with h the maximal number of ports. Every level of the reachability graph has no more than k type-state pairs. At every level one or more type-state pairs are added, hence the reachability analysis terminates and in the pyramid there are at most $k + 1$ levels. To build a new level from a previous one it is necessary to filter the successors of the components in the previous level by checking if their requirements are satisfied. Since a component has at most k successors and requires at most h ports, the cost of building a level is $O(hk^2)$. The pyramid has at most $k+1$ levels, hence Alg. 1 runs in $O(hk^3)$ time.

To select the bindings and the components (Alg. 2), for every type-state pair at most h ports and b parent pairs need to be considered. Since in every level there could be potentially k pairs and the total number of pairs in the reachability graph is $O(k^2)$, Alg. 2 takes $O(bhk^3)$ time.

The computation of the maximal paths in Alg. 3 can be performed in $O(k^3)$ since there are at most k^2 maximal paths of length k. The generation of the abstract plan can be done in $O(hk^2)$ since there could be at most k^2 actions, each of them having no more than $h+1$ outgoing precedence constraints.

Alg. 4 relies on duplicating an instance whenever the topological visit gets stuck, due to precedence constraint cycles. In the worst case, a duplication is needed for every node of every instance and to detect which node to duplicate all the nodes could be visited. Since every node has at most $2hk^2+1$ incoming arcs, detecting the node to duplicate has a worst case cost of $O(hk^4)$. The duplication procedure may update the plan adding or modifying at most an action for every node and binding involving the instance to duplicate. Since an instance could be involved in k actions and every action has up to $2hk^2+1+h$ (incoming and outgoing) arcs, the cost to perform a duplication is $O(hk^3)$. Therefore, in the worst case, the cost of all duplications is $O(hk^4)$.

The topological visit of the abstract plan is linear w.r.t. the number of nodes and thus requires $O(k^3)$ steps.

Summing up, the *DeploymentPlanner* algorithm has a total complexity of $O(bhk^3)+O(hk^4)$, which considering b bound by k, amounts to $O(hk^4)$. □

5 Related work and conclusions

In this work we address the problem of finding a suitable technique to automatize the deployment of complex systems assembled from a large number of interconnected components. We propose an algorithm that computes in polynomial time the actions needed to deploy such a system and prove soundness and completeness of this novel approach.

To describe a system we adopt the Aeolus component model [12]. According to it, components are grey-boxes with provide and require ports and with an associated automaton, describing the component life-cycle, and expressing for each internal state the corresponding ports that are (de)activated. The idea to specify a component by means of a black-box with an interface that exhibits to the (outside) environment its behavior is widely adopted. For instance, the standard definition of component in the UML specification [1] sees components as black-boxes that may provide and require certain interfaces. This sometimes is not enough and the inner structure of a component must be also considered. The use of automata as a formal model is a natural choice as testified, for instance, by interface automata [8,17]. These models allow to develop formal verification methods for properties of interest but, differently from our approach, they focus on checking component compatibility and behavior refinement. The

FraSCAti [23] platform, by leveraging on a concise and expressive description of a complex software system, defined by the Fractal component model [5], develops a framework for managing the deployment of applications in the cloud. It is up to the system designer, however, to select the components and to realize their interconnection. Process calculi approaches are also used to model software components, e.g. [3,22,6,19]. The focus of these approaches, however, is not on deployment but rather on modeling interaction and communication between components.

Industrial tools such as [7,16,21,20,24,15] are available to ease the deployment of software. They allow to automatize the process of carrying out the deployment of components on a pool of machines, provided a deployment plan is known in advance.

Related to our work are [14,10] that compute final configurations solving a Constraint Satisfaction Problem. Both these works however do not provide a sequence of actions to reach the desired configuration.

Engage [13] uses automata to specify a component's behaviour and it is able to deploy the resources completing a target partial configuration. However, it relies on the assumption that the dependency graph is acyclic, meaning that circular dependencies among components are not admitted.

Closely related is [2] that proposes an heuristic-based algorithm to remove build dependency cycles for bootstrapping a Linux software distribution. The building order of the packages is generated using a topological sort of a graph. However, differently from our work, one of the assumptions is that once a package is recompiled, its older version is no longer required.

A proof of concept implementation of the *DeploymentPlanner* algorithm has been developed and described in [18] with some preliminary validation modeling more complex use cases. Results are encouraging as the tool is able to produce plans in less than a minute, for scenarios involving hundreds of components. As future work we intend to study the impact of the selection heuristics on the length of the deployment plan. We deem that with the right heuristics the number of components involved in the plan could be greatly reduced. We aim to further refine the current technique by considering also reconfiguration plans, dealing with cases in which the initial configuration has already some deployed components. Finally, we would like to take into account conflicts, producing in a reasonable amount of time plans that do not violate them or minimize the time windows where a system is inconsistent.

References

1. OMG Unified Modeling Language (UML), Superstructure, V2.4.1.
2. P. Abate and S. Johannes. Bootstrapping Software Distributions. In *CBSE'13*, pages 131–142. ACM, 2013.
3. F. Achermann and O. Nierstrasz. A Calculus for Reasoning about Software Composition. *Theor. Comput. Sci.*, 331(2-3):367–396, 2005.
4. S. Baron, Z. Peter, T. Vadim, Z. Jeremy, L. Arjen, and B. D. J. *High performance MySQL, 2nd edition.* O'Reilly, second edition, 2008.

5. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL Component Model and its Support in Java. *Softw., Pract. Exper. '06*, 36(11-12):1257–1284, 2006.
6. M. Bundgaard, T. T. Hildebrandt, and J. C. Godskesen. A CPS Encoding of Name-Passing in Higher-Order Mobile Embedded Resources. *Theor. Comput. Sci.*, 356(3):422–439, 2006.
7. M. Burgess. A Site Configuration Engine. *Computing Systems*, 8(2):309–337, 1995.
8. L. De Alfaro and T. A. Henzinger. Interface Automata. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 109–120. ACM, 2001.
9. Circular Build Dependencies. http://wiki.debian.org/CircularBuildDependencies. Retrieved June 2013.
10. R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, and J. Zwolakowski. Optimal Provisioning in the Cloud. Technical report, Aeolus project, June 2013. http://hal.archives-ouvertes.fr/hal-00831455.
11. R. Di Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro. Component Reconfiguration in the Presence of Conflicts. In *ICALP 2013: 40th International Colloquium on Automata, Languages and Programming*, volume 7966 of *LNCS*, pages 187–198. Springer-Verlag, 2013.
12. R. Di Cosmo, S. Zacchiroli, and G. Zavattaro. Towards a Formal Component Model for the Cloud. In *SEFM 2012: 10th International Conference on Software Engineering and Formal Methods*, volume 7504 of *LNCS*, pages 156–171. Springer-Verlag, 2012.
13. J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: A Deployment Management System. In *PLDI'12: Programming Language Design and Implementation*, pages 263–274. ACM, 2012.
14. J. A. Hewson, P. Anderson, and A. D. Gordon. A Declarative Approach to Automated Configuration. In *LISA '12: Large Installation System Administration Conference*, pages 51–66, 2012.
15. Juju, devops distilled. https://juju.ubuntu.com/. Retrieved June 2013.
16. L. Kanies. Puppet: Next-generation configuration management. *;login: the USENIX magazine*, 31(1):19–25, 2006.
17. K. G. Larsen, U. Nyman, and A. Wasowski. Interface Input/Output Automata. In *FM'06*, volume 4085 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2006.
18. T. A. Lascu, J. Mauro, and G. Zavattaro. A Planning Tool Supporting the Deployment of Cloud Applications. In *IEEE 25th International Conference on Tools with Artificial Intelligence, ICTAI 2013*. To appear.
19. F. Montesi and D. Sangiorgi. A Model of Evolvable Components. In *Trustworthy Global Computing*, volume 6084 of *Lecture Notes in Computer Science*, pages 153–171. Springer, 2010.
20. Opscode. Chef. http://www.opscode.com/chef/. Retrieved June 2013.
21. Puppet Labs. Marionette Collective. http://docs.puppetlabs.com/mcollective/. Retrieved June 2013.
22. A. Schmitt and J.-B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *LNCS*, pages 146–178. Springer, 2004.
23. L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *IEEE SCC*, pages 268–275. IEEE, 2009.
24. VMWare. Cloud Foundry, deploy & scale your applications in seconds. http://www.cloudfoundry.com/. Retrieved June 2013.