Science of Computer Programming 113 (2015) 261–284

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Automatic deployment of component-based applications ☆

Tudor A. Lascu, Jacopo Mauro *, Gianluigi Zavattaro

Department of Computer Science and Engineering, FoCUS INRIA Research Team, University of Bologna, Italy

ARTICLE INFO

Article history:
Received 9 July 2014
Received in revised form 14 July 2015
Accepted 17 July 2015
Available online 23 July 2015

Keywords:
Cloud applications management
Component configuration
Deployment planning

ABSTRACT

In distributed systems like those based on cloud 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. Automation techniques and tools have been proposed to ease the deployment process of these complex system. By relying on a formal model of components, we describe a sound and complete algorithm for computing the sequence of actions that permits the deployment of a desired configuration even in the presence of circular dependencies among components. We give a proof for the polynomiality of the devised algorithm and exploit it to develop METIS, a tool for computing deployment plans. The validation of METIS has been performed in two ways: on the one hand, by considering artificial scenarios consisting of a huge number of different components synthesized by following typical configuration patterns and, on the other hand, by exploiting it to deploy real-life installations of a WordPress blogging service.

© 2015 Elsevier B.V. All rights reserved.

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 compared to 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 important 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 [1] or the Engage system [2]. According to the Juju approach, the system administrator decides which high-level services are 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 a partial specification of the system to deploy and then the entire system is automatically completed and the actual deployment is synthesized.

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

☆ Work partially supported by Aeolus project ANR-2010-SEGI-013-0 and by the EU project ENVISAGE FP7-610582.
* Corresponding author.
E-mail addresses: lascu@cs.unibo.it (T.A. Lascu), jmauro@cs.unibo.it (J. Mauro), zavattar@cs.unibo.it (G. Zavattaro).

http://dx.doi.org/10.1016/j.scico.2015.07.006
0167-6423/© 2015 Elsevier B.V. All rights reserved.

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 [3] 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 article, 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 [5], 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, in order to reach 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 automation. 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 [5] we have proved that the deployment problem is undecidable for the full Aeolus model. However, if capacity constraints are not considered the problem is decidable but Ackermann-hard, i.e., it cannot be solved in primitive-recursive time or space. In order to allow efficient algorithms for automatic deployment, in this article we further simplify the Aeolus model by abstracting away also from conflicts, as done for instance in the Juju and Engage frameworks.

We propose a novel solution for automatic component deployment based on an algorithm divided in three distinct phases. In a first phase the existence of a plan is established 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 cope 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, the proposed technique is guaranteed to return a corresponding deployment plan. Finally, we show the polynomial complexity of the given algorithm.

In order to assess the effective viability of the proposed approach, a proof of concept implementation of a tool called METIS (Modern Engineered Tool for Installing Software systems) for synthesizing deployment plans has been developed. On the one hand, we have evaluated the performances of METIS by applying it to synthetically obtained large instances of the problem. On the other hand, we have exploited METIS to compute the deployment plan for real installations involving dozens of components. The obtained results are encouraging. In the first case, METIS manages hundreds of components in less than a minute[1] while the feasible instance size with standard planners is in the order of tens. In the second case, METIS computed in few milliseconds the low-level deployment actions necessary to effectively deploy real instances of WordPress farms, i.e., highly available load-balanced replicated installations of the popular WordPress blogging service.

Article structure In Section 2 we give an overview of the deployment approaches currently used. In Section 3 we report the Aeolus component model and the formalization of the component deployment problem. In Section 4 we present our novel solution to this problem, and in Section 5 we provide the correctness, completeness and computational complexity results for the given algorithm. In Section 6 we describe the validation of METIS on artificial as well as real-life scenarios. Finally, in Section 7 we draw some concluding remarks.

This article extends [6,7] by presenting a road-map to facilitate the understanding of the proposed algorithm, a revised and more detailed formal assessment of its correctness, and a validation of METIS on a real-world case deployment scenario.

2. Related work

Usually the deployment task is conducted by a team of experts that establishes how the different components are to be installed and connected together. The deployment process is then automated by coding it in custom scripts. This approach is effective only if the architecture of the system is decided once and for all and is not expected to be customized for the different needs of the potential end-users, or shaped differently to, e.g., optimize the usage of the available resources.

Currently, developing an application for the cloud is accomplished by relying on the Infrastructure as a Service (IaaS) or the Platform as a Service (PaaS) levels. The IaaS level provides a set of low-level resources forming a “bare” computing en-

[1] A minute seems a reasonable time for computing a deployment plan since provisioning and booting even a single virtual machine on an IaaS provider may require several minutes.

vironment. Developers pack the whole software stack into virtual machines containing the application and its dependencies and run them on physical machines of the provider's cloud. Exploiting the IaaS directly allows a great flexibility but requires also a great expertise and knowledge of the cloud and application entities involved in the process. At the PaaS level (e.g., [8, 9]) a full development environment is provided. Applications are directly written in a programming language supported by the framework offered by the provider, and then automatically deployed to the cloud. The high-level of automation comes however at the price of flexibility: the choice of the programming language to use is restricted to the ones supported by the PaaS provider, and the application code must conform to specific APIs.

Concerning the IaaS level, two deployment approaches standing at opposite sides are gaining more and more momentum: the *holistic* and the *DevOps* one. In the former, also known as *model-driven* approach, one defines a complete model for the entire application and the deployment plan is then derived in a top–down manner. In the latter, put forward by the DevOps community,[2] an application is deployed by assembling available components that serve as the basic building blocks. This emerging approach works in a bottom–up direction: from individual component descriptions and recipes for installing them, an application is built as a composition of these recipes.

As of today, most of the industrial products, offered by major companies, such as Amazon, HP and IBM, rely on the holistic approach. In this context, one prominent work is represented by the TOSCA (Topology and Orchestration Specification for Cloud Applications) standard [11], promoted by the OASIS consortium [12] for open standards. TOSCA proposes an XML-like rich language to describe an application. Deployment plans are usually specified using the BPMN or BPEL notations, workflow languages defined in the context of business process modeling.

The most important representative for the DevOps approach is Juju [1], by Canonical. It is based on the concept of *charm*: the atomic unit containing a description of a component. This description in form of metadata is coupled with configuration data and *hooks* that are scripts to deploy and connect components. Lately, the Juju team has overcome one of the main limitations of the tool, namely the (heavy) assumption that each service unit must be deployed to a separate machine. However, in order to use Juju, some advanced knowledge of the application to install is mandatory. This is due to the fact that the metadata does not specify the required functionalities needed by a component. For instance, to install a WordPress blog in a basic scenario its only requirement is that the application should be connected to a database. However, Juju allows for the deployment of the WordPress blog without warning that it should be deployed only after it has been properly connected to a database. This would actually result in a run-time error, occurring only after having "successfully" deployed WordPress.

A comprehensive survey of all the proposed approaches for the application deployment is outside the scope of this work. In the following we review the solutions most relevant to our work by discussing first works proposed from the academia and then tools developed in industrial environments.

Academia Engage [2] is a deployment management framework consisting in a language used to describe resources and their dependencies, a configuration engine used to generate a full installation specification from an initial or partial one, and a deployment engine/runtime service used to carry out the installation and manage the components during the deployment. The Engage model introduces some important simplifications in order to reach a feasible solution. First of all, similarly to what happens in our approach, conflicts and capacity constraints are not modeled. Contrary to our approach however, the acyclicity of the dependencies among components, that is crucial for Engage, precludes the possibility of having resources that are mutually dependent. Moreover, dependencies in the Engage model are between resources regardless of their current state, the guarded actions are limited in their scope (i.e., the only two possibilities are downstream and upstream) and they assume that the state machine forms a strongly connected graph (i.e., each state is required to be reachable from any other in the state machine).

Another interesting project is ConfSolve [13,14] that consists basically of a definition of a *domain specific language* for describing configuration problems and a tool that uses constraint solving technology to solve them. ConfSolve is able to compute valid configurations that optimize one or more criteria like, e.g., maximizing the number of virtual machines per physical one. The ConfSolve language is object oriented and declarative and allows using quantification and summation over decision variables in constraints. The major limitation of this approach is that the ConfSolve language models the problem of optimal provisioning (of virtual machines) rather than focusing on the deployment process. It does not take into account the wiring aspect, i.e., how to bind the components in use and which are the steps needed to reach the final (optimal) configuration computed by the solver.

VAMP (Virtual Applications Management Platform) [15,16] is a framework constituted by a language to describe the global structure of the application and an environment to manage the runtime deployment of components. The language extends the OVF (Open Virtualization Format) [17] language, a proposed standard for a uniform format for applications to be run on virtual machines. The VAMP deployment process is implemented as a decentralized protocol in a self-configuration manner. The approach is interesting but limited for our purposes as it works under the assumption that the dependency graph is acyclic and requires the developer to specify the virtual machine in which a given component lives.

In [18] a framework has been developed to formally capture the problem of component deployment. The aim of that work is to model the deployment process of component systems and, based on this, to define a technology-agnostic tech-

[2] DevOps is a software development method that stresses communication, collaboration and integration between software developers and Information Technology professionals [10].

Table 1
Available techniques and tools for deployment automation.

Name	Family	Conf. type	Conf. lang.	Comp. lang.	Plan synt.	Platform	Cyclic dep.
Engage	DevOps	Partial	JSON	JSON	✓	Independent	✗
ConfSolve	–	–	ConfSolve	–	–	Independent	–
SmartFrog	–	Full	SmartFrog DSL	Recipe	✗	Independent	✗
VAMP	Holistic	Full	OVF	OVF	✓	Independent	✗
Juju	DevOps	–	–	Recipe (charm)	✗	Ubuntu	✗
TOSCA	Holistic	Full	TOSCA	TOSCA	✗	Independent	✗
Amazon AWS CloudFormation	Holistic	Full	JSON	JSON	✗	Amazon (AMI, EC2, S3)	✗
HP Cloud Service Automation	Holistic	Full	Graphical & TOSCA	Graphical & TOSCA	✗	Private & Hybrid cloud	?
IBM SmartCloud Orchestrator	Holistic	Full	Patterns & TOSCA	Proprietary & TOSCA	✗	OpenStack	✗
METIS	DevOps	Partial	JSON	JSON	✓	Independent	✓

nique to ensure some correctness properties. To this end a Labeled Transition System (LTS) is defined where states and edges represent, respectively, possible configurations of the system and deployment operations. The properties proved to hold, namely well-formedness and closure, basically amount to the fact that dependency constraints and version compatibility will be respected during deployment. However, in this framework, circular dependencies among components are allowed only in a weak form, as at installation time dependency constraints may be temporarily violated. Moreover, components are seen as monolithic/atomic entities (i.e., internal states representing their behavior are not part of the model), each component is considered to be inherently deployable as an independent unit, and dependency constraints must explicitly specify the name and version of the component that is expected to act as a provider for the required interface.

Another relevant direction of research is the one leveraging on traditional *planning* techniques and tools coming from the artificial intelligence area. In [19] the problem of multi-component deployment is translated into an instance of planning problems via an encoding into the PDDL language [20] (the *de facto* standard format to define problems in the planning domain). A tool, called Planit, relying on the LPG [21] planner, has been developed. In this work, components are seen as atomic entities, their (internal) behavior not being considered. They are only subject to start, stop and connect to other components. Scalability experiments were conducted with up to 120 components, the hardest instance being one with 40 components taking 412 seconds to be solved.

Industry SmartFrog [22] is a Java framework, developed at HP, for managing deployment in a distributed setting. It shares some similarities with the Engage approach as every component has a declarative description. It lacks, however, a way to use the declarative description to extract some information for the deployment plan or to perform some static checks.

DADL (Distributed Application Description Language) [23] is a language extension of SmartFrog that enables to express different kinds of constraints (such as Service Level Agreements SLAs and elasticity). This work, however, just focuses on the language aspects.

The Puppet language (and more generally the framework offered by PuppetLabs [24,25]) and CFEngine [26,27] are two successful tools aimed at configuration management in a distributed setting. Products that fall in this category are designed to simplify the task to manage the replicated deployment of the same system.

CloudFoundry [28] is a PaaS solution by VMware that allows for the selection, connection and pushing to a cloud of well defined services (databases, message buses, etc.), used as building blocks for writing applications with one of the supported infrastructures.

Finally, there is RedHat Aeolus [29], a project that despite the name has nothing to do with the Aeolus project that developed the Aeolus model considered in this article. RedHat Aeolus just focuses on allowing for the definition of a virtual machine (VM) that is exportable to all major cloud providers (Amazon, Rackspace, Heroku, etc.). This enables the possibility to migrate a VM to and from cloud providers and also private clouds.

A concise summary of the approaches mentioned above is presented in Table 1. The classification is based on the following categories:

Family whether a framework is based on a top–down (*holistic*) or bottom–up (*DevOps*) approach,
Configuration type (Conf. type) whether the description of the final configuration to reach is fully specified or not,
Configuration language (Conf. lang.) the language used to specify the final configuration,
Component language (Comp. lang.) the language used to specify components,
Plan syntheses (Plan synt.) whether the framework supports operations for the automatic synthesis of a deployment plan,
Platform the platform supported,
Cyclic dependencies (Cyclic dep.) whether the framework is able to deal with circular dependencies among components or not.

An entry "**–**" means that the corresponding classification element may not be applied to that particular tool. The "**?**" symbol instead means that it was not possible to establish the value by reading the official documentation available online.

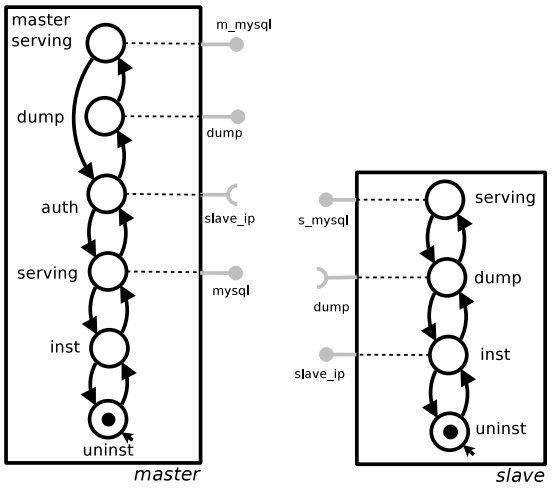


Fig. 1. MySQL master–slave components according to the Aeolus model.

3. 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 [30], 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 article 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.

The master component has 6 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 behavior 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 3.1 *(Component type).* The set Γ of *component types* ranged over by $\mathcal{T}, \mathcal{T}_1, \mathcal{T}_2, \ldots$ contains quadruples $\langle Q, q_0, T, D\rangle$ 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. $D(q_0) = \langle \mathbf{P}, \emptyset\rangle$).[3]

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.

[3] Notice that assuming that the initial state has no requirements does not hinder the usability of the model. A component that can be deployed in state q only if certain functionalities are provided can be modeled simply adding to its model a dummy initial state q_0 and a transition between q_0 and q.

Definition 3.2 *(Configuration).* A *configuration* $\mathcal{C}$ is a quadruple $\langle U, Z, S, B\rangle$ where:

- $U \subseteq \Gamma$ 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 a triple 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.

We call active the ports that are provided and required by the current state of the component. Please note that a port can have more than one binding, each one connecting it to other distinct ports. This allows, for instance, provide-ports to satisfy simultaneously the requirements of more than one component.

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

Definition 3.3 *(Correctness).* Let us consider the configuration $\mathcal{C} = \langle U, Z, S, B\rangle$.

We write $\models_{req}(C, z)r$ 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 $S(z) = \langle \mathcal{T}, q\rangle$ and for every $r \in \mathcal{T}.\mathbf{R}(q)$ we have that $\models_{req}(C, z)r$.

We now formalize how configurations evolve by means of actions.

Definition 3.4 *(Actions).* The set $\mathcal{A}$ contains the following actions:

- $stateChange(z, q, q')$: changes the internal 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 3.5 *(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 $\mathcal{C}[z].\texttt{state} = q$ and $(q, q') \in \mathcal{C}[z].\texttt{trans}$ and
$$S'(z') = \begin{cases} \langle \mathcal{C}[z].\texttt{type}, q'\rangle & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases}$$

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

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

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

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

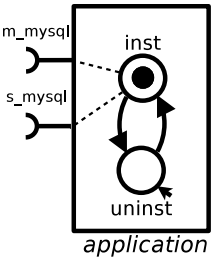


Fig. 2. Target.

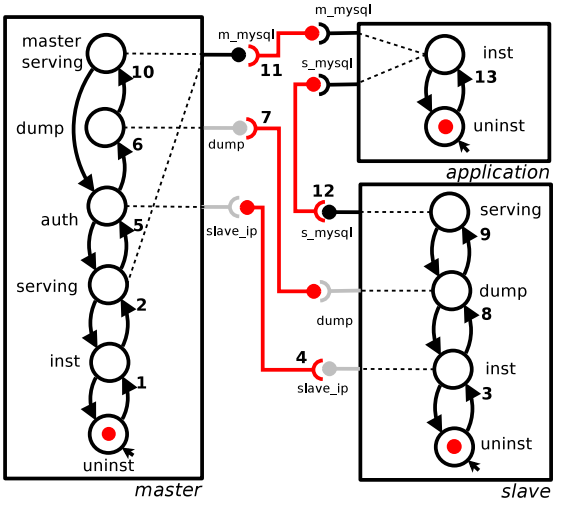


Fig. 3. Sample deployment plan for the running example.

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

Definition 3.6 *(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 $0 \leq i \leq m$.

We now have all the ingredients to define the *deployment problem*, that is our main concern: given a 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 3.7 *(Deployment problem).* The *deployment problem* has as input a 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 target set of components 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.

As an example of a deployment plan let us consider the configuration depicted in Fig. 1 extended with the target *application* component in Fig. 2 in its initial state uninst. As graphically described in Fig. 3,[4] if we want to move the *application* component in state inst, a possible deployment plan can perform two consecutive *stateChange* actions in the *master* to reach the serving state (actions 1 and 2) and one *stateChange* action in the *slave* to reach the inst state (action 3). At this point, to allow the master to enter into the auth state, a binding between the *slave_ip* ports is performed (action 4). The *master* can then reach the dump state that provides the port *dump*. After the binding of the *dump* port the *slave* can reach the

[4] For a textual representation of the actions of the plan please have a look at the end of Section 4.3, Listing 1.

Algorithm 1 DEPLOYMENTPLANNER(U, $\langle \mathcal{T}_{target}, q_{target} \rangle$).

```
1: REACHABILITYANALYSIS()                          ▷ Create reachability graph (final nodes in Nodes_n)
2: if ⟨T_target, q_target⟩ ∈ Nodes_n then
3:     COMPONENTSELECTION()                        ▷ Select a set of nodes sufficient to reach the target
4:     ABSTRACTPLAN()                              ▷ Generate an abstract plan
5:     PLANSYNTHESIS()                             ▷ Generate a concrete plan
```

serving state and the *master* the master serving state. With two additional bindings and one state change it is then possible to bring the dummy component *application* in the desired target state inst. 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 (action number 5 in Fig. 3) corresponds in a real-world deployment scenario to the execution of the command `GRANT REPLICATION SLAVE ON *.* TO 'slave-user'@'slave-ip';`.

4. Solving the deployment problem

We now present our technique for solving the deployment problem. We first describe the structure of our algorithm by discussing the main ideas behind it, then we report the presentation of the details in separate subsections.

Procedure DEPLOYMENTPLANNER(U, $\langle \mathcal{T}_{target}, q_{target} \rangle$) in Algorithm 1 computes, if there exists one, a deployment plan to reach a configuration with at least one component of type $\mathcal{T}_{target}$ in state q_{target}, considering the universe of component types U.

The algorithm first invokes REACHABILITYANALYSIS (described in Section 4.1) that generates a *reachability graph* used to check whether the required plan exists. Such a graph contains all those pairs $\langle \mathcal{T}, q \rangle$ for which it is possible to reach a configuration with at least one component of type $\mathcal{T}$ in state q. 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. With $Nodes_n$ we denote the nodes generated in the last phase.

If the target pair $\langle \mathcal{T}_{target}, q_{target} \rangle$ is in $Nodes_n$, at least one plan exists, and in order to synthesize it we proceed as follows.

First of all, COMPONENTSELECTION (described in Section 4.1) selects a set of component type-state pairs sufficient to reach the target. The idea is to proceed backward, from the target pair $\langle \mathcal{T}_{target}, q_{target} \rangle$ in $Nodes_n$, by selecting at each layer $Nodes_j$ a set of pairs sufficient to reach those that have been already selected in $Nodes_{j+1}$. More precisely, the following aspects must be taken into account while selecting the nodes in $Nodes_j$. A pair $\langle \mathcal{T}, q \rangle$ in $Nodes_{j+1}$ can be obtained either as a copy of a pair in $Nodes_j$ or by performing a state change from a pair $\langle \mathcal{T}, q' \rangle$ in $Nodes_j$, assuming that there exists a transition from q' to q. In this last case, it is also necessary to take into account new require ports that are activated by the state q; if there are new requirements, in order for the state change to be executed, it could be necessary to select additional pairs in $Nodes_j$ that provide the needed ports.

By using the nodes selected in the reachability graph, the subsequent procedure ABSTRACTPLAN (described in Section 4.2) generates the so-called *abstract plan*. The abstract plan is a graph with nodes representing the actions to be executed, and edges denoting the temporal dependencies among them. There are three kinds of dependencies: those representing ordering constraints on state transitions to be executed in sequence on the same component, those indicating that a state q in one component must be entered before a state q' in another component because q' requires ports provided by q, and those indicating that a state q in one component must be left before leaving state q' in another component because q requires ports provided by q'.

A deployment plan could be in principle generated by performing a topological visit of the nodes in the abstract plan. However, such topological visit is not always possible due to the presence of cycles. Intuitively, cycles represent situations in which a state change from q to q' is at the same time *needed*, to activate a new port p' provided by q', and *prevented*, because otherwise a port p provided by q would become inactive. The problem is solved by PLANSYNTHESIS (described in Section 4.3) that performs an *adaptive* topological sort: the graph is modified when a cycle is encountered by duplicating the component required to perform the state change from q to q'. One instance remains in state q thus continuing to provide the port p, while a second instance moves to q' in order to activate the new port p'.

In the remainder of the section we discuss separately the three main phases of the algorithm: the reachability analysis consisting of the generation of the reachability graph and the subsequent selection of the needed nodes, the generation of the abstract plan, and finally the synthesis of the concrete plan.

4.1. 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

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

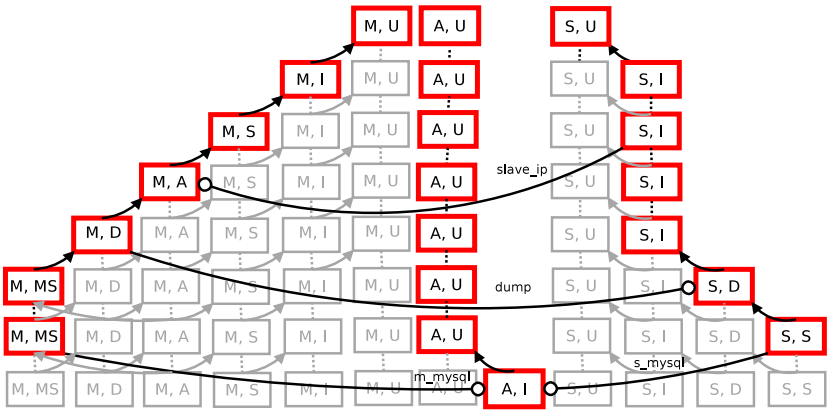


Fig. 4. Reachability graph and component selection for the running example. (For interpretation of the colors in this figure, the reader is referred to the web version of this article.)

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.

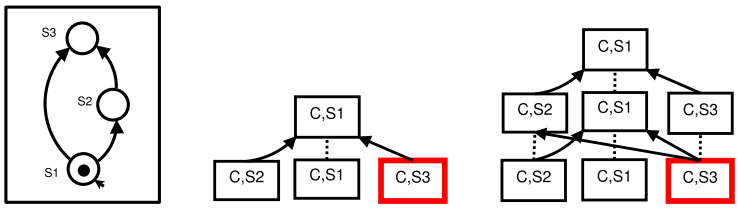
Algorithm 2 creates a *reachability graph* that visually could be seen as a pyramid where the top level contains all the component types in their initial state and, at every step, a new level is produced by adding new component type-state pairs, reachable from the ones at the previous level (see the grey part of Fig. 4). $Nodes_n$ is the set of the type-state pairs at level n, while $Arcs_n$ represents the possible ways a type-state pair can be obtained; $x \longrightarrow y$ means that component state y, at level $n + 1$, is obtained from x at level n 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 the component states at the previous level (Lines 5–8). Finally, variable $Arcs_n$ 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 because it is not possible to add infinitely often new component type-state pairs, as these are finite. When the fix-point is reached, if the last set does not contain the target component type-state pair, it means 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 Algorithm 3.

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, we select at level $i - 1$ 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

Algorithm 3 componentSelection().

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



(a) Component C (b) Lazy strategy (c) Default strategy

Fig. 5. Difference between lazy and default strategies.

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 article; a discussion about this specific point can be found in [31].

For every require port needed by the selected pairs of level i that are not copies, we select a pair at level $i-1$ 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 a heuristics, abstracted by the heuristic_prov function, to decide which pair is used as a provider.

Fig. 4 depicts the output of this first phase for the MySQL master–slave example. The grey and red part is the reachability graph generated by Algorithm 2, while the part only in red represents a possible selection performed by Algorithm 3. The master, slave and application component types are denoted by M, S and A respectively, and each state is referred by its initial upper-case letters: U for uninst, I for inst, S for serving, A for auth, D for dump and MS for masterserving.

The first level of Fig. 4 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 $\langle \mathsf{M}, \mathsf{S} \rangle$ is added. At next step, pair $\langle \mathsf{M}, \mathsf{A} \rangle$ can also be added since it derives from $\langle \mathsf{M}, \mathsf{S} \rangle$ and its requirement of the port *slave_ip* is fulfilled by $\langle \mathsf{S}, \mathsf{I} \rangle$, appearing at the previous level. The generation of the reachability graph proceeds as depicted until the pair $\langle \mathsf{A}, \mathsf{I} \rangle$ is added: this is the last level as no new type-state pairs can be generated.

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

Lazy strategy Let us denote with k the total number of different type-state pairs. In the worst case Algorithm 2 adds at every iteration only one pair and so the upper bound on the number of iterations is k. There are a few possible strategies to decide the level at which to stop the iteration for building up the reachability graph. Indeed, instead of adopting the *default strategy* described above that enforces to proceed until the fix-point is reached, one could employ a *lazy strategy* stopping as soon as the target node is produced, if ever. The lazy strategy minimizes the work to be done in this phase but has the drawback that not all paths to reach a certain pair may have been discovered yet. Fig. 5 shows a simple example highlighting the difference between the reachability graph obtained applying a lazy and the default strategy considering as target the component C in state S3. The reachability graph, built using the default strategy, has one layer more where a new path S2 $\longrightarrow$ S3 to the target is discovered by an additional iteration of the algorithm. In the general case an alternative path may present advantages over the others and so the lazy strategy may result in a poorer deployment plan as there may be less choices available.

Algorithm 4 ABSTRACTPLAN().

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

4.2. 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) $\xrightarrow{r}\!\!\!\!\rightarrow$: precedence of instances that provide a port r w.r.t instances requiring it;
(iii) $\xrightarrow{r}\!\!\!\!\dashrightarrow$: precedence of an instance requiring a port r w.r.t. actions that deactivate it.

Algorithm 4 is used to derive the abstract plan. To generate an abstract plan we consider an instance for every maximal path in the reachability graph 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. 4, for the master–slave example there are three maximal paths: one for the master (starting from $\langle$M, U$\rangle$ and ending in $\langle$M, MS$\rangle$), one for the dummy component, and one for the slave (starting from $\langle$S, U$\rangle$ and ending in $\langle$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.

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

The abstract plan for the running example is displayed in Fig. 6. The rows represent the life-cycles of master, slave and application, respectively. The $\xrightarrow{slave\_ip}\!\!\!\!\rightarrow$ from $\langle s, \mathsf{U}, \mathsf{I}\rangle$ to $\langle m, \mathsf{S}, \mathsf{A}\rangle$ expresses the fact that the *stateChange* of slave from uninst to inst must precede the *stateChange* of master from serving to auth because state auth of server requires the port *slave_ip*, provided by slave in state inst. The twin $\dashrightarrow$ arc states that master must switch from auth to dump before slave switches from inst to dump, as this state ceases providing the port *slave_ip*, otherwise its requirement would become unfulfilled. Following the same principle we can interpret the pair of arcs $\langle m, \mathsf{A}, \mathsf{D}\rangle \xrightarrow{r}\!\!\!\!\rightarrow \langle s, \mathsf{I}, \mathsf{D}\rangle$ and $\langle s, \mathsf{D}, \mathsf{S}\rangle \dashrightarrow \langle m, \mathsf{D}, \mathsf{MS}\rangle$ for the port *dump*. Finally, the target is represented by node $\langle a, \mathsf{U}, \mathsf{I}\rangle$, namely application entering state inst. This state requires two ports, *m_mysql* and *s_mysql* provided respectively by master in state masterserving and slave in state serving. Two $\xrightarrow{r}\!\!\!\!\rightarrow$ arcs (together with their $\dashrightarrow$ counterparts) are thus added with destination $\langle a, \mathsf{U}, \mathsf{I}\rangle$, one from $\langle s, \mathsf{D}, \mathsf{S}\rangle$ and the other one from $\langle m, \mathsf{D}, \mathsf{MS}\rangle$.

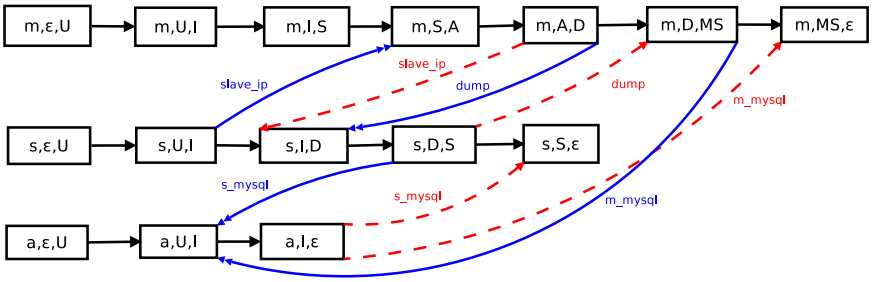


Fig. 6. Abstract plan for the running example.

Algorithm 5 PLANSYNTHESIS().

```
1: Plan = [ ]; ToVisit = [ ]; finished = false
2: for all n = ⟨i, x, y⟩ ∈ 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:         ⟨i, x, y⟩ = ToVisit.pop()
9:         if x == ε then
10:            processInstanceEdge(⟨i, x, y⟩)
11:        else if y == ε then
12:            Plan.append(del(i))
13:            processRedEdges(⟨i, x, y⟩)
14:        else
15:            Plan.append(stateChange(⟨i, x, y⟩))
16:            processRedEdges(⟨i, x, y⟩)
17:            processBlueEdges(⟨i, x, y⟩)
18:            processInstanceEdge(⟨i, x, y⟩)
19:        if InstMap[i] == T_target ∧ y == q_target then finished = true
20:        Act.remove(⟨i, x, y⟩)
21:    until ToVisit == [ ] ∨ finished
22:    if ¬finished then
23:        let n ∈ Act where ∄n′ ∈ Act . (n′ ⟶ n ∈ Prec ∨ n′ -r↠ n ∈ Prec) in
24:            Duplicate(n)
25:    for all n ∈ Act . no_incoming_edges(n) do
26:        ToVisit.push(n)
27: until finished
```

4.3. 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 5 (comprising the auxiliary functions and procedures in Algorithm 6 and Algorithm 7) builds the plan adding actions to a list called *Plan*. Nodes can be visited if they do not have precedence constraints, i.e., incoming arcs. Function `no_incoming_edges` is used to check this condition. Visitable nodes are stored in a stack, named *ToVisit*. When a node becomes visitable it is pushed onto *ToVisit* in order to be later processed.

The algorithm relies on three auxiliary procedures, processInstanceEdges, processBlueEdges and processRedEdges in Algorithm 6, aimed at dealing respectively with $\longrightarrow$, $\twoheadrightarrow$ and $\dashrightarrow$ edges, the three kinds of edges present in the abstract plan. When a node of the abstract plan is visited, procedure processRedEdges and procedure processBlueEdges deal with its outgoing $\dashrightarrow$ and $\twoheadrightarrow$ arcs. They add *unbind* and *bind* actions to the *Plan* list and remove the corresponding arcs from the abstract plan. Notice that the *unbind* actions are not added if the visited node corresponds to a *del* action: in fact, when an instance is deleted all its connections are implicitly removed. Moreover, if the removal of an arc makes a node visitable, they add it to the *ToVisit* stack. Similarly, procedure processInstanceEdge removes the precedence arc $\longrightarrow$, adding its target node to the *ToVisit* stack if it has no incoming arcs.

The topological visit finishes when a node is visited which corresponds to a *stateChange* towards the state q_{target} of an instance of type $\mathcal{T}_{target}$.

Algorithm 6 Auxiliary procedures for plan synthesis.

```
1: procedure processInstanceEdge(⟨i, x, y⟩)
2:     let n ∈ Act where ⟨i, x, y⟩ ⟶ n ∈ Prec in
3:         Prec.remove(⟨i, x, y⟩ ⟶ n)
4:         if no_incoming_edges(n) then ToVisit.push(n)
5: procedure processBlueEdges(⟨i, x, y⟩)
6:     for all ⟨i, x, y⟩ -r->> ⟨i′, x′, y′⟩ ∈ Prec do
7:         Plan.append(bind(r, i, i′)); Prec.remove(⟨i, x, y⟩ -r->> ⟨i′, x′, y′⟩)
8:         if no_incoming_edges(⟨i′, x′, y′⟩) then ToVisit.push(⟨i′, x′, y′⟩)
9: procedure processRedEdges(⟨i, x, y⟩)
10:    for all ⟨i, x, y⟩ -r--> ⟨i′, x′, y′⟩ ∈ Prec do
11:        if y ≠ ε then Plan.append(unbind(r, i′, i))
12:        Prec.remove(⟨i, x, y⟩ -r--> ⟨i′, x′, y′⟩)
13:        if no_incoming_edges(⟨i′, x′, y′⟩) then ToVisit.push(⟨i′, x′, y′⟩)
```

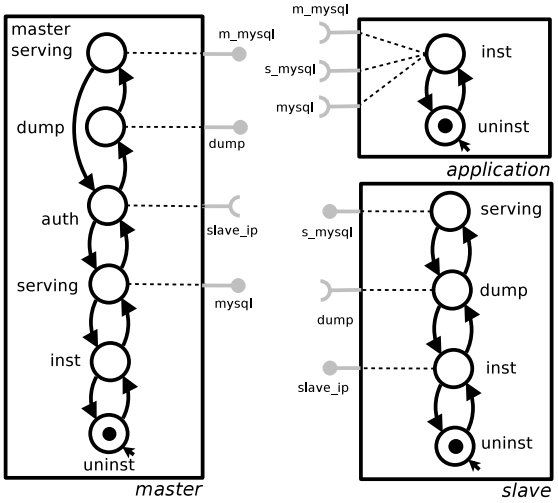


Fig. 7. Component types for the new scenario requiring duplication.

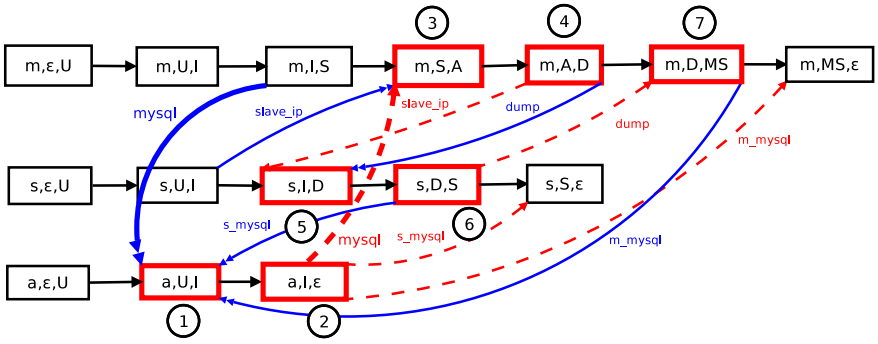


Fig. 8. Cyclic abstract plan for the modified running example. (For interpretation of the colors in this figure, the reader is referred to the web version of this article.)

Notice that the topological visit could be blocked by a cycle in the abstract plan before reaching the target. As an example, consider a slight modification of the running example in which the application architecture demands a secondary component of type master in state serving. This new scenario may be modeled by modifying the application component in a way that the target inst state has an additional required port *mysql* as depicted in Fig. 7.

The resulting abstract plan is shown in Fig. 8 where nodes forming a cycle are highlighted in red and tagged by identifiers. In this abstract plan there are two nested cycles, one containing the other: one formed by nodes $\{1, 2, 3, 4, 5, 6, 7\}$ and a smaller one formed by nodes $\{1, 2, 3, 4, 7\}$.

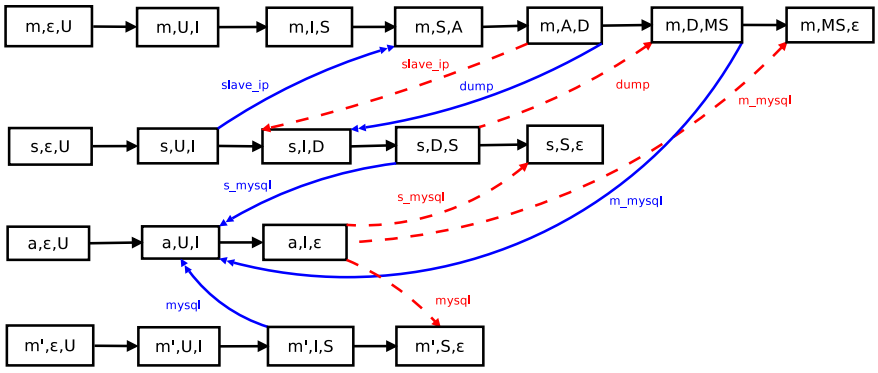


Fig. 9. Effect of the duplication of node $\langle m, \mathsf{S}, \mathsf{A}\rangle$ in Fig. 8.

Algorithm 7 DUPLICATE($\langle i, x, y\rangle$).

```
1: i' = getFreshName()
2: DUPLICATEPLAN(i, i')
3: for all n' -r-> n ∈ Prec . ⟨i, x, y⟩ ⟶* n ∧
4:                 ( ∄n'' ∈ Act . n'' ⟶* n ∧ n'' -r->> n''' ∈ Prec ) do
5:     Prec.remove(n' -r-> n)
6: while ∃n -r-> n' ∈ Prec . ⟨i, x, y⟩ ⟶* n ∧
7:             ( ∄⟨i, x', y'⟩ ∈ Act . ⟨i, x', y'⟩ ⟶* n ∧ r ∉ InstMap[i].P(x') ) do
8:     let m ∈ Act where m ⟶* n' ∧ ( ∄m' ∈ Act . m' ⟶ m ∈ Prec ) in
9:         DUPLICATE(m)

10: procedure DUPLICATEPLAN(i, i')
11:     for (j = Plan.size() − 1; j ≥ 0; j = j − 1) do
12:         if Plan[j] == bind(r, i, z) then Plan[j] = bind(r, i', z)
13:         else if Plan[j] == bind(r, z, i) then Plan.insert(bind(r, z, i'), j)
14:         else if Plan[j] == unbind(r, i, z) then Plan[j] = unbind(r, i', z)
15:         else if Plan[j] == unbind(r, z, i) then Plan.insert(unbind(r, z, i'), j)
16:         else if Plan[j] == new(i : T) then Plan.insert(new(i' : T), j)
17:         else if Plan[j] == stateChange(⟨i, x, y⟩) then
18:             Plan.insert(stateChange(⟨i', x, y⟩, j)
```

In general, cycles occur in the abstract plan when an instance z is required to be in two states q and q' at the same time as they enact different provide ports simultaneously demanded. It is then necessary to duplicate that instance. A new instance z' that will stay in state q is created, thus keeping the first provide port active. The original instance is then allowed to perform the $stateChange(z, q, q')$ action to activate the second provide port. In this way, both provide port will be contemporaneously activated by two distinct duplicated instances of the same component type. For this reason the technique used to deal with this issue takes the name of *duplication*.

In Fig. 8 the instance which is required at the same time to be in two different states is m of type master, that should contemporaneously activate the ports *mysql* (to satisfy a requirement of the instance a) and *dump* (to satisfy a requirement of the instance s). Fig. 9 depicts the effect of applying the duplication procedure. A new instance m', of type master, is created and its life cycle stops in state serving, keeping providing port *mysql*. Moreover, the pair of $\overset{mysql}{\dashrightarrow}$ and $\overset{mysql}{\dashrightarrow}$ arcs (highlighted in Fig. 8) are moved towards the new instance m' to reflect the fact that the requirement of the instance a is now served by the new instance m'. The elimination of the arc $\overset{mysql}{\dashrightarrow}$ incoming in the node $\langle m, \mathsf{S}, \mathsf{A}\rangle$ has the effect of removing both cycles.

To find the instance to be duplicated we rely on selecting one node in the abstract plan with only incoming $\dashrightarrow$ arcs (Line 23 of Algorithm 5). The existence of such node is guaranteed by Lemma 5.1 reported in the next Section. After execution of the duplication procedure, all the incoming $\dashrightarrow$ arcs will be redirected to the new instance, thus the selected node will have no incoming edges. This guarantees the possibility to re-start the topological visit.

Let $\langle i, x, y\rangle$ be the node selected for duplication. The DUPLICATE($\langle i, x, y\rangle$) function (see Algorithm 7) assigns a fresh name i' to the new instance (Line 1). Then, the procedure DUPLICATEPLAN is called to duplicate the actions already performed on i to generate also the new instance i' (Lines 11–18). The actions *new* and *stateChange* of i' are added to the plan immediately after the *new* and *stateChange* actions of i (Lines 16, 18). Similarly, *bind* and *unbind* actions where i requires ports provided by other instances are replicated (Lines 13, 15). The *bind* and *unbind* actions where i provides ports to other instances

are replaced with *bind* and *unbind* actions involving i' instead of i (Lines 12, 14). In this way, the new instance i' will be considered instead of i by all those instances that in the previous version of the deployment plan connected to some provide port of i. For this reason the abstract plan should be modified by removing $\dashrightarrow$ corresponding to bindings that have been redirected from i to the new instance i' (Lines 3–5).

The new instance i' will remain in state x until the end of the plan in order to keep its provide ports active. In case the state x activates also some require ports, we have to guarantee the presence for the rest of the plan of other instances activating the complementary provide ports. This is realized by applying the duplication procedure on those instances that are connected to such require ports (Lines 6–9).

Notice that the abstract plan is not extended with nodes and arcs corresponding to the new instances generated by DUPLICATE. In fact, all the actions involving such instances are directly added to the deployment plan by the DUPLICATEPLAN procedure. The effect on the abstract plan of the execution of the DUPLICATE procedure is simply the elimination of $\dashrightarrow$ arcs previously directed towards the instances that have been duplicated. When the duplication procedure has been completed, the topological visit can be restarted with new nodes that can be visited (Lines 25–26 of Algorithm 5); at least the initial node with only $\dashrightarrow$ incoming arcs becomes visitable because such arcs have just been removed.

Listing 1: Deployment plan for the running example.

```
Plan[1] = [Create instance application:Application:Uninst]
Plan[2] = [Create instance slave:Slave:Uninst]
Plan[3] = [Create instance master:Master:Uninst]
Plan[4] = [master : change state from Uninst to Inst]
Plan[5] = [master : change state from Inst to Serving]
Plan[6] = [slave : change state from Uninst to Inst]
Plan[7] = [slave : bind port slave_ip to master]
Plan[8] = [master : change state from Serving to Auth]
Plan[9] = [master : change state from Auth to Dump]
Plan[10] = [master : bind port dump to slave]
Plan[11] = [master : unbind port slave_ip from instance slave]
Plan[12] = [slave : change state from Inst to Dump]
Plan[13] = [slave : change state from Dump to Serving]
Plan[14] = [slave : bind port s_mysql to application]
Plan[15] = [slave : unbind port dump from instance master]
Plan[16] = [master : change state from Dump to MasterServing]
Plan[17] = [master : bind port m_mysql to application]
Plan[18] = [application : change state from Uninst to Inst]
```

As an example, starting from the abstract plan of Fig. 6 (the version that does not require duplication), the METIS tool described in Section 6 generates the deployment plan in Listing 1.

5. Formal analysis of the algorithm

In this section we prove that DEPLOYMENTPLANNER defined in Algorithm 1 terminates and it 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 description of the universe of component types and the component type-state target pair.

To prove the termination of DEPLOYMENTPLANNER we rely on the following lemma stating that, in presence of circularities blocking the topological visit of the abstract plan, there exists a node that has only $\dashrightarrow$ incoming arcs. This is the node chosen to start the duplication phase that will rearrange the plan ensuring that the selected node becomes visitable after duplication.

Lemma 5.1. *If every node of the abstract plan has at least an incoming arc then there exists a node that has only* $\dashrightarrow$ *incoming arcs.*

Proof. By contradiction, let us suppose that there is no node with only $\dashrightarrow$ incoming arcs. Let us select one node of the abstract plan. By assumption, this node has at least one $\twoheadrightarrow$ or $\longrightarrow$ incoming arc. Let us follow backward this arc and select the source node. Also this node has at least one $\twoheadrightarrow$ or $\longrightarrow$ incoming arc. We can therefore reiterate the procedure by selecting its source node. Since the number of nodes in the abstract plan is finite we have that the selection procedure will eventually select a node that has been already visited. This means that there exists a cycle having only $\twoheadrightarrow$ and $\longrightarrow$ arcs.

However, this cycle cannot exist for the following argument. All the nodes in an abstract plan correspond to nodes in the reachability graph (produced by REACHABILITYANALYSIS) which is partitioned in layers $Nodes_0, \cdots, Nodes_n$. By construction (see ABSTRACTPLAN, Lines 9–12 and 16) we have that both $\longrightarrow$ and $\twoheadrightarrow$ arcs connect nodes in a layer to nodes in a strictly greater layer. For this reason it is not possible to have cycles including only $\twoheadrightarrow$ and $\longrightarrow$ arcs. □

We are now ready to prove that the DEPLOYMENTPLANNER algorithm terminates.

Theorem 5.2 *(Termination).* DEPLOYMENTPLANNER *terminates.*

Proof. The "repeat until" cycle of Algorithm 2 (Lines 2–17) exits when the set $Nodes_n$ is equal to the set $Nodes_{n-1}$. This happens when there is no pair $\langle \mathcal{T}, q' \rangle$ that is not already present in $Nodes_{n-1}$ is added to $Nodes_n$ (Lines 8 and 11). Since the variable $Nodes_n$ is a set and the number of component type-state pairs is finite, the "repeat until" cycle cannot perform a number of iteration greater than the number of the possible component type-state pairs and therefore Algorithm 2 always terminates.

If the target component-type state pair is not reachable, DEPLOYMENTPLANNER terminates (Line 2 of Algorithm 1). Otherwise, it executes in sequence Algorithms 3, 4, and 5.

Algorithms 3 and 4 trivially terminate because their "for" cycles have a finite number of iterations and the "repeat until" cycle of Algorithm 4 (Lines 18–22) is bound by the length of the maximal paths, which coincides with n in the worst case.

Algorithm 5 contains two nested "repeat until" cycles. The inner one (Lines 7–21) always terminates because it executes a topological visit of the finite abstract plan. The termination of the external one (Lines 6–27) depends on termination of the possibly invoked procedure DUPLICATE (Algorithm 7). This is a recursive procedure whose termination is guaranteed by the following argument. Let $\langle i, x, y \rangle$ be a node selected for duplication. According to the "while" loop at Lines 6–9, the DUPLICATE procedure could be invoked recursively on $m = \langle i, x, y \rangle$ only if the instance i has a subsequent node n' having an incoming $\overset{r}{\dashrightarrow}$ arc for a provide port r which remains continuously active in all the current states traversed between node m and n. But this cannot occur because all the $\dashrightarrow$ arcs incoming in subsequent nodes of the instance i, without a corresponding twin $\twoheadrightarrow$ arc, are removed from the abstract plan (Lines 3–5). In fact, the provide port r activated by all the current states between node m and n guarantees that there is no arc $\overset{r}{\twoheadrightarrow}$ outgoing from any of such nodes, because an arc $\overset{r}{\twoheadrightarrow}$ should exit from a node corresponding to an action that activates the previously inactive provide port r. □

Before proving the correctness of the entire DEPLOYMENTPLANNER algorithm, we first focus on the initial reachability analysis phase. We prove that the reachability graph computed by Algorithm 2 contains all and only those pairs $\langle \mathcal{T}, q \rangle$ such that there exists a deployment plan generating at least one component of type $\mathcal{T}$ in state q.

Lemma 5.3. *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 2 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 for every $\langle \mathcal{T}, q \rangle \in Nodes_n$, and for every $h_{\langle \mathcal{T}, q \rangle} > 0$, there exists a deployment plan from an empty configuration to a configuration containing $h_{\langle \mathcal{T}, q \rangle}$ components of type $\mathcal{T}$ in state q for every $\langle \mathcal{T}, q \rangle \in Nodes_n$. 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.

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 2). Moreover, for every requirement r of a component of type $\mathcal{T}$ in state q there exists a pair $\langle \mathcal{T}'', q'' \rangle \in Nodes_i$ s.t. a component of type $\mathcal{T}''$ in state q'' provides r (Line 7). Therefore, starting from a configuration having for every $\langle \mathcal{T}, q \rangle \in Nodes_i$ exactly $h_{\langle \mathcal{T}, q \rangle} + \sum_{\langle \mathcal{T}, q' \rangle \in Nodes_{i+1} \setminus Nodes_i} h_{\langle \mathcal{T}, q' \rangle}$ components of type $\mathcal{T}$ in state q (this configuration could be obtained by inductive hypothesis) it is possible to perform for every pair $\langle \mathcal{T}, q' \rangle$ of $Nodes_{i+1} \setminus Nodes_i$ exactly $h_{\langle \mathcal{T}, q' \rangle}$ state changes to obtain a component of type $\mathcal{T}$ in state q'. At this point, the obtained configuration has for every pair $\langle \mathcal{T}, q \rangle \in Nodes_{i+1}$ a number of components of type $\mathcal{T}$ in state q that is greater or equal to $h_{\langle \mathcal{T}, q \rangle}$. From this configuration, with delete actions it is possible to obtain the desired configuration. These delete actions do not violate correctness since for every pair $\langle \mathcal{T}, q \rangle \in Nodes_{i+1}$ we require the presence of at least a component of type $\mathcal{T}$ in state q (i.e., $h_{\langle \mathcal{T}, q \rangle} > 0$).

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 initial hypothesis. □

We now move to the proof of soundness and completeness of the DEPLOYMENTPLANNER algorithm.

Theorem 5.4 *(Soundness). Given a universe of components U, a component type $\mathcal{T}_{target}$, and a state q_{target}, if* DEPLOYMENTPLANNER*$(U, \langle \mathcal{T}_{target}, q_{target} \rangle)$ 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 $\mathcal{T}_{target}$ in state q_{target}.*

Proof. We first observe that $\mathcal{C}_m$ contains at least one component of type $\mathcal{T}_{target}$ in state q_{target}. From Line 19 of Algorithm 5 we know that the last node of the topological visit is $\langle i, x, q_{target}\rangle$, where the type of the instance i is $\mathcal{T}_{target}$. This ensures that the action $stateChange(i, x, q_{target})$ is added to the synthesized plan (Line 15 of Algorithm 5), and it can be followed only by *bind* and *unbind* actions added by the processBlueEdges and processRedEdges procedures. This guarantees that in $\mathcal{C}_m$ there is a component i of type $\mathcal{T}_{target}$ in state q_{target}.

It remains to prove that the deployment plan $\mathcal{C}_0 = \langle U, \emptyset, \emptyset, \emptyset\rangle \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_m} \mathcal{C}_m$ is correct. We proceed by induction on m. For $m = 0$ it is enough to observe that $\mathcal{C}_0$ is the empty configuration which is correct by definition. Now consider $\mathcal{C}_{j-1} \xrightarrow{\alpha_j} \mathcal{C}_j$ with $\mathcal{C}_{j-1}$ correct by inductive hypothesis.

The correctness of $\mathcal{C}_j$ can be proven by case analysis on the kind of the 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 $\alpha_j = stateChange(i, x, y)$ then α_j may invalidate correctness in two ways: either i stops providing a port p needed by some other component, or state y of i requires a port r which is not provided in $\mathcal{C}_j$. In the first case, if i' is the component requiring p, by Algorithm 4 (Line 16) 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 (Line 23). This guarantees that in Algorithm 5 (Lines 13 or 16) an *unbind* action is added to the plan before the $stateChange(i, x, y)$, 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 (Line 17 of Algorithm 5) before the $stateChange(i, x, y)$ action, and correctness is not violated.

If $\alpha_j = del(i)$ correctness could be violated if i stops providing a port p needed by some other component. This corresponds to the first of the two sub-cases already analyzed for $\alpha_j = stateChange(i, x, y)$.

If $\alpha_j = unbind(r, i', i)$ we observe that such action is added to the plan only when the source node of an arc $\langle i, x, y\rangle \overset{r}{\dashrightarrow} \langle i', x', y'\rangle$ is visited where $y \neq \varepsilon$ (Line 11 of Algorithm 6). This guarantees that the $stateChange(i, x, y)$ has been just added to the plan (Line 15 of Algorithm 5) so the current state of i in $\mathcal{C}_j$ is y. Moreover, the arc $\langle i, x, y\rangle \overset{r}{\dashrightarrow} \langle i', x', y'\rangle$ guarantees that y does not activate the require port r (Line 23 of Algorithm 4). Hence, unbinding a require port which is not active cannot violate the correctness of the configuration. □

Theorem 5.5 *(Completeness). Given a universe of components U, a component type $\mathcal{T}_{target}$, and a state q_{target}, if a solution exists to the deployment problem on input $I = (U, \mathcal{T}_{target}, q_{target})$, then* deploymentPlanner$(U, \langle \mathcal{T}_{target}, q_{target}\rangle)$ *returns a deployment plan for I.*

Proof. By Theorem 5.2 the algorithm surely terminates. By Lemma 5.3 we have that $\langle \mathcal{T}_{target}, q_{target}\rangle$ is in the computed reachability graph, then the algorithm completes by returning a plan. By Theorem 5.4 the returned plan is a correct deployment plan for $\mathcal{T}_{target}$ and q_{target}. □

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

Theorem 5.6 *(Complexity). The* deploymentPlanner *algorithm runs in a time which is polynomial in the size of the input.*

Proof. The input of the problem includes the description of the universe of component types plus the target pair. We identify three relevant parts of the input: 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 number of ports.[5] 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 graph has 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 reachability graph has at most $k + 1$ levels, hence Algorithm 2 runs in $O(hk^3)$ time.

To select the bindings and the components (Algorithm 3), 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 $O(k^2)$, Algorithm 3 takes $O((b + h)k^3)$ time.

The computation of the maximal paths in Algorithm 4 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.

Algorithm 5 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 there are at most k nodes, the cost of detecting the node to duplicate has the worst case

[5] For instance, considering the running example, since the number of component type-state pairs is 6 for the master, 4 for the slave, and 2 for the dummy application component $k = 12$, $b = 2$, while $h = 9$ since the three components have 5 provide and 4 require ports.

cost of $O(k)$. Then, in the worst case, the cost of selecting all the nodes to duplicate is $O(k^2)$. The DUPLICATEPLAN procedure (Line 10) 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 of all the DUPLICATEPLAN calls is $O(hk^3)$. The DUPLICATE procedure examines all incoming and outgoing arcs of the remaining actions of the instance to duplicate. Since an instance has at most k actions and every action has $2hk^2+1+h$ arcs, the cost of the duplication procedure, excluding the recursive calls, 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 $O((b+h)k^3)+O(hk^4)$, which considering b bound by k, amounts to $O(hk^4)$. □

6. Validation

In order to assess the effective viability of the proposed approach, we have implemented METIS. This is a proof of concept implementation developed as open source project at http://www.aeolus-project.org/software/. The implementation is about 3.5K lines of code written in OCaml and it is distributed under GPL license.

METIS takes as input an encoding of the universe in the JSON format following the Aeolus Universe JSON Schema.[6] For example, Listing 2 shows the specification of the Application component type depicted in Fig. 2. Each component type is specified by means of a name (field `name`) and by the automaton describing its behavior. Each state is defined by a `name`, a list of `successors`, and a list of the provided and required ports.[7] METIS outputs the sequence of actions needed to deploy the target in textual format and the *abstract plan* as a directed graph in `dot` format [33,34].

Listing 2: Sample input.

```
{
   "states": [
    {
     "provide": {},
     "require": {},
     "initial": true,
     "name": "Uninst",
     "successors": [ "Inst" ]
    },
    {
     "provide": {},
     "require": { "m_mysql" : 1 , "s_mysql" : 1  },
     "successors": [ "Uninst" ],
     "name": "Inst"
    }
   ],
   "name": "Application"
}
```

Since to the best of our knowledge there is no specific benchmark for application deployment, in order to validate the performances of METIS in a systematic way we tested it by considering artificial instances of the problem of incremental complexity that are synthesized by reproducing a typical installation pattern. As an additional validation, we discuss how METIS has been applied by Mandriva [35] (one of our industrial partners) on real case examples involving the deployment of a so-called "WordPress farm", i.e., a load balanced, replicated blogging service based on WordPress.[8]

6.1. Systematic validation using synthetic scenarios

For a systematic evaluation of METIS, in order to simulate the deployment of big applications, we defined deployment scenarios inspired by a typical installation procedure for mutually dependent components [3]: if there are two mutually dependent components to be installed, the first one can be partially installed in order to provide the functionalities required by the second one, then the second component is fully installed, and finally the first component completes its installation.

We compared METIS with standard planning solvers using an encoding of the deployment problem into the Planning Domain Definition Language (PDDL) [20], nowadays the most used language to define and solve classical planning problems [36]. Each component instance is translated into one PDDL object with possible actions corresponding to state changes. These actions can be acted on the object only when other objects in the configuration provide the required interfaces. The

[6] https://github.com/aeolus-project/utils/tree/master/aeolus-json/data.

[7] Notice that the numerical attribute "1" associated to the require ports is here added just to comply with the Aeolus Universe JSON schema. These numbers, used to model replication criteria, are used by other tools like Zephyrus [32] that use this input format too.

[8] https://wordpress.com/.

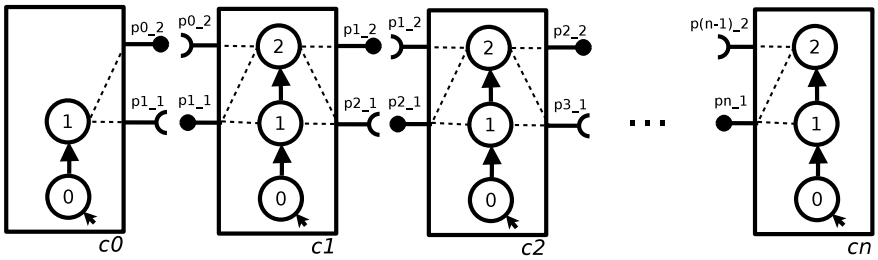


Fig. 10. Component types for the validation with artificial instances.

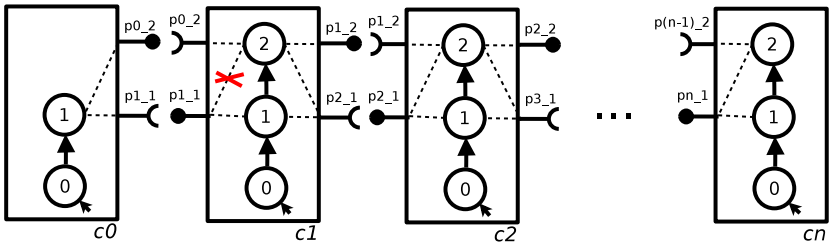


Fig. 11. Modified version requiring duplication.

tested PPDL encoding abstracts from the bind and unbind actions[9] and limits the number of objects that could be concurrently used.[10] We refer the interested reader to [31] for more details about the PDDL encoding.

We have considered scenarios composed by an increasing number of components having two or three states as depicted in Fig. 10. Every scenario is composed by $n+1$ components, c0, c1, …, cn and the target is represented by state 2 of component cn. A valid plan must first create all components, then perform from cn to c0, in sequence, the state change from 0 to 1, and finally perform the state change from 1 to 2 from c1 to cn.

In these scenarios instance duplication is not needed during the generation of the deployment plan. As duplication may introduce a computational overhead, we also considered modified scenarios where we remove the activation of the provide port *pX_1* from the states 2 in some randomly chosen components. Removing this activation of the provided port demands duplicating the instance of component type c*X* in order to simultaneously satisfy the requirements of its two neighbor components. For instance, consider the removal of the provided port as depicted in Fig. 11. If the provided port *p1_1* is missing from state 2 of component c1, then if we want to reach it, a duplicate instance of type c1 must be created. This will remain into state 1 in order to keep providing port *p1_1* needed by state 1 of c0.

In the modified scenarios, the number of random deletions applied was one fifth of the total number of components.

6.1.1. Experimental results

All the tests were performed using a dual core machine with a 2.50 GHz Intel i5 processor, 6 GB of RAM, Ubuntu 12.10 operating system with 64 bit support. We used a time cap of 130 seconds for all the runs. To solve the encoding of the problems in PDDL we used two planners: Metric-FF [37,38] and Madagascar-p [39].[11] The first solver is based on Graph-Plan [40], a standard planning algorithm to prune the search space. The second solver instead belongs to the Satplan [41] family and encodes the planning problem into a SAT formula and then uses state-of-the-art SAT solvers to find a solution. Since for decidability purposes the use of PDDL requires a finite use of objects, for reducing the search space of the solvers we set in the PDDL encoding the number of components that could be used concurrently to the minimum possible value.

The performance of the two planners is summarized in Table 2 where *error* indicates that the solver exited with an error state without computing the plan, *timeout* means that the solver took more than 130 s and was interrupted, a dash means that the test was not conducted because the previous execution already timed out or ended in error.[12]

The performances of the general planning solvers are quite limited: they are able to compute plans for just a small number of components. These poor performances are due to the fact that the size of the encoding of the planning problem increases exponentially w.r.t. the number of components that need to be deployed concurrently. In particular, Metric-FF

[9] Bind and unbind action can be added to form a valid deployment run in polynomial time in a post processing phase.

[10] This limitation was necessary because all the planning solvers assume a finite number of objects. Without this limitation the classical planning problem becomes undecidable.

[11] In the PDDL encoding that we have used, we required planners supporting the ADL fragment of PDDL. Since the current winners of the International Planning Competition did not support this fragment, we were not able to use them.

[12] By construction the test requiring duplication were not applicable to scenarios with less than 5 components. The dash symbol in this case means that the test was not conducted because not applicable.

Table 2
Performances of standard planners considering from 3 to 6 component types.

Size	No duplication		With duplication	
	Madagascar-p	Metric-FF	Madagascar-p	Metric-FF
3	0.07 s	0.07 s	–	–
4	0.47 s	timeout	–	–
5	2.21 s	error	3.71 s	error
6	error	–	error	–

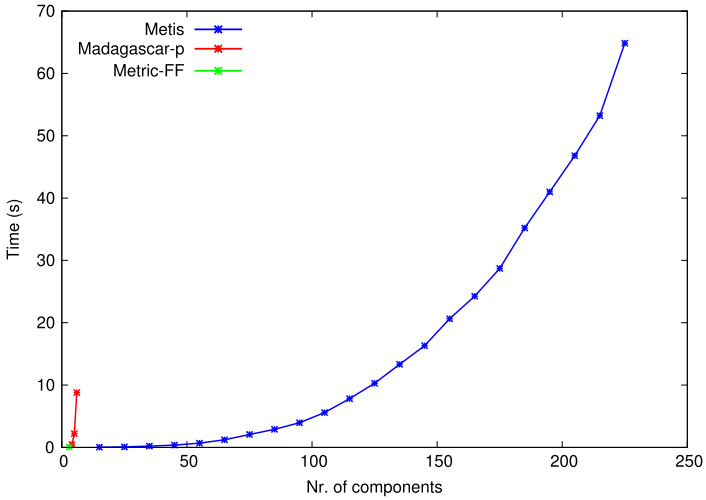


Fig. 12. METIS vs. general purpose planners for scenarios without duplication.

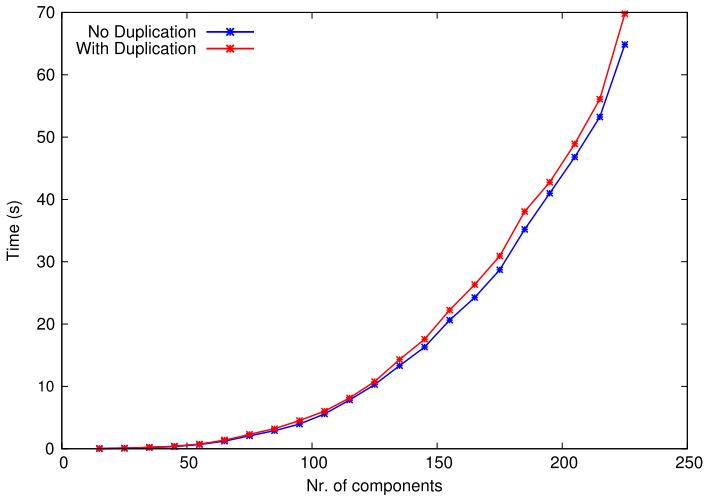


Fig. 13. Performance of METIS for scenarios with and without duplication.

times out because it wastes all its time trying to ground all the possible actions, while Madagascar-p fails because the encoding into SAT becomes too big to be handled. Since as the number of components increases, the number of possible PDDL actions increases exponentially, we conjecture that — no matter what PDDL encoding is used — the general planning solvers cannot be leveraged effectively for big deployment scenarios.

The best planner among the two considered is Madagascar-p, able just to solve instances with only 5 components. Clearly, using standard planners is not a viable solution for automatizing the deployment of applications and ad-hoc planning solution such as METIS have to be used. Indeed, as shown in Fig. 12 comparing the performance of METIS w.r.t. Metric-FF and Madagascar-p, METIS greatly outperforms the general planning tools and it is able to synthesize a deployment plan of 225 components in nearly 65 seconds (after that it consumes all the available memory, returning an error).

Even when duplication is required, as depicted in Fig. 13, METIS is still able to synthesize a deployment plan for more than 200 components in less than a minute. METIS can therefore be effectively used to compute deployment plans involving hundreds of components.

6.2. Real life usage of METIS

While to conduct a systematic validation of METIS we had to rely on "synthetic" scenarios, specific real use cases have been already considered to validate METIS in a production environment. In particular, METIS has been included in Aeolus Blender [42], an integrated solution for automatic application deployment developed by the Mandriva company. Blender is a tool that integrates in a unique solution METIS, the configuration optimizer Zephyrus [32], and the deployment engine Armonic [43].

Blender relies on a repository of components defined in an ad-hoc formalism [44]. Starting from this repository it first uses Zephyrus to compute an optimal configuration satisfying the user desiderata, then it generates a universe of component types described according to the Aeolus Universe JSON Schema processed by METIS to compute a deployment plan for the optimal configuration. This plan is later concretely executed by Armonic to actually deploy the desired system in an OpenStack [45] cloud environment.

Blender is an ongoing effort and, for now, supports a limited number of components, but sufficient to deploy several different installations of the blogging service WordPress.

A discussion of the Blender software is outside the scope of the current article. We just focus on how METIS is used by Blender to deploy simple installations of WordPress. We describe in details the simplest possible installation based on 3 components only (with a deployment plan of 16 actions), while we simply sketch a more complex installation requiring 12 components (with a deployment plan of 61 actions).

The installation of a WordPress server requires additional components. In particular WordPress requires the installation of a database and a Httpd server. Using Blender, Zephyrus already computes a final configuration containing one instance of WordPress, one instance of the MySQL database and one instance of Apache 2 (the Httpd server) producing a universe file containing all these components. For instance, what follows is the excerpt of the universe file in which the WordPress component is described.

Listing 3: WordPress JSON representation.

```
{
   "states": [
    {
     "provide": {},
     "require": {},
     "initial": true,
     "name": "Installed",
     "successors": [ "Template" ]
    },
    {
     "provide": {},
     "require": {},
     "successors": [ "Configured" ],
     "name": "Template"
    },
    {
     "provide": {},
     "require": { "@Mysql/Active/add_database": 1 },
     "successors": [ "Active" ],
     "name": "Configured"
    },
    {
     "provide": {},
     "require": {
      "@Httpd/Active/start": 1,
      "@Mysql/Active/add_database": 1,
      "@Httpd/Configured/get_document_root": 1
     },
     "name": "Active"
    } ],
   "name": "Wordpress"
}
```

The equivalent graphical representation of the WordPress component is depicted in Fig. 14. Fig. 15 shows instead the abstract plan generated by METIS for reaching the WordPress component in `Active` state while the deployment plan is as follows.

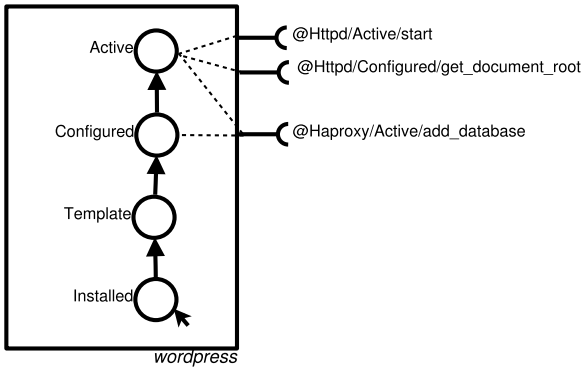


Fig. 14. Aeolus WordPress component generated automatically by Blender.

Listing 4: Deployment plan for the simple WordPress example.

```
Plan[1] = [Create instance wordpress:Wordpress:Installed]
Plan[2] = [Create instance httpd:Httpd:Installed]
Plan[3] = [Create instance mysql:Mysql:Installed.InstalledOnMBS]
Plan[4] = [mysql : change state from Installed.InstalledOnMBS to Installed]
Plan[5] = [mysql : change state from Installed to SetRootPassword]
Plan[6] = [mysql : change state from SetRootPassword to Configured]
Plan[7] = [mysql : change state from Configured to Active.ActiveOnMBS]
Plan[8] = [mysql : change state from Active.ActiveOnMBS to Active]
Plan[9] = [mysql : bind port @Mysql/Active/add_database to wordpress]
Plan[10] = [httpd : change state from Installed to Configured]
Plan[11] = [httpd : bind port @Httpd/Configured/get_document_root to wordpress]
Plan[12] = [httpd : change state from Configured to Active]
Plan[13] = [httpd : bind port @Httpd/Active/start to wordpress]
Plan[14] = [wordpress : change state from Installed to Template]
Plan[15] = [wordpress : change state from Template to Configured]
Plan[16] = [wordpress : change state from Configured to Active]
```

Starting from this plan and with the input of some configuration parameters (e.g., the database root password) Blender is able to deploy a fully functional WordPress server. METIS actions are translated into concrete actions performed on an OpenStack cloud. For instance, the create actions are translated in "apt-get commands" executed on the virtual machines while bind actions correspond to Armonic component method invocations.[13]

This simple WordPress installation requires the execution of METIS for just 1 ms since it involves just 3 components, the largest (i.e., MySQL) with only 7 states. We have also considered a more sophisticated installation where WordPress is replicated to increase its accessibility; in particular, we consider three instances which are load balanced by the HTTP accelerator Varnish. At https://github.com/aeolus-project/metis/tree/multipleTargets/tests it is possible to find the universe file, the abstract plan and the deployment plan computed by METIS for the installation of a WordPress farm constituted by 3 WordPress instances with NFS support for balancing the HTTP requests through the HTTP accelerator Varnish. In this case the final configuration automatically computed by Zephyrus is constituted by Varnish, a MySQL database, a NFS server, and 3 WordPress instances, each of them equipped with an Httpd server and an NFS client. Even in this more complex case (12 components, 61 deployment actions) the generation of the deployment plan was almost instantaneous (12 ms). For the interested reader, screencasts showing the use of Blender can be found at [46].

7. 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 able to compute in polynomial time the actions needed to deploy such a system and we prove soundness and completeness of this novel approach. We then present METIS, a proof of concept implementation of the proposed approach and we validate its efficiency and effectiveness in two ways. On the one hand, by testing its performances on extremely large synthetic problem instances and, on the other hand, by discussing its integration within a framework for automatic deployment of cloud applications called Blender [42].

The developed technique shares some commonalities with both the top–down holistic and the bottom–up DevOps approaches since, given a (partial) global description of the final configuration to obtain, the deployment plan is automatically inferred from a declarative description of individual components. Results are encouraging: METIS is able to produce plans in less than a minute, for scenarios involving hundreds of components.

[13] For more details related to Blender we refer the interested reader to [42].

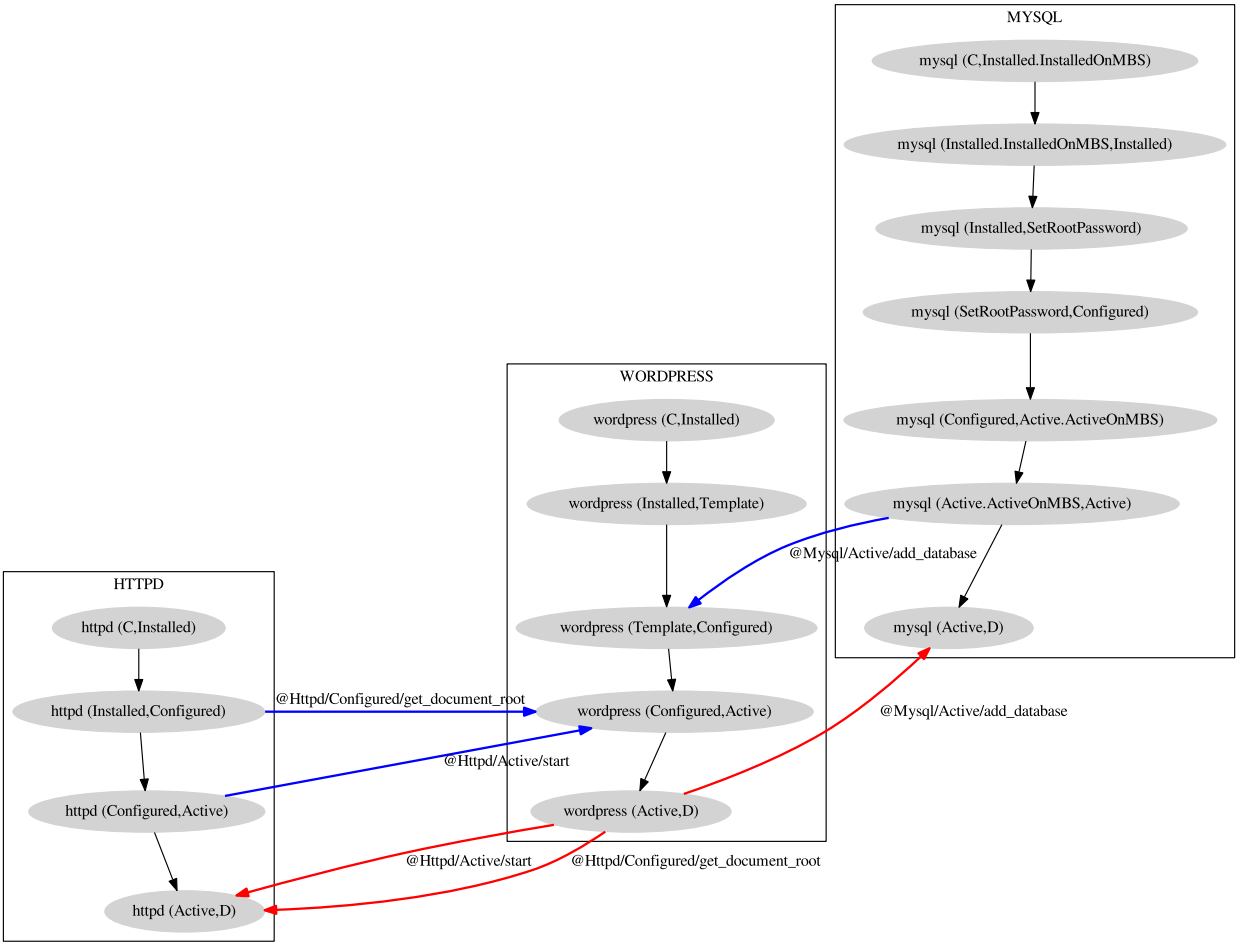


Fig. 15. Abstract plan generated by METIS.

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 significantly 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. To achieve this, it may be required to enrich the Aeolus Model with connectors such as those introduced by the Reo coordination language [47]. In this way it will be possible to model the exchange of configuration data between the components and deal with the connectors dynamic reconfiguration [48,49]. Finally, we would like to take into account also conflicts among components. In [5] we have proved that checking the existence of a deployment plan in which conflicts never occur is decidable but not tractable (more precisely, we prove that it is Ackermann-hard). In the light of this negative result, we plan to consider relaxed versions of the deployment problem in which either conflicts can be avoided by installing conflicting components on different (virtual) machines, or conflicts can be tolerated when they occur within specific phases of the deployment plan.

References

[1] Juju, DevOps distilled, https://juju.ubuntu.com/.
[2] J. Fischer, R. Majumdar, S. Esmaeilsabzali, Engage: a deployment management system, in: PLDI, ACM, 2012, pp. 263–274.
[3] Circular build dependencies, http://wiki.debian.org/CircularBuildDependencies.
[4] B. Schwartz, P. Zaitsev, V. Tkachenko, J.D. Zawodny, A. Lentz, D.J. Balling, High Performance MySQL, 2nd edition, O'Reilly, 2008.
[5] R.D. Cosmo, J. Mauro, S. Zacchiroli, G. Zavattaro, Aeolus: a component model for the cloud, Inf. Comput. 239 (2014) 100–121.
[6] T.A. Lascu, J. Mauro, G. Zavattaro, Automatic component deployment in the presence of circular dependencies, in: FACS, in: LNCS, vol. 8348, Springer, 2013, pp. 254–272.
[7] T.A. Lascu, J. Mauro, G. Zavattaro, A planning tool supporting the deployment of cloud applications, in: ICTAI, IEEE, 2013, pp. 213–220.
[8] Google App Engine, https://developers.google.com/appengine/.
[9] Microsoft Azure, http://azure.microsoft.com.
[10] DevOps, http://devops.com/.
[11] OASIS, Topology and orchestration specification for cloud applications (TOSCA) version 1.0, http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html.

[12] OASIS, Organization for the advancement of structured information standards (OASIS), https://www.oasis-open.org.
[13] J.A. Hewson, P. Anderson, Modelling system administration problems with CSPs, in: Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation, Mod-Ref11, 2011, pp. 73–82.
[14] J.A. Hewson, P. Anderson, A.D. Gordon, A declarative approach to automated configuration, in: LISA, USENIX Association, 2012, pp. 51–66.
[15] X. Etchevers, T. Coupaye, F. Boyer, N.D. Palma, Self-configuration of distributed applications in the cloud, in: IEEE CLOUD, IEEE, 2011, pp. 668–675.
[16] G. Salaün, X. Etchevers, N.D. Palma, F. Boyer, T. Coupaye, Verification of a self-configuration protocol for distributed applications in the cloud, in: Assurances for Self-Adaptive Systems, in: LNCS, vol. 7740, Springer, 2013, pp. 60–79.
[17] DMTF (Distributed Management Task Force), Open virtualization format specification version 2.0.1, http://dmtf.org/sites/default/files/standards/documents/DSP0243_2.0.1.pdf.
[18] Y.D. Liu, S.F. Smith, A formal framework for component deployment, in: OOPSLA, ACM, 2006, pp. 325–344.
[19] N. Arshad, D. Heimbigner, A.L. Wolf, Deployment and dynamic reconfiguration planning for distributed software systems, Softw. Qual. J. 15 (3) (2007) 265–281.
[20] M. Fox, D. Long, PDDL2.1: an extension to PDDL for expressing temporal planning domains, J. Artif. Intell. Res. 20 (2003) 61–124.
[21] A. Gerevini, I. Serina, LPG: a planner based on local search for planning graphs with action costs, in: AIPS, vol. 2, 2002, pp. 281–290.
[22] P. Goldsack, J. Guijarro, S. Loughran, A.N. Coles, A. Farrell, A. Lain, P. Murray, P. Toft, The SmartFrog configuration management framework, Oper. Syst. Rev. 43 (1) (2009) 16–25.
[23] J. Mirkovic, T. Faber, P. Hsieh, G. Malaiyandisamy, R. Malaviya, DADL: distributed application description language, USC/ISI Technical Report.
[24] L. Kanies, Puppet: next-generation configuration management, ;login: USENIX Mag. 31 (1) (2006) 19–25.
[25] Puppetlabs, Puppet, http://puppetlabs.com/.
[26] M. Burgess, A site configuration engine, Comput. Syst. 8 (2) (1995) 309–337.
[27] CFEngine, http://cfengine.com/.
[28] VMWare, Cloud Foundry, http://www.cloudfoundry.com.
[29] Aeolus by RedHat, http://www.aeolusproject.org/.
[30] R. Di Cosmo, S. Zacchiroli, G. Zavattaro, Towards a formal component model for the cloud, in: SEFM, in: LNCS, vol. 7504, Springer-Verlag, 2012, pp. 156–171.
[31] T.A. Lascu, Automatic deployment of applications in the cloud, Ph.D. thesis, University of Bologna, 2014.
[32] R.D. Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, A. Agahi, Automated synthesis and deployment of cloud applications, in: ASE, ACM, 2014, pp. 211–222.
[33] E. Koutsofios, S. North, et al., Drawing graphs with dot, Tech. rep., AT&T Bell Laboratories, Murray Hill, NJ, 1991.
[34] Graphviz—graph visualization software, http://www.graphviz.org/.
[35] Mandriva SA, www.mandriva.com.
[36] C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge University Press, 2003.
[37] Metric-FF, http://fai.cs.uni-saarland.de/hoffmann/metric-ff.html.
[38] J. Hoffmann, The metric-FF planning system: translating “Ignoring delete lists” to numeric state variables, J. Artif. Intell. Res. 20 (2003) 291–341.
[39] Madagascar-p, http://users.ics.aalto.fi/rintanen/jussi/satplan.html.
[40] A. Blum, M.L. Furst, Fast planning through planning graph analysis, Artif. Intell. 90 (1–2) (1997) 281–300.
[41] H.A. Kautz, B. Selman, et al., Planning as satisfiability, in: ECAI, vol. 92, 1992, pp. 359–363.
[42] R. Di Cosmo, A. Eiche, J. Mauro, G. Zavattaro, S. Zacchiroli, J. Zwolakowski, Automatic deployment of software components in the cloud with the Aeolus blender, Technical report, Inria Sophia Antipolis, 2015, https://hal.inria.fr/hal-01103806.
[43] Mandriva, Armonic, http://armonic.readthedocs.org/en/latest/index.html.
[44] Mandriva, Armonic, lifecycle anatomy, http://armonic.readthedocs.org/en/latest/lifecycle.html.
[45] OpenStack, http://www.openstack.org.
[46] Aeolus project, Aeolus blender, http://blog.aeolus-project.org/aeolus-blender/.
[47] F. Arbab, Reo: a channel-based coordination model for component composition, Math. Struct. Comput. Sci. 14 (3) (2004) 329–366.
[48] C. Krause, Reconfigurable component connectors, Ph.D. thesis, Universiteit Leiden, 2011.
[49] N. Oliveira, L.S. Barbosa, On the reconfiguration of software connectors, in: SAC, ACM, 2013, pp. 1885–1892.