A Planning Tool Supporting the Deployment of Cloud Applications

Tudor A. Lascu Jacopo Mauro Gianluigi Zavattaro
FOCUS team, University of Bologna and INRIA,
Dep. of Computer Science, Mura A. Zamboni 7, 40127 Bologna, ITALY
Email: {lascu,jmauro,zavattar}@cs.unibo.it

Abstract

Cloud computing offers the possibility to build sophisticated software systems on virtualized infrastructures at a fraction of the cost necessary just a few years ago. Nevertheless, the deployment of such complex systems is a serious issue due to the large number of involved software packages and services, and to their elaborated interdependencies. In this paper we address the challenge of automatizing this complex deployment process. We first formalize it as a planning problem and observe that standard planning tools can effectively solve it only on small and trivial instances. For this reason, we propose an ad hoc planning technique which we validate by means of a prototype implementation able to effectively solve this deployment problem also on instances of realistic size.

I. Introduction

Current technology makes it possible to run distributed software systems on-demand on a virtualized infrastructure, at a fraction of the cost which was necessary just a few years ago, thus fulfilling one of the many promises of what is generally referred to as "*cloud computing*". But realizing, maintaining and reconfiguring such software systems is a serious challenge. When the number of components grows it becomes essential to be able to specify at a certain level of abstraction a particular configuration of the distributed software system, and to rely on tools that provide a set of possible deployment and configuration actions leading to a system configuration that satisfies some user requests. Recent works have introduced formalisms which focus on this *automation* aspect of the deployment process, like the Juju initiative within Ubuntu [19], the Engage system [16], and the Aeolus model [14]. All of them have the goal to provide a planner to enable the automation of finding possible deployment paths. In particular, the Aeolus component model has been designed in order to investigate the computational boundaries of this specific kind of planning problem. Namely, components describe resources which provide and require different functionalities by means of ports, and that may be in conflict with each other. These components are equipped with *state machines* that describe *declaratively* how required and provided functionalities are enacted. Automatizing a deployment consists of planning a sequence of low-level actions like creation/deletion of components, port binding/unbinding, and internal state changes, that reaches a configuration with at least one component in a specific target internal state. In the full Aeolus model it is also possible to specify *capacity constraints*, i.e. for each provided functionality how many requirements it can satisfy, and for each requirement how many different instances of the provided functionality are needed to satisfy it. In [14] we have proved that the deployment problem is undecidable for the full Aeolus model. On the contrary, if capacity constraints are not considered, we have proved in [13] that the problem turns out to be decidable, but it is Exp-Space hard.

In this paper, we focus on the feasibility of developing effective tools able to tackle this problem, and for this reason we have decided to further simplify the Aeolus model by removing also conflicts. Automatic deployment tools like Juju and Engage abstract away from conflicts too. The absence of conflicts is useful, for instance in Engage, to complete partial configurations simply by adding new components without having to check whether these are incompatible with already present components.

In Engage, once the configuration has been completed, a low-level deployment plan is synthesized by topologically sorting a graph-based representation of the components in the configuration, where arcs represent dependencies between components. This approach works only if there are no circular dependencies among components. Unfortunately, this form of circularities frequently occur both at the level of fine-grained software packages and of coarse-grained services. For instance, as far as packages are concerned, circular dependencies have to be dealt with for bootstrapping a Linux distribution (see for instance [11] for a list of circular build dependencies between packages in Debian). Concerning services, another example of circular dependency can be found in the master-master replication pattern of *mysql* [9]. In this case, for reliability purposes, two *mysql* servers act at the same time both as master and slave of each other. In the Aeolus model, circular dependencies among components can be naturally represented (see, for instance, the example depicted in Fig. 2).

In this paper, we present a novel planning technique for the Aeolus model (without conflicts) able to synthesize a deployment plan, when it exists, or to terminate by indicating failure, otherwise. This new technique is based on three distinct phases. In a first phase the existence of a plan is checked by performing a forward symbolic *reachability analysis* of all possible reachable states of the components. If the target state is reachable, a second phase of *abstract planning* generates

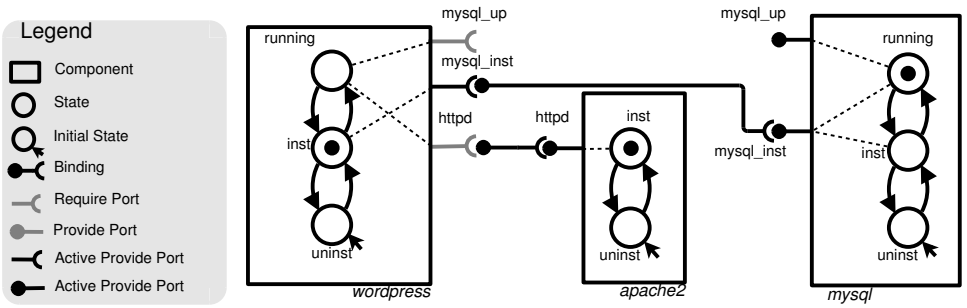

Fig. 1: Wordpress installation in the Aeolus model.

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 functionality before another component enters a state requiring such functionality. 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 the technique used to deal with those cases in which more instances of the same kind of component must be contemporaneously deployed, in different states, in order to enact different functionalities at the same time.

The validation of our planning technique has been achieved by means of a prototype implementation. We compared the prototype on some use cases against some standard general purpose planners providing an encoding of the problem in the Planning Domain Definition Language (PDDL) [17]. Results indicate that general purpose planners can not be used in practice since they do not efficiently handle scenarios involving just 4 or 5 components. Our prototype, instead, produces plans involving hundreds of components in less than one minute.

Paper structure. In Section II we recall the Aeolus component model and we report the corresponding formalization of the component deployment problem as a specific instance of planning. In Section III we present our algorithm for solving this planning problem while in Section IV we discuss its implementation and validation against standard planning tools. Finally, in Section V we discuss the related literature while in Section VI we report some concluding remarks.

II. Deployment as a Planning Problem

In this section we introduce the Aeolus model, considered in this paper to specify how packages/services can be deployed in a cloud application.

Indeed, the Aeolus model [14] is designed to uniformly represent both fine-grained software components, like packages to be installed on one single virtual machine, and coarse-grained services possibly obtained as composition of distributed and properly connected sub-services. A service, a package or a software component can be modeled as grey-box exposing (only) relevant behaviour by means of internal states and actions for changing them during deployment. For instance Fig. 1 depicts three components representing, respectively, *wordpress*, *apache2*, and *mysql* services : *wordpress* and *apache2* are installed while *mysql* is in running mode.

Each state may activate provide and require ports: active require ports represent the functionalities needed by the service to be in a given state and must be bound to active provide ports of other components. Ports are labeled with *interfaces* indicating the corresponding required/provided functionality. For instance *wordpress* to be installed requires at least an installation of *mysql*. This is captured by the fact that the *inst* state of *wordpress* requires the port with interface *mysql_inst* that is provided by *mysql* both in *running* and *installed* mode.

Component types are modeled as finite state automata indicating the possible internal states and the corresponding state transitions. Different states can activate different provide and require ports. For instance, *mysql* can go from *running* to *inst* state. Doing this it will no longer provide the port with interface *mysql_up* but just the port with *mysql_inst* interface. On the other hand *wordpress* can not switch its state to *run* because in order to do so it requires the port *mysql_up* that is not yet bound to an active provide port.

According to the Aeolus model, the deployment problem requires as input a *universe* of possible component types and a *target state*, i.e. an internal state of one specific component type that must be active in the final configuration. The output of the problem is a *sequence of low-level deployment actions* on components of the type in the input universe that, upon execution, produces a configuration containing at least one component in the target state. Moreover, all the intermediary configurations, as well as the finally reached one, must be correct in the sense that all active require ports should be bound to complementary active provide ports of other components.

The possible low-level actions are as follows:

- $create(C, id)$ that creates a new component of type C in an initial state. The new component is identified by a unique and fresh identifier id;
- $delete(id)$ that deletes the component identified by the identifier id;
- $bind(r, id_1, id_2)$ that creates a binding between the provided port r of the component identified by id_1 and the required port of the component identified by id_2;

- *unbind*(r, id_1, id_2) that deletes the binding between the provided port r of the component identified by id_1 and the required port of the component identified by id_2;
- *stateChange*(id, s_0, s_1) that changes the state of the component identified by id from s_0 to s_1.

It is worth noticing that there can be more than one way to reach a given configuration of components. For instance, one possible way to obtain the configuration depicted in Fig. 1 from scratch, is to first create the resources via the actions *create*$(wordpress, w)$, *create*$(apache2, a)$, and *create*$(mysql, m)$. These three actions create three new components identified by w, a, and m respectively. All these new components will be in the *uninst* state that is the initial state for all of them. Then the *apache2* and *mysql* components can be installed by performing the action *stateChange*$(a, uninst, inst)$ and *stateChange*$(m, uninst, inst)$. At this point, to be able to install *wordpress*, we need first to bind the *mysql_inst* port. This is done by performing *bind*$(mysql\_inst, m, w)$. After the creation of the binding, *wordpress* can be installed by performing *stateChange*$(w, uninst, inst)$. Finally the configuration depicted in Fig. 1 can be obtained by performing the *bind*$(httpd, a, w)$ and *stateChange*$(m, inst, run)$ actions.

Note that the *unbind*, *delete*, and *stateChange* actions sometimes cannot be performed since their execution would violate the constraint that each active require port must be bound to an active provide one. *bind* and *create* actions, instead, can always be performed as bindings are allowed between ports that are not active and we require that initial states do not activate require ports.

As a final remark, we observe that the decision to use one unique internal *target state* to specify the configuration to be reached is not a limitation. In fact, this target state could activate several require ports indicating an entire set of functionalities that must be present in the final configuration.

III. The Planning Algorithm

We now present our algorithm to solve the deployment problem defined in previous section. The algorithm is divided in three phases, namely, *reachability analysis*, *abstract planning* and *plan generation*.

The first phase computes the states of the components that can be obtained, starting from an empty configuration. If the target state can be reached, an abstract plan is generated describing the needed types of components and a path to reach the target state. Subsequently a concrete plan is obtained by specifically instantiating the component types selected in the abstract plan.

As a running example we model the compilation of package kerberos with ldap support in a Debian system. To build kerberos (*krb5*) the *libldap2-dev* package of *openldap* is needed. This package however depends on *libkrb5-dev* from *krb5*. There is therefore a circular dependency between *krb5* and *openldap*. In Debian the generic way to deal with these circular dependencies is profile builds: every package caters for multiple stages of staged/bootstrap build, so that if necessary a package can have *stage1*, *stage2*, ... before the final, *normal*, build. In the kerberos case, *krb5* is built in the first stage missing out the generation of the *krb5-ldap* package. Then *openldap* can be built directly into its *normal* build satisfying its dependencies. Once *openldap* is built, *krb5* can also be build into its *normal* stage. This process would be modeled in Aeolus as depicted in Fig. 2.

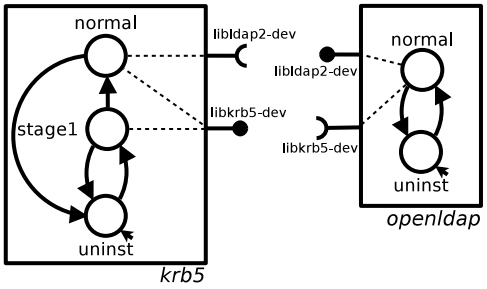

Fig. 2: Representation of the *krb5* and *openldap* components.

A. Reachability analysis

The first step in the proposed technique checks if the the desired target state can be reached. To do so all reachable states are computed, for each of the component types in the given universe. In the following we use the pair $\langle \mathscr{T}, q \rangle$ to denote a component type $\mathscr{T}$ and one of its state q.

An increasing sequence of sets of component-state pairs $S_0, \ldots, S_n$ is built in such a way that S_{i+1} extends S_i with the new states that can be reached upon execution of a *stateChange* action. The first set, S_0, contains all the components in their initial state, i.e. $S_0 = \{\langle \mathscr{T}, q_0 \rangle \mid q_0 \text{ initial state of } \mathscr{T}\}$. Formally S_{i+1} is the largest set satisfying the following constraints:

- $S_i \subseteq S_{i+1}$;
- $\langle \mathscr{T}, q \rangle \in S_{i+1}$ implies the existence of $\langle \mathscr{T}, q' \rangle \in S_i$ such that there is a transition from q' to q in the state automaton of $\mathscr{T}$;
- $\langle \mathscr{T}, q \rangle \in S_{i+1}$ implies that for every require port r activated by the state q of $\mathscr{T}$ there exists $\langle \mathscr{T}', q' \rangle \in S_i$ such that the state q' of $\mathscr{T}'$ activates a provide port r.

The generation of sets proceeds until a fix-point is reached (i.e. $S_{i+1} = S_i$). When the fix-point is reached, if the last set does not contain the target pair it means a plan to achieve the goal does not exist and therefore the procedure terminates. Otherwise, we continue with the next phase.

As input to the next phase, we consider a graph-like representation, called *reachability graph*, of the sets $S_0, \ldots, S_n$ that keeps track of all the possible ways to obtain the component state-pairs at level $i+1$ from those at level i. More precisely, the graph has as nodes the pairs in $S_0, \ldots, S_n$: if one node at level $i+1$ was already present at level i, the two nodes are connected with an arc $\cdots\cdots$, if a state pair $\langle \mathscr{T}, q \rangle$ at level $i+1$ can be obtained from $\langle \mathscr{T}, q' \rangle$ at level i by means of a *stateChange* action, an $\longrightarrow$ arc from the former to the latter is added. Visually the reachability graph can therefore be seen as

a pyramid of levels of component-states having arrows $\longrightarrow$ or arcs ····· between two consecutive levels as the one in Fig. 3.

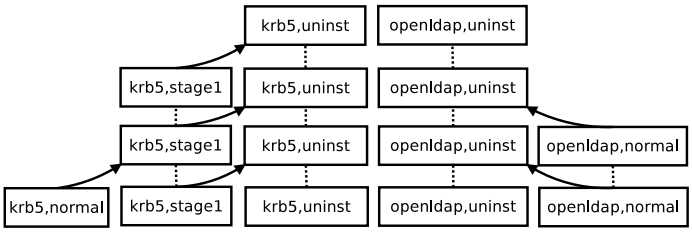

Fig. 3: Reachability graph for the kerberos running example.

The first level of Fig. 3 contains the two components *krb5* and *openldap* in their initial states. In the second level the component *krb5* in *stage1* state is added since it can be derived from the *krb5* component in state *uninst*. The component *openldap* in *normal* state can not be added at this level since it requires the interface *libkrb5-dev*, not yet provided. *openldap* in *normal* state is added however in the third level since *libkrb5-dev* is now provided by *krb5* in state *stage1*. Finally, in the fourth level, the target state is added deriving it from *krb5* in state *stage1*. This last level is also the fix-point since no new component-state pairs can be generated from it.

Note that keeping component copies allows one to consider different ways a component can use to reach a state. This adds flexibility in deciding how a target can be reached.

B. Abstract Planning

After generating the reachability graph we compute an *abstract plan*.

We first describe the structure of an abstract plan and then explain how this can be derived from the reachability graph. An abstract plan is a directed graph where the nodes represent either a *create*, *delete*, or *stateChange* action, and arcs represent action precedence constraints. In the following we denote with $\langle z,q,q'\rangle$ a *stateChange* from q to q' of instance z, with $\langle z,\varepsilon,q_0\rangle$ the *create* action of the instance z in the initial state q_0, and with $\langle z,q,\varepsilon\rangle$ the *delete* action on the instance z in state q. We consider three types of precedence arcs:

- $\longrightarrow$: states the precedence of *stateChange* actions on the same component instance; formally $\langle z,x,x'\rangle \longrightarrow \langle z,x',x''\rangle$ where x' is a state and x,x'' are either states or the special symbol ε denoting absence of the instance z;
- $\overset{r}{\twoheadrightarrow}$ where r is an interface: states that if an action deploys an instance z' in a state y' requiring r, provided by z in state y, then state y must be entered before entering state y', formally $\langle z,x,y\rangle \overset{r}{\twoheadrightarrow} \langle z',x',y'\rangle$;
- $\overset{r}{\dashrightarrow}$, where r is an interface, is the dual of the previous arrow: it states that if an action deploys an instance z' in a state y' requiring r, provided by z in state y, then state y' must be exited before exiting state y, formally $\langle z',y',u'\rangle \overset{r}{\dashrightarrow} \langle z,y,u\rangle$.

We are now ready to describe how an abstract plan is obtained. Starting from the reachability graph we select the target component-state pair at the bottom of the pyramid. From the bottom level we then proceed upward selecting the components that are used to deploy the selected component-state pairs at the lower level. To do so, for every selected component at level $i+1$, we select at level i one of its predecessors (i.e. a component-state pair connected via the $\longrightarrow$ arrow) or a copy (i.e. a component-state pair connected via the ····· arc). Moreover, for every require port activated by the selected component-state pairs of level $i+1$ that are not copies, we select a component-state pair at level i that is able to satisfy the requirement, and we keep track of this choice.

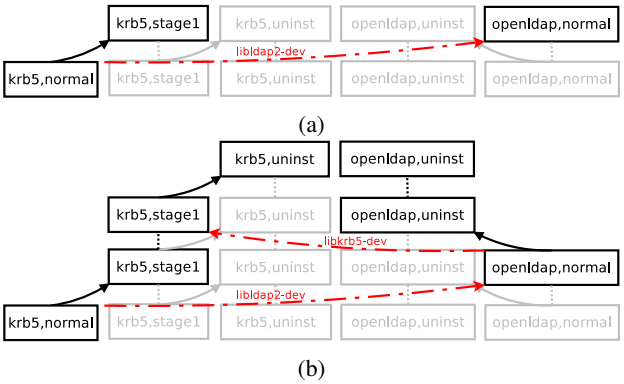

Fig. 4: Generation of abstract plan for the kerberos example.

For the kerberos case, Fig. 4a shows that in the last level *krb5* in *normal* state is selected. Since *krb5* can be only obtained via *krb5* in state *stage1* we select *krb5* in state *stage1* in the previous level. Moreover since *krb5* in state *normal* requires *libldap2-dev* we select at level 2 also the component *openldap* in state *normal*. Iterating this selection process we may end up in the scenario depicted in Fig. 4b.

We would like to underline that during the selection of component-state pairs different choices could be made. For instance in Fig. 4b at the second level we could have selected component *krb5* in state *uninst* to deploy the same component in state *stage1* and component *krb5* in state *stage1* to provide the *libkrb5-dev* interface. These choices have an impact on the number of instances employed to reach the goal. In order to minimize this number we rely on heuristics.[1] In particular, for the selection of component-state pairs, we choose the one that is able to satisfy the maximum number of (not already satisfied) requirements. In case of ties we select the component that can be obtained from an initial configuration satisfying less requirements. In case of ties we prefer a copy and, if the component is instead newly obtained, we select the one that can be obtained with less state changes. Similarly, when component-state pairs are selected to satisfy some requirements, we select first the one able to satisfy the maximum number of requirements, in case of ties the one that can be obtained with less interfaces and, in case of a tie, the

[1]. Heuristics are used to reduce the complexity of finding the best choice. Indeed, exploring all the possibilities to compute a (global) minimum can be done just at an exponential cost

one that can be obtained with less state changes.

Once all the component-state pairs have been selected, we consider a component instance for every maximal path that starts from a component-state in the top level and reaches a component-state that is not a copy. For instance in the kerberos case there are two maximal paths, one starting from the component *krb5* in state *uninst* and reaching the state *normal*, and one starting from the component *openldap* in state *uninst* and reaching the state *normal*. We identify the corresponding instances with z and w respectively.

For every instance we add to the abstract plan its *create*, *delete* and *stateChange* actions. Arrows $\longrightarrow$ are added to connect these actions in chronological order (i.e. first the instance creation, the state changes and then the deletion action). The arrows $\overset{r}{\twoheadrightarrow}$ and $\overset{r}{\dashrightarrow}$ are instead added between actions of instances requiring and providing an interface r.

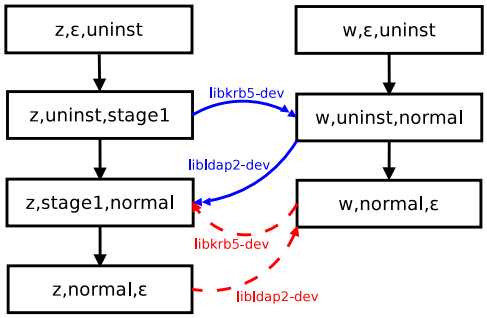

Fig. 5: Abstract plan for the kerberos running example.

Fig. 5 shows the abstract plan obtained for the kerberos case. The four actions on the left are related to instance z while the three on the right are actions related to instance w. z is first created, then it changes its state first into *stage1* and then to *normal* before being deleted. w instead is created, it changes state into *normal*, before being deleted. These precedences are encoded by $\longrightarrow$ arrows. z's requirement of *libldap2-dev* in state *normal*, satisfied by w in *normal* state, is encoded with $\overset{libldap2-dev}{\twoheadrightarrow}$ between $\langle w, uninst, normal\rangle$ and $\langle z, stage1, normal\rangle$ and $\overset{libldap2-dev}{\dashrightarrow}$ between $\langle z, normal, \varepsilon\rangle$ and $\langle w, normal, \varepsilon\rangle$. The fist one states that w must be in *normal* state before z moves to *normal* while the second states that the deletion of z must precede the deletion of w. Indeed, if one of these constraints does not hold it means that the abstract plan violates a requirement thus leading to a non correct configuration. Similarly, the *libkrb5-dev* interface requirement of w in state *normal*, satisfied by z in *state1*, is encoded with $\overset{libkrb5-dev}{\twoheadrightarrow}$ between $\langle z, uninst, stage1\rangle$ and $\langle w, uninst, normal\rangle$ and $\overset{libkrb5-dev}{\dashrightarrow}$ between $\langle w, normal, \varepsilon\rangle$ and $\langle z, stage1, normal\rangle$. In this case we can however notice that z continues to provide the port *libkrb5-dev* also when it is in state *normal*. Thus w does not need to be deleted before *krb5* moves to *normal* but it can stay until the *krb5* is not deleted. This relaxation corresponds to setting $\langle z, normal, \varepsilon\rangle$ as the target of $\overset{libkrb5-dev}{\dashrightarrow}$. In general all the constraints $\langle z, x, y\rangle \overset{r}{\dashrightarrow} \langle z, x', y'\rangle$ can be relaxed replacing $\langle z, x', y'\rangle$ with $\langle z, x'', y''\rangle$ where $\langle z, x'', y''\rangle$ is a *delete* action or it is the the first *stateChange* reaching a state y'' that does not provide r. After applying these relaxations we obtain the final version of abstract plan that, for the kerberos case, is the one depicted in Fig. 6.

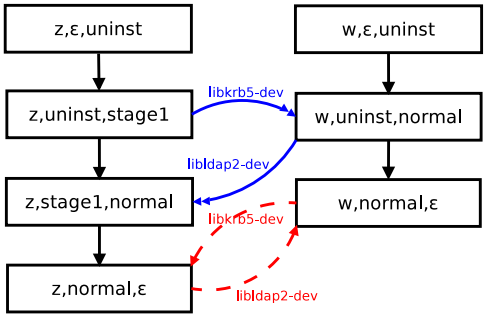

Fig. 6: Abstract plan for the kerberos example after relaxation.

C. Plan generation

The abstract plan is used to synthesize a concrete one. The idea is to visit the nodes of the abstract plan in topological order until the target component is obtained. Visiting a node consists of performing that action. Moreover, in order to properly satisfy component requirements, when an incoming $\overset{r}{\twoheadrightarrow}$ is encountered a new binding should be created, and when an outgoing $\overset{r}{\dashrightarrow}$ is encountered the corresponding binding should be deleted. Notice that it is not necessary to visit the entire abstract plan as it is sufficient to reach the target state. For this reason, we give priority to the visit of the actions of the components containing such state.

For instance, in the kerberos example, we can extract a concrete plan from the abstract plan in Fig. 6 as follows. Assume that the target state is state *normal* of component type *krb5*. As we give priority to the corresponding instance, the first action in the concrete plan is *create(krb5,z)* corresponding to the visit of $\langle z, \varepsilon, uninst\rangle$. The subsequent action is *stateChange(z,uninst,stage1)* corresponding to $\langle z, uninst, stage1\rangle$. The visit of the actions on the instance z cannot proceed due to the incoming arrow $\overset{libldap2-dev}{\twoheadrightarrow}$; for this reason the next action in the concrete plan is *create(openldap,w)* corresponding to the visit of $\langle w, \varepsilon, uninst\rangle$. The next node in the abstract plan to be visited is $\langle w, uninst, normal\rangle$, but as this node has an incoming $\overset{libkrb5-dev}{\twoheadrightarrow}$, two actions must be added to the concrete plan: *bind(libkrb5-dev,z,w)* and *stateChange(w,uninst,normal)*. At this point, the visit of the component instance z can continue by considering node $\langle z, stage1, normal\rangle$; as this node has an incoming $\overset{libldap2-dev}{\twoheadrightarrow}$, two actions must be added to the concrete plan: *bind(libldap2-dev,w,z)* and *stateChange(w,uninst,normal)*. This completes the generation of the concrete plan as the target state has been reached.

Unfortunately, the topological visit is not always possible as it may be inhibited by the presence of cycles in the abstract plan. Consider, for instance, a slightly modified version of the

kerberos example in which the component type *krb5* in *normal* state requires not only one *openldap* in *normal* state, but also one in *uninst* state. In this case the abstract plan will be as in Fig. 7 (note the addition of the pair of arcs labeled with *uninst*).

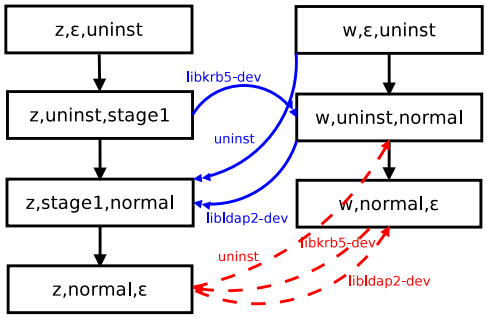

Fig. 7: Abstract plan for the modified kerberos example.

In the abstract plan in Fig. 7 there is a cycle among $\langle w, uninst, normal\rangle$, $\langle w, normal, \varepsilon\rangle$ and $\langle z, normal, \varepsilon\rangle$ that forbids the visit of $\langle w, uninst, normal\rangle$, which is a necessary step to visit the target node $\langle z, stage1, normal\rangle$.

In general, these cycles appear when an instance is expected to provide an interface during a specific phase of the plan, but this is not possible because in the same phase of the plan the instance is required to change its internal state. This problem can be solved by means of *instance duplication*: an additional component instance is deployed in such a way that the new instance can continue providing the required interface during that specific phase of the plan. The application of instance duplication to the abstract plan in Fig. 7 is reported in Fig. 8, where we add a new instance y of type *openldap* that does not proceed further than state *uninst*. This new resource is used to satisfy the requirement *uninst* of z. Notice that the topological visit until the target node $\langle z, stage1, normal\rangle$ becomes now possible.

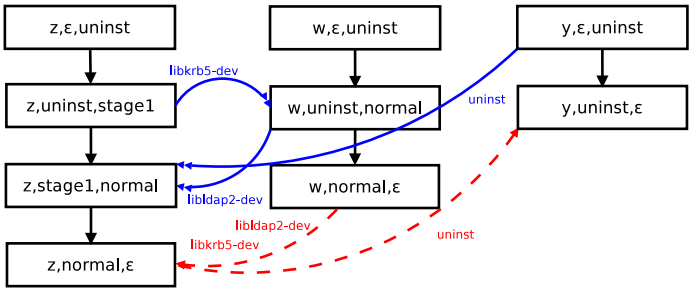

Fig. 8: Abstract plan in Fig. 7 after duplication of instance w.

In general, we have to consider an *adaptive* topological visit, where by adaptive we mean that the abstract plan is transformed when the visit is blocked by a cycle. When a node cannot be visited, we proceed as follows. There is at least an $\overset{r}{\dashrightarrow}$ or a $\overset{r}{\twoheadrightarrow}$ arc, incoming to this node. We consider two cases: when there is no incoming $\overset{r}{\twoheadrightarrow}$ arc, and otherwise. In the first case, we proceed by removing the incoming $\overset{r}{\dashrightarrow}$ by duplicating the instance containing the node that could not be visited. The duplication of the instance is done from its initial node to such node. Concerning the arcs, all the incoming $\overset{r}{\twoheadrightarrow}$ and outgoing $\overset{r}{\dashrightarrow}$ arcs of the instance (representing its requirements on the other components) will have to be duplicated and, more important, the incoming $\overset{r}{\dashrightarrow}$ to be removed are moved to the new instance. This is obtained by moving the pairs of outgoing $\overset{r}{\twoheadrightarrow}$ / incoming $\overset{r}{\dashrightarrow}$ representing the interface that the new instance must provide to the other components. See, for instance, the pair of arcs labeled with *uninst* that are moved from their initial position in Fig. 7 to the new instance y in Fig. 8. In this way, the considered node does not any longer have incoming arcs $\overset{r}{\dashrightarrow}$ and it can be visited. Consider now the case in which there are incoming arcs $\overset{r}{\twoheadrightarrow}$. We consider the node sources of these arcs. If those nodes cannot be visited, we apply to them the instance duplication procedure. Upon this duplication phase, the initial node could be visited or will have impediments due to incoming arcs $\overset{r}{\dashrightarrow}$. In this case the problem can be solved as described in the previous case.

It is interesting to note that this adaptive visit will eventually terminate because new instances do not introduce new cycles, and because there exists no cycle involving only $\overset{r}{\twoheadrightarrow}$ arcs. This is guaranteed by the fact that this class of arcs represents the dependencies selected during the transformation from the reachability graph to the abstract plan; by construction, these dependencies cannot be circular because they always go from nodes at a lower to nodes at a higher level in the pyramid.

IV. Validation

In the context of knowledge representation and reasoning, a very important application of artificial intelligence, is that of developing languages and tools for reasoning about actions and change and, more specifically, for the problem of planning [8]. Since 1998, a declarative language for planning has been defined for establishing a common syntax for different tools in order to allow different research groups to test their solvers. This language is known as PDDL [17].

Our tool solves a planning problem and therefore we tried to validate our ad-hoc planner against standard planners. To do so we have defined an encoding of our specific planning problem into PDDL: 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 the other objects in the configuration provide the required interfaces. The encoding abstracts from the bind and unbind actions[2] and limits the number of objects that could be concurrently used.[3]

As a benchmark we have considered Aeolus instances automatically generated following the pattern of component interdependency discussed in Section III with the kerberos

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

[3]This limitation was necessary because all the solvers assume a finite number of objects –without this limitation the planning problem is undecidable.

Size	Test $T+$		Test T	
	Madagascar-p	Metric-FF	Madagascar-p	Metric-FF
3	0.13 s	timeout	0.05 s	0.05 s
4	0.77 s		0.41 s	timeout
5	3.44 s		1.96 s	
6	error		7.22 s	
7			error	

TABLE I: Performances of standard planners.

running example. We have considered configurations composed by an increasing number of components having 3 states (excluding one component having only 2 states used to trigger the plan).

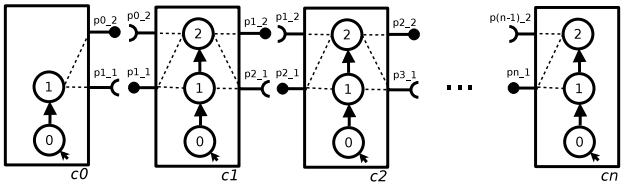

Fig. 9: Component types considered in our tests.

As depicted in Fig. 9, the components have provide and require ports in such a way that a valid plan to reach the state *s2* of the rightmost component would require first to create the components, then perform from *cn* to *c0* the state change from *s0* to *s1*, and then perform the state change from *s1* to *s2* from *c1* to *cn*. In these scenarios instance duplication is not needed during the generation of the concrete plan. In order to test also this feature we modified the component types by removing for nearly one fifth of the components the activation of the provide port *pX_1* from the states *s2*, where *X* is the number of the selected component. Removing this activation of the provide port requires the duplication of the instance of the component type *cX* in order to contemporaneously satisfy the requirements of its two neighbor components. The activation of the provide port to be deleted were selected randomly. In the following we will denote respectively with T the test of the scenario where duplication is not needed and with $T+$ the test where duplication is necessary.

The tests were performed using a dual core machine with 2.60 GHz Intel i5 processors, 8GB of RAM, and Ubuntu 13.04 operating system. We used a time cup of 120 seconds and two planners that support the ADL fragment of PDDL (other popular solvers support only fragments of PDDL): Metric-FF [5] and Madagascar-p [3]. The first solver is based on GraphPlan, a standard planning algorithm to prune the search space, while the second encodes the planning problem into a SAT formula and then uses state of the art SAT solvers to solve it. For reducing the search space of these solvers we set to the minimum the number of components that could be used concurrently.

We notice that the performances of the general purpose solvers are quite poor. Table I reports the time performances for both the T and $T+$ scenarios. With *error* we mean that the solver exited without computing the plan, with *timeout* that the solver took more than 120s and was thus interrupted.

Even though the best solver among the two is Madagascar-p, it is able to solve scenarios with just 6 components for the simpler T test. Metric-FF, in the $T+$ case, times out even when considering a scenario with 3 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. Metric-FF 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.

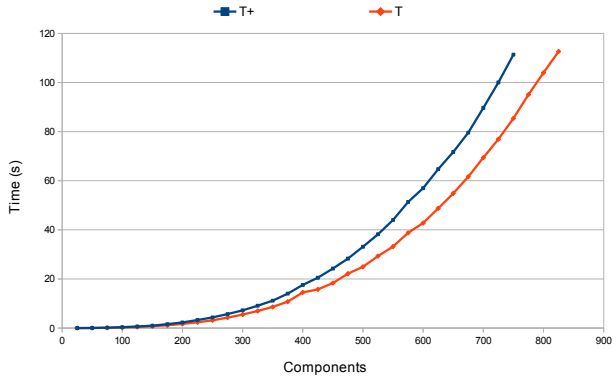

Fig. 10: Time performances using our tool

Fig. 10 shows instead the time taken by our tool to produce a valid plan for tests T and $T+$. Clearly our ad-hoc planner outperforms the general purpose planners. In both cases it is able to produce plans in less than two minutes for scenarios involving more than 700 components. As expected, duplication affects the performance of the planner. The degradation of performances, however, is still manageable: plans can be provided in less than a minute even for scenarios with hundreds of components that need replication.

The tool was developed in OCaml and is about 3.5 K lines of code. The source code of our tool and the problem encoding into the PDDL language are available at [2].

V. Related work

The problem of managing networks of interconnected machines has promoted the realization of many popular tools like CFEngine [10], Puppet [20], MCollective [22] and Chef [21]. Despite their differences, they all allow to declare the components to be installed on each machine, together with their configuration files, and then employ various mechanisms to deploy components accordingly. However, the problem of specifying which components to deploy where, and how to interconnect them is left to the user. Another system specifically targeted at application deployment in the cloud is CloudFoundry [23], but it has the same limitations described above.

The problem of automatizing the selection/distribution/interconnection of the components has been addressed by other tools like ConfSolve [18] and Zephyrus [12]. ConfSolve [18] relies on a constraint solver to propose an optimal allocation

of virtual machines on servers and applications on virtual machines, but it does not handle the problem of connections among services. Zephyrus [12] automates assembling complex distributed systems, allowing a user to compute a valid configuration satisfying a high level specification. The description of the model is given in Aeolus, and the tool takes into account conflicts and capacity constraints, but it does not consider components as automata since the only concern is to provide a valid final configuration and not the steps to reach it. Juju [19] and Engage [16], are more similar to our approach: they both rely on a solver to perform the deployment part and they avoid the problem of dealing with conflicts among components. In our approach however we are able to treat circular dependencies that cannot be defined in Engage and should instead be manually resolved using low level programming techniques in Juju. Another approach to automating deployment is proposed in [15]; it uses an Architecture Description Language with information on the relationships among software services, which needs to be explicitly provided by the user in full detail, and uses a decentralized protocol to perform automatic configuration. Closely related to our work is [7] that proposes an heuristic-based algorithm to remove build dependency cycles for bootstrapping a Linux software distribution. The building order of the packages is generated using a topological sort of a graph. The problem addressed there is, however, slightly but significantly different as the aim is to build packages from scratch. At every iteration the newer version of the recompiled package replaces the old one. In our case this is not always true as the same service may be required in both installed and running state to support fault tolerance via replication.

VI. Conclusions

We presented a tool that, given a representation of software packages/services following the Aeolus model, computes the necessary steps to deploy a final configuration. Our solution handles circular dependencies among components and produces plans deploying hundreds of different components in less than a minute. Moreover, the deployment actions can be easily parallelized; once the concrete plan is generated, the knowledge of the precedence constraints among different actions allows one to identify the minimal number of synchronizations needed to perform the deploying actions in parallel.

In order to practically experiment and trial our tool in a real-life environment, with the collaboration of our industrial partner within the ANR project "Aeolus: Mastering Cloud Complexity" [1] –Mandriva SA [4]– we are currently developing a n-tiers application deployment engine. This engine deploys, on an IaaS, a final configuration found by the Zephyrus tool. First, the required virtual machines are provisioned by relying on OpenStack. We then proceed with the installation and configuration phase in which the engine connects to each virtual machine involved in the n-tiers application. The required packages are installed and services are configured following the steps specified by the plan produced by our tool. This is done using the deployment tool MSS (Mandriva Server Setup) [6]. It is worth noticing that these steps are performed without any human interaction, by combining the MSS configuration information, the final configuration computed by Zephyrus and the plan output by our tool. At present we are able to successfully deploy a configuration of several instances of Wordpress backed by a Varnish load balancer. We are currently working on integrating examples with complex database configurations containing master and slave requirements.

In future work, we aim to improve our tool by considering also reconfiguration plans, dealing with cases in which the initial configuration is not empty, but already deployed components could be used and rearranged to reach the desired new configuration. We would also like to take into account conflicts producing a plan that does not violate them or, if this is not feasible in practice, devise plans that minimize the time windows where a system is inconsistent.

References

[1] Aeolus: Mastering the Complexity of the Cloud. http://www.aeolus-project.org/.

[2] DACLAMP - Deployment Applications in the CLoud Aeolus Min Planner. https://github.com/talascu/daclamp.

[3] Madagascar-p. http://users.cecs.anu.edu.au/~jussi/satplan.html. Retrieved June 2013.

[4] Mandriva SA. www.mandriva.com.

[5] Metric-FF. http://fai.cs.uni-saarland.de/hoffmann/metric-ff.html. Retrieved June 2013.

[6] MSS - Mandriva Server Setup. http://doc.mandriva.com/en/mes5/Enterprise-Server-Manual-EN.html/CS-mss-intro.html. Retrieved June 2013.

[7] P. Abate and S. Johannes. Bootstrapping Software Distributions. In *CBSE*, 2013.

[8] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[9] S. Baron, Z. Peter, T. Vadim, Z. Jeremy, L. Arjen, and B. D. J. *High performance mysql, 2nd edition*. O'Reilly, second edition, 2008.

[10] M. Burgess. A Site Configuration Engine. *Computing Systems*, 8(2):309–337, 1995.

[11] Circular Build Dependencies. http://wiki.debian.org/CircularBuildDependencies. Retrieved June 2013.

[12] R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, and J. Zwolakowski. Optimal Provisioning in the Cloud. Technical report, Aeolus project, Juin 2013. http://hal.archives-ouvertes.fr/hal-00831455.

[13] R. Di Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro. Component Reconfiguration in the Presence of Conflicts. In *ICALP 2013*.

[14] R. Di Cosmo, S. Zacchiroli, and G. Zavattaro. Towards a Formal Component Model for the Cloud. In *SEFM 2012*.

[15] X. Etchevers, T. Coupaye, F. Boyer, and N. de Palma. Self-Configuration of Distributed Applications in the Cloud. In *CLOUD 2011*.

[16] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: A Deployment Management System. In *PLDI '12*.

[17] M. Fox and D. Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

[18] J. A. Hewson, P. Anderson, and A. D. Gordon. A Declarative Approach to Automated Configuration. In *LISA '12*.

[19] Juju, devops distilled. https://juju.ubuntu.com/. Retrieved June 2013.

[20] L. Kanies. Puppet: Next-generation configuration management. *;login: the USENIX magazine*, 31(1):19–25, 2006.

[21] Opscode. Chef. http://www.opscode.com/chef/. Retrieved June 2013.

[22] Puppet Labs. Marionette Collective. http://docs.puppetlabs.com/mcollective/. Retrieved June 2013.

[23] VMWare. Cloud Foundry, deploy & scale your applications in seconds. http://www.cloudfoundry.com/. Retrieved June 2013.