Cloud Ready Applications Composed via HTN Planning

Ilche Georgievski
Sustainable Buildings
Groningen, The Netherlands
ilche@sustainablebuildings.nl

Faris Nizamic
Sustainable Buildings
Groningen, The Netherlands
faris@sustainablebuildings.nl

Alexander Lazovik
Johann Bernoulli Institute
University of Groningen
Groningen, The Netherlands
a.lazovik@rug.nl

Marco Aiello
Johann Bernoulli Institute
University of Groningen
Groningen, The Netherlands
m.aiello@rug.nl

***Abstract*—Modern software applications are increasingly deployed and distributed on infrastructures in the Cloud, and then offered as a service. Before the deployment process happens, these applications are being manually – or with some predefined scripts – composed from various smaller interdependent components. With the increase in demand for, and complexity of applications, the composition process becomes an arduous task often associated with errors and a suboptimal use of computer resources. To alleviate such a process, we introduce an approach that uses planning to automatically and dynamically compose applications ready for Cloud deployment. The industry may benefit from using automated planning in terms of support for product variability, sophisticated search in large spaces, fault tolerance, near-optimal deployment plans, etc. Our approach is based on Hierarchical Task Network (HTN) planning as it supports rich domain knowledge, component modularity, hierarchical representation of causality, and speed of computation. We describe a deployment using a formal component model for the Cloud, and we propose a way to define and solve an HTN planning problem from the deployment one. We employ an existing HTN planner to experimentally evaluate the feasibility of our approach.**

***Index Terms*—service composition, automated planning, application configuration, software deployment, cloud computing**

I. Introduction

Cloud computing brings new possibilities of experiencing benefits from software applications. These are no longer installed and running on a single machine, but they are composed of assorted software components that are transparently deployed and distributed on several machines in Cloud infrastructures, and are always available on a reliable network. Consider as an example an application for intelligent energy management of office buildings [1]. The application is supposed to provide office occupants with various representations of energy and environment information, and control a wide range of devices and systems, for instance, a lighting system. Such an application consists of multiple components each of which offers its capabilities as services deployed on the Cloud infrastructure belonging to some office building or building corporation. These services are not necessarily accessible over a network that is open for public use, but they are typically accessible only by the corporations providing or using them (thus greater control and privacy). We refer to such services as *Cloud services*.

The problem

Cloud applications are usually composed manually or with some predefined scripts, either involving strenuous effort and being error prone. Several factors contribute to this. The first one is that although each service is responsible for addressing a specific and separate aspect of an application, there is often high *interdependency* between services [2]. Second, each service may have *multiple versions* each of which includes a different set of requirements for communication, exchange of information, and functionalities of other services [3]. Third, each service may have *multiple instances* running in the same setting [4]. Say there are 300 rooms in some office building. A single instance of a service with some specific functionality, for example, lighting control, may have difficulties with such scaling of the number of offices. This implies that the *number of services* for an actual deployment may vary and increase, which is a fourth factor.

Considering these factors, one has to find, choose and properly configure appropriate services so that they compose applications ready for deployment. We refer to this as a *deployment problem*. The solutions to deployment problems involve *deployment actions*, which are simple operations performed on services, such as installing a service instance, binding service instances, terminating a service instance, etc. With the proliferation of services and requests for application deployments, solving deployment problems requires a lot of resources in the development, configuration, integration and maintenance of applications in Cloud infrastructures. It is therefore vital to search for and decide on deployment actions automatically and dynamically such that these actions configure a required application by interacting with existing service instances and/or creating new ones on the Cloud.

Proposed solution

As a necessary direction to automate the composition of Cloud applications ready for deployment, it appears natural to resort to automated planning [5]. Planning provides powerful methods for searching in large and complex Cloud infrastructures to find "good" compositions of Cloud ready applications. Applications are composed dynamically, thus services need not to be fixed in advance in scripts and always available (the same holds for the servers of the Cloud). Additionally,

planning can be used to handle the Cloud uncertainty (e.g., failures of hardware resources), find deployments optimal with respect to the use of computer resources, etc.

There is an evident basic correspondence between planning problems and deployment problems: planning goals correspond to requests for application deployments, planning states correlate to current deployments or configurations of Cloud infrastructures, and planning actions correspond to deployment actions. In the Cloud setting, however, deployment actions are simple operations without any semantics, keeping the actions separate from the configuration knowledge. To support this modularity of deployment actions and still consider the configuration knowledge when composing Cloud applications, we turn to Hierarchical Task Network (HTN) planning [6]. HTN planning provides support through its rich domain knowledge and hierarchical representation of causality. HTN planning is suitable also due to its speed of computation.

The contributions

We summarise our contributions next.

- We propose to solve the problem of composing applications ready for deployment on Cloud infrastructures via HTN planning. To the best of our knowledge, this is the first proposal to compose Cloud applications using a generic planning technique in contrast to special-purpose planning techniques (see [2], [3]). On the other hand, this sort of problems has a close resemblance with Web service composition, a problem well studied by the planning community. There are however a few notable differences. The first one is that Web services are distributed on the Internet, thus publicly available, and assumed to be registered to some repository. Cloud services, in contrast, are commonly part of well-controlled environments. The second and important issue with Web services lies in the lack of consistent semantic annotations such that make their composition feasible in practice. Even though various ways to describe Web services exist (e.g., SOAP, WSDL, OWL-S), some already deprecated or never used in practice, the reality of Web services is that they are associated only with syntactic specifications and free-text descriptions, leading to the consideration of Web services as nothing more than data sources [7]. Being part of controlled environments, Cloud services have different characteristics: they tend to be structured and described using consistent (in-house) ontologies [8], [9], or even provided with machine-interpretable annotations [10]. Corporations tend to make use of well-established standards and best practices they gain in the domain of service-oriented architectures to support a standardised way of access to Cloud services [11]. In contrast to Web service composition, these considerations foreground the possibility to make the composition of Cloud applications feasible in practice. Third, the configuration processes in Cloud infrastructures involve creation of new service instances, making the composition of Cloud services and our approach distinct in this respect. Another issue that differentiates the two problems but we do not deal here with is the deployment of Cloud services on multiple servers under various resource constraints.
- We establish a formal correspondence between deployment problems and HTN planning problems. In fact, we propose a strategy to create HTN planning problems from deployment problems described using an existing formal model called Aeolus [12]. The Aeolus model enables configuring applications deployable on the Cloud.
- We encode a domain model and use our own domain-independent HTN planner to examine it.
- We evaluate the planner's performance under increasing difficulty of deployment problems, and show that the planner is able to compose applications fast. We then compare it to the performance of an existing planner implemented specifically to handle Aeolus-based deployment problems. As expected, the domain-specific planner outperforms our domain-independent HTN planner, however, the results show the feasibility of HTN planning to compose Cloud applications.

The paper is organised as follows. Section II provides brief descriptions of HTN planning, the Aeolus model and a running example. Section III introduces our modelling strategy and the deployment-based HTN planning problem. Section IV provides details on the experimental evaluation. Section V discusses related work, followed by Section VI that concludes the paper.

II. Preliminaries

HTN planning provides the means for solving deployment problems, and the Aeolus model enables specifying them. We also provide a running example that helps in demonstrating our approach.

A. HTN planning

In HTN planning, the domain model consists of tasks that can be accomplished by operators or methods. An operator represents a transition from a state to another one, while a method predefines how to decompose some task into greater details. Given an HTN planning problem, which consists of an initial state, an initial task network and sets of operators and methods, planning is performed by repeatedly decomposing tasks from the initial task network until operators executable in the initial state are reached.

A primitive task is an expression of the form $pt(\tau)$, where pt is a primitive-task symbol, and $\tau = \tau_1, \ldots, \tau_n$ are terms. A compound task is defined similarly. The set of primitive and compound tasks is a finite set of task names TN. A state s is a set of ground predicates with the closed-world assumption. An *operator* o is a triple $\langle pt(o), pre(o), \mathit{eff}(o)\rangle$, where $pt(o)$ is a primitive task, $pre(o)$ and $\mathit{eff}(o)$ are preconditions and effects, respectively. An operator o is *applicable* in a state s iff $pre(o) \subseteq s$. Applying o to s results into a new state $s[o] = s \cup \mathit{eff}^+(o) \setminus \mathit{eff}^-(o)$. A *task* t is a pair $\langle ct(t), M_t\rangle$, where $ct(t)$ is a compound task, and M_t is a set of methods. A *method* m is a pair $\langle pre(m), tn(m)\rangle$, where $pre(m)$ are

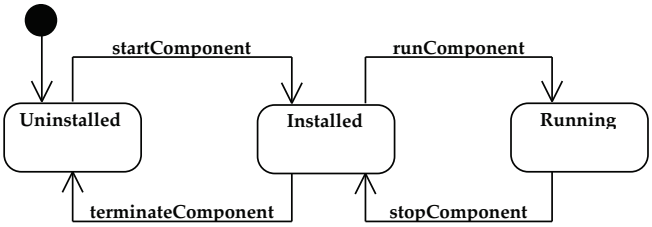

Fig. 1. FSM depicting the state transitions of a component specified in UML.

preconditions and $tn(m)$ is a task network. A method m is *applicable* in a state s iff $pre(m) \subseteq s$. Given a task t such that $m \in M_t$, applying m to s results into a task network $s[m] = tn(m)$. A *task network* tn is a pair $\langle T_n, \prec \rangle$, where $T_n \subseteq TN$, and $\prec$ defines the order of tasks in T_n.

Definition 1 (HTN planning problem): An *HTN planning problem* $\mathcal{P}$ is a tuple $\langle s_0, tn_0, O, T \rangle$, where s_0 is an initial state, tn_0 is an initial task network, O and T are sets of operators and tasks, respectively.

Definition 2 (Solution): Given an HTN planning problem $\mathcal{P}$, a sequence of operators $o_1, \ldots, o_n$ is a *solution* to $\mathcal{P}$, if and only if there exists a task $t \in T_0$, where $tn_0 = \langle T_0, \prec_0 \rangle$, such that $(t, t') \in \prec_0$ for all $t' \in T_0$ and 1) t (or o_1) is primitive and applicable in s_0 such that $o_2, \ldots, o_n$ is a solution to $\mathcal{P} = \langle s_0[o_1], tn_0 \setminus \{o_1\}, O, T \rangle$; or 2) t is compound and there exists an applicable method m such that $tn(m) = (s_0[m], t)$, $tn' = tn_0 \setminus \{t\} \cup tn(m)$, and $o_1, \ldots, o_n$ is a solution to $\mathcal{P} = \langle s_0, tn', O, T \rangle$.

B. Deployment model

We define the problem of configuring and deploying applications on the Cloud by using the Aelous model [12]. The main element of the model is a *component*, describing a manageable resource that provides and requires functionalities. Through the use of state machines, the Aeolus model provides a way to encode specific components declaratively by specifying how functionalities are accomplished. Let us consider a component as the Finite State Machine (FSM) shown in Figure 1. The FSM defines the *state transition* processes of a component, i.e., the states and the order in which a component can transition from one state to another. A component is initially in an *uninstalled* state. Upon start, it transitions into an *installed* state, and then to a *running* state. State transitions are accomplished using *deployment actions*. For example, given some component in its initial state, it is installed by invoking the *startComponent* action.

In most cases, however, a component can transition in some state only if the functionalities that particular state requires through *require ports* are communicated by components that can provide them through *provide ports*. We can observe such transitions in configuration patterns (see Figure 2). A pattern contains a set of components interrelated among each other through the ports on the level of states. The components are abstract, meaning that they will be replaced by concrete

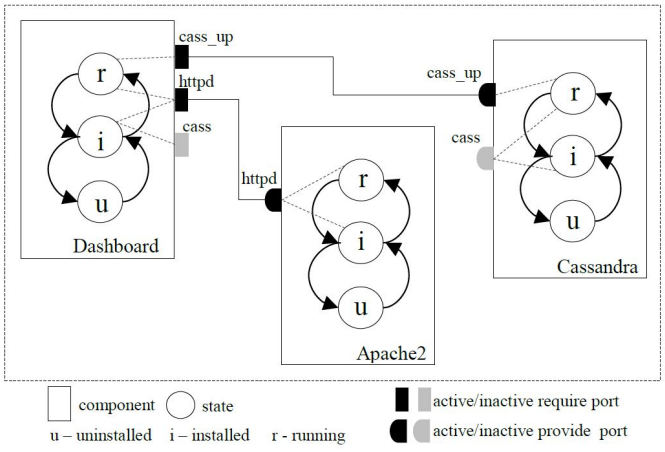

Fig. 2. A pattern for the Public Dashboard application.

components, or *instances*, at runtime. A single configuration pattern therefore defines a number of actual compositions.

A *component* c is a 5-tuple $\langle Q, q_0, U, P, R \rangle$, where Q is a finite set of states, q_0 is the initial state, $U \subseteq Q \times Q$ is the set of state transitions, P is the set of provide ports, and R is the set of require ports. We denote the set of all available components as C, and the set of all ports as F. The set A consists of the deployment actions used upon the elements in C and F. A *configuration* D is a tuple $\langle C, I, \phi, B \rangle$, where C is a set of available components, I is a set of currently deployed component instances, ϕ is a function that associates $i \in I$ with a pair $\langle c, q \rangle$, where $c \in C$ and $q \in Q$ is the current component state; and $B \subseteq F \times I \times I$ is a set of bindings.

A deployment problem consists of an initial configuration, a set of deployment actions, and a request for a new configuration (i.e., application). The solution to the problem is a deployment run representing a sequence of deployment actions on components that, when deployed, produce the required configuration.

C. A running example

Let us consider again the application for energy management in office buildings and suppose that its only capability is to present energy and environment information to office occupants on public screens using Web interfaces. We refer to this application as *Public Dashboard*. Figure 2 graphically represents a simplified Aeolus pattern for composing the Public Dashboard application in a running state. The main and top-level component represents *Dashboard*, which operates using several software services among which essential ones are a Web server and a database. The application requires a database to store all energy and environment information (e.g., energy consumption, light level, weather information, etc.). Cassandra database is preferred and commonly used, but other databases are compatible too. A recommended server is Apache, but any other server that supports the underlying scripting language and database is suitable too. We use *Cassandra* and *Apache2* as components that *Dashboard* depends on.

III. Deployment as an HTN planning problem

Next we introduce the strategy to create an HTN planning problem from a deployment problem. We use the Hierarchical Planning Definition Language (HPDL) [13] when describing the planning structures. In the following, we refer to a state transition that does not depend on any functionality provided by other components as *simple transition*. Otherwise, we use the term *complex transition*.

A. Hierarchical planning domain model

Components, states and ports of components: We encode components, instances, ports as domain types `component instance port`, which are all subtypes of the type `object`. In fact, each component type, such as *Dashboard*, is represented as an object of type `component`.

While FSMs associate components with states abstractly, component instances are the ones to be in a specific state at planning time. We encode an instance state using a predicate "*(state instance)*", where *state* is a string representing the type of an FSM state, and *instance* is a variable representing the component instance. An example of a *Dashboard* instance $d1$ in an installed state is `(installed d1)`.

A component state may be associated with require and provide ports. To represent the association of a port to a state, we use a predicate "*(statePort component port)*", where *statePort* is a string representing the type of port in a specific state, *component* is a variable representing the type of component that requires or provides a port represented by the variable *port*. For example, if *Dashboard* requires the $httpd$ port in the installed state, we encode it as `(installed-require dashboard httpd)`. Such knowledge holds for all instances of the respective component. These predicates are therefore grounded in the initial state and static during planning.

Creating new component instances: One of the features of the composition of Aeolus applications is that one or more component instances must be created from existing (abstract) components. We address the creation of new uninitialised instances using a *domain function*. This function returns a number that we use to represent instance variables in a special predicate `(instance ?iNum - number)`. The `instance-number` function practically serves us as a counter to keep track of the current value that can be assigned for new instances. The domain function does not take arguments. We use an additional predicate `(type ?iNum - number ?c - component)` to associate the instance with a particular component. We increase the instance number, and assert the association by manipulating the effect of the operator that creates new instances as showed in the following encoding.

```
(:action createInstance
 :parameters (?c - component)
 :precondition ()
 :effect (and (instance (instance-number))
              (type (instance-number) ?c)
              (increase (instance-number) 1)))
```

Deployment actions: In addition to `createInstance`, we consider the actions that accomplish simple transitions. These are the deployment actions, including the binding ones. The binding actions are responsible for low-level binding of ports – the require ports are bound to the provide ports. We encode all these actions as HPDL operators. The parameters of operators corresponds either to a component instance variable or to variables of a port and two instances (in the case of binding actions). The preconditions and effects of each operator capture the semantics of the respective action. The following is an operator that corresponds to the *startComponent* deployment action, which makes the state of a instance to become installed and activates all the ports associated with the installed state of the component which the current instance belongs to.

```
(:action start
 :parameters (?i - instance)
 :precondition (and (not (installed ?i)))
 :effect (and (installed ?i)
         (forall (?p - port) (when
         (and (installed-provide ?c ?p)
              (type ?i ?c))(active ?p ?i)))))
```

Other deployment actions are encoded similarly. As for the binding ones, the `bind` operator creates a binding between the provide port of some instance and the require port of another one, and the `unbind` operator deletes an already established binding between two components' instances.

Configuration processes: Although each different type of an application has its own installation and running configuration pattern, the process of configuring applications is general and can be abstracted away. Let us detail how we can accomplish that.

The process of configuring an application requires satisfaction of the dependencies to functionalities provided by components. Let us assume that an instance in an uninstalled state cannot have requirements to be satisfied. We may then consider two abstractions for complex transitions of components. The first abstraction refers to acquiring a component functionality in the installed state, while the second one refers to establishing a functionality in the running state. We point out that complex transitions representing other configuration types can be easily incorporated in the current domain model with minor modifications. HTNs naturally enable encoding knowledge at different levels of abstraction. This support for modularity enables us to focus on a particular level at a time [6]. We can formulate tasks and encode high-level strategies in the methods of these tasks before reasoning on low-level tasks (operators).

We encode each abstraction as a task in the domain model, namely `install` and `run` tasks. Each method of these tasks encodes a specific case. One such method involves port activation. If a component state is associated with one or more require ports, the **port activation** process makes sure that the need of the current instance for specific functionalities is addressed. That is, if the current component instance has require ports that are not active, the method first activates each port and calls recursively its corresponding task until all necessary ports are activated. The actual process of port

activation is encoded in a separate task. The task not only activates a required functionality, but also finds and installs (or runs) a component instance that provides that functionality. An instance with active require ports can then use the functionalities of other components with active provide ports. This is accomplished by another method that involves port binding. The process of **port binding** binds require ports to appropriate provide ports. For this process, the method depends directly on the binding actions. Once we have methods that involve port activation and binding, we can proceed to the method that deals with the case when all require ports are active and bound. To address the satisfaction of all require ports, we use a *forall* expression in the method for both tasks, `install` and `run`. The following expression is used for the `install` task.

```
(forall (?p - port)
        (and (installed-require ?c ?p)
             (bound ?p ?i ?i1))))
```

After this constraint check, we are ready to start or run an instance. In the case of the `run` task, when running an instance, we have to deactivate the ports that will be no longer provided by the instance in the installed state. The process of **port deactivation** is accomplished using a separate task with multiple methods. Each method represents a different case to be handled, such as a provide port that is bound but needed for the running state, a provide port free to be unbound, etc. The port deactivation task uses port unbinding. The process of **port unbinding** is more complex than the binding one, and requires checking for constraint violation. That is, we have to take care of active provide ports bound to active require ports. We use a separate task for this process, that is, `unbindPorts`. This task does nothing when the port is bound and needed for the next transition. When all necessary constraints are satisfied, it unbinds a specific port and recursively calls itself, shown in the following encoding. Being a recursive task, it includes a base case that performs phantomisation [6].

```
:tasks (sequence (unbind ?p ?i ?i1)
                 (unbindPorts ?i))
```

There are methods in the `install` and `run` tasks that deal with the case when there are no required functionalities for an instance. This means that we have a simple transition which can be handled by installing the component instance directly. In the case of running an instance, we invoke the port deactivation task to ensure a valid transition to the running state.

The modelling of the transitions from a running state to an installed state and further to an uninstalled state is analogous to the encoding of the tasks we described so far.

One of the features of these kinds of compositions is that a cycle may occur between states of different component instances. That is, an instance is expected to provide a functionality at a specific point in the composition, but it is not possible because at the same point the instance is required to change its state [3]. We address this feature using the process of **instance duplication**. Instance duplication deals with such cycles by creating as many instances of the same component as needed, and deploying them in different states at the same time. We encode instance duplication as a separate method. The method makes sure that the current component instance is in a specific state and it has at least one provide port bound. Consequently, a new component instance is created either in an installed state or in a running state, depending on the type of configuration.

Algorithm 1 shows the high-level steps of the strategy we described for the creation of an HTN domain model.

Algorithm 1 Transformation of an Aeolus model into an HTN planning domain model

Input: a set of components C, a set of deployment actions A
Output: HTN planning domain model $\langle O, T \rangle$
1: Encode component, instance, port as types
2: Choose $c = \langle Q, q_0, U, P, R \rangle$ from C
3: **for** $j = 1$ to $|Q|$ **do**
4: Create state predicate and port predicates for q_j, $q_j \in Q$
5: **end for**
6: Encode an operator o for creating instances
7: **for** $j = 1$ to $|A|$ **do**
8: Encode a_j as an operator o_j, $a_j \in A$
9: **end for**
10: Ask the user questions regarding the configuration processes in $\langle C, A \rangle$, and encode the corresponding tasks

B. Deployment-based HTN planning problem

A *deployment problem* $\mathcal{P}^D$ is a tuple $\langle D_0, A, G \rangle$, where D_0 is the initial configuration, A is the set of deployment actions, and G is the requested configuration. δ is a satisfying deployment run for $\mathcal{P}^D$ if and only if δ is a sequence of deployment actions that transform D_0 into G. A requested configuration, G, is achievable if and only if there exists at least one satisfying deployment run for it.

Given a deployment problem $\mathcal{P}^D$, we define the corresponding deployment-based HTN planning problem $\mathcal{P}$ according to Definition 1, where 1) s_0 is the initial state consisting of a list of the following ingredients derived from D_0: components and ports as objects, component states, currently deployed instances, the current state of deployed instances and bindings as the special predicates we defined in the HTN planning domain model. s_0 also contains a domain function initialised to 0. 2) tn_0 is the initial task network encoding the requested configuration G; 3) O is the set of operators that represent actions in A, and T is the set of tasks derived from the configuration processes with respect to Algorithm 1. A plan π is a solution to $\mathcal{P}$ according to Definition 2.

Theorem 1: Let $\mathcal{P}^D$ be a deployment problem and $\mathcal{P}$ be the corresponding HTN planning problem. If a requested configuration G is achievable, then there exist a plan π for $\mathcal{P}$.

Let δ be a satisfying deployment run for $\mathcal{P}^D$ such that G is achievable. Under the assumption that the user provides reasonable answers – there is a correspondence between $\mathcal{P}^D$ and $\mathcal{P}$ as defined previously, then there must exist a solution for $\mathcal{P}$.

We can now obtain that the solution of the deployment-based HTN planning problem is a deployment run for the corresponding deployment problem.

Theorem 2: Let $\mathcal{P}^D$ be a deployment problem and $\mathcal{P}$ be the corresponding HTN planning problem such that Theorem 1 holds. We can then construct a sequence of deployment actions based on π that is a satisfying deployment run for $\mathcal{P}^D$.

Let us present a constructive proof for which we consider the deployment problem $\mathcal{P}^D$ shown in Figure 2. Let $\mathcal{P}$ be the corresponding deployment-based HTN planning problem. Furthermore, consider the following plan for $\mathcal{P}$: [createInstance(d0), createInstance(a1), start(a1), bind(httpd,d0,a1), start(d0), createInstance(c2), start(c2), run(c2), bind(cassup,d0,a2), run(d0)]. We can construct a deployment run in which the actions from the plan are deployment actions. The resulting deployment run is a satisfying deployment run for $\mathcal{P}^D$.

IV. Experimental evaluation

Motivation: Consider extending the application for managing office buildings and suppose that its capabilities go beyond those of the Public Dashboard. Typically, such an application consists of a number of primary components responsible for implementing core processes, and several secondary components that complete the operation cycle of the application [1]. The primary and secondary components are all highly interdependent. Say that some building is equipped with numerous heterogeneous devices, such as sensors and actuators. A primary component wraps up and interacts with these devices in such a way that it gathers the information they provide (e.g., light level), and executes low-level commands (e.g., turn on a lamp). Some of these functionalities are used by another component that amasses the device information and provides it as unified raw data to other interested components. Among those, an essential one processes the raw data and exposes it as meaningful context information. The component that provides automated control reasons over the context information and selects device actions that achieve some building objective. These actions are further processed by another component and send out to the component responsible for executing low-level commands. Other primary components may focus on more specific issues, such as collection and measurement of only electricity consumption of devices. As secondary components, different databases are used, for example, one for storing raw and context data and another for saving descriptive information about the building; message brokers are used for asynchronous communication between the components, etc.

The primary and secondary components are implemented as Cloud services, which can be in all three states described earlier. We see that services are dependent among each other, thus they have require and provide ports. We consider the *degree of dependence* a computational factor. Furthermore, the final application is intended to be deployed in a private Cloud. Given that such an application may be run in environments of varying size (e.g., small and large office buildings), the number of components involved in the application may reach relatively high. We therefore evaluate the efficiency of our approach under increasing *number of components*. Finally, components may have multiple instances running, for instance, to cover different building spaces (e.g., floors, offices, common spaces, etc.). The need for *instance duplication* increases the difficulty of planning problems too.

Set-up: We make planning problems more interesting and challenging with respect to component interdependencies by having the requested configuration of applications to appear deeply in the right of the search space. We use a set of components $c_1, \ldots, c_n$, where each c_i has require and provide ports as follows. Given that we want to have the rightmost component c_n in its running state, the dependencies between components will require to first create instances for components from c_1 to c_n, then to perform transition from uninstalled to installed state in the reverse order of component instances, and finally, to transition from installed to running state in the order from c_1 to c_n. Then, we increase the difficulty of planning problems with respect to the number of components by varying the number from 3 to 300, resulting in more than 50 problems. These constitute our first test case.

Using the setting of the first test case, we create a second test case to increase the difficulty of planning problems in such a way that configurations require instance duplication. We randomly select several components and, for a selected component c_i, we remove the activation of a provide port p_i^1 from its running state. The removal requires another instance of c_i to be created so as to satisfy the requirements of c_{i-1} and c_{i+1}.

We use our own HTN planner, called Scalable Hierarchical (**SH**) planning system [14], to solve the planning problems of the two test cases and to evaluate the feasibility of HTN planning for composing Cloud applications. **SH** is a domain-independent HTN planner implemented entirely in the Scala programming language. It consists of two main modules, namely HPDL processor and Planner. HPDL problem and domain descriptions are transformed into programming-level constructs through the HPDL processor. The Planner includes the main algorithm which is based on depth-first search. **SH** shares similarities with two existing HTN planners: the support for HPDL with SIADEX [15] and the search mechanism with SHOP2 [16].

We run **SH** on a Intel Core i7-3517U @1.90GHz, 8GB RAM machine running Windows 8.1 and Java 1.8.0_31.

To assess the impact of using HTN planning, we compare the results of the performance of **SH** with the results of the performance of a planner developed specifically for solving Aeolus-based deployment problems [3]. This domain-specific planner is evaluated in an experimental set-up similar to ours, thus we use their reported results directly.

Results: Figure 3 shows the results of the both planners, where the number of generated instances equates to the number of components. Even though **SH** shows worse performance than the domain-specific planner, which is expected, deployment problems with 200 components can be solved in less than 15 seconds.

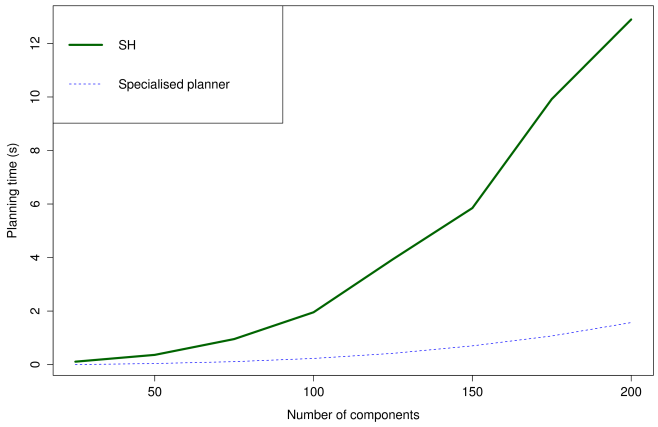

Fig. 3. Performance of our domain-independent HTN planner and a domain-specific planner on deployment problems without instance duplication.

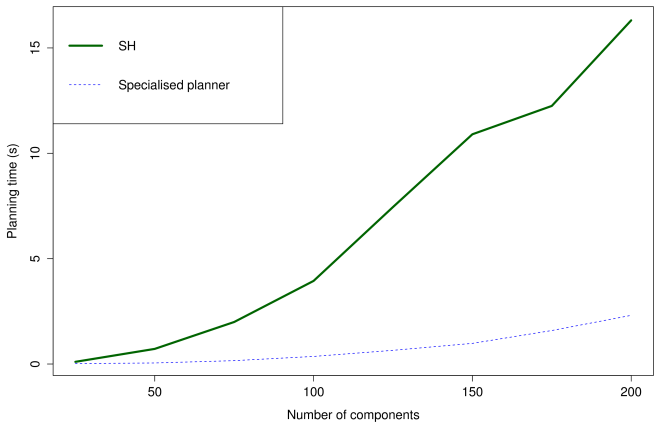

Fig. 4. Performance of our domain-independent HTN planner and a domain-specific planner on deployment problems with instance duplication.

Figure 4 shows the results of our planner and the domain-specific planner when used in the second test case. The number of created instances is strictly greater than the number of components. With the creation of a new instance, the size of the state is increased by adding two predicates, and the state is modified by updating the domain function. Here too, the performance of **SH** falls behind the one of the domain-specific planner. Additionally, when the number of components is larger than 120, the need for instance duplication degrades the performance of **SH** as compared to the case without instance duplication.

On a side note, Lascu et al. [3] report on the performance of two domain-independent (non-HTN) planners on the same set of deployment problems. One planner can solve deployment problems with 3 components without instance duplication in 0.05 seconds. The other planner solves problems with up to 7 components without instance duplication in 7.22 seconds, and up to 5 components with instance duplication in 3.44 seconds. Our HTN planner outperforms both planners significantly, though their results seem symptomatic and unexpected even for pure domain-independent planners.

V. Related work

The problem of composing applications ready for deployment via automated planning has been addressed, to the best of our knowledge, in two studies:

- Arshad *et al.* describe a problem of deploying software systems, and uses a temporal-based planner to search for an optimal plan with respect to plan duration [2]. While we also deal with configuring software application, we tackle two important issues, not addressed in this study, namely new instance creation and modelling configuration processes, making it possible to apply planning to Cloud-based applications. In addition, we use a formal model for the Cloud to derive planning problems, we allow for more than once instance of a service to exist at a time, and the goal does not need to include the ports for connection – the planner figures that out automatically.
- Lascu *et al.* describe a deployment problem based on the Aeolus formal model and presents a domain-specific planner to search for a solution [3]. This means that all configuration processes and features are implemented and embodies in the planning process. We however encode all domain-specific knowledge in the domain model, making the approach flexible and extensible to new features and capabilities. Also, our approach does not require the initial configuration to be empty.

More generally, the problem of composing Cloud services have a close resemblance with the problem of Web service composition. Various aspects of Web service composition have already been addressed by numerous planning approaches, e.g., [17]–[19]. Existing approaches however overlook an important characteristic about the Web service composition: a Web service can represent either an abstract Web service type or one or more instances of a specific Web service [4]. In the existing approaches, the Web service composition consists of synthesising a Web service type, which seems to be sufficient for the scenarios considered – too small to involve multiple service instances. In practice, however, there is a choice among many instances of a Web service. One of the distinct features of our approach is the creation of new and multiple instances of Cloud services during runtime.

Looking at HTN planning, it is employed to represent and compose Web services in several studies summarised in [6]. Common among those studies is the assumption that Web services are represent in OWL-S and can be transformed into HTNs. OWL-S is a language specifically designed to support the discovery, composition and monitoring of Semantic Web services. In reality, however, the language supports essentially only behavioural descriptions of services [7], [20]. Such descriptions seem insufficient to be correctly translated into HTNs, and moreover, inappropriate to reason over. This drawback prevents OWL-S from being used in practical and real-world cases at all. On the other hand, our approach is

not dependent on a specific modelling language, but on a formal model that captures the semantics of current and future controlled Cloud infrastructures. Additionally, the studies assume the existence of OWL-S compound Web services which can be translated to HTN methods and compound tasks (for details, see [17]). On the other hand, we do not use any compound Cloud services, but we use compound tasks to model configuration processes.

Contrary to the approach taken in [21], where assignment expressions encoded in the precondition of SHOP2's operators are used to create new streams, we create new instances using domain functions and numerical fluents modelled in the effects of HPDL actions. It would be interesting to analyse whether there are performance benefits from these two different encoding approaches. Additionally, we allow for existence of multiple instances.

In Cloud computing, the problem of managing interconnected machines has been addressed by many tools, such as Wrangler [22], SmartFrog [23], CFEngine [24], Puppet [25], Chef,[1], and Ansible.[2] These tools support specifying the components, together with their configuration files, to be installed on machines, and then, by using various mechanisms, deploy the components accordingly. The task of specifying which component to deploy where, and how to interconnect it to other components is however left to the user. Furthermore, ConfSolve [26] is used to search for an optimal allocation of virtual machines on servers and applications on virtual machines. However, the tool does not handle the problem of composing interdependent services. Juju[3] and Engage [27] are focused on a problem similar to ours, avoiding some issues related to the connection between components. For example, while our approach supports circular dependencies, these cannot be defined in Engage. In Juju, circular dependencies must be resolved manually.

VI. Conclusions

We examined the connections between the task of composing Cloud applications and automated planning. We proposed the use of HTN planning, described a deployment problem based on a formal model for the Cloud, and presented how to model an HTN planning problem from the deployment one. We showed that HTN planning offers a possibility to express various constraints on the composition, dynamic instance creation, recursion through the use of tasks, and instance duplication provided in the domain model.

The experimental evaluation illustrated that HTN planning can compose Cloud applications of 100 components in less than 4 seconds, and applications of 200 components in about 15 seconds. This gives a concrete advantage of automated planning over the popular tools used in Cloud computing. We also showed that our domain-independent HTN planner is comparable to a planner developed specifically for this type of problems. In contrast to prior findings [3], we showed that even domain-independent planners are able to compose Cloud applications fast.

The advantages of our approach include the modularity and flexibility of the approach to further improvements and developments; the speed of computation; and the amount of effort spent to model HTN planning problems as compared to the effort spent developing (and extending) a domain-specific planner and/or tool. The contributions of our study include the establishment of a stronger relationship between Cloud computing and HTN planning; a model of deployment-based HTN planning problems; the dynamic instance creation; and the support for instance duplication.

[1]https://www.chef.io/chef/

[2]https://www.ansible.com/

[3]https://jujucharms.com/

References

[1] F. Nizamic, T. A. Nguyen, A. Lazovik, and M. Aiello, "Greenmind - an architecture and realization for energy smart buildings," in *International Conference on ICT for Sustainability*, 2014.

[2] N. Arshad, D. Heimbigner, and A. L. Wolf, "Deployment and dynamic reconfiguration planning for distributed software systems," in *IEEE International Conference on Tools with Artificial Intelligence*, 2003, pp. 39–46.

[3] T. A. Lascu, J. Mauro, and G. Zavattaro, "A planning tool supporting the deployment of cloud applications," in *IEEE International Conference on Tools with Artificial Intelligence*, 2013, pp. 213–220.

[4] B. Srivastava, "Planning with workflows - an emerging paradigm for web service composition," in *ICAPS Workshop on Planning and Scheduling for Web and Grid Services*, 2004, pp. 78–85.

[5] A. Lazovik, M. Aiello, and M. Papazoglou, "Planning and monitoring the execution of web service requests," in *Service-Oriented Computing - ICSOC 2003*, ser. Lecture Notes in Computer Science, M. E. Orlowska, S. Weerawarana, M. P. Papazoglou, and J. Yang, Eds. Springer, 2003, vol. 2910, pp. 335–350.

[6] I. Georgievski and M. Aiello, "HTN planning: Overview, comparison, and beyond," *Artificial Intelligence*, vol. 222, no. 0, pp. 124–156, 2015.

[7] J. Fan and S. Kambhampati, "A snapshot of public web services," *SIGMOD Rec.*, vol. 34, no. 1, pp. 24–32, 2005.

[8] R. Ranjan, L. Zhao, X. Wu, A. Liu, A. Quiroz, and M. Parashar, "Peer-to-peer cloud provisioning: Service discovery and load-balancing," in *Cloud Computing*, ser. Computer Communications and Networks, N. Antonopoulos and L. Gillam, Eds., 2010, pp. 195–217.

[9] J. Kang and K. M. Sim, "Towards agents and ontology for cloud service discovery," in *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, 2011, pp. 483–490.

[10] Z. Zhang, C. Wu, and D. W. Cheung, "A survey on cloud interoperability: Taxonomies, standards, and practice," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 4, pp. 13–22, 2013.

[11] Y. Wei and M. B. Blake, "Service-oriented computing and cloud computing: Challenges and opportunities," *IEEE Internet Computing*, vol. 14, no. 6, pp. 72–75, 2010.

[12] R. Di Cosmo, S. Zacchiroli, and G. Zavattaro, "Towards a formal component model for the cloud," in *International Conference on Software Engineering and Formal Methods*, 2012, pp. 156–171.

[13] J. Fdez-Olivares, L. Castillo, O. García-Pérez, and F. Palao, "Bringing users and planning technology together. Experiences in SIADEX," in *International Conference on Automated Planning and Scheduling*, 2006, pp. 11–20.

[14] I. Georgievski, "Coordinating services embedded everywhere via hierarchical planning," Ph.D. dissertation, University of Groningen, October 2015. [Online]. Available: https://www.rug.nl/research/portal/files/23863757/Complete_thesis.pdf

[15] L. A. Castillo, J. Fernández-Olivares, Ó. Garca-Pérez, and F. Palao, "Efficiently handling temporal knowledge in an HTN planner," in *International Conference on Automated Planning and Scheduling*, 2006, pp. 63–72.

[16] D. S. Nau, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman, "SHOP2: An HTN planning system," *Journal of Artificial Intelligence Research*, vol. 20, no. 1, pp. 379–404, 2003.

[17] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. S. Nau, "HTN planning for Web service composition using SHOP2," *Web Semantic*, vol. 1, pp. 377–396, 2004.

[18] S. Sohrabi, N. Prokoshyna, and S. A. Mcilraith, "Web service composition via generic procedures and customizing user preferences," in *International Semantic Web Conference*, 2006, pp. 597–611.
[19] E. Kaldeli, A. Lazovik, and M. Aiello, "Continual planning with sensing for Web service composition," in *AAAI Conference on Artificial Intelligence*, 2011, pp. 1198–1203.
[20] S. Balzer, T. Liebig, and M. Wagner, "Pitfalls of owl-s: A practical semantic web use case," in *International Conference on Service Oriented Computing*, 2004, pp. 289–298.
[21] S. Sohrabi, O. Udrea, and A. Riabov, "HTN planning for the composition of stream processing applications," in *International Conference on Automated Planning and Scheduling*, 2013, pp. 443–451.
[22] G. Juve and E. Deelman, "Automating application deployment in infrastructure clouds," in *IEEE International Conference on Cloud computing technology and science*, ser. CloudCom, 2011, pp. 658–665.
[23] J. Kirschnick, J. M. Alcaraz Calero, P. Goldsack, A. Farrell, J. Guijarro, S. Loughran, N. Edwards, and L. Wilcock, "Towards an architecture for deploying elastic services in the cloud," *Softw. Pract. Exper.*, vol. 42, no. 4, pp. 395–408, 2012.
[24] M. Burgess, "Cfengine: A site configuration engine," *Computing systems*, vol. 8, no. 2, pp. 309–337, 1995.
[25] L. Kanies, "Puppet: Next-generation configuration management," *Computing systems*, vol. 31, no. 1, pp. 19–25, 2006.
[26] J. A. Hewson, P. Anderson, and A. D. Gordon, "A declarative approach to automated configuration," in *Large Installation System Administration Conference*, 2012, pp. 51–66.
[27] J. Fischer, R. Majumdar, and S. Esmaeilsabzali, "Engage: A deployment management system," *SIGPLAN Not.*, vol. 47, no. 6, pp. 263–274, 2012.