Policy-Aware Provisioning of Cloud Applications

Uwe Breitenbücher, Tobias Binz, Oliver Kopp, Frank Leymann, Matthias Wieland
Institute of Architecture of Application Systems
University of Stuttgart, Stuttgart, Germany
{breitenbuecher, lastname}@iaas.uni-stuttgart.de

***Abstract*—The automated provisioning of complex composite Cloud applications is a major issue and of vital importance in Cloud computing. It is key to enable Cloud properties such as pay-as-you-go pricing, on-demand self-service, and elasticity. The functional aspects of provisioning such as instantiating virtual machines or installing software components are covered by several technologies on different technical levels: some are targeted to a pretty high level such as Amazon's Cloud Formation, some deal with deep technical issues based on scripts such as Chef or Puppet. However, the currently available solutions are tightly coupled to individual technologies without being able to consider non-functional security requirements in a non-proprietary and interoperable way. In this paper, we present a concept and framework extension enabling the integration of heterogeneous provisioning technologies under compliance with non-functional aspects defined by policies.**

***Keywords*—Cloud Applications; Provisioning; Security; Policies**

I. Introduction

The increasing use of IT in almost every part of enterprises leads to a higher management effort in terms of development, deployment, delivery, and maintenance of applications. For enterprises, IT management becomes a challenge as each new technology increases the degree of complexity while operator errors account for the largest fraction of failures [1]. These issues have been tackled by outsourcing IT to external providers and automating the management of IT—both enabled by Cloud computing [2]. The modular component-based architectures that are the consequence of using Cloud services allow to benefit from Cloud properties such as elasticity, scalability, and high availability without the need to have technical insight [3]. Unfortunately, the necessary balance between functional possibilities and non-functional security issues has been often skewed toward the first. Cloud services are typically easy to use on their own but hard to configure and extend in terms of non-functional aspects that are not covered natively by the offering. Creating applications that integrate different heterogeneous components which are hosted on or interact with Cloud services while fulfilling non-functional security requirements can quickly degenerate to a serious problem, especially if the technical insight is missing. Even the initial provisioning of applications can be a difficult challenge if non-functional requirements of different domains with focus on heterogeneous technologies have to be fulfilled.

In this paper, we present a concept and extend an existing Management Framework [4] to tackle these issues. The extension enables the fully automated provisioning of composite Cloud applications complying with non-functional security requirements from different domains which are expressed by *Provisioning Policies*. We introduce *Policy-Aware Management Planlets* that allow the implementation of policy-aware provisioning logic in a reusable way independently from individual applications. The presented framework enables Cloud providers as well as application developers to specify security requirements for the provisioning without the need to have the deep technical management knowledge needed in other approaches. In addition, the concept allows security experts of different domains to work together in a collaborative way. We evaluate the approach in terms of performance, complexity, economics, feasibility, extensibility, and the implementation of a prototype. The remainder of this paper is structured as follows. In Section II, we explain fundamentals and motivate our approach by a scenario presented in Section III. In Section IV, we introduce Provisioning Policies. Section V describes the Management Framework that is extended by our approach, which is presented in Section VI. Section VII evaluates the approach and Section VIII reviews related work. We conclude this paper and give an outlook on future work in Section IX.

II. Fundamentals

In this section, we explain two fundamental concepts providing the basis for the remainder of the paper: (i) Application Topologies and (ii) Management Plans.

A. Application Topologies

An application topology is a directed, possibly cyclic graph describing the structure of an application. It consists of nodes, which represent the different components of an application such as Virtual Machines (VM) or Java applications, and the relations among them, which are the edges between the nodes. Nodes and relations are called *elements* of the topology and have a type attribute defining its semantics. Each element may have a set of arbitrary properties used to describe the element in detail. Figure 1 shows an example, which describes a PHP-based Web shop application that stores data in a database backend. We use the visual notation *Vino4TOSCA* [5] to depict topologies graphically. Following this notation, the type of an element is enclosed by parentheses whereas its name is undecorated text. The application's infrastructure is provided by Amazon's public Cloud in the form of two virtual machines of type *AmazonEC2VM*. Thereon, operating systems are installed of type *Windows7* and *UbuntuLinux*. A PHP runtime of type *ApachePHPWebServer* hosts the application of type *PHP*. This application connects to a database of type *MySQL* which is hosted on the Linux operating system of the right stack. We modeled all relations as *hostedOn* relation and left out properties to simplify the diagram. Our approach employs application topologies to describe the structure of applications to be provisioned and to attach policies to the contained elements.

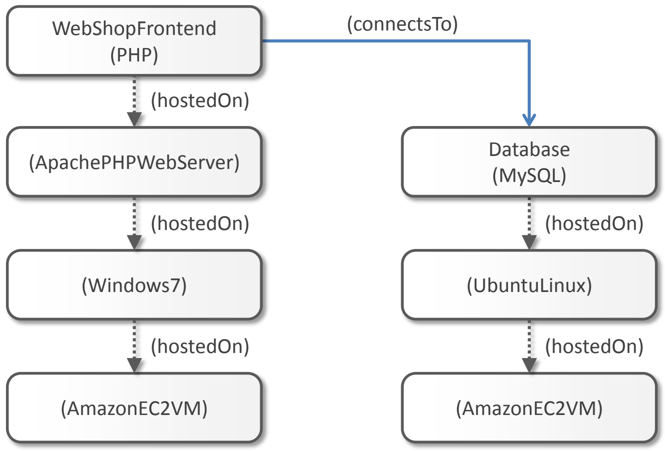

Figure 1. Example of a PHP-based application topology.

B. *Management Plans*

Management Plans are executable workflows used to automate the management of applications such as the provisioning of applications. They enable a much more robust and reliable way to manage applications than the manual or script-based management through technologies on a deep technical level. They inherit features from workflow technology such as recoverability, compensation, and fault handling mechanisms. One workflow language to implement these plans is the Business Process Execution Language (BPEL), which is, for example, used for the provisioning of applications [6]. Management Plans are often created manually by the application developers themselves. However, this is a difficult and time-consuming task and, as plans are typically coupled tightly to single applications, of limited value: Plans are mostly sensitive to structural differences and, therefore, hardly reusable for the management of other applications. In this paper, we generate so-called *Provisioning Plans*, which are a subclass of Management Plans, to fully automate the provisioning of applications based on their topologies.

III. Motivating Scenario

The following scenario is used to motivate the approach presented in this paper. Let us assume a company wants to host the PHP-based Web shop application described in the previous section on Amazon's public Cloud (Figure 1). Depending on the importance of that application for the company, there may be different non-functional security requirements. For example, the Web server hosting the functional logic typically comes up with default credentials, i. e., username and password are both "admin". If this is ignored, the Web server is open for attacks and may be misused. Thus, we need a mechanism to ensure that the strength of these credentials is considered during the deployment. The second example deals with the products data stored in the database. If the company financially depends on this Web shop, data loss caused by a failure may ruin the business. Thus, a frequent data backup of this database is needed. The last example deals with the geographic location of the company's customer data, which is also stored in the database. Legal aspects may require that this data has to remain in the European Union (EU). Thus, the virtual machine must be hosted on a physical server located in one of its states.

IV. Policies for Provisioning of Applications

In this section, we introduce *Provisioning Policies* which are used by our approach to express non-functional security requirements on the provisioning of applications. Provisioning Policies are a subclass of Management Policies which are employed in systems and network management. We describe Management Policies in detail and refine them into Provisioning Policies afterwards to clearly define the kinds of policies that are supported by the presented approach.

A. *Management Policies*

Management Policies are a well-known concept and common in research as well as in industry. They are derived from management goals and used to influence the management of applications, resources, and IT in general based on non-functional aspects such as security, performance, or cost requirements. They provide a (semi-) formal concept used to capture, structure, and enforce the objectives [7]. A lot of work on policies exists dealing with classifications, methodologies, and applications. To classify and identify the policies covered by our approach, we use the hierarchy of Wies [7] that classifies policies based on the level on which they influence the management. The hierarchy was developed based on criteria such as policy life-time, how they are triggered and performed, and the type of its targets. Wies differentiates between four classes: (i) Corporate/High-Level Policies, (ii) Task Oriented Policies, (iii) Functional Policies, and (iv) Low-Level Policies. Corporate Policies are directly derived from corporate goals and embody *strategic business management* rather than technical management aspects. The other three classes embody *technology oriented management* in terms of applying management tools, using management functions, and direct operation on the managed objects. Our approach covers the technology oriented management in terms of security. The most important requirement of security policies is to ensure strict adherence. The system must prevent that the security requirements defined by a policy get violated. This is vital as many types of policies cause actions that cannot be undone if once violated. For example, if a Data Location Policy defines that the application data must not leave a certain region (legal rights), i. e., the physical servers must be located in that region, and the data gets distributed over the world by a public Cloud, then the policy is violated and it is impossible to undo this violation. Violating such policies may result in high penalties.

B. *Provisioning Policies*

Provisioning Policies are a subclass of Management Policies and responsible for ensuring non-functional requirements during the provisioning of applications. During our research on application provisioning, we identified three different kinds of policies that must be considered: (i) Configuring Policy, (ii) Guarding Policy, and (iii) Extending Policy. We explain these three kinds in the following and give examples based on the motivating scenario introduced in Section III. Our approach supports primarily these three kinds of policies. A *Configuring Policy* configures the provisioning of components or relations. For example, a *Data Location Policy* attached to a virtual machine with *region* value "EU" configures the provisioning in a way that the virtual machine is hosted on a server located in the European Union. A *Guarding Policy* guards the provisioning of components or relations, i. e., it supervises the instantiation in terms of certain specified values or thresholds. For example, a *Safe Credentials Policy* ensures that the strength of login credentials, i. e., username and password, is strong enough. A *Extending Policy* extends the provisioning in terms of structure, i. e., it may add new components or relations which are not contained in the original application topology to be provisioned. For example, a *Frequent Data Backup Policy* for a database causes the installation of an additional software component on the operating system which backups specified database tables in a certain time interval to a certain location.

V. USED MANAGEMENT FRAMEWORK

In this section, we explain the Management Framework that gets extended to support processing non-functional security policies. We presented this framework in a former paper [4], which describes all details about the framework, Management Planlets, and Management Annotations.

The main functionality of the framework is managing running applications by generating Management Plans that are executed to perform the desired changes. Therefore, a Plan Generator gets a so-called *Desired Application State Model* as input which defines the desired state the application shall have after executing the management task. This model is an application topology which reflects the desired state: Components may have to be added or removed, relations must be established, or updates must be installed, to mention a few examples. To define the low-level management tasks that have to be performed to achieve the desired state, we use so-called *Management Annotations*, which are described in the next section. Based on this topology, a Management Plan is generated that brings the application from the current state to this desired state, i. e., performs the management tasks. This generation orchestrates so-called *Management Planlets* that provide small management tasks such as installing or removing components or establishing a database connection. Details are explained in Subsection V-A. All available planlets are stored in a repository which is used by the plan generator to find appropriate planlets to generate the plan. For example, if such a management task defines that a database of type *MySQL* must be installed on an existing operating system of type *UbuntuLinux* and data needs to be imported afterwards, the plan generator uses a planlet that installs the database and another planlet to import the data. Thus, planlets enable separating different functional low-level tasks in order to be orchestrated to provide a higher level of management tasks.

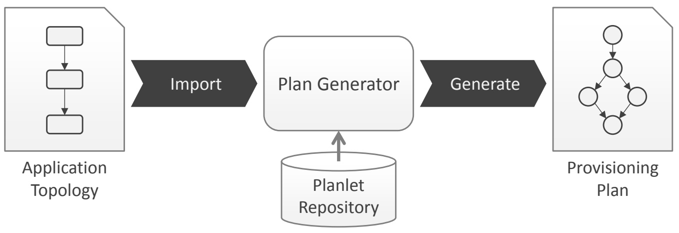

Figure 2. Management Framework for generating Provisioning Plans.

The framework also supports generation of provisioning plans, i. e., plans that provision a complex composite Cloud application on each level: from infrastructure to software components with all required relations among them. This is shown in Figure 2: The plan generator gets an application topology as input and generates the corresponding Provisioning Plan by orchestrating several planlets. In addition, the whole application to be provisioned is packaged into a so-called *Application Package* that contains all artifacts needed for provisioning such as installables. Planlets may be also contained in this package. This enables the developer to implement own custom planlets that are taken into account by the plan generator. Thus, the concept of planlets enables even the developers of the application themselves to define and influence the provisioning logic. However, the framework currently does not support the consideration of non-functional requirements. Therefore, one contribution of this paper is making the framework policy-aware to tackle this issue. In the next subsections, we provide more details about the framework to help understanding our extension presented in Section 6.

A. *Orchestration of Management Planlets*

Management Planlets are small workflows implementing reusable low-level management tasks such as creating a virtual machine on Amazon EC2, installing a Tomcat on an operating system, or establishing a connection between an application's frontend and its database backend. Planlets are intended to be orchestrated by so-called Management Plans, which provide a higher level of management tasks such as the provisioning of a whole application consisting of multiple interconnected nodes. Thus, they serve as generic building blocks for the generation of Management Plans. In general, planlets are used to manage a running application, but they can be used for the initial provisioning as shown in [4], too. Management Planlets express their functionality through an annotated application topology fragment. This fragment contains a small application topology consisting of typed nodes and relations which may be annotated with so-called *Management Annotations*. These annotations express small low-level management tasks that are performed by the planlet on the respective nodes or relations. The types of components and relations and the annotations are used by the plan generator to find suitable planlets for the desired tasks. A more detailed description of these annotations is given in Section V-B. A single planlet may perform multiple of these small management tasks in order to implement a more sophisticated task, e. g., a single planlet may install a database on an operating system and imports data afterwards. Planlets often need to express preconditions which must be fulfilled to make the planlet applicable, e. g., the previous mentioned planlet which installs the database on an operating system and imports data afterwards requires the operating system to be already running. These preconditions are also expressed through the topology fragment: all therein contained elements, i. e., all nodes and relations, and their properties which have no Management Annotations are treated as precondition. Thus, the operating system would be contained in the topology fragment without any annotation but with state-property value "Instantiated" as shown in Figure 4. This indicates that the operating system must be running. The state-property may be set by another planlet which provisioned the virtual machine and the operating system before. Thus, the ordering and selection of planlets in the overall Management Plan is calculated based on these properties and Management Annotations. During the plan generation, a virtual representation of the current state of the application gets

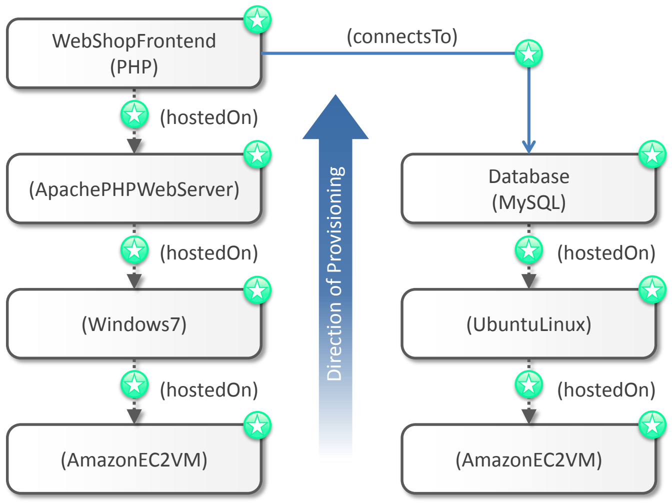

Figure 3. Annotated application topology to be provisioned.

transferred towards the goal state, i. e., all elements are created. In each state, all planlets whose preconditions match the current virtual state are called *Candidate Planlets*. These planlets are eligible to be applied. The plan generator then decides which planlet to choose to transfer the application into the next state. Planlets may also communicate with each other via properties, e. g., one planlet writes the IP-address and credentials of a Web server to the corresponding node which are used by another planlet to deploy applications on it. In addition, planlets may have input parameters used to get information they need from the user. These parameters are exposed to the input message of the overall generated Management Plan. For example, if a planlet needs Amazon credentials to acquire a virtual machine, it defines them in its local input message and the system exposes them to the global input message of the overall generated plan. Therefore, the plan generator adds activities transferring the data internally to the planlet. In the next section, we explain the used Management Annotations in detail.

B. Management Annotations

Management Annotations express low-level management tasks which have to be performed on the nodes and relations they are attached to. They are subdivided into two disjoint classes: *Structural Management Annotations* and *Domain-Specific Management Annotations*. The first class contains annotations that structurally change the application in terms of creating or destroying nodes or relations. Thus, there is a *Create-Annotation* and a *Destroy-Annotation*. For plain provisioning of applications only the Create-Annotation is used and annotated to all elements in the topology model as shown in Figure 3. The green circle with the star inside represents the Create-Annotation. These annotations tell the system that the corresponding nodes and relations shall be created. Figure 4 shows a planlet fragment that may be used to provision a part of this topology: The shown planlet creates a node of type *MySQL* and a relation of type *hostedOn* to an already existing node of type *Linux*, i. e., it installs a MySQL database on a Linux operating system. The precondition of the planlet is that the operating system is already running which is expressed by its state-property set to "Instantiated". The MySQL node to be created also has this state-property but with a Create-Annotation, which means that the planlet sets this property to the specified value "Instantiated". This property may be a precondition for another planlet which connects an application to this database or imports initial data. Thus, the installation planlet has to be executed before them. Application components connected by *hostedOn* relations are typically instantiated bottom up, i. e., the first planlets matching the topology are those creating the infrastructure and middleware components. After creating the basis, other planlets are applicable that have the existence of those components as precondition.

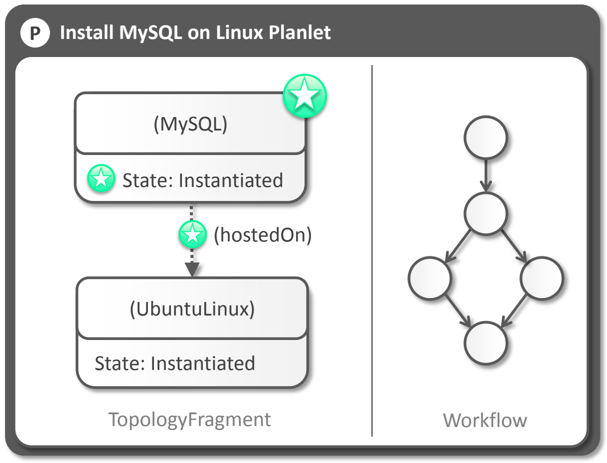

Figure 4. Planlet installing a database on a running operating system.

The second class is used to express domain-specific management tasks such as importing data into a database. Management Annotations are identified by a unique id, which is used for matchmaking. Thus, this enables defining any additional management task needed to configure the overall provisioning.

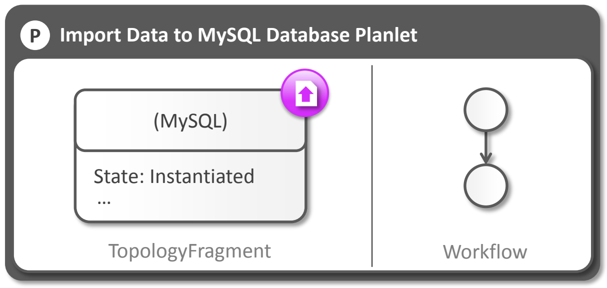

Figure 5. Planlet doing a database import.

Figure 5 shows a planlet that imports data to a database. This is expressed by the domain-specific *ImportData-Annotation* attached to the MySQL node, which is depicted by the purple cycle. This planlet must not be executed before the database is instantiated because of the state-property precondition. Therefore, considering the two example planlets shown in Figure 4 and Figure 5, the planlet that installs the database has to be executed before the planlet doing the import. Based on this concept, planlets get selected and ordered by the plan generator. Of course, the second planlet may have much more property preconditions such as database credentials and endpoint information. We ignore these to simplify the figures.

The concept allows distributing logic across several planlets which do not need to know each other. Each planlet implements a small functionality and can be used in combination with other planlets to achieve an overall goal. The approach also enables separating concerns in terms of management domains: Database experts are able to implement knowledge about databases into planlets without the need to know anything about the PHP applications connecting to databases. All in all, this provides a holistic and collaborative framework for managing applications.

VI. Policy-aware Provisioning of Complex Composite Cloud Applications

In this section, we present the two major contributions of this paper. We propose (i) a concept for defining and processing the Provisioning Policies introduced in Section IV and (ii) show how the Management Framework presented in Section V is extended to support policy-aware provisioning of applications.

The general concept is based on attaching policies to elements in application topologies and elements in planlet fragments which are matched during plan generation to compare *non-functional requirements* of the topology and *non-functional capabilities* offered by planlets. Therefore, we introduce *Policy-aware Management Planlets* that consider policies. This concept allows a fine grained definition of non-functional requirements and capabilities targeted directly to individual elements while preserving the functional description. In contrast to many existing bidirectional policy processing approaches, we use a strict one-way requirement-driven perspective for policies: policies attached to an application topology define requirements whereas policies attached to the fragment of a planlet define the planlet's capabilities. Planlets cannot express non-functional requirements and topologies cannot express capabilities. Thus, the planlet's policies may be ignored if they are not required.

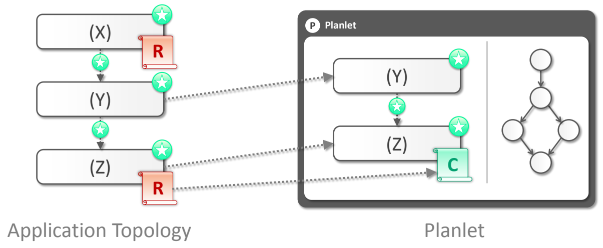

Figure 6. General concept of the presented approach.

Figure 6 explains the concept visually: On the left, it depicts an application topology consisting of three components connected by hostedOn relations that have to be provisioned. The node of type X and Z have a policy attached, which defines the non-functional security requirements that have to be fulfilled during the provisioning. On the right, there is a Policy-aware Management Planlet that provisions nodes of type Z and Y connected by a hostedOn relation. The policy attached to the node of type Z expresses the non-functional capabilities that may be provided by the planlet for that provisioning. During plan generation, the policies are compared and checked for compatibility. If the planlet fulfills all policies attached to elements in the topology that are covered by its topology fragment, the planlet is applicable. In the next section, we introduce the technical structure of Provisioning Policies.

A. Structure and Properties of Provisioning Policies

In this section, we introduce the format for Provisioning Policies to bind non-functional security requirements and capabilities to the tasks performed by planlets. A Provisioning Policy has a mandatory unique *id* within the topology it is contained in and an optional *type* defining the semantics of the policy, e. g., a policy of type *Safe Credentials Policy* ensures that username and password of a component to be provisioned are strong enough to resist attacks. The semantics of a policy type have to be well-defined and documented, i. e., application topology modelers and planlet developers must be aware of its meaning and how to use and implement it. There are a lot of existing policy languages in research and industry such as WS-Policy, Ponder, or KAOS [8]. Our approach supports their integration through an optional *content* field and an optional *language* attribute: The content field enables to fill in any policy- or language-specific information whereas the language attribute defines the used language. A processing mode attribute defines how the policy has to be fulfilled, e. g., whether it is sufficient to compare only the types of topology and planlet policy or if the content of the policy needs to be analyzed. That is the reason why the type and language attributes are optional: if only the type need to be compared, the language attribute is unnecessary. This is explained in detail in Section VI-D. The processing mode is used only by policies attached to elements in the topology as only they impose requirements. Each policy has an attribute *optional* that defines if the processing of the policy is mandatory. Topologies may use optional policies to express security requirements which are "nice to have" but not necessarily required. Planlets may use optional policies to enable configuration and reusability by offering additional non-functional capabilities that are fulfilled only if explicitly required. To target policies exactly to the affected management tasks, each policy in the topology may define the management tasks that have to consider the policies. Therefore, Provisioning Policies have a *MustProcess* list that may contain Management Annotations. All planlets implementing one of these tasks must consider the policy. If this list is empty, all planlets must consider the policy. To add exceptions, each policy has a *NeedNotProcess* list containing the Management Annotations that do not have to consider the policy. This concept allows separating concerns and targeting policies exactly to the affected tasks: Only planlets executing the Management Annotations defined in the MustProcess list must consider the policy while all other planlets do not have to care.

Figure 7 shows Data Location Policies attached to the application used in the motivating scenario. It depicts two Data Location Policies which are attached to the virtual machine and database node. Both are defined in the same language and must be processed by a type specific plugin. The difference lies in the MustProcess lists: The virtual machine policy must be considered only for its creation, which is depicted by the Create-Annotation in the MustProcess list. The database policy must be considered only by planlets that handle data, e. g., import or export data. This is expressed by the domain-specific *DataHandling-Annotation* depicted as blue circle with white paper inside. This differentiation makes sense as the policies express requirements on different tasks. As the location of the physical servers the virtual machine is hosted on determines also the geographic location of the database and, thus, of the data itself, the VM has to be located in the region the data has

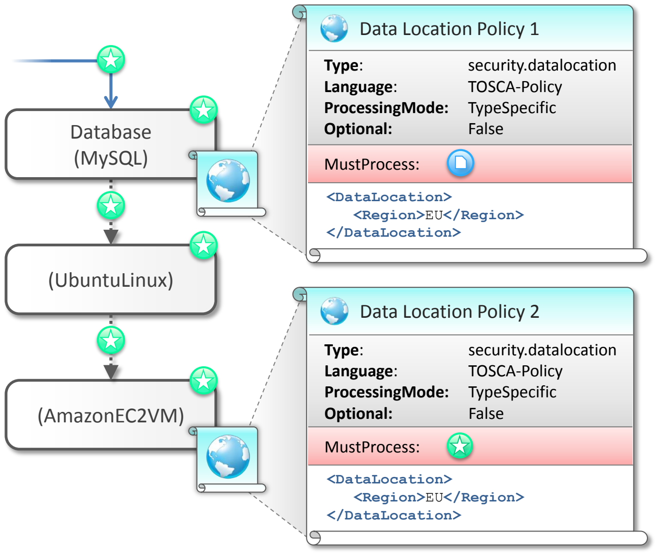

Figure 7. Two Data Location Policies attached to an application topology.

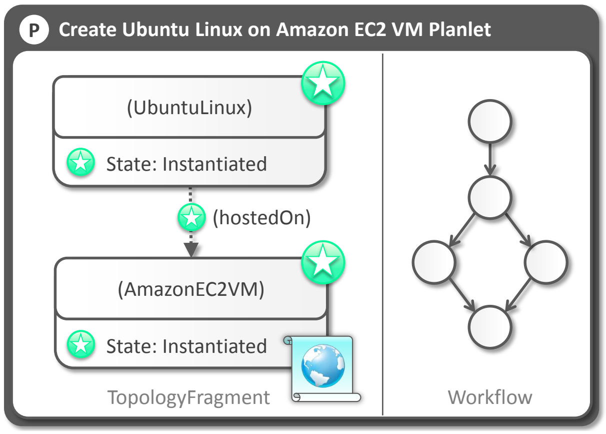

Figure 8. Planlet that instantiates a virtual machine with Linux operating system fulfilling a Data Location Policy.

to remain. Thus, the planlet instantiating the VM must consider this policy, e. g., as shown by the planlet depicted in Figure 8. In contrast, the location of the database needs not to be considered by the planlet installing it on the operating system. Therefore, the planlet shown in Figure 4 can be reused although it does not define any non-functional location information at all. However, handling data, e. g., export data, needs special considerations on the database layer because the data has to remain in the European Union, too. Thus, this concept allows a fine-grained definition of requirements on different levels.

B. Extending Management Annotations

Management Annotations are atomic entities that define either structural or domain-specific self-contained management tasks as explained in Section V-B. This is not sufficient for working with policies as the atomicity allows no abstract definition of those tasks. For example, the Data Location Policy attached to the MySQL database as shown in Figure 7 needs to be processed by all planlets having Management Annotations that deal with data, e. g., planlets exporting data must ensure that they do not store the backup at locations violating the policy. As the complete set of annotations may be unknown in advance, we need a mechanism to classify annotations of certain kinds of tasks. In particular, there are Management Annotations of type Import Data or Export Data as shown in Figure 5 that need to fulfill the Data Location Policy, i. e., data to be imported or exported must not be stored on servers outside the European Union. Listing all these annotations in the MustProcess list of the policy is not sufficient as mentioned before. Thus, we extend the concept by introducing inheritance: The data import and data export Management Annotations inherit all properties from a superclass annotation of type *DataHandling*. For example, the Data Location Policy in Figure 7 specifies that all Management Annotations having this superclass must process the policy. This extension allows defining abstract tasks which can be bound to policies in a generic way. Thus, if the framework processes a policy having this annotation, it ensures that all planlets handling data take this Data Location Policy into account. To achieve flexibility, we also allow multi-inheritance.

C. Plan Generator Extension

The plan generator of the framework tries to find planlets that can be orchestrated to provision the application. During this generation, a set of candidate planlets is calculated for each state and the planner decides which of the candidate planlets is applied next—as introduced in Section V-A. This calculation is based on compatibility: A planlet is applicable if each element in the planlet's fragment can be mapped to a compatible element in the topology. This means, that all preconditions of the planlet are fulfilled and that the management tasks which are implemented by the planlet and expressed in the form of Management Annotations are also specified in the topology. Details about this compatibility check can be found in [4]. We extend this compatibility check by taking policies into account: For each element in the topology that has an attached policy, the policy needs to be processed as defined by the processing mode attribute. How to deal with this processing mode attribute during matchmaking is explained in the next section.

D. Policy Processing Modes and Matchmaking

Each policy specifies a processing mode that defines how the policy has to be checked during the matchmaking of topology and planlets. We introduce three processing modes: (i) Type Equality, (ii) Language Specific, (iii) Type Specific. The mode defines the *minimum criterion* that must be met to fulfill the policy. Thus, the modes are ordered: from weak (Type Equality) to strong (Type Specific). Every stronger criterion outvotes the weaker criteria, i. e., if a policy defines processing mode *Type Equality*, which cannot be fulfilled by any planlet but there is a planlet fulfilling the *Language Specific* processing mode, the policy is fulfilled. Thus, if a criterion cannot be met, the system tries the next stronger criterion automatically. If it does not matter which criterion has to be met, this is equal to set processing mode to *Type Equality*.

1) Type Equality: This processing mode defines that only the types of two policies must be equal to fulfill the policy attached to an element in the topology. That means, that for each policy attached to an element in the topology there must be a policy of the same type attached to the corresponding element in the planlet's fragment. This processing mode is sufficient for policies that can express all their requirements and needs by a well-defined keyword used as type, for example, a *No Connection To Internet Policy* attached to a virtual machine node is expressive enough to define the requirement.

2) Language Specific: Language specific processing means that the policy must be processed by a dedicated framework plugin that is responsible for the used language. For example, if a policy is written in WS-Policy, there must be a corresponding WS-Policy plugin. All the language-specific logic is implemented by the plugins themselves, i. e., in the case of WS-Policy, operations such as intersection or normalization are up to the language plugin. The language plugin gets a reference to the policy to be checked, the whole topology, the candidate planlet's fragment, and a mapping of elements as input. The mapping defines which elements in the topology would correspond to which elements in the planlet's fragment if the policy can be fulfilled by the planlet. The plugins are free to interpret their policy language in any way. For example, if a certain language defines a key-value format for defining policy requirements, the plugin is allowed to compare these requirements with properties of the corresponding fragment node. If requirements and properties are compatible, the policy is fulfilled. Thus, there is no explicit need that a matching policy exists in the fragment at all.

3) Type Specific: This processing mode is the most specific one and bound to policy language and policy type, i. e., if there is a policy of type *Data Location Policy* written in WS-Policy, there must be a plugin registered for exactly that combination. Otherwise, the policy cannot be fulfilled. If such a plugin exists, the processing is equal to language specific: The plugin gets the same kind of information as input and decides if the planlet fulfills the policy's requirements. This processing mode enables a very specific processing of policies as the mode is bound to the policy type directly. For example, if a Data Location Policy with region "EU" (cf. Figure 7) is attached to a MySQL node that should be created and there are no planlets available in the system that have a compatible policy attached, the plugin may analyze the stack the MySQL database shall be hosted on and recognizes that the virtual machine below runs on Amazon's EC2 with region-property set to "EU". In this case, the policy would be fulfilled by the simple MySQL-Creation Planlet shown in Figure 4 that does not know anything about policies at all. This kind of processing enables complex logic which can be only known by a very specific type plugin.

E. Language and Type plugins

The processing mode attribute of a policy decides if a language or type specific plugin has to assess whether the policy can be fulfilled by a candidate planlet or not. Plugins may need to pass information about the matchmaking to the candidate planlet if it fulfills the policy's requirement, e. g., to configure it. Therefore, each plugin may return an XML-document and a list containing the policy ids of the planlet which have to be fulfilled as result that is passed to the planlet via its input message from the calling provisioning plan. This enables configuring if optional policies provided by the planlet have to be fulfilled, for example. This document is also linked with the id of the fragment's policy. Thus, the planlet is able to retrieve the policy language- or type-specific information.

F. Framework Architecture

In this section, we describe how the presented approach is implemented in the used Management Framework. Figure 9 shows the simplified architecture of the framework with the new integrated policy extension (gray background).

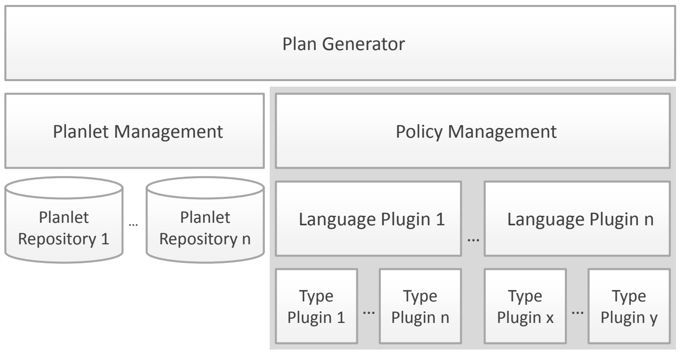

Figure 9. Extended framework architecture.

The basic architecture of the Management Framework substantially consists of a *Plan Generator* that uses a *Planlet Management* to retrieve the planlets and their descriptions stored in *Planlet Repositories*. The Plan Generator has a planlet orchestrator inside, which is responsible for scheduling planlets in the right order. We extend this orchestrator by the integration of a new component called *Policy Management* that is responsible for policy matchmaking and invoking *Language Plugins* and *Type Plugins*. Each planlet that is analyzed by the orchestrator gets now additionally checked if it fulfills the attached policies by a simple call to this new API. The integration is straight forward as the basic architecture of the Management Framework was built in a modular way.

G. Implementing Policy-aware Planlets—Lessons learned

In this section, we describe our experiences from the implementation of policy-aware planlets. A planlet providing additional non-functional capabilities expressed in the form of attached policies on elements contained in its fragment has to ensure that the semantics of the policies are only fulfilled if explicitly needed. This is important as the policy matchmaking is directed: only policies of the application topology are considered by the plan generator, not the policies of the planlet. Thus, if a planlet provides an extending policy, e. g., a Frequent Data Backup Policy, which exports data frequently from a database, this additional functionality should be installed only if needed. If the planlet is able to offer both modes, with and without fulfilling the policy, the planlet should declare this policy as optional. Therefore, the planlets get the mapping of elements in the topology to the elements in its fragment as input. Based on this mapping, the planlet is capable of recognizing if a policy is optional.

In many cases, the extension of planlets to policy-aware planlets is possible by only adding additional activities to the original planlet workflow—especially for Guarding and Extending Policies: The Frequent Data Backup Policy and the Safe Credentials Policy can be implemented by adding activities that install the additional software or check the chosen credentials. The actual process needs not to be modified. In contrast to this, Configuring Policies often need to adapt the original process: for example, the Data Location Policy may influence the provisioning of a virtual machine. Thus, the activity that creates this VM must be modified.

VII. Evaluation

In this section, we evaluate the presented approach. We prove the (i) feasibility, (ii) economics, (iii) analyze performance and complexity, (iv) describe our prototypical implementation, and describe (v) how the approach can be extended.

A. Feasibility

To prove the approach's feasibility, we evaluated the framework in terms of the three kinds of Provisioning Policies discussed in Section IV. We implemented planlets fulfilling the policy examples which were used throughout the paper. To prove the feasibility of Configuring Policies, we implemented the planlet described in Figure 8 that creates a virtual machine with an Ubuntu Linux operating system on Amazon EC2 complying with a Data Location Policy which defines that all data must remain in the European Union. The planlet extracts the region from the policy and uses this value as *region-attribute* for creating the VM using the Amazon Web Services API. This configures the provisioning in a way that the VM is hosted on physical servers located in states of the European Union. To prove the feasibility of Guarding Policies, we defined a Safe Credentials Policy attached to an Apache PHP Web server to ensure that username and password are strong enough. We implemented a planlet that provisions this Web server on Windows 7 complying with this policy by generating a safe password on its own. To prove the feasibility of Extending Policies, we implemented a Frequent Data Backup Policy which is attached to a MySQL database node. The corresponding planlet executes an additional bash script via cron job that frequently backups the data as MySQL dump to an external storage. This script execution may be seen as additional node hosted on the operating system. Thus, it extends the application structurally in order to fulfill a non-functional requirement. For all policy definitions, we used the properties-based policy language shown in the TOSCA specification [9].

The approach also enables defining complex non-functional security requirements that occur in real enterprise systems. This is enabled by the individual content field of Provisioning Policies. The field allows specifying any information about the policy language- or type, e. g., complex system configuration options and tuning parameters. As planlets that match such a policy are built to process exactly the information stored in the content field, the tight coupling of policies to the planlets processing them enables the implementation of any policy language and type. Thus, the corresponding planlets may deal with any individual policy-specific semantics or syntax. For example, if a security policy of type X specifies a set of complex system configuration files that must be taken into account during the provisioning of a certain component, the planlets complying with this policy type X expect these files and know how to process them. This enables to integrate expert knowledge about individual domains through defining own policy types and the corresponding planlets that deal with them.

B. Economics

The economic goal of our approach is to lower operating cost of provisioning. It is obvious that automating IT operations in order to reduce manual effort leads to a cost reduction in many cases. However, Brown and Hellerstein [10] analyzed the automation of operational processes and how this influences costs. They found out that three issues must be considered which counteract this reduction by causing additional effort: (i) Deploying and maintaining the automation environment, (ii) structured inputs must be created to use automation infrastructures, and (iii) potential errors in automated processes must be detected and recovered which is considerably more complicated than for manual processes. The presented approach tackles these issues. Planlets are reusable building blocks for the generation of provisioning plans. They are developed by expert users of several domains and provided to communities similar to DevOps. Therefore, free accessible planlet repositories enable continuous maintenance without the need for individual effort. Of course, maintaining local management infrastructure and the development of own custom planlets for special tasks causes additional effort, but this is a general problem that cannot be solved generically. The second issue of upfront-costs for creating structured input is reduced to a minimum as we provide tools for the modeling of application topologies and policies. The third issue of occurring errors is tackled implicitly by the workflow technology: every planlet defines its own error and compensation handling. Thus, errors are handled either locally by planlets themselves or by the generated provisioning plan, which triggers the compensation of all executed planlets to undo all operations for errors that cannot be handled.

C. Performance and Complexity

The performance of the approach is of vital importance as the generation of provisioning plans must be possible within a few seconds to obtain Cloud properties such as scalability or on-demand self-service. The employed Management Framework presented in Section V uses a partial order planning algorithm [11] for the generation of Management Plans [4]. As described by Bylander [12], the complexity of planning varies from polynomial to PSPACE-complete, depending on the preconditions, effects, and goals. The Management Framework tackles this issue by introducing restrictions on the design of planlets: it is forbidden to use multiple planlets providing the same or overlapping functionality but having different effects. For example, it is forbidden to have two planlets installing a MySQL database on an Ubuntu Linux operating system that differ in the set properties. This reduces the number of nondeterministic choices that have to be made during plan generation in terms of selecting planlets: if a Management Annotation can be processed by multiple different planlets, it does not matter which one is chosen because they have the same effects and may differ only in their preconditions. Thus, planlets having preconditions on the effects of these planlets are not affected by the plan generator's choice. This decreases the complexity to polynomial time [11]. Extending the framework by policies must follow this restriction: If a policy-aware planlet implements a functionality that is provided already by an existing planlet, the existing planlet has to be merged with the new policy-aware planlet. The new planlet must also support the original functionality which is trivial in most cases as policies only deal with non-functional requirements but do not change the original functionality. The only difference is additional effort as calling plugins might be necessary. In worst case, each policy in a topology must be processed by a plugin. As the number *n* of used policies within a topology is constant, the extension has no influence on the complexity. We evaluated the performance of the policy-aware framework with the following setting: Based on a set of 25 fully-implemented real planlets, we developed different application topologies of different size - from small (5 nodes, 4 hostedOn relations) to large (100 nodes, more than 100 relations). We added 1000 randomly generated test planlets without workflow implementation to simulate a real environment. Thus, the plan generator has to consider 1025 planlets during plan generation. The generation of provisioning plans for all test application topologies (small to large) required only a few milliseconds. Serialization to BPEL and deployment on the used process engine (WSO2 Business Process Server 2.1.2) increased this by approx. 7 seconds. Thus, the time for generating plans is negligible in comparison to the time-consuming serialization to BPEL and the deployment.

D. Prototype

To validate the concept technically, we implemented the approach on the basis of the Management Framework prototype presented in Section V. This prototype is implemented in Java and uses OSGi in order to provide a flexible and dynamic plugin system. Planlets are implemented in the Business Process Execution Language (BPEL) whereas topologies and planlet fragments are implemented using an internal data model similar to the structure of TOSCA [9]. We extended this topology model with the possibility to attach policies to nodes and relations. The policies are provided as XML files following the simple properties-based policy language used in this paper. We use declarative OSGi services to build the plugin system for language- and type-plugins, as described in Section VI-F. To prove the technical feasibility of the conceptually evaluated policies described in Section VI we implemented these policies and others by extending already existing planlets. The successful implementation of this prototype extension proves that the presented approach works in reality.

E. Extensibility

As there are many different existing policy types and languages, the presented approach must support extensibility. The used Management Framework (cf. Section V) supports creating own custom planlets that implement Management Annotations for any conceivable management task. As the approach presented in this paper relies on this concept, it is possible to implement new policy types the same way. The plugin-based architecture for language and type plugins complements the planlet-based policy extension: If a new policy type needs a dedicated type plugin for advanced processing, the architecture allows installing new plugins that handle these types. In addition, the architecture enables the integration of any existing policy language as well as the development of own languages. We successfully validated this criterion based on the integration of WS-Policy. As WS-Policy has its own type system in the form of assertions, the type attribute of the Provisioning Policy is not needed. In addition, the created plugin retrieves the information about domain-specific processing of assertions by extracting the policy-specific content field which defines this kind of information.

VIII. Related Work

There are several works focusing on the automated provisioning of Cloud applications. In this section, we describe the most related ones and compare them to our approach. The work of Eilam et al. [13] focuses on deployment of applications by orchestrating low-level operation logic similarly to planlets by so-called *automation signatures*. El Maghraoui et al. [14] present a similar approach that orchestrates provisioning operations provided by existing provisioning platforms and is, thus, much more restricted than using planlets, which are able to integrate any technology and system. Both works do not consider non-functional requirements at all—especially not in the form of explicitly attached policies, which are able to define explicitly the tasks that must consider the policy. In contrast to both works, planlets and Management Annotations allow the application developer to bind policies to abstract tasks which are executed during provisioning. Thus, Policy-aware Management Planlets introduce an additional layer of abstraction in terms of defining and processing non-functional requirements.

Mietzner and Leymann [15] present an architecture for a generic provisioning infrastructure based on Web services and workflow technology that can be used by application providers to define provisioning flows for applications. These flows invoke so-called provisioning services that provision a certain component or resource. Policies can be used by the provisioning flow to select the specified provisioning services based on non-functional properties of the resource to be provisioned, e. g., availability of the provisioned resource. The general idea of implementing provisioning services is similar to planlets. However, planlets allow a much more fine grained differentiation between provisioning tasks, e. g., the provisioning of a database and the following initial data import are done by different planlets. Thus, policies can be bound more specifically to tasks and allow, therefore, a more precise definition of non-functional security requirements.

The Composite Application Framework (Cafe) [16] is an approach to describe configurable composite service-oriented Cloud applications that can be automatically provisioned across different providers. It allows expressing non-functional requirements in WS-Policy that can be matched to properties of resources in an environment. However, these policies are restricted to the selection of services and lack mechanisms to configure, guard, or extend application provisioning as enabled by our approach.

The CHAMPS System [17] focuses on Change Management to modify IT systems and resources by processing so-called *Requests For Change (RFC)* such as installation, upgrade, or configuration requests. After receiving an RFC, the system assesses the impact of the RFC on components and generates a so-called Task Graph which is afterwards used to generate an executable plan. The system can be used for initial provisioning of composite applications, too. Although the system's plan generator considers policies and SLAs, the work does not describe how the executed tasks have to process the artifacts during provisioning.

Kirschnick et al. [18] present a system architecture and framework that enables the provisioning of Cloud applications based on virtual infrastructure whereon the application components get deployed. However, the framework does not support non-functional requirements, is tightly coupled to virtual machines, and lacks integrating any XaaS offerings. Thus, the system is not able to provision Cloud applications that consist of several XaaS offerings complying with non-functional security requirements.

The DevOps community provides tooling to automate the provisioning of Cloud applications. To mention the most important, Chef and Puppet are script-based frameworks used to automate the installation and configuration of software artifacts in distributed systems. The DevOps community also provides additional tooling such as Marionette Collective, ControlTier, and Capistrano used to improve the orchestration capabilities on a higher level. The frameworks are extensible in terms of adding new installation, configuration, and—in general—provisioning functionalities. This enables to integrate provisioning logic that considers non-functional requirements. However, all these frameworks focus on a deep technical level of provisioning and do not provide a means to express and integrate non-functional security requirements on such a high level as enabled by Management Annotations and planlets. The reusability in terms of provisioning different applications, the interoperability between script-based and non-script-based technologies as needed to build complex composite Cloud applications, and the holistic integration of different policy languages in order to achieve broad acceptance is not supported yet.

The Topology and Orchestration Specification for Cloud Applications (TOSCA) is an upcoming standard to describe composite Cloud applications and their management [9]. It tackles current challenges in Cloud computing such as portability and interoperability of Cloud applications, prevention of vendor lock-in, and the automated management of applications including provisioning [19]. TOSCA provides a XML-based format for describing Cloud applications as application topologies and enables the management of applications through Management Plans that capture management knowledge in an executable way. TOSCA provides a similar mechanism to attach policies to nodes and relations in topologies but, however, only provides a means to integrate policies into the topology but lacks a detailed description of their processing.

IX. Conclusion and Future Work

In this paper, we presented an approach that enables fully automated provisioning of complex composite Cloud applications in compliance with non-functional security requirements. Based on this approach, we extended an already existing Management Framework to support policy-aware provisioning. We introduced a new format for Provisioning Policies that are considered by Policy-aware Management Planlets during provisioning. In addition, the extended framework allows Cloud providers as well as application developers to implement their own policy-aware provisioning logic in a flexible and reusable manner independently from individual applications. The paper evaluates the presented approach in terms of performance, feasibility, and extensibility. In addition, we implemented a prototype that serves as a proof of concept of the presented conceptual work. In future work, we will extend this concept by a policy-aware preprocessing of topologies in order to increase the reusability of planlets to a higher level. Afterwards, we plan to apply the presented concept of policy-aware planlets also to runtime management of applications and extend the approach to support policies of other domains such as Green IT.

Acknowledgment

This work was partially funded by the BMWi project CloudCycle (01MD11023).

References

[1] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in USITS, June 2003.

[2] F. Leymann, "Cloud Computing: The Next Revolution in IT," in Proc. 52th Photogrammetric Week, September 2009, pp. 3–12.

[3] M. Armbrust et al., "Above the Clouds: A Berkeley View of Cloud Computing," University of California, Berkeley, Tech. Rep., 2009.

[4] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, "Pattern-based runtime management of composite cloud applications," in CLOSER, Mai 2013, pp. 475–482.

[5] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and D. Schumm, "Vino4TOSCA: A visual notation for application topologies based on tosca," in CoopIS, September 2012, pp. 416–424.

[6] A. Keller and R. Badonnel, "Automating the provisioning of application services with the BPEL4WS workflow language," in DSOM, November 2004, pp. 15–27.

[7] R. Wies, "Using a classification of management policies for policy specification and policy transformation," in IFIP/IEEE IM, June 1995, pp. 44–56.

[8] W. Han and C. Lei, "Survey paper: A survey on policy languages in network and security management," Comput. Netw., vol. 56, no. 1, January 2012, pp. 477–489.

[9] OASIS, Topology and Orchestration Specification for Cloud Applications Version 1.0, May 2013. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs02/TOSCA-v1.0-cs02.pdf

[10] A. B. Brown and J. L. Hellerstein, "Reducing the cost of it operations: is automation always the answer?" in HOTOS, June 2005, pp. 12–12.

[11] D. S. Weld, "An introduction to least commitment planning," AI Magazine, vol. 15, no. 4, Winter 1994, pp. 27–61.

[12] T. Bylander, "Complexity results for planning," in IJCAI, August 1991, pp. 274–279.

[13] T. Eilam, M. Elder, A. Konstantinou, and E. Snible, "Pattern-based composite application deployment," in IFIP/IEEE International Symposium on Integrated Network Management, May 2011, pp. 217–224.

[14] K. El Maghraoui, A. Meghranjani, T. Eilam, M. Kalantar, and A. V. Konstantinou, "Model driven provisioning: bridging the gap between declarative object models and procedural provisioning tools," in Middleware, November 2006, pp. 404–423.

[15] R. Mietzner and F. Leymann, "Towards provisioning the cloud: On the usage of multi-granularity flows and services to realize a unified provisioning infrastructure for saas applications," in SERVICES, July 2008, pp. 3–10.

[16] R. Mietzner, T. Unger, and F. Leymann, "Cafe: A Generic Configurable Customizable Composite Cloud Application Framework," in CoopIS, November 2009, pp. 357–364.

[17] A. Keller, J. L. Hellerstein, J. L. Wolf, K. L. Wu, and V. Krishnan, "The CHAMPS system: change management with planning and scheduling." in NOMS, April 2004, pp. 395–408.

[18] J. Kirschnick, J. M. A. Calero, L. Wilcock, and N. Edwards, "Toward an architecture for the automated provisioning of cloud services," Comm. Mag., vol. 48, no. 12, December 2010, pp. 124–131.

[19] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, "Portable Cloud Services Using TOSCA," IEEE Internet Computing, vol. 16, no. 03, May 2012, pp. 80–85.