Institute of Architecture of Application Systems

Policy-Aware Provisioning and Management of Cloud Applications

Uwe Breitenbücher, Tobias Binz, Christoph Fehling, Oliver Kopp, Frank Leymann, Matthias Wieland

Institute of Architecture of Application Systems,
University of Stuttgart, Germany,
lastname@iaas.uni-stuttgart.de

BibTeX:

```
@article{Breitenbuecher2014_SEC,
  author    = {Uwe Breitenb{\"u}cher and Tobias Binz and Christoph Fehling and
               Oliver Kopp and Frank Leymann and Matthias Wieland},
  title     = {Policy-Aware Provisioning and Management of Cloud Applications},
  journal   = {International Journal On Advances in Security},
  number    = {1\&2},
  volume    = {7},
  year      = {2014},
  pages     = {15-36},
  publisher = {Xpert Publishing Services (XPS)}
}
```

Original Publication Reference:
http://thinkmind.org/index.php?view=article&articleid=sec_v7_n12_2014_2
http://www.iariajournals.org/security/

Universität Stuttgart
Germany

Policy-Aware Provisioning and Management of Cloud Applications

Uwe Breitenbücher[1], Tobias Binz[1], Christoph Fehling[1], Oliver Kopp[2], Frank Leymann[1], and Matthias Wieland[2]
[1]Institute of Architecture of Application Systems, University of Stuttgart, Stuttgart, Germany
[2]Institute of Parallel and Distributed Systems, University of Stuttgart, Stuttgart, Germany
{breitenbuecher, lastname}@informatik.uni-stuttgart.de

***Abstract*—The automated provisioning and management of composite Cloud applications is a major issue and of vital importance in Cloud Computing. It is key to enable properties such as elasticity and pay-per-use. The functional aspects of provisioning and management such as instantiating virtual machines or updating software components are covered by various technologies on different technical levels. However, 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 addition, due to their heterogeneity, the integration of these technologies in order to benefit from their individual strengths is a major problem—especially if non-functional aspects have to be considered and integrated, too. In this article, we present a concept that enables executing management tasks using different heterogeneous management technologies in compliance with non-functional security requirements specified by policies. We extend the Management Planlet Framework by a prototypical implementation of the concept and evaluate the approach by several case studies.**

Keywords–Cloud Computing; Application Management; Provisioning; Security; Policies.

I. INTRODUCTION

The steadily increasing use of Information Technology (IT) in enterprises leads to a higher management effort in terms of application development, deployment, and operation. IT management becomes a serious challenge when additional technologies increase the complexity of management—especially if non-functional security requirements must be considered, too [1]. Since manual operator errors account for the largest fraction of failures, automating IT management becomes of vital importance [2]. These issues have been tackled by outsourcing IT to external providers and automating the management of IT, which are both enabled by Cloud Computing [3]. Cloud Computing reuses well-known concepts and makes them easily accessible. The modular architectures that are the consequence of using Cloud services enable to benefit from Cloud properties such as elasticity without the need to have technical insight [4]. 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 that 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 security requirements of different domains with focus on heterogeneous technologies have to be fulfilled. Application management additionally increases the complexity.

At the *International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2013)*, we presented a first step to tackle these issues by introducing a policy-aware *provisioning* concept [1] that enables defining non-functional security requirements on the execution of provisioning tasks using policies. We realized the concept by extending the Management Planlet Framework [5][6][7][8]. In this article, we continue this work and show how the presented policy-concept can be used to specify non-functional security requirements also on the *management* of applications. We, therefore, illustrate how the concept of *Policy-Aware Management Planlets* [1] can be used to enforce non-functional requirements on management tasks in a reusable way independently from individual applications and how they are enforced during execution. The article shows how policy-aware management tasks can be specified either (i) manually or (ii) automatically using the concept of *Automated Management Patterns* [5][8] and how the automated application of management patterns deals with declared policies. We show (i) that attaching *Management Annotation Policies* on components and relations of application topologies provides a fine grained means to specify non-functional security requirements to be fulfilled directly by the affected management tasks and (ii) how policies implemented in different policy languages can be processed in a uniform manner. In addition, we show how heterogeneous management technologies can be integrated using the presented approach in consideration of security policies. We realize the presented concepts by a policy-aware Management Planlet Framework extension that enables application developers, administrators, and Cloud providers to specify security requirements on the provisioning and management of applications without the need to have the deep technical management knowledge required in other approaches. In addition, the framework enables security experts of different domains to work together in a collaborative way. We evaluate the management approach through several case studies that are conducted throughout the paper and in terms of performance, complexity, economics, feasibility, extensibility, and a prototypical implementation. To provide an overview, we first explain the concepts of policy-aware provisioning we have presented in Breitenbücher et al. [1] at the SECURWARE 2013 conference and show in Section VII how they are used to enable policy-aware management.

In Section II, we explain fundamentals and motivate our approach in Section III. Section IV describes the framework that is extended by our approach. In Section V, we present Policy-Aware Management Planlets that are used in Section VI for policy-aware provisioning and in Section VII for policy-ware management of applications. In Section VIII, we present the architecture of the extended framework. Section IX evaluates the approach and Section X reviews related work. We conclude this paper and give an outlook on future work in Section XI.

II. FUNDAMENTALS

In this section, we explain four fundamental concepts that are required to understand the policy-aware provisioning and management approach presented in this paper. These are (i) Application Topologies, (ii) Enterprise Topology Graphs, (iii) Management Plans, and (iv) Management Policies.

A. Application Topology

An application topology is a directed, possibly cyclic graph describing the structure of an application. It defines nodes, which represent the different components of an application such as Web Servers, virtual machines, or Java applications, and the relations among them, which are the edges between the nodes. Nodes and relations of a topology are called *topology elements*. Each topology element has a certain type that defines its semantics and defines a set of *static* properties that describe details about the element, e. g., the configuration of a Web Server. Application topologies can be used to describe the provisioning of applications *declaratively*: they define the nodes and relations to be provisioned including all configuration properties, but not how to actually execute the provisioning.

Figure 1 shows an example that describes a LAMP-based (Linux, Apache, MySQL, PHP) application topology. To render application topologies graphically, we use the visual notation *Vino4TOSCA* [9] in this paper to depict all kinds of topology models. Following this notation, nodes are depicted as rounded rectangles, relations as arrows, and the type of a topology element is enclosed by parentheses. The application's infrastructure is provided by Amazon's public Cloud [10] in the form of two virtual machines of type "Ubuntu12.04VM" that are hosted on nodes of type "AmazonEC2". Both AmazonEC2 nodes provide login information in the form of properties, both virtual machines specify the desired configuration in terms of CPU and RAM. Such information are used to provision the corresponding elements. On the left virtual machine, a PHP runtime of type "ApachePHPServer2.2" is installed that hosts the business logic implemented as "PHP" application. The Web Server node defines its desired configuration in terms of login credentials and the HTTP port, under which the PHP application shall be reachable. The PHP node specifies the files to be deployed in the form of a referenced ZIP file that contains the business logic. This application connects to a database of type "MySQLDB" which is hosted on the MySQL Database Management System node of type "MySQLDBMS" that runs on the Ubuntu12.04VM node of the right stack. Similarly to the Web Server, the MySQLDBMS node defines the port under which the database shall be accessible. To ease accessing the application from the outside, an internet domain points to the application's PHP frontend. Therefore, a node of type "Domain" is connected via a "refersTo" relation to the application's PHP node. Of course, this topology is simplified: especially on the middleware layer (i. e., Apache Web Server and MySQL Database Management System), typically more properties are used to configure the respective component in more detail. Also in the following figures, we omitted most of these properties to simplify the diagrams. Our approach employs application topologies to describe the structure of applications to be managed and to attach policies to the affected elements.

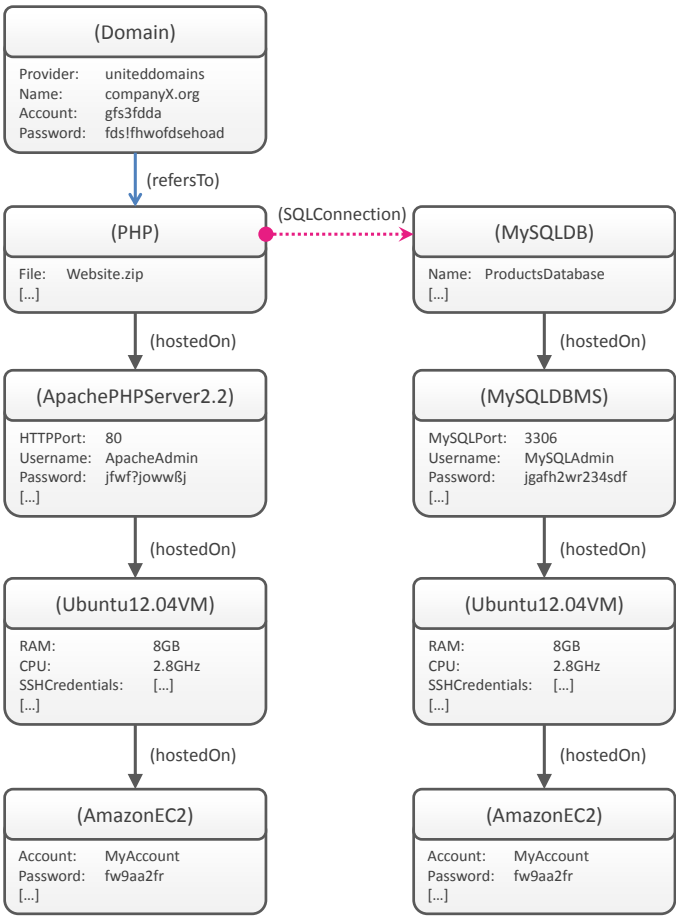

Figure 1. Example of a LAMP-based application topology.

B. Enterprise Topology Graph

Enterprise Topology Graphs (ETG) [11] are a special kind of application topology. They extend the static properties of application topologies by dynamic properties that provide runtime information about application components and relations such as current CPU load or IP-addresses. Thus, Enterprise Topology Graphs can be used to capture the *current state* of a running application as a fine-grained technical snapshot that formally describes all components and relations including their types, configurations, and runtime information. Enterprise Topology Graphs also capture runtime information in the form of the *lifecycle state* of components and relations, e. g., that a component is currently starting, running, or terminating.

ETGs are used in various domains to adapt [12], analyze [13], manage [5], and optimize application structures, e. g., to improve the ecological sustainability of business processes [14] and to consolidate duplicate components [11]. Enterprise Topology Graphs of running applications can be discovered fully automatically using the ETG *Discovery Framework* [15]. This framework requires only an entry point of the application, e. g., the URL of the application's Web frontend, to discover the whole ETG fully automatically including all software, middleware, and infrastructure components of the application. To render ETGs graphically, we also use Vino4TOSCA.

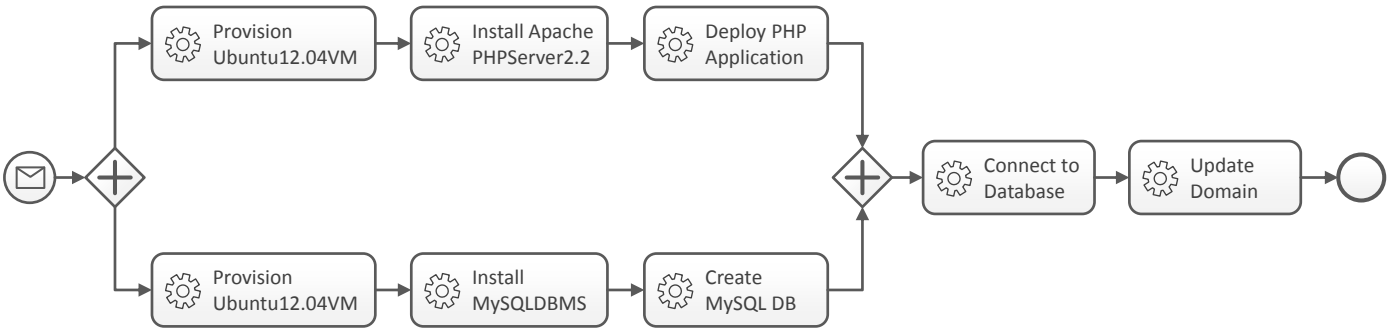

Figure 2. BPMN Management Plan that provisions the LAMP-based application described in Section II-A.

C. Management Plan

Management Plans are executable workflows [16] used to automate the management of applications, e. g., application provisioning, scaling an application, or updating components. They enable a more robust and reliable way to manage applications than manual or script-based management [6] due to features such as recoverability, compensation, and fault handling mechanisms [16]. Typical workflow languages to implement plans are the *Business Process Execution Language (BPEL)* [17] and the *Business Process Model and Notation (BPMN)* [18]. For example, BPEL can be used for application provisioning as presented by Keller et al. [19], BPMN to manage applications based on the TOSCA [20] standard [21].

Figure 2 shows a BPMN Management Plan that provisions the LAMP-based application shown in Figure 1. It consists of service tasks that provision the individual components and gateways that enable processing tasks in parallel: after receiving a start message, the shown plan provisions the two virtual machines on Amazon EC2 in parallel, installs the middleware components, and deploys the PHP application on the installed Web Server. After both parallel paths finished, the PHP application is connected to the MySQL database and the last activity updates the domain to the application's URL. This plan can be executed automatically to provision the application.

Management Plans are often created manually by the application developers [22]. 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 [6]. In addition, if non-functional security requirements must be considered, the complexity of creating Management Plans increases additionally when plans are authored manually. In this paper, we present an approach to generate Management Plans that consider non-functional security requirements expressed by policies.

D. Management Policy

In this section, we introduce *Management Policies*, which are used by our approach to express non-functional security requirements on the provisioning and management of applications. Management Policies are a well-known concept and common in research as well as in industry [23]. They are derived from management goals and employed in systems and network management 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 [24]. A lot of work on policies exists dealing with classifications, methodologies, and applications. To classify and identify the policies covered by our approach, we follow the hierarchy of Wies [24] 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 focuses on the technology oriented management.

Considering policies that define security requirements, it is of vital importance to ensure their strict adherence: management systems must prevent that the security requirements defined by a policy get violated because many types of security 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 due to legal rights, i. e., also the physical servers storing the data must be located in that region, and the data gets distributed over the world through decentralized Cloud servers located in other regions, the policy is violated and it is impossible to undo this violation. Depending on the agreements, such violations often result in high penalties.

In Breitenbücher et al. [1], we employed three kinds of *Provisioning Policies* to specify non-functional requirements on the *provisioning* of applications: (i) Configuring Policy, (ii) Guarding Policy, and (iii) Extending Policy. In this paper, we employ these policy types also for the *management* of applications. Therefore, we call these policy types *Management Policies* in the following. We explain these three kinds briefly to provide the basis for the motivating scenario introduced in the next section. A *Configuring Policy* configures the management of components or relations. For example, a Data Location Policy attached to a virtual machine with "region" value "EU" configures the deployment in a way that the virtual machine is hosted on a server located in the European Union. A *Guarding Policy* guards the management, i. e., it supervises management tasks in terms of specified values or thresholds. For example, a *Secure Password Policy* ensures that the strength of login

credentials, i. e., username and password, is strong enough. An *Extending Policy* extends the management in terms of structure, i. e., it may add new components or relations which are not contained in the original application topology. For example, defining a *Frequent Data Backup Policy* for a database may cause the installation of an additional software component on the underlying operating system that backups specified database tables in a certain time interval to a certain location.

III. Motivating Scenario

In this section, we describe a motivating scenario that is used throughout the paper to explain the presented approach. The basis for the scenario provides the LAMP-based application shown in Figure 1, which was explained in Section II-A. In our scenario, this application is a customer facing Website of a company for which we consider three different management tasks: (i) the initial provisioning of the application, (ii) making a backup of the database, and (iii) updating the employed Apache Web Server to a new version. All management tasks shall be executed in compliance with different non-functional security requirements, which are expressed by Management Policies that are attached to the affected components.

Depending on the importance of an application for a company, there are typically different non-functional security requirements of various types. In our scenario, six Management Policies are attached to components that define non-functional security requirements that must be complied with by the management tasks that are performed on these components during provisioning and further management of this application. Figure 3 shows the enriched application topology model: Data Location Policies are attached to both virtual machines and to the MySQL database, Secure Password Policies are attached to the middleware components, i. e., the Apache Web Server and the MySQL DBMS, as well as to the MySQL database itself. The two Data Location Policies attached to the virtual machines restrict the allowed geographic locations of the virtual machines and define that both must be hosted in the European Union (EU). Thus, the virtual machines must be hosted on a physical server located in one of the EU's states. In contrast to these policies that define requirements on the physical location of components, the Data Location Policy attached to the MySQL database defines that also the data itself must never leave the EU, e. g., if data is exported for backup, also this data copy must remain in the EU. The reason for these requirements may be legal aspects on the location of data that have to be considered by the company when outsourcing this application to the Cloud, e. g., if personal data is stored in the database. The second kind of policies used in this motivating example are the three Secure Password Policies attached to the middleware components and the database, which define that the employed passwords must have a certain strength. This security requirement results from the fact that there are many cases in which unsafe passwords are used by administrators or even the default passwords of middleware components are used. The Secure Password Policy ensures that the chosen passwords are strong enough to resist common attacks. Of course, this is not a complete list of security requirements a company may have on such an application and the scenario provides only a simplified description. The intention is to give a general overview on the kind of non-functional security issues that are tackled in this article.

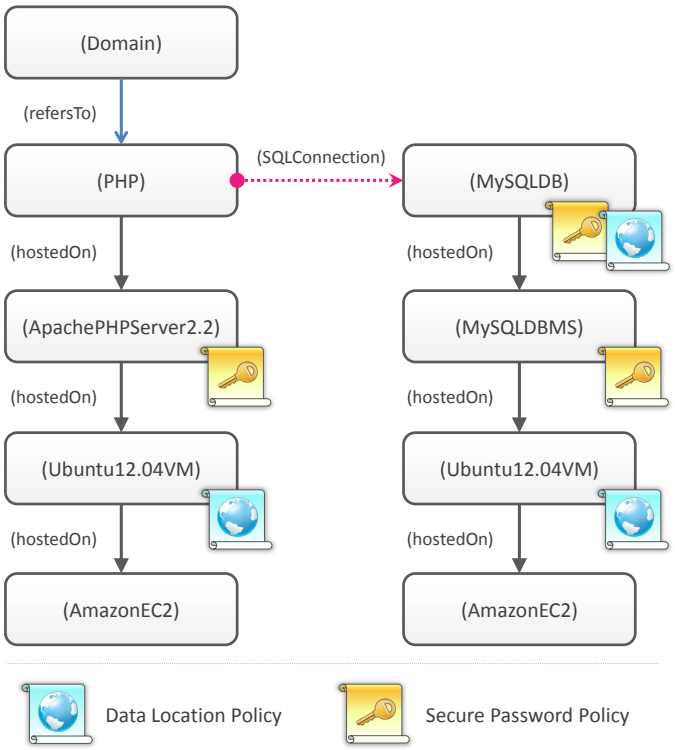

Figure 3. Motivating scenario: LAMP-based application topology with attached Management Policies.

The three management tasks that have to be performed on this application are the (i) initial provisioning of this application in the Amazon Cloud, (ii) making a database backup, and (iii) updating the Apache Web Server from version 2.2 to version 2.4. We now discuss how the attached policies may influence these management tasks. First, during provisioning, all policies attached to middleware components including the MySQL database must be considered. Only the Data Location Policy attached to the MySQL database node, which focuses on data handling, defines a requirement that must be considered only during management, i. e., after the application is provisioned and data shall be exported. The second management task of making a backup of the whole database is such a task that must consider this policy: the location to which the database backup is exported must comply with this policy. In this case, the target location must be a storage located in the European Union. The third management task to be performed is a typical use case that considers security problems of middleware components. In our scenario, the employed Apache Web Server in version 2.2 must be updated to a new version due to critical security issues found in the old version. In addition, due to the vital importance of this application for the company, the application's availability must be ensured, i. e., the update must be executed without application downtime. The challenge of executing this management task is that the Web Server component gets physically replaced by a new version of this component but the attached Management Policies must be fulfilled also by the new installation of the Web Server. Thus, policies that primarily affect the provisioning of components must be ensured also during the execution of such management tasks. This additionally increases the difficulty of complying with security policies when executing management tasks on the application.

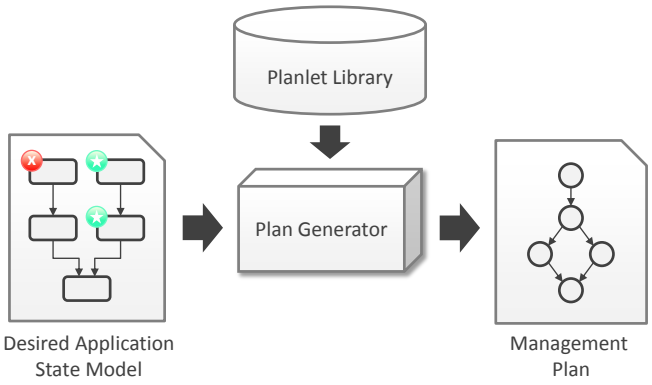

Figure 4. Architecture and concept of the Management Planlet Framework used to generate Provisioning and Management Plans (adapted from [5][7][8]).

IV. Management Planlet Framework

In this section, we explain the *Management Planlet Framework* that gets extended in this article to support the provisioning and management of applications in compliance with non-functional security requirements defined by Management Policies. The framework was presented in former papers [1][5][6][7][8], which describe all conceptual and technical details about the provided management functionalities. We describe the framework briefly in this section to provide all information required to understand the presented approach. We first give a high-level overview on the general concept and explain the details in the following subsections.

A. Conceptual Overview

The main functionality of the Management Planlet Framework is managing applications by generating Management Plans that can be executed fully automatically to perform the desired management tasks on applications. The general concept is shown in Figure 4. The framework provides a language to specify the management tasks to be performed on applications in an *abstract* and *declarative* manner using *Desired Application State Models* (DASM). These models can be transformed fully automatically to executable Management Plans by a *Plan Generator* that orchestrates so-called *Management Planlets*, which implement management logic as executable workflows. The Management Planlet Framework supports the initial provisioning of applications as well as application management. We extended the Management Planlet Framework in Breitenbücher et al. [1] to support security policies on the *provisioning* of applications. In this article, we show how the approach can be used for policy-aware *management* of applications, too.

B. Desired Application State Model

A Desired Application State Model (DASM) is a formal model specifying the desired state in which an application shall be transferred and all management tasks that have to be executed to reach this state. It consists of (i) the application's ETG, which describes the current structure and runtime information of the application, and (ii) so-called *Management Annotations*, which are attached to nodes and relations of the ETG to specify the management tasks to be executed on the respective running node or relation. Management Annotations express *low-level* management tasks such as creating components,

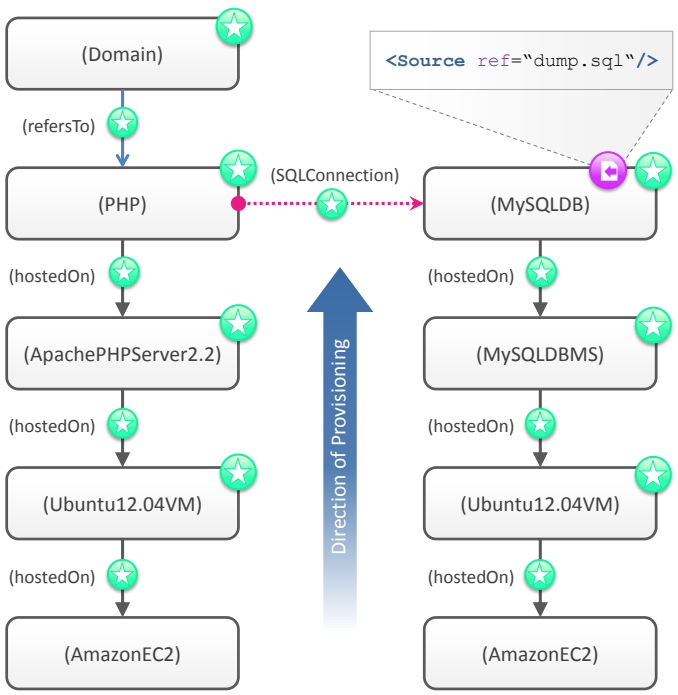

Figure 5. Desired Application State Model that describes the provisioning of the LAMP-based application described in the motivating scenario.

establishing relations, updating components, or importing data. They can be combined to model higher-level management task such as scaling an application, which typically requires executing multiple low-level tasks on different components. A Management Annotation specifies only the *abstract semantics* of a certain management task, e. g., that the corresponding node or relation shall be created, but not the technical realization of its execution. Therefore, each Management Annotation has a certain *type* that defines the represented task's semantics.

Management Annotations are subdivided into two disjoint classes: (i) *Structural Management Annotations* and (ii) *Domain-Specific Management Annotations*. The first class consists of two annotations that structurally change the application in terms of creating or destroying nodes or relations. Thus, there is a (i.a) *Create-Management Annotation* and a (i.b) *Destroy-Management Annotation*. Figure 5 shows a DASM that describes the provisioning of the motivating scenario. We depict all Management Annotations in DASMs as coloured circles, e. g., the green circle represents the Create-Annotation. Therefore, the topology elements to be provisioned are annotated with Create-Annotations. These annotations tell the system that the corresponding nodes and relations shall be created. The second class contains Domain-Specific Management Annotations, which express special management tasks for particular elements. For example, Figure 5 shows an *ImportData-Management Annotation* (purple circle with paper inside) attached to the MySQLDB node that defines that data shall be imported into the database. Domain-Specific Management Annotations typically provide additional annotation-specific information. For example, the ImportData-Annotation also specifies the data to be imported by a reference to the corresponding SQL dump. Both kinds of annotations may additionally declare that they must be executed before, after, or concurrently with another annotation.

In contrast to Management Plans that define all technical details required for their automatic execution, a DASM describes the tasks to be performed only declaratively, i. e., only *what* has to be done is described, but not *how*—all technical details about the execution of the specified management tasks are missing. Only the task's abstract semantics are defined by the declared Management Annotations. As a result, DASMs are not executable. Therefore, they are transformed by the framework's Plan Generator into executable Management Plans by orchestrating so-called Management Planlets, which implement the management logic required to execute the abstract management tasks specified by the Management Annotations in DASMs. In the next section, we explain Management Planlets in detail.

C. Management Planlets

Based on non-executable DASMs that declare the management tasks to be executed only declaratively, the framework's Plan Generator transforms DASMs automatically into executable Management Plans. These generated plans execute the Management Annotations declared in the DASM and bring the application from the current state into the desired state. To generate Management Plans out of DASMs, the Plan Generator orchestrates so-called *Management Planlets*. Planlets are small, executable workflows that provide the management logic required to execute particular Management Annotations on a certain combination of nodes and relations. They are used as reusable management building blocks implementing low-level management tasks such as creating a virtual machine on Amazon EC2, installing an Apache Web Server on an Ubuntu operating system, or exporting data from a MySQL database. Management Planlets are developed by technology experts of different domains and can be orchestrated to higher-level Management Plans, which implement more complex management tasks such as the provisioning of a whole application, scaling an application, or updating application components.

Management Planlets consist of two parts: (i) Annotated Topology Fragment and (ii) executable workflow. The *Annotated Topology Fragment* formally describes the planlet's functionality by a small application topology that is annotated with the Management Annotations the planlet executes on the combination of nodes and relations defined by this topology. It defines (a) the planlet's *effects* in the form of Management Annotations that are declared on nodes or relations and (b) *preconditions* in the form of nodes, relations, and properties that must be fulfilled to execute the planlet. The workflow implements the execution of the annotations declared on the respective elements.

Figure 6 shows a planlet that executes a Create-Annotation on a node of type "Ubuntu12.04VM" and a Create-Annotation on the associated relation of type "hostedOn", which connects to an existing node of type "AmazonEC2". Thus, this planlet creates a new Ubuntu virtual machine on Amazon's public Cloud offering EC2. Planlets often need to express preconditions that must be fulfilled to execute the planlet. Preconditions are defined by (i) all elements in a planlet's topology fragment and (ii) all properties of these elements that have no Create-Annotation attached. For example, the shown planlet requires a node of type "AmazonEC2" that provides the properties "Account" and "Password". The property value "⋆" denotes wildcard: the corresponding property must be set to any value. In this case, the planlet reads these properties

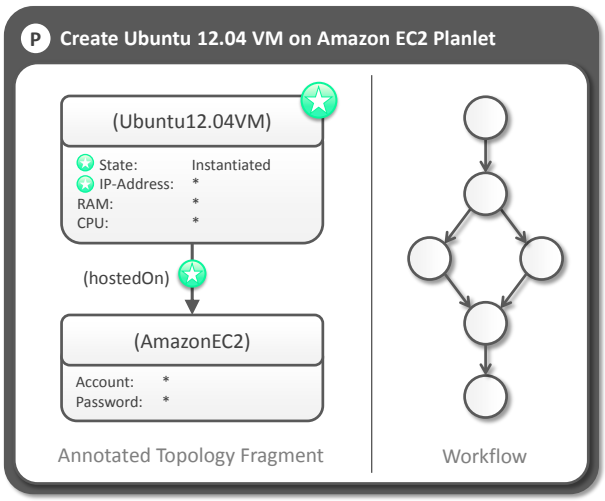

Figure 6. Planlet that creates an Ubuntu virtual machine on Amazon EC2.

to retrieve the information needed to use the corresponding Amazon account for creating the virtual machine. The planlet's effects are expressed by the attached Create-Annotations: the "Ubuntu12.04VM" node as well as the "hostedOn" relation to the "AmazonEC2" node will be created. In addition, the Create-Annotation attached to the Ubuntu12.04VM node's "State" and "IP-Address" properties define that the planlet sets these properties to the specified values: "State" to "Instantiated", "IP-Address" to a value that is not known before, which is expressed by the ⋆, as the actual IP-address can be determined not until the real provisioning of the virtual machine.

Figure 7 shows a planlet that imports data into a MySQL database. This is expressed by the domain-specific *ImportData-Annotation* attached to the MySQLDB node (depicted as purple cycle). This planlet must not be executed before the database is instantiated because of the state-property precondition. Thus, another planlet that creates this node, i. e., that creates a new MySQL database on a MySQL DBMS, sets this property that is used by the shown planlet as precondition. Based on this property, the order of the two planlets is determined: the planlet creating the database must be executed before the shown planlet that imports data. In addition, the shown planlet defines preconditions to execute this task by declaring required properties: endpoint information, i. e., IP-address and port, user, password, and database name. All these properties must be provided by the DASM itself or planlets that are executed before.

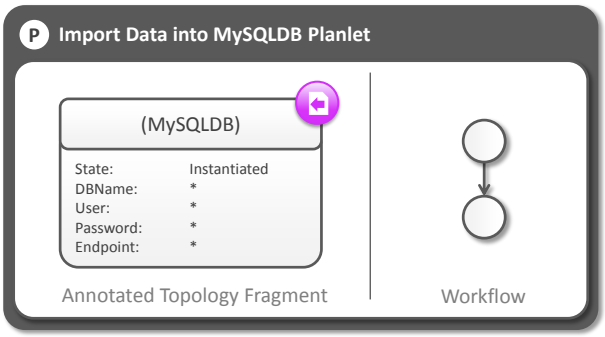

Figure 7. Planlet that executes the ImportData-Annotation on a MySQLDB.

To execute multiple Management Annotations declared in a DASM, typically multiple planlets have to be orchestrated into an overall Management Plan. All available planlets are stored in a library, which is used by the Plan Generator to find appropriate planlets for generating the desired Management Plan. The Plan Generator uses the planlet's fragment for selecting suitable planlets to process all Management Annotations in the DASM and to order planlets based on preconditions and effects. During the plan generation, a virtual representation of the current state of the application gets transferred towards the desired goal state defined by the DASM. In each intermediate state, all planlets whose preconditions match the current virtual state are called *candidate planlets*. These planlets are eligible to be applied. The Plan Generator decides which planlet transfers the application into the next state. The order of planlets is determined based on their preconditions and effects: all preconditions of a planlet must be fulfilled by the DASM itself or by another planlet that is executed before. This enables administrators to specify *static* information directly in the DASM, e. g., the desired RAM of the virtual machine created by the planlet shown in Figure 6 or the database name for the planlet shown in Figure 7. *Dynamic* information such as the IP-Address of a VM are directly written by planlets to the application's instance model, i. e., to its ETG. Thus, planlets communicate with each other via element properties in the application's ETG.

The framework enables distributing logic across several planlets that do not need to know each other. Each planlet implements a small functionality and can be used in combination with other planlets. This enables integrating different management technologies seamlessly into one holistic and collaborative management framework: all technology specific details are implemented by the planlet's workflow but not exposed to the Plan Generator. Thus, the framework provides a uniform manner to integrate heterogeneous technologies.

D. Provisioning of Applications

Besides management, the Management Planlet Framework supports also the initial provisioning of applications through generating executable *Provisioning Plans*, which are a subclass of Management Plans. The Provisioning Plan generation is based on the same concept as the Management Plan generation: a DASM that describes the management tasks to be performed to provision the application is transformed into the corresponding plan by the Plan Generator. Therefore, the DASM contains the application's topology and a Create-Annotation attached to each topology element in the model that shall be created. Figure 5 shows the DASM that describes the provisioning of the LAMP-based motivating scenario: each topology element that has to be created is annotated with a Create-Annotation, nodes as well as relations. This DASM can be used to generate the corresponding Provisioning Plan fully automatically. How these management tasks are finally executed depends on the orchestrated planlets that implement the corresponding Management Annotations. To configure the provisioning, additional Management Annotations may be declared on topology elements to define additional management tasks. For example, in the shown DASM, an ImportData-Annotation is declared on the MySQL database node to define that a certain dataset shall be imported after the database is installed. This Management Annotation can be, for example, executed by the planlet described in the previous section.

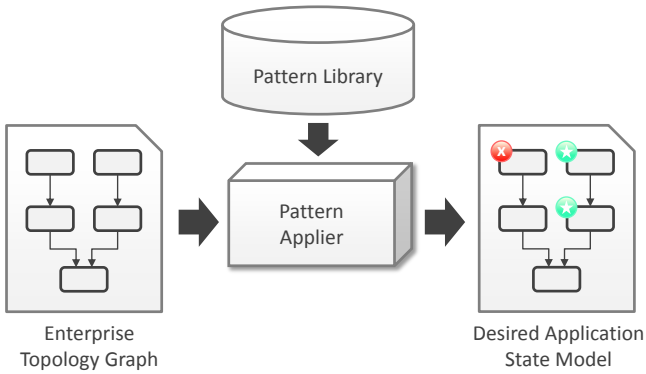

Figure 8. Automated Management Pattern approach.

E. Management of Applications

The framework supports the creation of DASMs for management tasks by two different methods: (i) manual creation and (ii) automated pattern-based creation. To support the *manual* DASM creation, the ETG Discovery Framework [15] is used to automatically discover the current application snapshot in the form of an XML-based ETG. The discovered ETG provides the basis for manually creating a DASM afterwards that specifies the management tasks to be executed by attaching Management Annotations to the corresponding elements. As Desired Application State Models are described in XML, they can be created easily by hand using XML tools. The frameworks Plan Generator is then used to generate an executable Management Plan, which may be customized by the administrator afterwards. In the final step, the generated plan is executed. This manual creation method is suitable when only few management tasks have to be specified, e. g., to export data from a database.

However, if more complex, high-level tasks have to be executed, e. g., scaling an application or updating a Web Server, the manual creation of DASMs quickly degenerates to a serious challenge: these tasks require an overall understanding about which low-level Management Annotations have to be declared to achieve the desired goal. Therefore, the framework employs *Automated Management Patterns* (AMPs) to *automatically* generate DASMs for this kind of tasks based on discovered ETGs [5]. In IT, patterns are a well-established means to document reusable solution expertise for frequently recurring problems [25]. The automation of this concept eases the creation of DASMs to specify complex high-level management tasks. Automated Management Patterns consist of three parts: (i) Topology Fragment, (ii) a Topology Transformation, and (iii) a textual description of the pattern. The *Topology Fragment* is a small application topology that defines to which combinations of nodes and relations the pattern is applicable. Thus, it is used for matchmaking of AMPs and ETGs: if the elements in the fragment match elements in the ETG, the pattern can be applied to these matching elements. The second part is a *Topology Transformation* that automatically applies the pattern by transforming an input ETG into a DASM that describes the tasks to be performed by attaching Management Annotations to the corresponding ETG elements. AMPs are stored in a *Pattern Library* and executed by a *Pattern Applier*, as shown in Figure 8. The third part is a *textual description* of the pattern in natural language to provide human-readable information about the pattern, i. e., the solved problem and the corresponding solution.

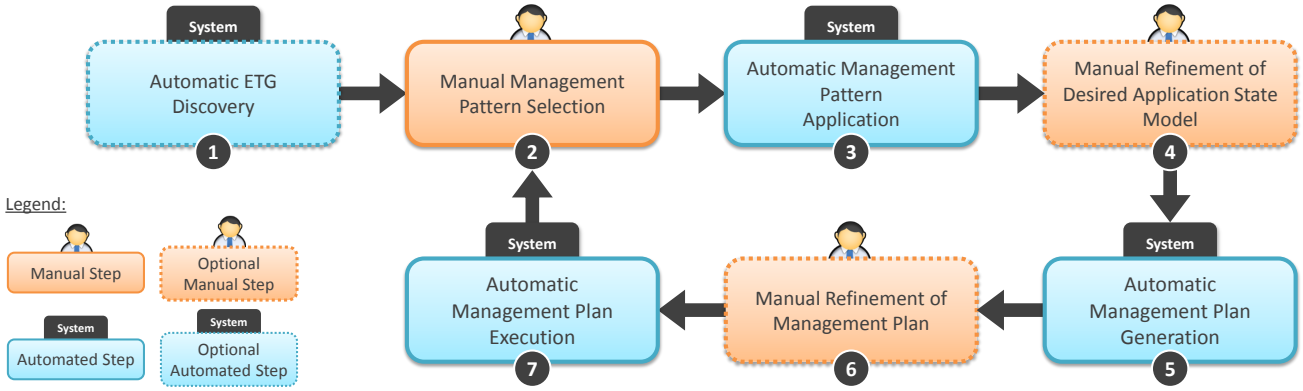

Figure 9. Pattern-based Application Management Automation Method (adapted from [8]).

F. Pattern-based Application Management Automation Method

To ease understanding the policy-aware management approach that gets introduced in this article, we explain the Management Planlet Framework's pattern-based management automation concept in more detail to provide a clear overview on the steps that are performed to apply a management pattern automatically to individual running applications. Based on Breitenbücher et al. [8], we describe the concept as method that automates management based on applying Automated Management Patterns to discovered ETGs. The method's overall process is shown in Figure 9 and consist of seven steps that are either automated by the framework or executed manually by an administrator. As explained by the legend on the bottom left of Figure 9, manual steps are depicted as orange rectangles having an icon attached that depicts an administrator whereas automated steps are rendered as blue rectangles having a "System"-caption attached. Some of the steps are optional, which is expressed by a dotted line surrounding the shape.

In the method's first step, a runtime snapshot of the application to be managed is discovered automatically using the ETG Discovery Framework [15]. The result is an ETG that describes (i) the current application structure and (ii) all runtime information in the form of properties. This step is optional: if the Management Planlet Framework was used to initially provision the application, its ETG was already created by the Management Planlets that executed the provisioning tasks and can, therefore, be used directly for the next step.

In the second step, the management task to be executed is specified by manually selecting an Automated Management Pattern. For example, in Breitenbücher et al. [8], we presented how the "Stateless Component Swapping Pattern" [26] can be implemented as Automated Management Pattern. This pattern captures the required management knowledge to migrate a stateless application component without downtime from a source environment into a target environment. Thus, sophisticated tasks can be applied fully automatically by a simple AMP selection.

The third step is automated by the framework: the Topology Transformation of the selected AMP is executed fully automatically on the ETG of Step 1. The transformation declares the management tasks to be executed in the form of Management Annotations and may modify the topology structure to add new nodes or relations. The result of this step is a DASM that describes the tasks to be executed for applying the pattern.

In Step 4, the resulting DASM may be refined manually for customization purposes or to additionally refine the declared tasks. For example, if an AMP declares how to migrate an application component to the Cloud, in this step the desired target Cloud provider may be changed manually by replacing the corresponding node. As AMPs can implement fully refined patterns, this step is optional (cf. Breitenbücher et al. [8]).

As DASMs are not executable, they must be transformed into executable processes. In Step 5, this is done fully automatically by the framework based on orchestrating Management Planlets into Management Plans, as explained in Section IV-C.

In Step 6, the generated Management Plan may be refined manually. For example, additional activities can be inserted for which there are no Management Annotations. However, since there are often no refinements needed, this step is optional.

In the last Step 7, the generated Management Plan is deployed on a workflow engine and executed to apply the management pattern to the real running application. As Management Planlets update the application's ETG automatically to reflect their changes, further AMPs can be applied directly afterwards. Therefore, the method continues in Step 2.

In Breitenbücher et al. [8], we classified two kinds of AMPs: (i) *Semi-Automated Management Patterns* and (ii) *Automated Management Idioms*. The semi-automated class represents AMPs that implement only the *abstract solution* of a certain management pattern, i. e., the DASMs resulting from Step 3 typically need to be refined manually in Step 4. For example, a semi-automated migration AMP copies only the component node, defines an abstract runtime environment, and attaches the corresponding Management Annotations. Thus, information that is required to select appropriate Management Planlets, e. g., a concrete target environment, need to be refined manually in the resulting DASM. In contrast, Automated Management Idioms implement a *refined solution* of a management pattern for a certain use case. For example, the aforementioned migration management pattern can be implemented as idiom refined for the concrete use case of migrating Java-based Web applications hosted on a Tomcat Servlet Container to the Amazon Cloud. Thus, all required refinement information is implemented directly in the idiom's transformation and a manual refinement is not required. The policy-aware management approach presented in this paper is agnostic to this distinction. Therefore, we do not distinguish in this paper and simply refer to AMPs.

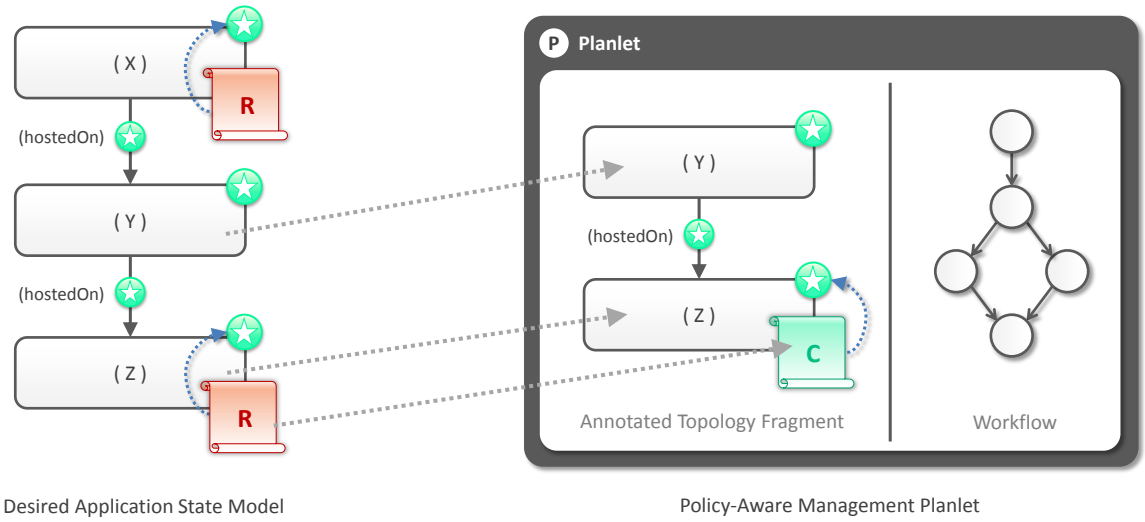

Figure 10. General concept of the approach: policies attached to a topology element are bound to the Management Annotations that must comply with the policy.

V. Policy-aware Management Planlets

In this section, we present the first part of the approach that enables the policy-aware execution of management tasks based on the Management Planlet Framework. We introduce (i) Policy-Aware Management Planlets, which are able to enforce policies during the execution of management tasks, (ii) Management Annotation Policies, which provide a format for defining and processing Management Policies, and (iii) show how the Management Planlet Framework is extended to support the policy-aware execution of management tasks.

The general concept is based on attaching Management Policies to elements in DASMs and elements in Management Planlet fragments that are bound directly to the management tasks the policies apply to. These policies are then analyzed during the plan generation to determine if a candidate planlet fulfills the non-functional requirements on management tasks defined by the policies in the DASM. To enable this, we introduce the concept of *Management Annotation Policies*, which provides a formal policy format that enables binding a policy directly to the Management Annotations it applies to. As these policies can be attached to elements in DASMs as well as elements in planlet fragments, we distinguish between two semantics: (i) a Management Annotation Policy attached to an element in a DASM defines the Management Annotations that must comply with the policy (called "topology policy") whereas (ii) a policy attached to an element in a planlet fragment defines the Management Annotations for which the planlet guarantees fulfilling the policy (called "planlet policy"). Thus, a candidate planlet that executes a Management Annotation in a DASM must consider all policies the DASM specifies on this annotation. This enables creating *Policy-Aware Management Planlets*, which specify the non-functional capabilities they ensure for the Management Annotations they execute on the respective nodes and relations modelled in their fragments. During the plan generation, each Management Annotation Policy bound to a Management Annotation in the DASM must be fulfilled by the Management Planlet that executes the respective annotation. This ensures that all policies specified in a DASM are enforced when the associated annotations they apply to are executed.

Figure 10 explains the presented concept visually: on the left, it depicts a DASM consisting of three components connected by hostedOn-relations that have to be provisioned, which is expressed by the attached Create-Annotations. The nodes of type X and Z have Management Annotation Policies attached defining the non-functional security requirements that have to be fulfilled during the execution of the associated Create-Annotations they apply to. 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 are provided by the planlet for executing the Create-Annotation. During the plan generation, the policies are compared and checked for compatibility. If a candidate Management Planlet fulfills all Management Annotation Policies attached to elements in the DASM that are applied to Management Annotations it executes on these elements, the planlet is applicable.

In contrast to many existing bidirectional policy approaches, we define a strict one-way requirement-driven perspective: policies attached to elements in a DASM define requirements on the tasks 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 are ignored if they are not required to fulfill the requirements defined by the DASM.

A. Management Annotation Policies

In this section, we introduce the format of Management Annotation Policies, which enables specifying non-functional security requirements and capabilities directly on the Management Annotations they apply to. Management Annotation Policies can be attached to elements, i. e., nodes and relations, contained in DASMs and to elements in the Annotated Topology Fragments of planlets. Policies attached to DASM elements express non-functional requirements whereas policies attached to fragment elements of planlets express the non-functional capabilities on tasks represented by annotations. A Management Annotation Policy consists of eight different parts, which define its semantics and how the policy must be processed. In the following subsections, we explain these parts in detail.

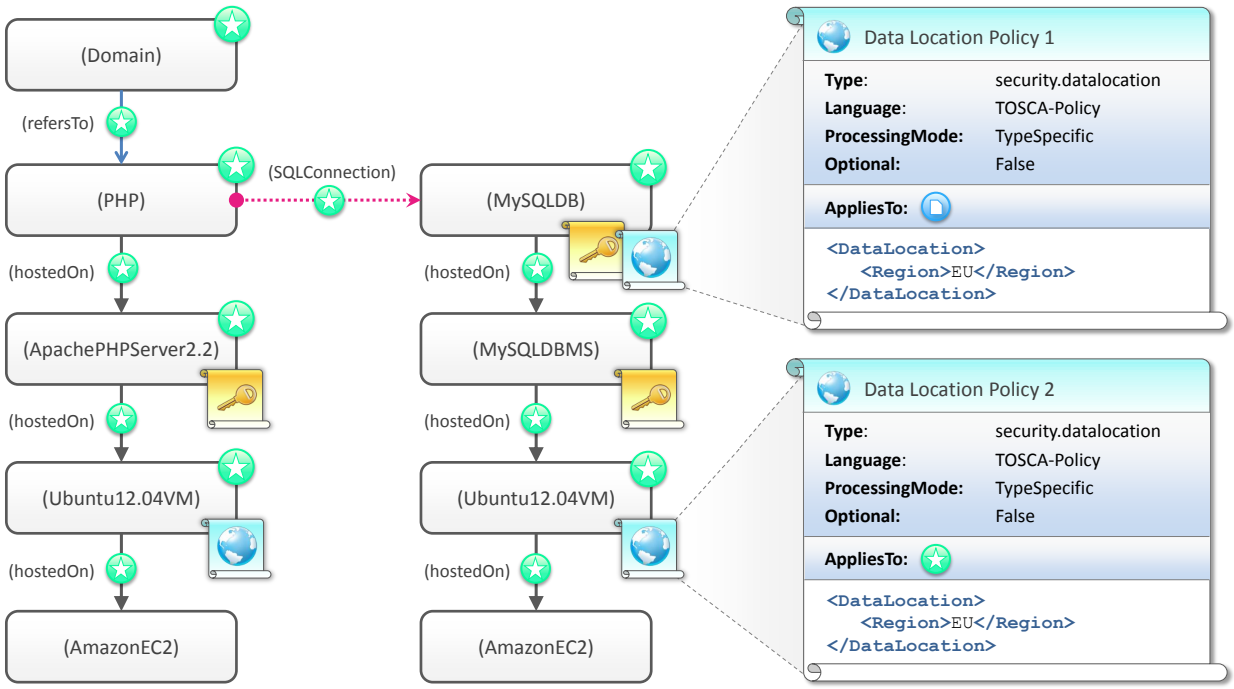

Figure 11. Management Annotation Policies attached to nodes in the DASM that describes the provisioning of the motivating scenario.

1) Identifier and Type: Each Management Annotation Policy has a unique *id* within the model it is contained and an optional *type* defining the semantics of the policy. For example, a policy of type "Secure Password Policy" ensures that the login password of a component is strong enough to resist attacks. The semantics of a policy type have to be well-defined and documented, i. e., DASM modellers and planlet developers must be aware of its meaning and how to use and enforce it.

2) Content Field and Language Attribute: As there are a lot of existing policy languages, such as WS-Policy, Ponder, or KAOS [27], our approach supports their integration through an optional *content* field and an optional *language* attribute: the content field enables to fill in any policy-specific information whereas the language attribute defines the used policy language.

3) Processing Mode: The *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 types need to be compared, the language attribute is unnecessary. This is explained in detail in Section V-C.

4) Optional Attribute: Each Management Annotation Policy has an attribute *optional* that defines if the processing of the policy is mandatory. In DASMs, this attribute can be used to define optional policies that express security requirements that are "nice to have" but not necessarily required. Planlets can declare optional policies to vary their execution: optional policies are fulfilled only if explicitly required by the DASM.

5) AppliesTo-List: To specify the Management Annotations that must consider the policy, each Management Annotation Policy explicitly defines an *AppliesTo-list* that contains the affected Management Annotations. We distinguish here between two sides: (i) a topology policy attached to an element in a DASM specifies in this list the Management Annotations that must comply with the policy, i. e., all planlets that execute one of the annotations in this list must consider the policy. If this list is empty, all planlets executing annotations on the element must consider the policy. (ii) A policy attached to a planlet fragment defines in this list the Management Annotations for which the planlet ensures the policy. If this list is empty, the planlet guarantees the policy for all Management Annotations it executes on the corresponding element. This concept allows binding Management Policies directly to the affected tasks.

6) Ignore-List: If a policy must be processed by all tasks except a few exceptions, using the AppliesTo-List would require to specify all these annotations. Therefore, to add exceptions easily, a Management Annotation Policy may specify annotations the policy does not apply to in the so-called *Ignore-list*. For policies in DASMs, all Management Annotations specified in this list do not have to consider the policy. For policies on elements in Annotated Topology Fragments of planlets, the list specifies the annotations for which the planlet does not guarantee enforcing the policy. Thus, if the AppliesTo-List of a policy is empty, i. e., the policy applies to all Management Annotations, adding Management Annotations to the Ignore-List enables defining exceptions on both sides.

Figure 11 shows two Data Location Policies attached to the virtual machine and database of the motivating scenario in detail. Both are defined in the same language and must be processed by a type specific plugin. The difference lies in the AppliesTo-lists: the Management Annotation Policy attached to the virtual machine must be considered only for its creation, which is depicted by the Create-Annotation in the AppliesTo-list whereas the Management Annotation Policy attached to the database must be considered only by planlets that handle data, e. g., by planlets that execute ImportData- or ExportData-Management Annotations. This is expressed by the domain-

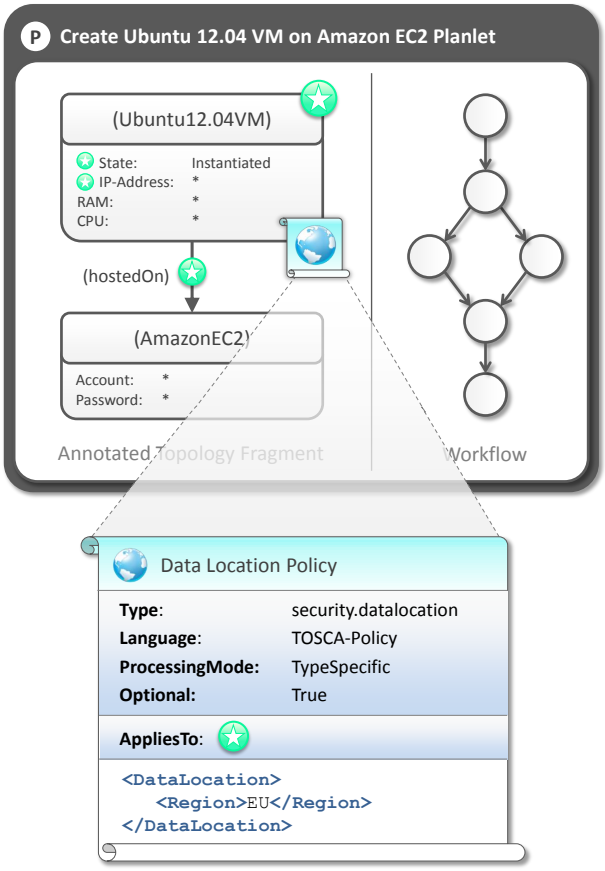

Figure 12. Policy-Aware Management Planlet that creates a virtual machine on Amazon EC2 ensuring a Data Location Policy on the Create-Annotation.

specific *DataHandling-Annotation* depicted as blue circle. 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 to remain. Thus, the planlet instantiating the VM must enforce this policy, e. g., as shown by the planlet depicted in Figure 12. In contrast, the location of the database needs not to be considered by the planlet that installs it on the operating system as the physical location of the underlying virtual machine is essential, not the installation of the database on this machine. Therefore, a normal planlet without any policy can be used that 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 also the data itself has to remain in the EU. Thus, this concept allows a fine-grained definition of requirements on different levels for different kinds of tasks.

On the other side, Management Annotation Policies attached to elements in planlet fragments define in the AppliesTo-list for which tasks the planlet ensures the Management Annotation Policy. For example, the planlet shown in Figure 12 that instantiates a new Ubuntu virtual machine on Amazon EC2 provides an optional policy that ensures that this task (executing the Create-Annotation on the VM node) can be executed in consideration of the attached policy. Thus, the planlet is able to fulfill this policy for the instantiation of the VM if

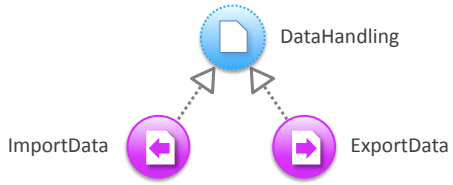

Figure 13. Management Annotation inheritance for data handling.

required by the DASM. As a result, the AppliesTo-list enables binding *non-functional capabilities* to the tasks executed by the planlet through attaching policies and linking them with the corresponding Management Annotations. This enables a direct binding of non-functional capabilities to management tasks.

B. Management Annotation Inheritance

Management Annotations are atomic entities that define either structural or domain-specific management tasks as explained in Section IV-B. However, this is not sufficient for working with policies as it allows no abstract classification of tasks. For example, the Data Location Policy attached to the MySQL database as shown in Figure 11 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 "ImportData" or "ExportData" as shown in Figure 7 that need to fulfill the Data Location Policy, e. g., data to be imported or exported must not be stored on servers outside the European Union. Listing all these annotations in the AppliesTo-list of the policy is not efficient. Thus, we extend the concept by introducing inheritance as depicted in Figure 13: the ImportData- and ExportData-Management Annotations inherit all properties from the superclass annotation of type "DataHandling". For example, the Data Location Policy in Figure 11 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. Policy Processing Modes and Matchmaking

Management Annotation Policies specify a processing mode that defines how the topology policy has to be checked during the matchmaking of DASM and the Annotated Topology Fragments of candidate planlets. We introduce three processing modes: (i) Type Equality, (ii) Language Specific, and (iii) Type Specific. The processing mode defines the *minimum criterion* that must be met to fulfill the topology 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 topology policy defines processing mode *Type Equality*, which cannot be fulfilled by any planlet but there is a planlet fulfilling the policy for the *Language Specific* processing mode, the topology policy is fulfilled by this planlet. Thus, if a criterion of a topology policy cannot be met, the system tries the next stronger criterion for this policy automatically.

1) Type Equality: This processing mode defines that only the types of topology policy and candidate planlet policy must be equal. Thus, for each policy attached to an element in the DASM there must be a policy of the same type attached to the corresponding element in the candidate planlet's Annotated Topology Fragment. If the framework finds a planlet providing a policy with compatible type, AppliesTo-Lists and Ignore-Lists of both topology and planlet policy must be also compatible, i. e., each Management Annotation contained in the AppliesTo-List of the topology policy must be either (i) contained in the AppliesTo-List of the planlet policy or (ii) the AppliesTo-List of the planlet policy is empty and its Ignore-List does not contain the corresponding annotation. This is required to ensure that the planlet fulfills the policy for the desired tasks. Using this processing mode is sufficient for policies that can express all their requirements by a well-defined term that is used as type, e. g., 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 topology policy must be processed by a dedicated 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 that implements all the language-specific logic. The language plugin gets a reference to the policy to be checked, the whole DASM, the candidate planlet, and a mapping of elements as input. The mapping defines which elements in the topology correspond to elements in the candidate planlet's Annotated Topology 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 policy exists in the Annotated Topology Fragment of the candidate planlet at all. However, plugins must analyze to which Management Annotations a topology policy is bound and have to consider this information when they execute their language-specific matchmaking logic to evaluate candidate planlets.

3) Type Specific: This processing mode is the most specific one and bound to both policy language and policy type, i. e., if there is a policy of type "Data Location Policy" written in "WS-Policy" having this mode, there must be a plugin registered for exactly that combination to process the policy in terms of evaluating candidate planlets. Otherwise, the policy cannot be fulfilled. If such a plugin exists, the processing is equal to language specific: the plugin gets the same information as input and decides if the candidate 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 is attached to a MySQLDB node that shall be created, the policy applies to the Create-Annotation, and there are no Policy-Aware Management 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 a simple MySQLDB-Creation planlet that provides no policy at all. This kind of processing enables complex logic that can be only known by a specific type plugin.

VI. Policy-Aware Provisioning of Applications

In this section, we describe how the concept of Policy-Aware Management Planlets is used to automate the provisioning of applications in compliance with non-functional security requirements specified on the execution of provisioning tasks. The developer of the application manually creates an application topology that models the application to be provisioned and declares the desired security Management Annotation Policies. Based on this model, the framework is able to automatically generate a DASM that describes the application's provisioning: it employs a generic Automated Management Pattern that annotates Create-Management Annotations to all elements in the application topology that have to be provisioned without changing the declared policies. Thus, the application topology becomes a DASM by applying this generic *Provisioning* AMP automatically, which can be executed on any application topology as its Topology Transformation is independent from individual structures. The resulting DASM is typically adapted manually to configure the provisioning by adding additional Management Annotations. For example, additional management tasks such as importing data into a database are added. The Provisioning AMP does not change the Management Annotation Policies that are attached in the application topology. It neither adds, nor changes, nor removes policies and attaches only Create-Annotations. Therefore, the non-functional requirements specified by the attached policies are not changed and originally contained in the resulting DASM. Thus, the first management task that considers the initial provisioning of the motivating scenario is specified by the DASM shown in Figure 11, which was created automatically by applying the Provisioning AMP. As the AmazonEC2 nodes represent the lowest layer, these nodes are not annotated by the AMP as this layer describes infrastructure or platform services or physical hardware and, therefore, already exists. The policy-aware provisioning of this application is ensured by the Management Annotation Policies attached to the elements in the DASM that must be considered by the Management Planlets that execute the Management Annotations they apply to. In addition, the planlets directly create the corresponding instance model in the form of the application's ETG and copy all Management Annotation Policies to the corresponding elements to ensure that further management tasks also consider these policies.

VII. Policy-Aware Management of Applications

The Management Planlet Framework supports managing applications by two different methods: (i) manual creation of Desired Application State Models and (ii) applying Automated Management Patterns to create DASMs automatically following the pattern-based method described in Section IV-F [5]. Each management task described in the motivating scenario is suited for one of these approaches. To backup the database, the DASM can be created manually as only a simple attachment of an ExportData-Annotation is required to specify the task. In contrast to this, updating the Apache Web Server is more complex if downtime must be avoided. Therefore, we employ the concept of Automated Management Patterns to specify the annotations to be executed. In this section, we show how both management tasks can be executed automatically using the Management Planlet Framework while the non-functional security requirements specified by the attached Management Annotation Policies are considered by both approaches.

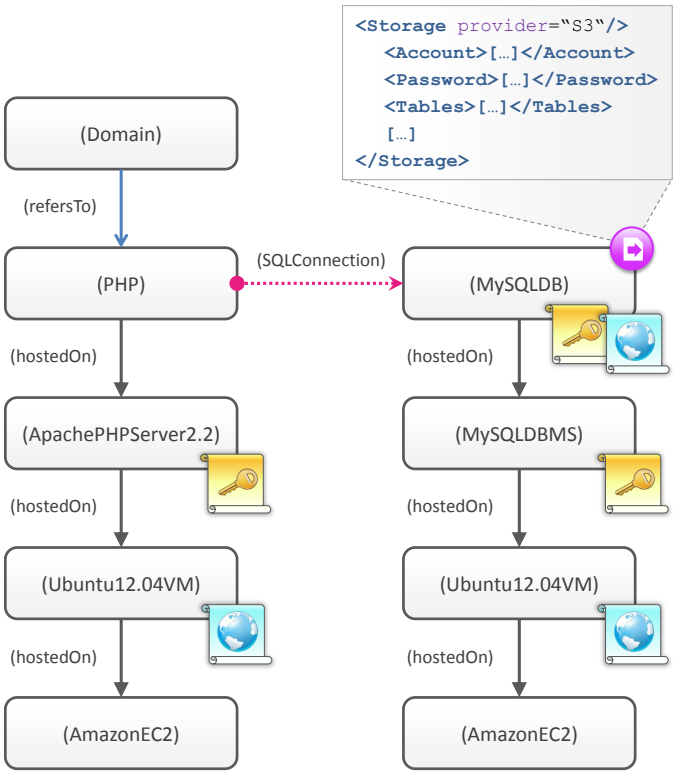

Figure 14. DASM that describes the second management task of the motivating scenario with manually attached ExportData-Management Annotation.

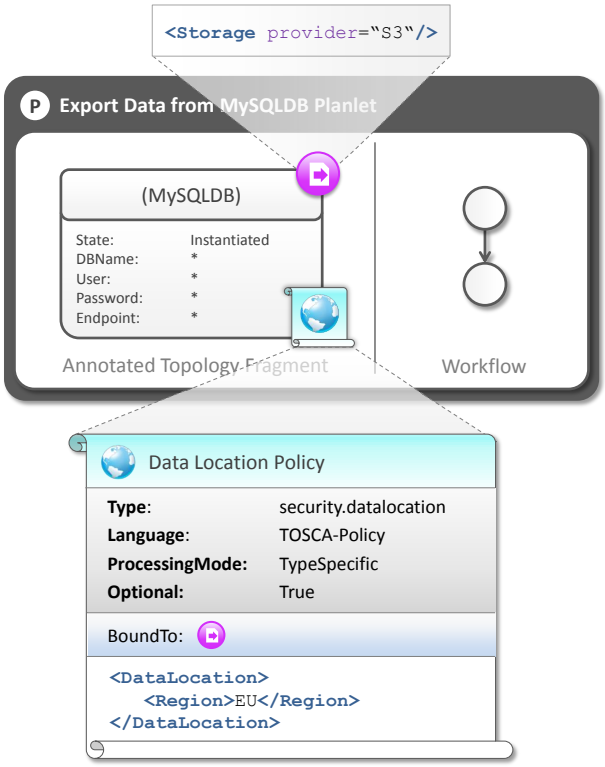

Figure 15. Policy-Aware Management Planlet that exports data from a MySQLDB node to an Amazon S3 storage located in the European Union.

A. *Manual Specification of Policy-Aware Management Tasks*

The Management Planlet Framework supports creating Desired Application State Models manually, i. e., the administrator attaches the management tasks to be performed in the form of Management Annotations to the corresponding topology elements in the DASM by hand (cf. Section IV-E). In the motivating scenario, a backup of the application's database shall be made, which is a typical management task that can be specified manually: only a simple ExportData-Management Annotation has to be attached to the MySQLDB node and configured in terms of the target storage. The resulting DASM is shown in Figure 14. In our scenario, we specify in the annotation that the data backup shall be stored in Amazon's Simple Storage Service "Amazon S3" [28], which is defined by the annotation-specific content of the ExportData-Annotation. Additional information for configuring the export task are also provided, e. g., the tables that have to be exported and the Amazon account that shall be used. All these information are used by the planlet that executes the Management Annotation. The corresponding Management Planlet must, in addition, fulfill the Data Location Policy attached to the MySQL database node that defines that all data handling tasks must consider this policy (the details of the Data Location Policy were shown in Figure 11). As the ExportData-Management Annotation inherits from the DataHandling-Management Annotation, the Management Planlet exporting the data must comply with this policy. Therefore, the Management Planlet exporting the data must also specify a Data Location Policy on the MySQLDB node ensuring that the planlet is aware of storing the data not outside the declared region—in this case, the European Union.

The planlet shown in Figure 15 fulfills all these requirements. It executes the ExportData-Management Annotation on the MySQLDB node and ensures by the attached Data Location Policy that the exported data remains in the European Union. The shown planlet is able to export data to Amazon S3, which is declared by the Export-Data Management Annotation. Thus, as the annotation in the Desired Application State Model defines the same storage service, the planlet is applicable and selects a storage on Amazon S3 that is hosted on a physical server located in the EU, which is defined by the policy. The matchmaking of the Management Annotation's specific content, here the S3 description, is based only on the main element's name (here "Storage") and all attributes of this element (here "provider"). Therefore, the annotation defined by the planlet matches the annotation declared in the DASM shown in Figure 14.

Based on the manual creation of DASMs, administrators are able to define the management tasks to be executed on a very high level of abstraction. They only have to declare the abstract Management Annotations on topology elements without the need for detailed technical expertise of the underlying management technologies. For example, the presented export data task requires the administrator only to attach and configure the annotation by defining the storage information and the tables to be exported. All technical execution details are inferred automatically by the framework through invoking the corresponding Management Planlet. In addition, the Management Planlet Framework automatically ensures that the execution of Management Annotations complies with the non-functional requirements. Thus, the administrator only defines Management Annotations and does not have to care about the defined policies.

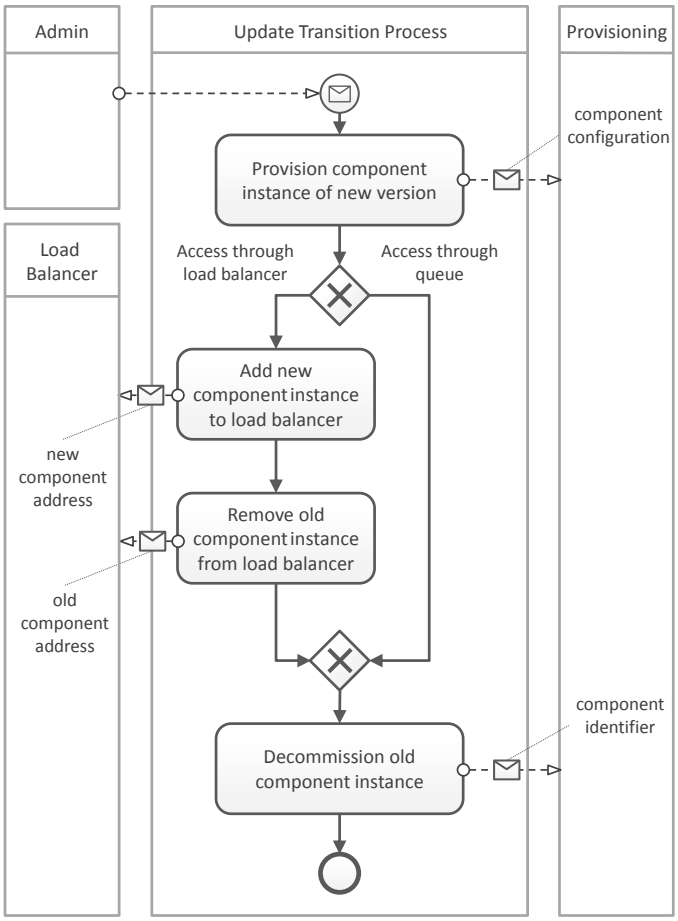

Figure 16. Abstract Update Transition Process modelled in BPMN [29].

B. Automated Specification of Policy-Aware Management Tasks

The Management Planlet Framework supports the pattern-based method described in Section IV-F to automatically create DASMs through applying Automated Management Patterns to ETGs. The administrator only selects the pattern to be applied and the pattern's Topology Transformation attaches all management tasks that have to be performed automatically in the form of Management Annotations to the corresponding topology elements in the DASM. To execute the third management task of the motivating scenario—updating the Apache Web Server to a new version without downtime—we employ this concept. Therefore, we first (i) describe the pattern that is able to do this update, (ii) show how this pattern can be implemented as Automated Management Pattern, and (iii) show that all policies are considered during the Management Plan generation.

Patterns are a well-established concept to document proven solution expertise for problems that frequently occur in a certain context in a generic and abstract manner. This enables capturing the core of problem and solution expertise in an abstract fashion that can be refined for individual use cases. In the domain of Cloud Computing, patterns are of vital importance to build, manage, and optimize IT. In this paper, we focus on management patterns that describe abstract management processes for typical problems in Cloud environments [30]. For example, how to manage resiliency, elasticity, or the migration of application components [26][29]. According to Christopher

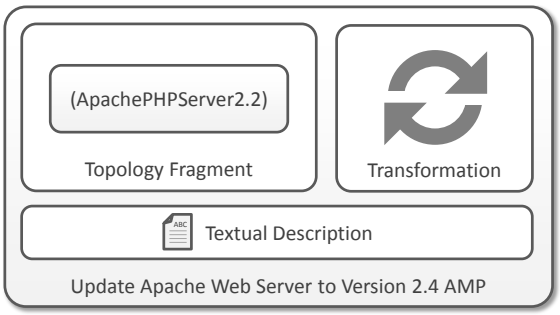

Figure 17. Automated Management Pattern that updates the Apache Web Server from version 2.2 to version 2.4 without downtime.

Alexander, a pattern is a three-part rule that captures the relation between (i) a certain context, (ii) a problem, and (iii) a solution [31]. Following this definition, patterns provide a suitable means to describe the execution of management tasks, such as updating a Web Server without application downtime, in consideration of a certain context that is constrained by non-functional security requirements. Therefore, we automate a management pattern to update the Apache Web Server from version 2.2 to version 2.4 without downtime and make the corresponding pattern implementation aware of policies.

The corresponding pattern is called *Update Transition Process Pattern* [29] and originates from the Cloud Computing pattern language developed by Fehling et al. [26][29][30]. The question answered by this pattern is *"How can application components of a distributed application be updated seamlessly?"*. The context observed by the pattern is that during the lifetime of an application, new versions of used middleware, operating systems, or application components may become available. If the application has to ensure high availability, the transition time of updating a component from an old to a new version shall be minimized to avoid downtime of individual application components or the overall application. Thus, its intent is updating an application component to a new version seamlessly without downtime, i. e., the overall application's functionality is still available during and after the updating process. A component shall, therefore, be updated transparently to the overall system. Thus, the pattern fits exactly to our management task of updating the Web Server while ensuring the application's availability. The pattern's solution is depicted as BPMN [18] process in Figure 16: an administrator triggers the update transition process that first provisions a component instance of the new version. The new component runs simultaneously with the old application component. Afterwards, the load balancing is switched to the new component instance of the new version. This avoids downtime of the updated component. Finally, the old application component is decommissioned.

To apply this abstract management pattern to our concrete use case of updating an Apache Web Server, we automate its abstract process by an Automated Management Pattern. The corresponding AMP is shown in Figure 17. The pattern defines by its topology fragment that it is only applicable to elements of type "ApachePHPServer2.2". The Topology Transformation implements the management logic that is refined explicitly for this individual use case, i. e., for migrating the Web Server from version 2.2 to version 2.4. Thus, the transformation is capable of transforming an input ETG that contains a node of type

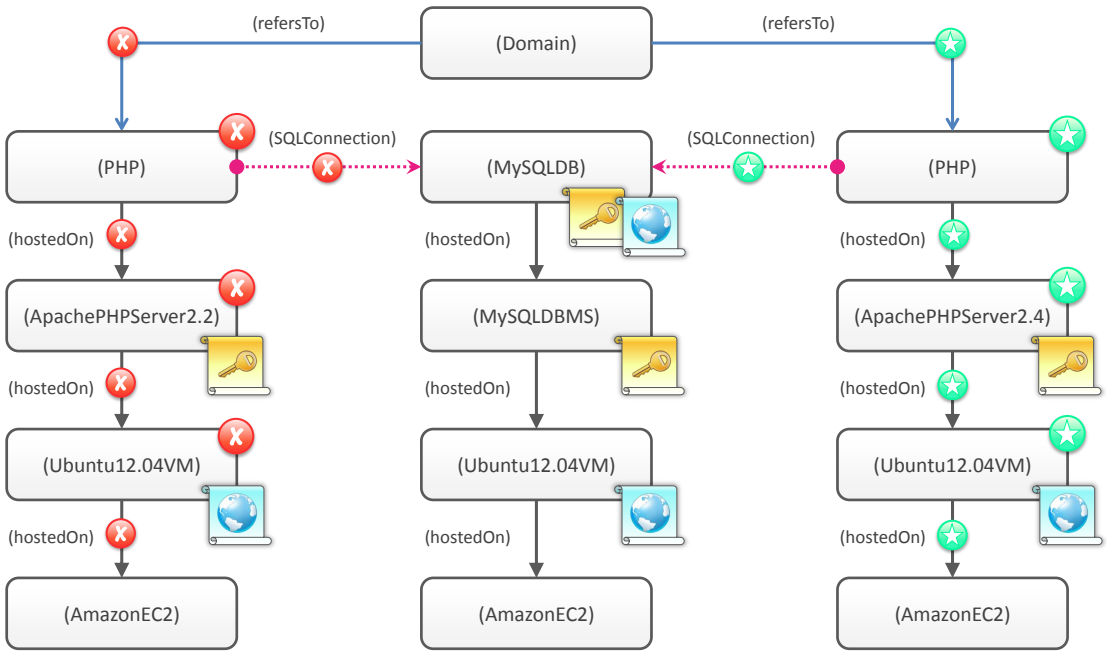

Figure 18. Desired Application State Model describing the Management Annotations to be executed for updating the Apache Web Server without downtime.

"ApachePHPServer2.2" to an output DASM that describes all Management Annotations that have to be executed for replacing this Web Server by the new version 2.4 without downtime.

The refined process is described in the following. To update the Web Server without downtime, the old deployment must remain active until the new deployment is completely provisioned. Otherwise, during replacing the old installation, the system would go down. As it is not possible to install two Apache Web Servers on a single virtual machine that listen both to the same port—in this case, the HTTP standard port 80—a new virtual machine has to be created to install the new Web Server on a different operating system. This is important as the new installation must have exactly the configuration of the old server. As the pattern is specialized for exactly that replacement of Apache Web Servers, it is able to extract the configuration of the old Web Server and to specify the configuration of the new Web Server accordingly. Thus, the functional behaviour is identical. As soon as the new installation is running, updating the internet domain can be triggered. However, to prevent downtime reliably in this step, the old installation must not be decommissioned until the Domain Name System (DNS) servers were updated with the new URL. Therefore, we employ a DNS Propagation Checker that checks when all servers are updated and the old installation can be terminated. All these considerations are important to ensure correct operation.

The Topology Transformation of the corresponding Automated Management Pattern is implemented as follows. The resulting DASM after applying the pattern is shown in Figure 18. The pattern copies the whole stack hosting the Web Server including all attached policies and replaces the Web Server node by the new version. The new stack is annotated with Create-Annotations to specify the nodes and relations that must be created. All incoming and outgoing relations of the old stack have to be destroyed and added to the nodes of the new stack to redirect the relations, i. e., "refersTo" relations of the domain and "SQLConnection" to the database. The old stack is partially annotated with Destroy-Annotations: all nodes that are hosted on the Web Server and the Web Server node itself must be destroyed. Therefore, the PHP node and the Web Server node are annotated with Destroy-Annotations. In addition, the pattern analyzes the underlying stack of the Apache Web Server to be updated and recognizes that the virtual machine has no incoming or outgoing relations except its hostedOn relation to AmazonEC2, i. e., the virtual machine is only used to host the Web Server. Therefore, the pattern annotates also this node to be destroyed as it creates a new virtual machine for the new Web Server. If the node has any incoming relations, e. g., another component is also hosted on this VM, the pattern would only create a new virtual machine but not annotating the old VM to be destroyed. To ensure that downtime is avoided during this update, both stacks must be active until the domain is switched completely to the new deployment. To achieve this, the Destroy-Annotation and the Create-Annotation that are attached to the "refersTo" relations must be processed concurrently. It is not appropriate to employ one planlet that destroys the old "refersTo" relation and another to create the new one as between these two executions, the application would be not available. Therefore, the transformation declares that these two annotations must be executed concurrently, which is only possible using a single planlet that executes both annotations. This planlet finishes not until all DNS servers are updated with the new URL and has the precondition that the old target node as well as the new target node of the domain are running. As only this planlet can be used for generating a complete plan (due to the concurrency requirement specified on the two annotations), its precondition avoids that the old stack is decommissioned before the new stack is ready. Thus, the domain-switching Management Planlet is executed before the planlets that decommission the old stack. Similarly, the transformation declares that the Create-Annotation on the new SQL connection must be executed before the Create-Annotation of the new refersTo-relation, the Destroy-

Annotation on the old SQL connection thereafter. The execution of this DASM is completely policy-aware as the policies of the old stack are copied completely to the new stack. Thus, each provisioning task has to consider the respective policies. This ensures that the non-functional security requirements are not violated and that the new stack complies with all policies that were ensured during the initial provisioning of the application.

By having a fine-grained description of the running application and the management tasks to be performed in one formal model, complex processing can be implemented using Automated Management Patterns through traversing the graph. Thus, every rule that would be considered by a manual execution can be implemented in such Topology Transformations. We showed how complex transformations can be implemented to analyze the context in which the Management Annotations are executed in Breitenbücher et al. [7]. However, some management tasks may be too complex to be automatically specified by an Automated Management Pattern completely. For example, if the internal state of the application needs to be considered, too. Therefore, a manual adaptation of the resulting DASM may be required to ensure a complete and correct execution of complex tasks. This is possible in Step 4 of the framework's pattern-based management method described in Section IV-F. Following this method, Management Annotation Policies can be declared at three points in time. Either (i) the developer of the application defined the policies when creating the initial application topology that was used by the Provisioning AMP to provision the application (cf. Section VI), (ii) policies are added by the AMP in step 3, or (iii) they are declared manually on the DASM resulting from applying an AMP in Step 4. The next section discusses this in detail.

C. Policy-Preserving versus Policy-Changing AMPs

In this section, we generally classify two basic kinds of Automated Management Patterns in terms of handling declared Management Annotation Policies: (i) Policy-Preserving Automated Management Patterns and (ii) Policy-Changing Automated Management Patterns. We explain both classes in this section and provide examples to illustrate their differences.

The first class of *Policy-Preserving Automated Management Patterns* does not change policies attached in the ETG at all when generating DASMs, i. e., they neither add, nor modify, nor remove attached policies. They only define tasks to be executed by attaching Management Annotations and may add new components or relations. As a result, they are not aware of any policy at all: as they only add the management tasks to be executed, all policies are ensured by the Policy-Aware Management Planlets that execute the corresponding Management Annotations to which the policies are applied. For example, if there is a "KeepAlive-Policy" attached to a topology element that defines that the element never must be destroyed, the policy applies to all Management Annotations that stop the element in any form, i. e., the Stop-Management Annotation and the Destroy-Management Annotation. This policy is ensured automatically by the system through the executing planlets, independently from the Management Annotations that are added: even if a Destroy-Annotation is attached to the topology element, the policy is never violated as there cannot exist a Management Planlet that fulfills that policy for the Destroy-Management Annotation. The reason lies, obviously, in the semantics of policy and Destroy-Management Annotation, which are contradicting: the policy forbids stopping or destroying the element while the Destroy-Management Annotation defines exactly that task. Thus, the Plan Generator cannot generate the corresponding Management Plan. This indicates that at least one policy is violated. As a result, if an AMP does not change policies at all, it implicitly considers each policy through the decoupling of *specifying* management tasks and *executing* management tasks. Only the execution must be aware of the policies, not the patterns that only specify tasks. Thus, as long as Automated Management Patterns do not change policies, the sole specification of Management Annotations and their execution by planlets does not violate policies.

In contrast to this class, *Policy-Changing Automated Management Patterns* may change Management Annotation Policies attached in the ETG, i. e., they may add, adapt, or remove policies. For example, the Update Transition Process AMP introduced in the previous section changes the policies in the ETG as it copies the application stack including the attached Management Annotation Policies. Thus, it adds not only Management Annotations to be executed and new topology elements to the DASM but also new policies. As a result, applying this class of AMPs to an ETG changes the defined non-functional requirements in Step 3 of the framework's pattern-based method described in Section IV-F. Therefore, depending on the use case and the pattern to be applied, a manual check and adaptation of the resulting DASM may be required in the method's following Step 4 to ensure a correct specification of the management tasks to be executed and the changed policies before the Management Plan gets generated in Step 5. In addition, the administrator may even analyze the resulting generated Management Plan in Step 6 for correcting problems manually afterwards. However, to detect possible problems, analyzing the DASM is more appropriate as this model describes also the context in which the management tasks are executed [7]. In our motivating scenario, there is no problem to apply the Update Transition Process Pattern as it does not modify the policies attached to the original model. Thus, the non-functional requirements defined in the original ETG are not changed as it only attaches Destroy-Annotations to the original stack. Even if those annotations violate a policy attached to this stack, e. g., a KeepAlive-Policy that is attached to the PHP application, this policy is ensured as the plan generation will fail (similar as described for Policy-Preserving AMPs). Attaching new policies to the newly created stack containing the new Web Server node does not change the previous non-functional semantics as the same kind of policies are attached to a semantically equal stack having the same functionality.

VIII. Policy-Aware Management Framework

In this section, we describe how the presented approach is realized in the used Management Planlet Framework [5]. Figure 19 shows the extended architecture of the framework with the new integrated policy extension (gray background).

A. Architecture

The basic architecture of the Management Planlet Framework consists of a *Plan Generator* that uses a *Planlet Manager* to retrieve the planlets and their descriptions stored in a *Planlet Library*. The Plan Generator has a planlet orchestrator

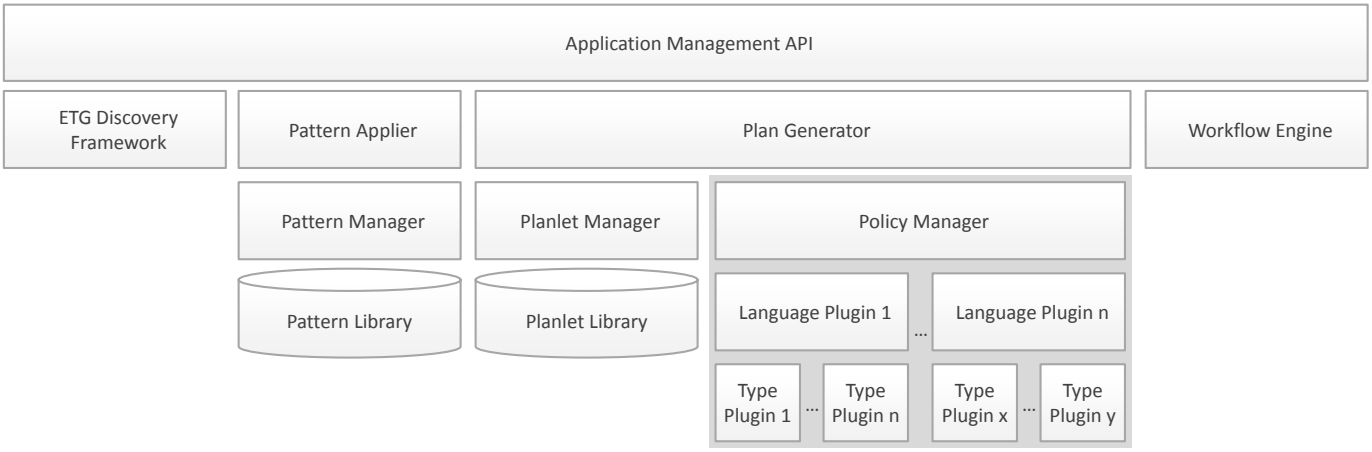

Figure 19. Policy-Aware Management Framework architecture.

inside, which is responsible for scheduling planlets in the right order. We extend this orchestrator by the integration of a new component called *Policy Manager* that is responsible for policy matchmaking and invoking the corresponding *Language Plugins* or *Type Plugins*, respectively. Each planlet that is analyzed by the orchestrator gets 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 Planlet Framework was built in a modular way. To execute the generated plans, a workflow engine is employed. All generated workflows are deployed on this engine to execute the implemented tasks fully automatically. To create DASMs automatically for provisioning and management tasks, the framework provides a pattern layer that consists of a *Pattern Library* that is managed by a *Pattern Manager* component, which is responsible for retrieving Automated Management Patterns. Thereon, a component called *Pattern Applier* is responsible to find the AMPs that are applicable to a certain ETG and to invoke the pattern's Topology Transformation to generate the corresponding DASM.

B. Plan Generator Extension

The Plan Generator of the framework tries to find appropriate Management Planlets that can be orchestrated to execute the Management Annotations defined in the Desired Application State Model. 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 explained in Section IV-C. 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 Desired Application State Model. This means, that all preconditions of the planlet are fulfilled and that the management tasks that 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 [5]. The calculation of potential candidate planlets is extended by policy processing: if a Management Planlet executes a Management Annotation in the DASM that is bound to a Management Annotation Policy, i. e., that is contained in the policy's AppliesTo-list, the policy needs to be processed as defined by the processing mode attribute and the Management Annotations specified in its AppliesTo-list. This is required to analyze if the candidate planlet fulfills all defined requirements. How to deal with this processing mode attribute during matchmaking is explained in the next section.

C. Language and Type plugins

The processing mode attribute of a Management Annotation 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 that have to be fulfilled. These are passed to the planlet via its input message from the calling plan. This enables configuration 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.

D. Lessons learned

In this section, we describe our experiences from the implementation of Policy-Aware Management Planlets and Management Annotation Policies. 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 Policy-Aware Management Planlet is capable of recognizing if a Management Annotation Policy is optional.

In many cases, extending planlets to Policy-Aware Management 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 Secure Password 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.

IX. 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 feasibility of the approach, we evaluated the framework in terms of the three kinds of Management Policies discussed in Section II-D. We implemented planlets fulfilling the policy examples that were used throughout the paper. To prove the feasibility of Configuring Policies, we implemented the planlet described in Figure 12 that creates a virtual machine with an Ubuntu Linux operating system on Amazon EC2 complying with a Data Location Policy that defines that all data must remain in the European Union. The planlet creates the VM using the Amazon Web Services API and specifies that the virtual machine shall be provisioned in the EU. 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 Secure Password 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 Ubuntu complying with this policy. Such a policy can be implemented by a planlet in two different ways: (i) either the planlet requires username and password as input data (which may be taken from properties in the DASM or exposed to the input message of the generated plan), then the planlet checks if the strength is strong enough or (ii) the planlet sets the credentials with a high strength itself. Both implementations guard the provisioning in a way, that the password is strong enough. To prove the feasibility of Extending Policies, we implemented a Frequent Data Backup Policy that is attached to a MySQLDB node. The corresponding planlet executes an additional bash script as 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 [20].

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 Management 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 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 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 and management. 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 [32] analyzed the automation of operational processes and how this influences costs. They found that three issues must be considered that 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 Management Plans. They are developed by expert users of various domains and provided to communities. Therefore, free accessible planlet libraries enable continuous maintenance without the need for individual effort. Of course, maintaining local management infrastructure and the development of 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 there are tools for the modeling of application topologies and policies and the discovery of ETGs: the TOSCA modelling tool *Winery* [33] supports modelling of TOSCA-based application topologies that can be used as import format for our framework. Winery also supports modelling and attaching policies to elements in the topology. As TOSCA policies provide a content field to fill in any policy-related information, Management Annotation Policies can be specified using this field. Thus, to define declarative provisionings in the form of application topologies, this tool eases the creation of the corresponding models. To discover ETGs, we presented a ETG Discovery Framework in Binz et al. [15]. This framework enables discovering ETGs fully automatically. Therefore, only an entry point of the application has to be specified, e. g., the URL of the application. The framework discovers the corresponding topology including all runtime information fully automatically. 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 Management 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 Management Plans must be possible within a few seconds to obtain Cloud properties such as scalability or on-demand self-service. The employed Management Planlet

Framework presented in Section IV uses a partial order planning algorithm [34] for the generation of Management Plans [5]. As described by Bylander [35], 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 have multiple planlets providing the same or overlapping functionality in the planlet library. For example, it is forbidden to have two planlets installing a MySQL database on an Ubuntu operating system that differ only in the properties they set. This eliminates the non-deterministic choices that have to be made during the plan generation in terms of selecting planlets: each Management Annotation can be processed by exactly one planlet. This decreases the complexity to polynomial time [34]. Extending the framework by policies must follow this restriction: if a Policy-Aware Management Planlet implements a functionality that is provided already by an existing planlet, the existing planlet has to be merged with the new Policy-Aware Management 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 (cf. Section VIII-D). The added policies should be implemented and declared as optional. The only difference is additional effort as calling plugins might be necessary. In worst case, each Management Annotation Policy in a DASM must be processed by one plugin. As the number n of policies attached to elements in a DASM is constant, the extension has no influence on the complexity.

D. Prototype

To validate the concept technically, we implemented the approach on the basis of the Policy-Aware Management Framework architecture presented in Section VIII. The prototype is based on former implementations of the Management Planlet Framework and, therefore, implemented in Java and uses OSGi in order to provide a flexible and dynamic plugin system. Management Planlets are implemented in the Business Process Execution Language (BPEL) [17] whereas DASMs and Annotated Topology Fragments of planlets are implemented using an internal data model similar to the structure of TOSCA [20]. In our prototype extension, we extended this meta-model with the possibility to attach Management Annotation Policies to nodes and relations of topologies. 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 VIII-C. To prove the technical feasibility of the conceptually evaluated policies described in Section V, we implemented several policies by extending already existing planlets to Policy-Aware Management Planlets. In addition, we modified existing Automated Management Patterns to consider attached policies. The successful implementation of this prototype proves the technical feasibility.

E. Extensibility

As there are many different existing policy types and languages, the presented approach must support extensibility. The Management Planlet Framework (cf. Section IV) 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 Management Annotation 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.

F. Limitations

The presented approach has some limitations that are discussed in this section. First, the kinds of Management Policies that are currently supported by the presented approach are limited to policies that can be considered by a single Policy-Aware Management Planlet. For example, a "MustExistOnlyOnce-Policy" is not possible using the presented approach as this policy currently cannot be enforced by one Management Planlet: planlets are only able to consider the policies, elements, and properties in the DASM that directly match their Annotated Topology Fragments. Thus, if there are two MustExistOnlyOnce-Policies attached to elements in the DASM, a Policy-Aware Management Planlet that creates one element cannot be aware of these kinds of conflicting policies. We plan to solve that issue be extending the matchmaking rules of Policy-Aware Management Planlets that can be currently defined only by the Annotated Topology Fragment, which is sufficient for most policy types but not for all.

Second, the approach currently allows modelling conflicting policies in DASMs: administrators as well as Automated Management Patterns may specify policies that are in conflict with each other, i. e., they apply to the same Management Annotation but specify conflicting requirements on the execution of this annotation. Because the framework is of generic nature, it is not able to detect these conflicts in DASMs at design time as it is not aware of the semantics of the policies. If the conflicting policies specify the processing mode *Type Equality*, this is only a modelling issue and does not lead to policy violations as a Policy-aware Management Planlet would need to be found that executes this Management Annotation while complying with both policies. However, as the policies are in conflict, such a planlet cannot exist if it is implemented correctly. Thus, the plan generation fails as no Policy-aware Management Planlet can be found to execute the Management Annotation while enforcing both conflicting policies. As a result, the application's policies are not violated as nothing will be executed on the real running application. To ensure this, a Policy-aware Management Planlet is not allowed to specify conflicting policies on the Management Annotations it executes—even not if they are declared as optional. If a Management Annotation Policy is checked by a language or type plugin, i. e., possibly a *normal* planlet that does not specify policies at all is evaluated if it fulfills the topology policies for a certain Management Annotation, the plugin is responsible for analyzing also the other policies that are bound to the

corresponding Management Annotation. If the plugin cannot guarantee that the policy it is responsible for is not in conflict with the other declared policies for a certain candidate planlet, it has to reject the planlet. Thus, plugins must be implemented carefully. Especially the implementation of language plugins, which typically do not understand the defined requirements and only execute generic matchmaking operations on the language-specific content of policies, must be very defensive: if there are other policies associated with the same Management Annotation that are not implemented in the same policy language, they have to reject each candidate planlet as they cannot ensure that the other policies are not in conflict with the policy it is responsible for. Only if other policies are specified in the same language, language plugins may be able to analyze them. But this mainly depends on the policy language and its semantics.

Third, the presented approach currently provides a basic framework that supports defining security policies on the execution of provisioning as well as management tasks. This requires an explicit specification of the (i) tasks to be performed and the (ii) Management Annotation Policies that must be considered by the corresponding tasks. Although the Management Planlet Framework, in particular the concept of Automated Management Patterns, enables specifying low-level management tasks automatically, security issues must be specified manually by developers and administrators by declaring appropriate Management Annotation Policies on the affected components, relations, and Management Annotations. This requires security expertise and causes effort to apply well-known security concepts and methodologies, e. g., for applying security patterns [36], to individual applications using our framework. In addition, Management Annotation Policies may be specified that cannot be processed by the framework, i. e., no capable planlet is available in the library. This leads to DASMs that cannot be transformed into the corresponding, executable Management Plan. We plan to tackle these issues in the future by (i) supporting developers and administrators in expressing security requirements as Management Annotation Policies and (ii) employing the concept of Automated Management Patterns to automatically attach policies and management tasks to individual applications. Especially the latter one may be a powerful way to enable secure management automation. For example, automated migration patterns could directly attach additional security policies to the components to be migrated that specify possible geographic regions. This enables capturing and automating (i) management expertise as well as (ii) security expertise, e. g., by implementing Automated Management Patterns that execute a sophisticated management functionality while complying with certain laws.

X. Related Work

There are several works focusing on the automated provisioning and management of Cloud applications. In this section, we describe the most related ones and compare them to our approach. The work of Eilam et al. [37] focuses on deployment of applications by orchestrating low-level operation logic similarly to planlets by so-called *automation signatures*. El Maghraoui et al. [38] 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—especially not in the form of explicitly attached policies, which are able to define the tasks that must consider the policy. In contrast to both works, Management Annotation Policies enable application developers to bind policies directly to the abstract tasks that must comply with the policy. Thus, Policy-aware Management Planlets introduce an additional layer of abstraction in terms of defining and processing non-functional security requirements.

Mietzner and Leymann [39] 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. In addition, our approach supports also management of applications.

The Composite Application Framework (Cafe) [40] 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 and management as enabled by our approach.

Mietzner et al. [41] present ProBus, a standards-based extended enterprise service bus that is capable of policy-based service and resource selection to optimize service selection in dynamic environments. ProBus enables clients to submit service invocation requests that include policies, to which service providers need to comply with, in one message. They show how these policies can be evaluated by ProBus and, in addition, how policies can be used to define non-functional requirements on stateful resources.

The CHAMPS System [42] 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* that 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 these artifacts.

Kirschnick et al. [43] 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 various kinds of different XaaS offerings. Thus, the system is not able to provision Cloud

applications that consist of several XaaS offerings in compliance with non-functional security requirements defined as policies. In addition, the framework currently supports no management.

The DevOps community provides tooling to automate configuration management of Cloud applications. To mention the most important, Chef [44] and Puppet [45] 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—management functionalities. This enables to integrate management logic that considers non-functional requirements. However, all these frameworks focus on a deep technical level of management and do not provide a means to express and integrate non-functional security requirements on such a high level as enabled by Management Annotation Policies and Policy-Aware Management Planlets. The reusability in terms of managing different applications, the interoperability between script-based and non-script-based technologies as needed to provision and manage complex composite Cloud applications, and the holistic integration of different policy languages is not supported yet. Of course, script-based frameworks can support the policies presented in this paper directly implemented in the affected scripts.

The Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard to describe composite Cloud applications and their management [20]. It tackles current challenges in Cloud Computing such as portability and interoperability of Cloud applications, prevention of vendor lock-in, and the automated provisioning and management of applications [46][47]. TOSCA specifies an XML-based format for describing Cloud applications as application topologies and enables the management of applications through Management Plans, which 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 attach policies to the topology but lacks a detailed description of their processing. To tackle this issue, we demonstrate how non-functional requirements on the provisioning and management of applications can be defined in TOSCA using policies and propose a mechanism for automatic processing of formal policy definitions in Waizenegger et al. [48]. However, this approach is based on manually authoring Management Plans which is time-consuming, complex, and error prone [7].

In Waizenegger et al. [49], we presented a framework architecture for the provisioning and management of Cloud services and applications based on Management Plans that support processing non-functional security requirements in the form of policies. However, the framework employs only the concept of plans without stating how these plans have to be created. This issue is tackled by the presented Policy-Aware Management Planlets approach presented in this article that enables a fully automated generation of policy-aware Provisioning as well as Management Plans. We additionally defined in Waizenegger et al. [49] different stages in the lifecycle of applications where policies may be defined, the layer of the topology to which the policy applies, and classified fundamental effects of policies. Similarly to this article, policies are attached to elements in topology models if they are targeted to the corresponding element directly. In Waizenegger et al. [49], we showed also how policies may be attached to the topology itself to specify non-functional requirements that do not affect only one single element. We plan to extend the Management Planlet Framework to support these *global policies*, too.

In Breitenbücher et al. [22], we present an approach to combine declarative and imperative provisioning of Cloud applications based on a plan generator for Provisioning Plans, which is based on a similar concept as planlets. The plan generator is implemented in the open source TOSCA runtime environment *OpenTOSCA* [50]. However, the plan generator currently supports only provisioning of TOSCA-based applications, neither the management of applications nor non-functional requirements in the form of Management Policies.

XI. Conclusion and Future Work

In this paper, we presented an approach that enables to automate the provisioning and management of composite Cloud applications in compliance with non-functional security requirements defined by policies. We extended the Management Planlet Framework to support policy-aware provisioning and management based on a certain kind of Management Policies that enables binding non-functional security requirements directly to the management tasks that must enforce them. We introduced a new format for Management Policies that are considered by Policy-aware Management Planlets during the execution of management tasks. In addition, the extended framework allows Cloud providers as well as application developers to implement their own policy-aware management logic in a flexible and reusable manner independently from individual applications. The paper evaluates the presented approach in terms of performance, feasibility, economics, limitations, 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. This extension shall enable to define global policies that are attached to the topology itself.

Acknowledgment

This work was partially funded by the BMWi project CloudCycle (01MD11023) and by the Co.M.B. project of the Deutsche Forschungsgemeinschaft (DFG) under the promotional reference SP 448/27-1.

References

[1] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and M. Wieland, "Policy-Aware Provisioning of Cloud Applications," in *SECURWARE*. Xpert Publishing Services, August 2013, pp. 86–95.

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

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

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

[5] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, "Pattern-based Runtime Management of Composite Cloud Applications," in *CLOSER*. SciTePress, May 2013, pp. 475–482.

[6] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and J. Wettinger, "Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies," in *CoopIS*. Springer, September 2013, pp. 130–148.

[7] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and M. Wieland, "Context-Aware Cloud Application Management," in *CLOSER 2014*. SciTePress, April 2014, pp. 499–509.

[8] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, "Automating Cloud Application Management Using Management Idioms," in *PATTERNS*. Xpert Publishing Services, May 2014, pp. 60–69.

[9] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and D. Schumm, "Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA," in *CoopIS*. Springer, September 2012, pp. 416–424.

[10] Amazon Elastic Compute Cloud (Amazon EC2). [Online]. Available: http;//www.aws.amazon.com/ec2

[11] T. Binz, C. Fehling, F. Leymann, A. Nowak, and D. Schumm, "Formalizing the Cloud through Enterprise Topology Graphs," in *CLOUD*. IEEE, June 2012, pp. 742–749.

[12] V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch, "How to Adapt Applications for the Cloud Environment," *Computing*, vol. 95, pp. 493–535, 2013.

[13] T. Binz, F. Leymann, A. Nowak, and D. Schumm, "Improving the Manageability of Enterprise Topologies Through Segmentation, Graph Transformation, and Analysis Strategies," in *EDOC*, September 2012, pp. 61–70.

[14] A. Nowak, T. Binz, F. Leymann, and N. Urbach, "Determining Power Consumption of Business Processes and Their Activities to Enable Green Business Process Reengineering," in *EDOC*. IEEE, September 2013, pp. 259–266.

[15] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "Automated Discovery and Maintenance of Enterprise Topology Graphs," in *SOCA*. IEEE, December 2013, pp. 126–134.

[16] F. Leymann and D. Roller, *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000.

[17] OASIS, *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, OASIS Std., April 2007.

[18] OMG, *Business Process Model and Notation (BPMN), Version 2.0*, Object Management Group Std., Rev. 2.0, January 2011.

[19] A. Keller and R. Badonnel, "Automating the Provisioning of Application Services with the BPEL4WS Workflow Language ," in *DSOM*. Springer, November 2004, pp. 15–27.

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

[21] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications," in *Business Process Model and Notation*. Springer, September 2012, pp. 38–52.

[22] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA," in *IC2E*. IEEE, March 2014, pp. 87–96.

[23] R. Boutaba and I. Aib, "Policy-based Management: A Historical Perspective," *Journal of Network and Systems Management*, vol. 15, no. 4, pp. 447–480, December 2007.

[24] R. Wies, "Using a Classification of Management Policies for Policy Specification and Policy Transformation," in *IFIP/IEEE IM*, June 1995, pp. 44–56.

[25] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996.

[26] C. Fehling, F. Leymann, S. T. Ruehl, M. Rudek, and S. Verclas, "Service Migration Patterns - Decision Support and Best Practices for the Migration of Existing Service-based Applications to Cloud Environments," in *SOCA*. IEEE, December 2013, pp. 9–16.

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

[28] Amazon Simple Storage Service (Amazon S3). [Online]. Available: http;//www.aws.amazon.com/s3

[29] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, January 2014.

[30] C. Fehling, F. Leymann, J. Rütschlin, and D. Schumm, "Pattern-Based Development and Management of Cloud Applications." *Future Internet*, vol. 4, no. 1, pp. 110–141, 2012.

[31] C. Alexander, *The Timeless Way of Building*. Oxford University Press, 1979.

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

[33] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery – A Modeling Tool for TOSCA-based Cloud Applications," in *ICSOC*. Springer, December 2013, pp. 700–704.

[34] D. S. Weld, "An Introduction to Least Commitment Planning," *AI Magazine*, vol. 15, no. 4, pp. 27–61, Winter 1994.

[35] T. Bylander, "Complexity Results for Planning," in *IJCAI*. Morgan Kaufmann, August 1991, pp. 274–279.

[36] A. V. Uzunov, E. B. Fernandez, and K. Falkner, "Securing distributed systems using patterns: A survey," *Computers & Security*, vol. 31, no. 5, pp. 681–703, May 2012.

[37] T. Eilam, M. Elder, A. Konstantinou, and E. Snible, "Pattern-based Composite Application Deployment," in *IM*. IEEE, May 2011, pp. 217–224.

[38] 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*. Springer, November 2006, pp. 404–423.

[39] 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*. IEEE, July 2008, pp. 3–10.

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

[41] R. Mietzner, T. van Lessen, A. Wiese, M. Wieland, D. Karastoyanova, and F. Leymann, "Virtualizing Services and Resources with ProBus: The WS-Policy-Aware Service and Resource Bus," in *ICWS*. IEEE, July 2009, pp. 617–624.

[42] 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*. IEEE, April 2004, pp. 395–408.

[43] 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, pp. 124–131, December 2010.

[44] Opscode, Inc., "Chef Official Site," 2012. [Online]. Available: http://www.opscode.com/chef

[45] Puppet Labs, Inc., "Puppet Official Site," 2012. [Online]. Available: http://puppetlabs.com/puppet/what-is-puppet

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

[47] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, ser. Advanced Web Services. Springer, January 2014, pp. 527–549.

[48] T. Waizenegger *et al.*, "Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing," in *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*. Springer, September 2013, pp. 360–376.

[49] T. Waizenegger, M. Wieland, T. Binz, U. Breitenbücher, and F. Leymann, "Towards a Policy-Framework for the Deployment and Management of Cloud Services," in *SECURWARE*. Xpert Publishing Services, August 2013, pp. 14–18.

[50] T. Binz *et al.*, "OpenTOSCA – A Runtime for TOSCA-based Cloud Applications," in *ICSOC*. Springer, December 2013, pp. 692–695.

All links were last followed on May 30, 2014.