A Method to Automate Cloud Application Management Patterns

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

***Abstract*—Management patterns are a well-established concept to document reusable solutions for recurring application management issues in a certain context. Their generic nature provides a powerful means to describe application management knowledge in an abstract fashion that can be refined for individual use cases manually. However, manual refinement of abstract management patterns for concrete applications prevents applying the concept of patterns efficiently to the domain of Cloud Computing, which requires a fast and immediate execution of arising management tasks. Thus, the application of management patterns must be automated to fulfill these requirements. In this paper, we present a method that guides the automation of Cloud Application Management Patterns using the Management Planlet Framework, which enables applying them fully automatically to individual running applications. We explain how existing management patterns can be implemented as Automated Management Patterns and show how these implementations can be tested afterwards to ensure their correctness. To validate the approach, we conduct a detailed case study on a real migration scenario.**

Keywords*—*Application Management; Cloud Computing; Management Patterns; Management Automation.

I. Introduction

Management patterns are a well-established concept to document reusable solution expertise for frequently recurring application management problems in a certain context [1]. They provide the basis for the implementation of management processes and influence the architecture and design of applications. The generic nature of patterns enables management experts to document knowledge about proven solutions for challenging management issues in an abstract, structured, and reusable fashion. This supports application managers in solving concrete instances of the general problem. Applying management patterns, e. g., to scale or to migrate application components, to concrete real use cases in the form of running applications requires, therefore, typically a *manual refinement* of the pattern's abstract high-level solution towards the individual use case [2]. However, the manual refinement and application of management patterns is time-consuming and, therefore, not appropriate in the domain of Cloud Computing since the immediate and fast execution of arising management tasks is of vital importance to achieve Cloud properties such as pay-as-you-go pricing models and on-demand computing [1]. This is additionally underscored by the fact that human errors are the largest cause of failures of internet services and large systems [3][4]. Especially the rapid evolution of management technologies additionally strengthens this effect: complex management tasks can be executed much easier and quicker due to powerful management interfaces offered by Cloud providers that abstract from technical details. However, this increases the probability of human errors because there is hardly any notion of the underlying physical infrastructure and the actual impact the executed tasks may have [1]. As a consequence, to use the concept of patterns efficiently in the domain of *Cloud Application Management*, the (i) refinement of management patterns for individual use cases as well as (ii) the execution of the refined solution must be *automated* since manual realizations are too slow, costly, and error prone [2].

However, the difficulties of automating management patterns are manifold. Especially the immense technical expertise required to refine a pattern's abstract solution towards a concrete use case is one of the biggest challenges in terms of automation. To tackle these issues, we presented the pattern-based *Management Planlet Framework* in former works [2][5][6][7][8], which enables applying management patterns automatically to concrete running applications for executing typical management tasks such as migrating applications or updating components without downtime [2]. The framework employs so called *Automated Management Patterns*, which implement a certain management pattern in a way that enables its application to various individual use cases either semi-automatically or even fully-automatically. However, the implementation of these automated patterns is a non-trivial task that requires special attention to ensure a high quality and correctness of their automated executions on real applications. This issue is tackled in this paper. We present a method that enables automating *Cloud Application Management Patterns* using the Management Planlet Framework introduced above. We show how management patterns described in natural text can be analyzed and implemented in a generic way that enables applying the captured solution logic automatically to concrete use cases in the form of running applications—independently from individual manifestations. To guide this analysis, the method describes how the relevant information required to automate a management pattern can be extracted from its textual description. The presented method can be used to automate various kinds of management patterns, which enables applying this concept efficiently in the domain of Cloud Application Management. We prove the feasibility of our approach by a detailed case study that considers the automation of an existing migration pattern and various applications of the presented method to automate other management patterns.

The remainder of this paper is structured as follows: in Section II, we explain the employed Management Planlet Framework, which provides a generic means to automatically apply management patterns to individual applications. Section III presents the main contribution of this paper in the form of a method to automate existing Cloud Application Management Patterns using the employed Management Planlet Framework. We conduct a detailed case study in Section IV to illustrate how the method can be applied to automate an existing migration pattern. Section V discusses related work. Section VI concludes the paper and provides an outlook on planned future work.

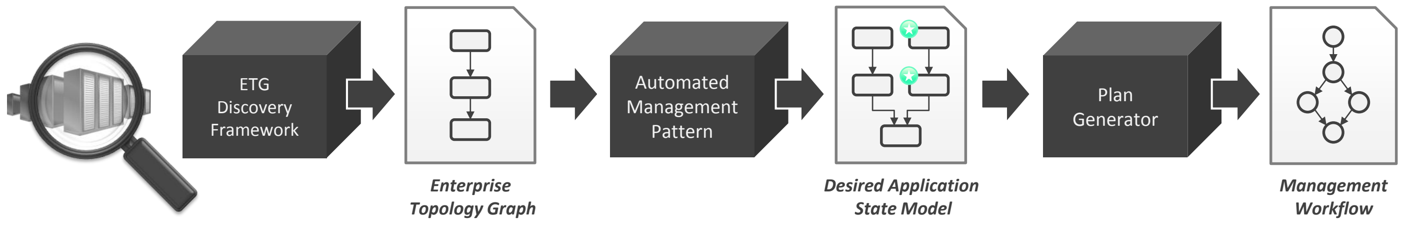
ETG Discovery Framework
Enterprise Topology Graph
Automated Management Pattern
Desired Application State Model
Plan Generator
Management Workflow

Figure 1. Architecture and concept of the Management Planlet Framework (adapted from [2][6][7][8]).

II. EMPLOYED MANAGEMENT FRAMEWORK

In this section, we present the employed *Management Planlet Framework* [2][5][6][7][8] that provides the basis for automating management patterns. The framework can be used for the (i) initial provisioning of applications [7] as well as (ii) for runtime management of applications [6], which is the focus of this paper. The framework's management approach is shown in Figure 1 and can be summarized as follows: first, the application to be managed is captured by a formal model called *Enterprise Topology Graph*. In the second step, an *Automated Management Pattern* is applied that transforms this Enterprise Topology Graph into a *Desired Application State Model*, which declaratively specifies the management tasks to be performed. This DASM is then transformed into an executable *Management Workflow* by a *Plan Generator* in the last step. In the following, we explain these steps and the involved artifacts in detail.

A. Enterprise Topology Graph (ETG)

An Enterprise Topology Graph (ETG) [9] is a formal model that describes the current structure of a running application including its state. ETGs are modelled as directed graphs that consist of nodes and relations (*elements*) representing the application's components and dependencies. Each element has a certain type and provides properties that capture runtime information. For example, a node may be of type "VirtualMachine" and provides properties such as its "IP-Address". ETGs can be discovered fully automatically using the ETG *Discovery Framework* [10]. This framework only requires an entry point of the application, e. g., the URL of an application's Web frontend, to discover the whole ETG fully automatically including all software and infrastructure components of the application.

B. Automated Management Patterns (AMP)

An Automated Management Pattern (AMP) is a generic implementation of a management pattern that can be applied automatically to individual applications that match a predefined application structure, e. g., to migrate an application without downtime. An AMP consumes the ETG of the application to which the implemented pattern shall be applied and automatically specifies the management tasks that have to be performed on the application's nodes and relations. Therefore, AMPs consist of two parts: the (i) *Topology Fragment* describes the application structure to which an AMP can be applied, i. e., it models the nodes and relations that must match elements in an ETG to apply the pattern to these matching elements. The (ii) *Topology Transformation* consumes the application's ETG and automatically creates a *Desired Application State Model*, in which it specifies the management tasks to be performed in the form of abstract *Management Annotations* that are declared by the transformation on nodes and relations of the ETG.

C. Desired Application State Model (DASM)

A Desired Application State Model (DASM) describes management tasks to be performed on nodes and relations of a running application in a *declarative* manner. It consists of (i) the application's ETG and (ii) *Management Annotations*, which are declared on nodes or relations of the ETG. A Management Annotation (depicted as coloured circle) specifies a small management task to be executed on the associated element, but defines only the *abstract semantics* of the task, e. g., that a node shall be created, but not its technical realization. For example, a *Create-Annotation* attached to a "MySQLDatabase" that has a "hostedOn" relation to an "UbuntuVM" means that the database shall be installed on the VM. Similarly, there are annotations that specify specific management tasks, e. g., an *ImportData-Annotation* attached to a database defines that data has to be imported. Management Annotations may additionally define that they must be executed before, after, or concurrently with another annotation. Due to the declarative nature of DASMs, only the *what* is described, but not the *how*. Thus, in contrast to imperative descriptions such as executable workflows [11] that define all technical details, DASMs can not be executed directly and are transformed into workflows by the *Plan Generator*.

D. Management Planlets & Plan Generator

In the last step, the created DASM is automatically transformed into an executable Management Workflow. This is done by the framework's Plan Generator, which orchestrates so called *Management Planlets*. A Management Planlet is a small workflow that executes one or more Management Annotations on a certain combination of nodes and relations. For example, a Planlet may deploy a Java application on a Tomcat Webserver. The Plan Generator tries to find a suitable Management Planlet for each Management Annotation specified in the DASM that executes the corresponding management task. Thus, Management Planlets are reusable building blocks that provide the low-level imperative management logic to execute the declarative Management Annotations declared in DASMs.

E. Development of Automated Management Patterns

DASMs provide the basis to implement AMPs on a high-level of abstraction: management tasks need to be specified only abstractly in the form of declarative Management Annotations without the need to deal with the complex, low-level, and technical issues required for their execution. These technical details are considered only by the responsible Management Planlets. However, since management patterns typically capture multiple steps to be performed, the development of AMPs is a challenging task and needs careful consideration. Therefore, we present a method that guides the development of AMPs.

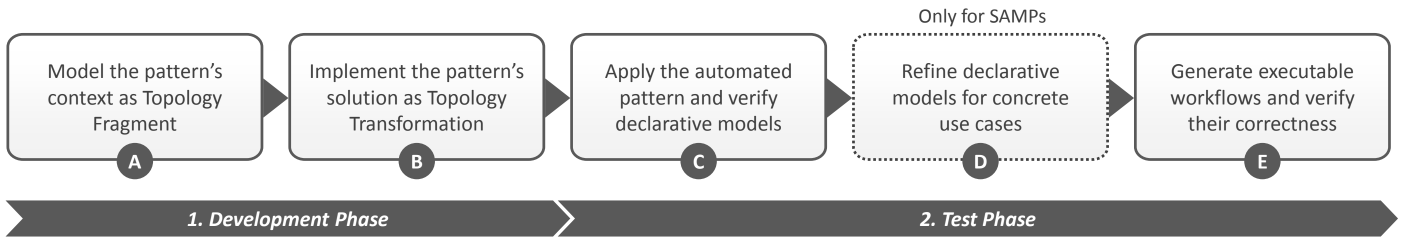
Only for SAMPs
Model the pattern's context as Topology Fragment
A
Implement the pattern's solution as Topology Transformation
B
Apply the automated pattern and verify declarative models
C
Refine declarative models for concrete use cases
D
Generate executable workflows and verify their correctness
E
1. Development Phase
2. Test Phase

Figure 2. Method to automate existing Cloud Application Management Patterns.

III. A Method to Automate Cloud Application Management Patterns

In this section, we present a method to automate existing management patterns through implementing them as AMPs. We distinguish here between two kinds of AMPs: *Semi-Automated Management Patterns* (SAMPs) [6] implement patterns on an abstract level, e. g., to migrate any kind of application to another location. Therefore, applying SAMPs typically requires a manual refinement of the resulting DASMs before they can be transformed into the corresponding workflows, e. g., abstract nodes types must be replaced, additional nodes or relations have to be inserted, or further Management Annotations have to be added. An *Automated Management Idiom* (AMI) [2] implements a refined version of a management pattern for a concrete use case, e. g., to migrate a Java application hosted on an Apache Tomcat Webserver to the Amazon Cloud. As a consequence, applying AMIs results in already refined DASMs that can be translated directly into workflows. The method is shown in Figure 2 and consists of two phases: in the (i) *Development Phase*, the management pattern to be automated is analyzed and implemented as SAMP or AMI. In the (ii) *Test Phase*, the implementation is verified for correctness. In the following subsections, we explain the five steps of the method in detail.

A. Model the pattern's context as Topology Fragment

Analyze the management pattern's context with focus on the application structures to which the pattern can be applied and model this information as Topology Fragment.

Cloud Application Management Patterns typically consist of several parts that describe the pattern in natural text [1]. In the method's first step, the textual description of the pattern to be automated is analyzed in terms of the application structures to which the AMP shall be applicable. This information is then modelled as Topology Fragment. If a concrete AMI shall be created, the fragment typically needs to be refined for the respective use case. For example, the aforementioned refinement of the pattern that migrates a Java application to the Amazon Cloud results in a Topology Fragment that models a node of type "JavaApplication" that has a relation of type "hostedOn" to a node of type "ApacheTomcat". This AMI is then applicable to all ETGs that contain this combination of elements. Thus, as the Topology Fragment provides the basis for matchmaking SAMPs and AMIs with ETGs, it must define exactly the nodes and relations to which the automated pattern is applicable. This kind of information is typically described in the *context* section of management patterns, but also other sections such as *problem* or even the *solution* can be used to extract this information.

B. Implement the pattern's solution as Topology Transformation

Analyze the management pattern's solution and implement the described management logic as Topology Transformation.

In the second step, the pattern's solution logic is captured in a way that enables its automated application to individual applications. Therefore, the textual description of the pattern's solution is analyzed in terms of the management tasks that have to be executed to apply the pattern. The analyzed procedure is then implemented as Topology Transformation that acts on the nodes and relations defined by the Topology Fragment: the Topology Transformation must declare exactly the Management Annotations on the ETG that declaratively specify the analyzed management tasks to be executed following the pattern's solution. In case of automating a management pattern as abstract SAMP, the Topology Transformation implements only the pattern's original abstract solution logic and remains rather vague: executing this transformation on an ETG typically results in a DASM that additionally needs to be refined manually afterwards. If the pattern is automated as detailed AMI for a concrete use case, the abstract solution logic must be first (i) refined towards this use case in order to provide detailed information about the management tasks to be executed. These are (ii) implemented afterwards in the Topology Transformation that specifies the corresponding Management Annotations. Executing such fine-grained AMI-transformations typically results in fully refined DASMs that can be transformed directly into executable workflows without further manual effort.

C. Apply the automated pattern and verify declarative models

Apply the created SAMP / AMI to concrete use cases and compare the resulting DASMs with the solution described by the original pattern or refinement to verify their correctness.

After the textual description of the pattern is translated into its corresponding SAMP / AMI, the correctness of the realization needs to be verified. Therefore, in the first part of the *Test Phase*, the implemented patterns are tested against various concrete use cases, i. e., ETGs of different running applications. First, a set of appropriate use cases must be identified that captures all possible application structures to which the pattern can be applied and that are affected by the pattern's transformation. This requires an explicit and careful analysis of the pattern's Topology Fragment and Topology Transformation to cover all possible scenarios. Additional use cases must be identified to which the pattern can *not* be applied to additionally ensure the correctness of the pattern's Topology Fragment. Afterwards, the pattern is tested against these cases. The test is subdivided into two steps: (i) testing the Topology Fragment and (ii) testing the Topology Transformation. In the first step, the automated pattern is applied to different use cases which are not all suited for the pattern. Thus, some use cases match with the Topology Fragment, others not. The results of these matchmakings are then compared with the semantics of the original pattern or refinement, respectively, to verify the correct modelling of the Topology Fragment: the Automated Management Pattern must be applicable exactly to the same use cases as the original pattern or the refinement, respectively. In the second step, the Topology Transformation is executed on the correctly matching use cases. The resulting DASMs are then compared with (i) the *solution* section of the original pattern / refinement to verify if the declaratively specified tasks comply with the description and (ii) the *result* section to verify that the results are equal. This is possible as the created DASMs contain information about both solution and result: they describe the management tasks to be executed as well as the final application structure and partially the application's state after executing the workflow.

D. Refine declarative models for concrete use cases

Only for creating SAMPs: refine the resulting DASMs for concrete use cases in order to provide all missing information required to generate executable workflows.

If a pattern is semi-automated as SAMP, the DASMs resulting from the previous steps cannot be translated directly into executable workflows as the refinement is missing: the SAMP's Topology Transformation implements only the pattern's abstract solution and provides not all information required to generate the workflow. As a result, the resulting DASMs must be refined manually in this step for providing all required information. DASMs resulting from the application of AMIs are not affected.

E. Generate executable workflows and verify their correctness

Transform the final DASMs into the corresponding workflows, compare them with the solution described by the original pattern or refinement, and verify the result of their execution.

In the last step, the final DASMs resulting from the previous steps are transformed into the corresponding executable Management Workflows using the framework's Plan Generator. Then, the correctness of these imperative management description models are verified by two manual steps: (i) verifying the correctness of the generated workflow implementations and (ii) verifying the correctness of the final application states after executing the Management Workflows. In the first step, the implementation of the generated workflows are compared with the abstract *solution* of the pattern or its refined incarnation if the tested pattern is implemented as AMI. This last verification ensures that the finally executed management tasks including all technical details are correct. In particular, this step is required to ensure that the employed Management Annotations lead to correct workflows, e. g., that there are Management Planlets available to execute all the Management Annotations declared in the DASMs. In the second verification step, the generated Management Workflows are executed on the real running test applications and the results are compared with the *result* section of the original management pattern or the refined result if the tested automated pattern is implemented as AMI.

IV. Case Study and Validation

In this section, we validate the approach by a detailed case study that considers the automation of an existing management pattern using the proposed method. Due to the important issue of vendor lock-in in the domain of Cloud Computing, we automate a migration management pattern. First, we describe the most important facts of the original pattern and derive a refined idiom afterwards that enables migrating Java-based applications to the Amazon Cloud. The refined idiom is then implemented as Automated Management Idiom using the presented method.

The pattern to be automated is called *Stateless Component Swapping Pattern* [12] and originates from the Cloud Computing pattern language developed by Fehling et al. [1][12][13]. The pattern deals with the problem *"How can stateless application components that must not experience downtime be migrated?"*. The context observed is that for many business applications downtime is unacceptable, e. g., for customer-facing applications. Hence, its intent is migrating stateless applications from one environment into another transparently to the accessing human users or other applications. Therefore, the stateless application is active in both environments concurrently during the migration to avoid downtime. Here, "stateless" means that the application does not handle internal session state [12].

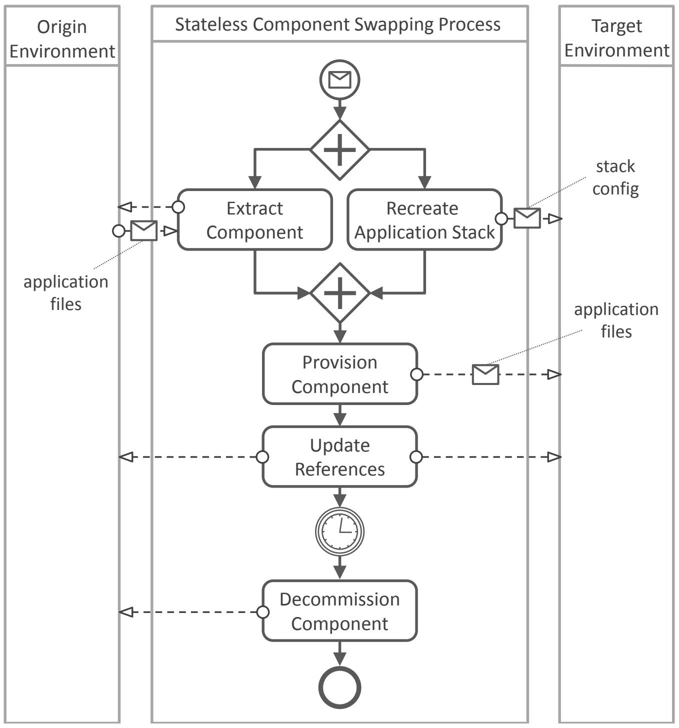

Figure 3. Abstract Stateless Component Swapping Process (adapted from [12]).

Figure 3 describes the pattern's solution as Business Process Model and Notation (BPMN) [14] diagram: first, the component to be migrated is extracted from the origin environment while the required application stack is provisioned concurrently in the target environment. After the new application stack is provisioned, a new instance of the component is deployed thereon while the original component is still active. When deployment has finished and the new component is running, all references pointing to the old component are updated to the new one and the original component gets decommissioned.

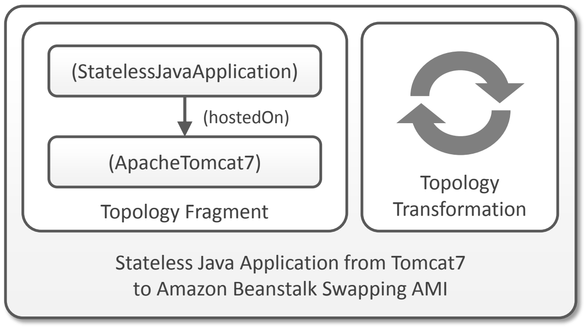
(StatelessJavaApplication)
(hostedOn)
(ApacheTomcat7)
Topology Fragment
Topology Transformation
Stateless Java Application from Tomcat7 to Amazon Beanstalk Swapping AMI

Figure 4. Automated Management Idiom.

Since this case study aims at fully automating this management pattern, we implement it as Automated Management Idiom using the presented method. Therefore, we have to refine the pattern towards a more specific context to which it can be applied fully automatically in a generic manner, i. e., all applications that correspond to this context can be managed by applying the created AMI. In this case study, the refined context is defined by migrating a stateless Java application that is hosted on an Apache Tomcat 7 Webserver to Amazon's Cloud offering "Beanstalk", which provides a platform service (PaaS) for directly hosting Java applications. The pattern's core intent of migrating the application without downtime remains.

We now apply the presented method to automate the pattern for this refined context. In the first step, the kinds of applications to which the AMI can be applied must be defined. Therefore, we analyze the description of the pattern and the described refinement to model the Topology Fragment shown in Figure 4; according to the description, the automated pattern can be applied to all nodes of type "StatelessJavaApplication" that are connected by a relation of type "hostedOn" with a node of type "ApacheTomcat7". In the second step, we implement the pattern's solution as executable Topology Transformation. We first have to refine the pattern's abstract solution to the concrete use case considered in this study. Therefore, we (i) analyze the abstract solution process shown in Figure 3, (ii) transfer and refine the described information to our Java-based migration use case, and (iii) implement the Topology Transformation accordingly. We now step through this process and transfer each activity into our transformation implementation in consideration of the refinement. The transformation described in the following is implemented in a generic manner. It acts exclusively on the nodes and relations defined by the Topology Fragment or applies generic transformation rules that are not bound to a particular application. Therefore, it can be executed on all applications that match the pattern's Topology Fragment defined above. We explain the transformation directly on a real scenario that is depicted in Figure 5; on the left, there is the current ETG of a Java-based application that runs on a Apache Tomcat 7 installation hosted on a local physical server. The application implements a stateless Webservice that is publicly reachable via an internet domain. This Webservice shall be migrated to Amazon Beanstalk. As the defined Topology Fragment shown in Figure 4 matches this ETG, our refined pattern can be applied. All nodes and relations that are surrounded by dotted lines and Management Annotations are inserted by the transformation. The numbers in white circles represent the transformation steps.

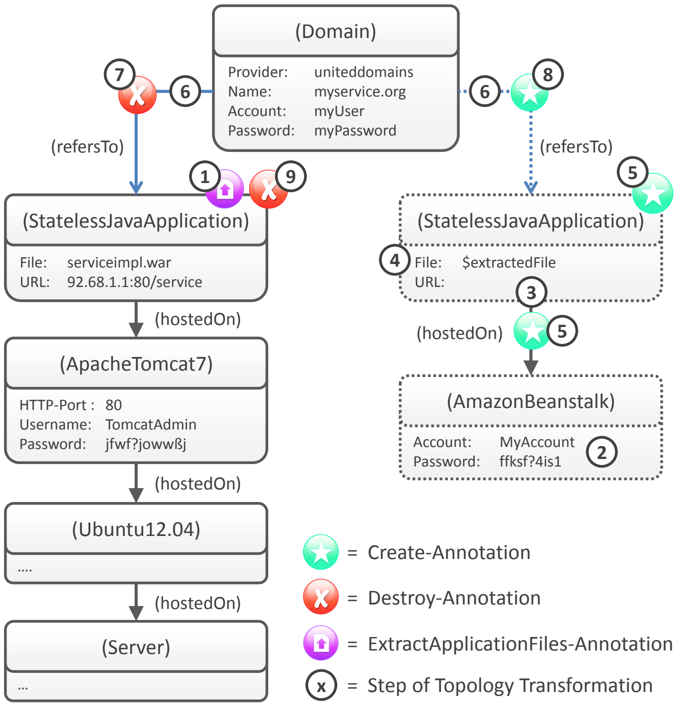

Figure 5. DASM that results from applying the created AMI.

First, the component to be migrated has to be extracted and the new application stack needs to be created. Therefore, the transformation (1) attaches a *ExtractApplicationFiles-Annotation* to the "StatelessJavaApplication" node and (2) inserts a new node of type "AmazonBeanstalk" to the DASM. As the Beanstalk node provides "Account" and "Password", the transformation requests these properties as input parameters and writes them directly to the node. We need no Management Annotations on this node since the Beanstalk service is always running. The *ExtractApplicationFiles-Annotation* is configured to export the Java files of the application to a location that is stored in a variable "$extractedFile", which is used in the next step. Afterwards, the transformation (3) inserts a new node of type "StatelessJavaApplication" and a "hostedOn" relation to the Beanstalk node, (4) specifies the files to be deployed using the "$extractedFile" variable, and (5) attaches *Create-Annotations* to the new node and the new relation. To update the references of the old component, we first (6) copy all incoming and outgoing relations except its "hostedOn" relation and replace the old component node by the new node, (7) attach *Destroy-Annotations* to the old relations, and (8) attach *Create-Annotations* to the new relations. Then, the transformation (9) attaches a *Destroy-Annotation* to the old component that specifies to undeploy the application from the local Webserver. To avoid downtime, the execution order of some annotations must be defined, e. g., creating and destroying the "refersTo" relations must be done concurrently by one planlet whereas the old component must not be decommissioned before the new one is ready. These orders are specified in the last step. The resulting DASM is complete and ready to be translated into the corresponding executable Management Workflow. In the following Test Phase, similar use cases are taken and the AMI gets applied to them. The resulting DASMs, as well as the generated workflows, are then analyzed for correctness.

V. Related Work

We applied the method to automate several management patterns, e. g., the *Stateless Component Swapping Pattern* [12] and the *Update Transition Process Pattern* [13] (published in Breitenbücher et al. [2][8]). In Nowak et al. [15], we automated the *Green Compensation Pattern* [16] to reduce the CO_2 emission of virtual machine-based business processes.

Fehling et al. [17][18] present a (i) step-by-step process for the traceable identification of Cloud patterns, a (ii) pattern format, and a (iii) pattern authoring toolkit that can be used to support the identification process. Despite these works focus mainly on Cloud architecture patterns, the core concepts can be adapted and used to create management patterns and idioms that can be automated afterwards using the presented method. Fehling et al. [17] also show how the identified architectural Cloud patterns can be applied using an existing provisioning tool. However, they consider only application provisioning and do not consider the automation of management patterns.

Reiners et al. [19] present an iterative pattern formulation approach for developing patterns. They aim for documenting and developing knowledge from the very beginning and continuously developing findings further. From this perspective, patterns are not just final artifacts but are developed based on initial non-validated ideas. They have to pass different phases until they become approved patterns. In contrast to our work that focuses on automating patterns, this iterative pattern formulation approach can be used to develop management patterns that capture problem and solution in natural text. Thus, the approaches are complementary: the formulation approach can be used to create management patterns that are automated afterwards using our method. Our pattern automation helps testing the captured knowledge in each phase of the iterative process to validate the pattern's correctness and suitability.

Falkenthal et al. [20] present an approach that enables reusing concrete implementations of patterns by attaching them as so called *Solution Implementations* directly to the patterns they originate from. The approach can be used to create workflows that implement a management patterns solution for a certain use case as Solution Implementation that is linked with the original pattern. However, a manual implementation of the corresponding workflows requires a lot of management expertise for handling the technical complexity of refinement [2]. In addition, such management workflows are typically tightly coupled to particular application structures and are not able to provide the flexibility of Topology Transformations that may analyze the whole application topology to ensure a correct specification of the Management Annotations to be performed.

VI. Conclusion and Future Work

In this paper, we presented a method that enables automating existing management patterns using the Management Planlet Framework. We showed that the method enables analyzing and implementing existing management patterns in a generically applicable fashion. The method provides a structured means to create and test Automated Management Patterns that guides developers in transforming natural text into automated routines. To validate the presented approach, we conducted a detailed case study that shows how a Cloud Application Management Pattern for application migration can be automated using our method. In future work, we plan to investigate how the method can be used to automate architectural patterns to support the development and initial provisioning of Cloud applications, too.

Acknowledgment

This work was partially funded by the BMWi project Cloud-Cycle (01MD11023).

References

[1] 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, March 2012.

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

[3] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *USITS 2003*. USENIX Association, June 2003, pp. 1–16.

[4] A. B. Brown and D. A. Patterson, "To Err is Human," in *EASY 2001*, July 2001, p. 5.

[5] 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 2013*. Springer, September 2013, pp. 130–148.

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

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

[8] U. Breitenbücher *et al.*, "Policy-Aware Provisioning and Management of Cloud Applications," *International Journal On Advances in Security*, vol. 7, no. 1&2, 2014.

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

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

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

[12] 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 2013*. IEEE, December 2013, pp. 9–16.

[13] 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.

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

[15] A. Nowak, U. Breitenbücher, and F. Leymann, "Automating Green Patterns to Compensate CO_2 Emissions of Cloud-based Business Processes," in *ADVCOMP 2014*. IARIA Xpert Publishing Services, August 2014.

[16] A. Nowak, F. Leymann, D. Schleicher, D. Schumm, and S. Wagner, "Green Business Process Patterns," in *PLoP 2011*. ACM, October 2011.

[17] C. Fehling, F. Leymann, R. Retter, D. Schumm, and W. Schupeck, "An Architectural Pattern Language of Cloud-based Applications," in *PLoP 2011*. ACM, October 2011.

[18] C. Fehling, T. Ewald, F. Leymann, M. Pauly, J. Rütschlin, and D. Schumm, "Capturing Cloud Computing Knowledge and Experience in Patterns," in *CLOUD 2012*. IEEE, June 2012, pp. 726–733.

[19] R. Reiners, "A Pattern Evolution Process - From Ideas to Patterns," in *Informatiktage 2012*. GI, March 2012, pp. 115–118.

[20] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, and F. Leymann, "From Pattern Languages to Solution Implementations," in *PATTERNS 2014*. IARIA Xpert Publishing Services, May 2014, pp. 12–21.