Decisions, Models, and Monitoring – A Lifecycle Model for the Evolution of Service-Based Systems

Christian Inzinger*, Waldemar Hummer*, Ioanna Lytra†, Philipp Leitner*, Huy Tran†,
Uwe Zdun†, and Schahram Dustdar*

*Distributed Systems Group
Vienna University of Technology
Argentinierstr. 8/184, A-1040 Vienna, Austria
{lastname}@dsg.tuwien.ac.at

†Research Group Software Architecture
Faculty of Computer Science, University of Vienna
Währingerstr. 29, A-1090 Vienna, Austria
{firstname.lastname}@univie.ac.at

***Abstract*—The process of engineering and provisioning service-based systems (SBS) follows a complex and dynamic lifecycle with different phases and levels of abstraction. We tackle the problem of making this lifecycle explicit, providing development time and runtime support for evolutionary changes in such systems. SBSs are modeled as integrated ecosystems consisting of four conceptual layers (or phases): design, implementation, deployment, and runtime. Our work is driven by the notion that identifying the right changes (monitoring) and effecting of these changes (adaptation) usually takes place individually on each layer. While considering changes on a single layer (e.g., runtime adaptation) is often sufficient, some cases require systematic escalation to adjacent layers. We present a generic lifecycle model that provides an abstracted view of the problem domain and can be mapped to concrete artifacts on each individual layer. We introduce a real-life scenario taken from the telecommunications domain, which serves as the basis for discussion of the challenges and our solution. Based on the scenario and our experience from a research project on Virtual Service Platforms, we evaluate three concrete use cases which illustrate the diversity of evolutionary changes supported by the approach.**

I. Introduction

Current enterprise applications are usually built on the notion of a service-oriented architecture (SOA), i.e., they use and reuse existing infrastructure assets and platform services while themselves providing services to be used by other applications. Such service-based systems (SBSs) are typically built for the long haul. Consequently, adapting SBSs to changing environments, or simply improving SBSs to eliminate problems of earlier versions, become central.

While the overall development process of SBSs is by now well-understood, the design of adaptive systems that evolve automatically or semi-automatically along with the environment they live in is still rather uncharted. Specifically, little research work exists that feeds runtime monitoring data back into the artifacts of previous phases of the development process. There are existing approaches in the field of self-adaptation that enable explicit feedback-loops in order to help software systems adjusting their behaviors according to their perception of the surrounding environment [1], [2]. A number of studies in the area of log mining aim at supporting the extraction of off-line log information for analysis and verification [3]. To the best of our knowledge, none of the existing approaches targets the reflection of runtime data to early development phases in order to support the continuous evolution of software systems. Examples of the reflection are to validate the configuration options selected during system modeling or deployment or to verify the assumptions and rationale of architectural decisions. Without adequate links between runtime monitoring and design-time artifacts, targeted improvement and evolution of SBSs become much more difficult.

In this paper, we present a novel approach to support a continuous development lifecycle of SBSs. Our approach is a realization of the model-driven development paradigm that extends the traditional development process with feedback loops that can feed runtime information to the corresponding artifacts of the adequate phase. During the course of the development phases, software architects and developers use different models to capture various types of development artifacts, such as architectural design decision, component models, or monitoring rules. Based on these models, deployment configurations, monitoring directives, and adaptation rules can be automatically generated. At the heart of our approach, we introduce a lifecycle evolution model to formally represent the relationships between monitoring information and the development artifacts. The evolution model can be (semi-)automatically achieved with reasonable efforts by extending model-to-model and model-to-code transformation rules. Monitoring information collected from the running code can be fed back into the artifacts of each phase to support on-line or off-line analysis and evolution of all artifacts of the SBS. The evolution model, on the one hand, can help to identify which particular artifacts at which phase may influence a certain unexpected or undesired incident, for instance, performance reducing, policy violation, and so forth. On the other hand, the trace links recorded in the evolution model can significantly enrich the context of the incident for better understanding and analysis. For instance, if monitoring unveils a performance problem at runtime, it is non-trivial to decide if the best way of coping with this situation is to simply reconfigure the system, or to improve system design or even architecture.

The rest of this paper is structured as follows. Section II concretizes the setting of the paper using an illustrative case study. The main contributions of the paper are discussed in Sections III and IV. In Section V, we evaluate our approach using a qualitative discussion, which is based on a case study from the INDENICA[1] project. Section VI discusses related research. Finally, the paper is concluded in Section VII.

II. Illustrative Example

The motivation for this paper is based on a scenario from the telecommunications services domain. The enhanced Telecom Operations Map[2] (eTOM) is a widely adopted industry standard for implementation of business processes promoted by the TeleManagement Forum (TMF). Our scenario is condensed from the TMF's Case Study Handbook [4] as well as two eTOM-related IBM publications on practical application of SOA in such systems [5], [6]. Figure 1 depicts the service delivery process in Business Process Modeling Notation (BPMN). It consists of six activities $i_1, \ldots, i_6$ (referred to as *interfaces* or *abstract services*). Each abstract service activity has alternative subactivities which we denote as *concrete service implementations* (denoted $c_1, \ldots, c_{12}$ in the figure). At runtime the process selects and executes one concrete service for each service interface.

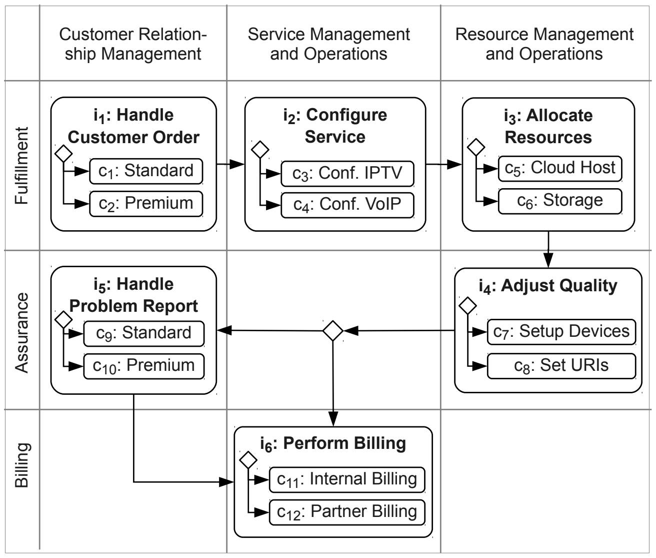


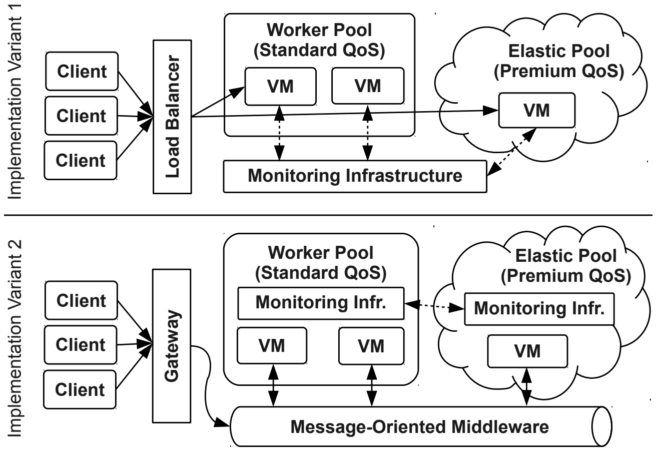
Figure 1. Service-Based Scenario Application

[1]http://www.indenica.eu

[2]http://www.tmforum.org/BusinessProcessFramework/1647/home.html

The process is initiated by the abstract service i_1 (Handle Customer Order) which is offered in two variants for standard and premium users. Depending on the order input, the process then configures a particular service (IPTV or VoIP). The third abstract service allocates the resources required for delivering the service (e.g., a cloud host or storage). Telecommunication services are typically associated with Quality of Service (QoS) attributes, which are fine-tuned by abstract service i_4. For instance, this activity configures parameters in the VoIP device or sets the location URI (Uniform Resource Identifier) of IPTV endpoints, in correspondence with QoS requirements. If a problem is detected at runtime, the optional reporting service is executed in activity i_5. Finally, the process terminates after storing billing information, either for paying partner providers or for internal accounting if the service was delivered in-house.

A. Service Variants and Evolution



Implementation Variant 1
Client
Client
Client
Load Balancer
Worker Pool (Standard QoS)
VM
VM
Elastic Pool (Premium QoS)
VM
Monitoring Infrastructure
Implementation Variant 2
Client
Client
Client
Gateway
Worker Pool (Standard QoS)
Monitoring Infr.
VM
VM
Elastic Pool (Premium QoS)
Monitoring Infr.
VM
Message-Oriented Middleware

Figure 2. Scenario Implementation Variants for a Single Provider

When the abstract business process in Figure 1 is mapped to a concrete infrastructure, several implementation variants are possible. Two variants are illustrated in Figure 2. In variant 1, the clients access a load balancer, which forwards requests to selected virtual machines (VMs). A fixed pool of worker machines supports the standard QoS, and additional VMs are requested from an elastic pool to serve premium clients. A centralized monitoring infrastructure collects performance metrics from VMs in both pools. As the system evolves, we anticipate that the load balancer becomes a bottleneck and hence gets replaced in variant 2 by a message-oriented middleware (MOM) to achieve stronger decoupling. Moreover, the infrastructure becomes decentralized to achieve better locality of the monitoring components within the VM pools. Providing support for implementation variants reflects architectural decisions made at design time or runtime based on given requirements and desired goals.

B. Challenges for Service Delivery and Evolution

The scenario outlined in Section II entails the following challenges that are typically encountered when engineering service-based applications:

Architectural Decisions: The process of developing service platforms follows recurring architectural design decisions [7], which should be explicit, systematic, and reusable.
View-Based Modeling: The platform models need to capture the architectural components and processes, thereby distinguishing multiple external (e.g., service interfaces) and internal (e.g., monitoring infrastructure) views for different stakeholders [8].
Cross-Provider Platform Integration: The scenario integrates service platforms from different providers, hence requiring well-defined communication interfaces as well as shared application models.
Platform Monitoring: The service platform is subject to fluctuations in request load, hence the service delivery process is governed by monitoring of QoS (Quality of Service) metrics [9].
Lifecycle and Adaptation: Based on the monitoring information and changes in the environment, the platform needs to support short-term adaptation [10] (e.g., scale-out due to load bursts) and long-term evolution (e.g., architectural reconfiguration).

III. Evolution Lifecycle Model

In this section we present a novel application evolution lifecycle model to allow for runtime adaptation of service-based applications, as well as their controlled evolution in a unified manner.

A. Application Lifecycle Overview

First, we discuss the application development lifecycle phases usually implemented by iterative and agile development processes, such as the Rational Unified Process [11], Goal-driven Software Development [12], and Scrum [13]. These approaches facilitate the necessary flexibility required for implementing complex SBSs. Complementary to these approaches we suggest to adapt existing software models and artifacts based on the feedback from monitoring and adaptation rules at runtime. Figure 3 summarizes the application development phases at design time and runtime into *Architectural Design*, *Modeling/Implementation*, *Deployment/Configuration*, and *Monitoring/Adaptation*. Before getting into the details of the feedback propagation between and across the different phases during software evolution, we briefly discuss each phase in the lifecycle process.

An emerging practice in architectural design is to not only document solution structures, but explicitly record architectural decisions that led to these structures [14]. Recurring architectural decisions can be documented in architectural decision models, thus increasing reuse and minimizing documentation effort [15].

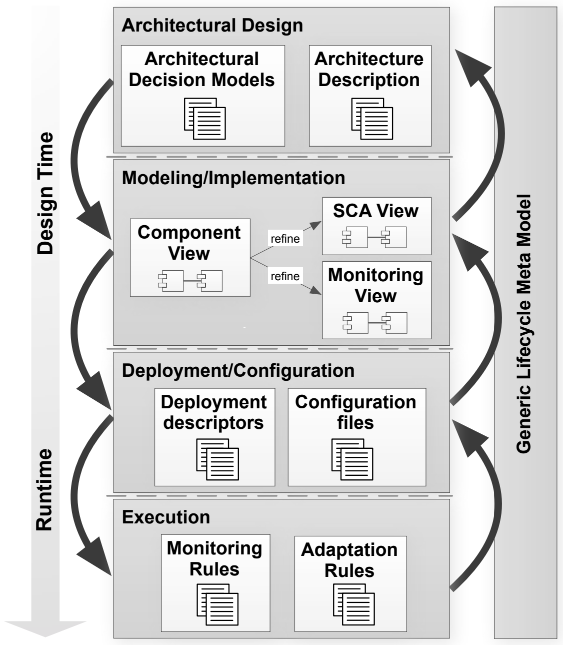


Figure 3. Application Lifecycle Overview

In practice, in order to describe software architectures various architectural views for different stakeholders' needs are used [16]. Model-driven techniques allow us to transform actual architectural decisions into architectural views automatically in a reusable manner [17]. In the modeling phase, high level views—as the component-and-connector view—can be refined and enriched to generate lower level technology and platform-specific views using techniques such as View-based Modeling Framework (VbMF) [8]. The view models are used to generate code skeletons, configuration and implementation artifacts for common boilerplate constructs. The generated code is then augmented with hand-written code by developers to complete the application implementation.

After development leads to an application release, a deployment plan is derived, according deployment descriptors are created, and configuration files are prepared. Deployment descriptors contain information about where to physically deploy the application, which dependencies need to be satisfied for a successful deployment, and how to actually instantiate the application code [18].

At runtime, the SBS is controlled using previously defined monitoring queries and adaptation rules that allow the application to react to changes in the environment in order to maintain desired behavior. In our approach we extend the notion of adaptation not only to the deployed application and its configuration, but also to the view models and architectural decisions from the earlier phases of software development.

The lifecycle allows for iterations across phases to address necessary changes in architecture, modeling, implementation and runtime management according to the used development method. In the following we introduce an evolution lifecycle meta model that augments the artifacts produced during each lifecycle phase to assist application evolution.

B. Lifecycle Model Overview

The lifecycle model is designed to assist application adaptation decisions using relevant data from artifacts in each phase to enable reasoning over costs and benefits of possible application changes. This model contains the required links between the artifacts of the aforementioned phases for propagating the feedback and enabling the appropriate adaptation. Results from monitoring are used to automatically perform adaptations. If automatic adaptation is not possible, the model provides the system operators with recommendations about possible escalation strategies to achieve a given goal. The feedback propagation and adaptation actions can apply to different phases of the application lifecycle. For example, configuration files may change or an alternative application version needs to be deployed at runtime. If the given goals are still not achieved this way, a refactoring of the architectural design and a reconsideration of the existing architectural decisions will have to take place. In the following, we present the evolution lifecycle model in more detail and discuss relevant properties and instances in each lifecycle phase.

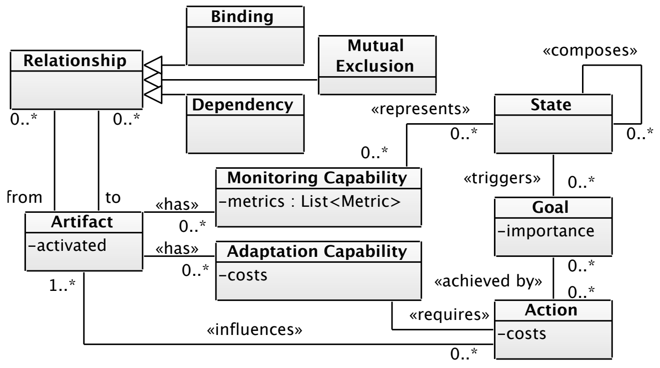


Figure 4. Lifecycle Evolution Model

An overview of the proposed model is shown in Figure 4. System state information aggregates observed metrics from various application artifacts to represent relevant status information. Application goals are specified along with actions that can be taken to achieve these goals. Artifacts from all lifecycle stages are represented along with their adaptation and monitoring capabilities. Adaptation capabilities represent assets of artifacts that can be modified externally. Similarly, monitoring capabilities provide information about artifact assets. Actions aggregate possible steps necessary to achieve a given application goal, specifying required adaptation capabilities for execution, as well as monitoring capabilities to verify goal fulfillment.

In the following, we discuss the model elements presented in Figure 4 in more detail.

State: This class represents possible application states, composed of state properties and their values. If (and only if) a State entity accurately reflects the current system state, the *activated* attribute is set to true; The data is gathered from monitoring capabilities of artifacts, and low-level monitoring information is aggregated into higher-level representations suitable for triggering adaptation decisions.

Goal: Application goals are derived from requirements and represent system behavior objectives. These objectives are gathered from nonfunctional requirements and represent concerns such as response time or service quality. Application goals modeled in the evolution lifecycle model augment the functional requirements implemented in the traditional development process.

Action: Actions encapsulate high-level measures for achieving goals. A goal such as ‘minimize response time’ can have multiple actions associated with it, e.g., add additional resources, reduce service quality, or change application architecture. For any given action multiple adaptation capabilities offered by application artifacts might be suitable.

Adaptation capability: Artifacts in the lifecycle model can have adaptation capabilities associated, representing means of changing them. Adaptation capabilities contain indicators for cost of performing adaptations as well as the supported degree of automation.

Monitoring capability: Monitoring capabilities represent relevant properties of artifacts that can be observed and are aggregated in system states to represent high-level application status information.

Artifact: All relevant artifacts produced during the application lifecycle are represented in the model, along with indicators representing their value for the application, cost of changing them, as well as their name. These indicators are used to improve adaptation decisions. Artifacts may be related to other artifacts, which allows us to introduce dependencies between the artifacts of the different phases, e.g., between architectural decisions and views, between design views and deployment descriptors, etc.

An excerpt of the model invariants is printed using Object Constraint Language (OCL) notation in Listing 1:

```
context Dependency inv: self.from.activated implies
  self.to.activated

context MutualExclusion inv: not self.from.activated
  or not self.to.activated

context Binding inv:
  self.from.activated = self.to.activated

context Action inv: self.requires.forAll(c |
  Artifact.allInstances()->exists(a |
    a.adaptationCapability.includes(c)))
```

Listing 1. Excerpt of Invariants for Lifecycle Model

In the following we discuss specific artifacts used in different lifecycle stages.

C. Architectural Design

Architectural decision models and architectural decisions are the basic artifacts produced during the architectural design phase. An architectural decision model contains reusable architectural decisions addressing recurring design issues. An architectural decision contains alternative options which can be realized using design patterns. One or more architectural decisions get reflected on the elements of the architectural design view. The links between architectural decisions and designs allow us to trace back affected decisions from a monitoring rule. These links can be (semi-)automatically established using the mapping techniques presented in [17] for bridging architectural decisions and design models.

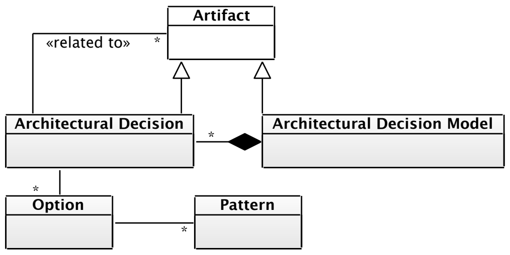


Figure 5. Architectural Decision Model

In addition, one or more architectural decisions might have various adaptation capabilities. Imagine, for instance, the case of a selected option of a low-level decision for satisfying low response time. The triggering of an adaptation rule will switch to an alternative option.

D. Modeling and Implementation

In this stage, the design models are created according to the architecture decision that have been made in the previous stage. The initial creation of design models can be performed automatically using the transformation in [17] based on the architectural decision model and/or manually manipulated by the developers. We show in Figure 6 an excerpt of the view-based design models and their relationship with the architectural decisions. View models are composed of view elements representing application subsystems, components, and their interactions. Views are used to capture various perspectives of modeling software systems and help the developers focusing on particular aspects of the system under consideration [8].

The links between view models or view elements and architectural decisions described in Figure 6 are based on the mapping techniques mentioned above and actually derived from the association "*related to*" shown in Figure 5. Artifacts created in this phase (i.e., views and view elements) can furthermore expose appropriate adaptation capabilities,

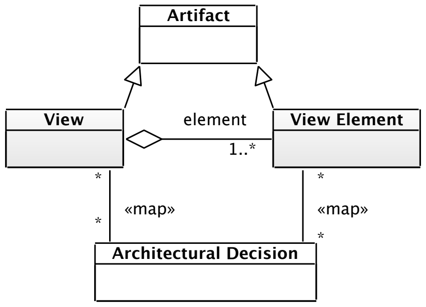


Figure 6. View-based Model

specifying the cost of changes as well as the supported degree of automation. Mature applications can benefit from all previously implemented design decisions and model elements by (semi-)automatically reusing components derived in earlier iterations.

View models can be used to generate code artifacts that comprises monitoring capabilities to observe all relevant aspects of the developed application along with adaptation capabilities to modify behavior at runtime. The modeled capabilities are mapped to according actions in the lifecycle evolution model, signifying their influence on the fulfillment of actions leading to desired goals.

E. Deployment and Configuration

In this stage, the physical deployment structure, as well as the configuration of the application instance to be run are created. As shown in Figure 7, code artifacts are bundled in deployable packages, and deployment descriptors are populated with information necessary to instantiate the created application on physical infrastructure.

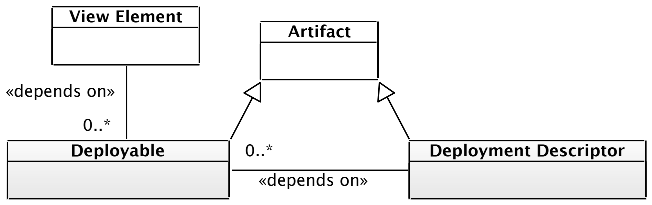


Figure 7. Deployment Model

The created artifacts are represented in the lifecycle model. Deployable packages include dependency relations to model elements implemented in the previous phase. Deployment descriptors encapsulate relevant information about component configuration, dependencies, as well as deployment structure. Deployment-specific monitoring capabilities are provided, allowing to observe how the application is instantiated, e.g., how many physical machines are used, and how components are distributed among them. Furthermore, adaptation capabilities allow for the modification of the deployment structure. As mentioned previously, the modeled monitoring and adaptation capabilities are used by actions representing objectives to be achieved by the application and are later used to allow application evolution strategies to consider changes in the deployment structure.

F. Execution and Runtime Monitoring

After successfully deploying an application, runtime monitoring provides comprehensive data about the fulfillment of specified application goals. Figure 8 shows monitoring queries that are executed alongside the SBS are represented in the lifecycle model and map to according monitoring capabilities. Monitoring capabilities represent low-level data, such as response time, number of service calls, and occurred errors, emitted from the SBS at runtime that is assigned to appropriate actions by system designers.

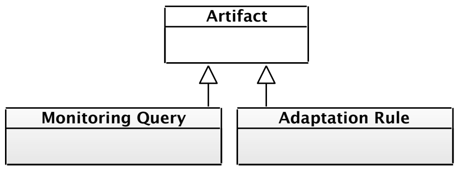


Figure 8. Runtime Model

Similarly, adaptation rules adjusting application behavior at runtime are modeled to document links to according adaptation capabilities. As mentioned above, adaptation capabilities represent changes to the application that can be performed at runtime, such as modify service quality, defer background processing, and modify external service binding. These capabilities are used by actions representing higher-level objectives that can be achieved by executing adaptations.

IV. Adaptation and Escalation Strategy

In this section we discuss how the lifecycle evolution model is used to enable adaptation and evolution across the complete development process leading to more efficient, cost-effective and higher quality applications.

Our objective is to perform adaptation actions to achieve application goals in the most cost-effective and automatic way possible. Whenever an application goal violation is detected, an adaptation strategy is derived from the SBS's lifecycle evolution model.

Available adaptation actions are performed automatically at runtime if possible to adjust the SBS's behavior towards the defined goals. Runtime adaptations are usually cheap and can be performed quickly, allowing for fast reaction to environmental changes. This is similar to how current applications perform runtime adaptation. However, the presented approach is designed to improve on the state of the art by allowing for incorporation of information from all phases of the application development lifecycle to enable more sophisticated, higher quality decisions about SBS evolution considering not only runtime, but also deployment, implementation and design aspects.

When runtime adaptation is not sufficient to reach a defined application behavior goal, modifying the application deployment might be suitable. The deployment model can be modified in several ways, e.g. by replicating application parts, migrating deployable artifacts to different physical machines, or adjusting deployment configuration. After modifying the deployment model, the adaptation strategy will incrementally redeploy the application to minimize downtime. Deployment adaptation can be executed automatically using defined adaptation rules.

Algorithm 1 Action selection

```
Input: $S$: currently active states
Output: $A$: actions to execute

   $G^u$      currently unfulfilled goals
   $A_g^{ac}$  actions achieving goal $g$
   $G_a^{ac}$  goals achieved by action $a$
   $a_g^e$    escalation action for goal $g$

 1: $A \leftarrow \emptyset$
 2: $G^u \leftarrow \{g' \in G | «triggers»(s, g'), s \in S\}$
 3: while $G^u \neq \emptyset$ do
 4:     $v_{actions} \leftarrow empty\ dictionary$
 5:     for all $g \in G^u$ do
 6:         $A_g^{ac} \leftarrow \{a' \in A | «achieved\_by»(g, a')\}$
 7:         if $A_g^{ac} = \emptyset$ then
 8:             $A \leftarrow A \cup a_g^e$
 9:             $G^u \leftarrow G^u \setminus g$
10:         end if
11:         for all $a \in A_g^{ac}$ do
12:             $i_a \leftarrow 0$
13:             $G_a^{ac} \leftarrow \{g' \in G | «achieved\_by»(g', a)\}$
14:             for all $g_a \in G_a^{ac}$ do
15:                 $i_a \leftarrow i_a + g_a.importance$
16:             end for
17:             $v_{actions}[a] \leftarrow i_a / a.costs$
18:         end for
19:     end for
20:     $a_{max} \leftarrow \arg\max_{a \in A}(v_{actions}[a])$
21:     $G_{a_{max}}^{ac} \leftarrow \{g' \in G | «achieved\_by»(g', a_{max})\}$
22:     $G^u \leftarrow G^u \setminus G_{a_{max}}^{ac}$
23:     $A \leftarrow A \cup a_{max}$
24: end while
25: return $A$
```

The process for selecting adaptation actions to be executed is illustrated in Algorithm 1. The algorithm is periodically executed and supplied with the set of system states S that currently hold, and aims to return a set of actions A that should improve system state towards currently unfulfilled goals G^u. We first gather the set of goals that are currently not satisfied, as seen on line 2. Then, the algorithm creates an associative array mapping available actions to a utility value incorporating the action's execution costs as well as the importance of currently unsatisfied goals (lines 6, 11–18). The most valuable action a_{max} is added to the set of actions A to be executed, and goals $G^{ac}_{a_{max}}$ that are achieved by a_{max} are removed from the set of unfulfilled goals G^u (lines 20–23). If no actions are found to achieve a goal g, a system-provided escalation action a^e_g is added to the set of actions A to be executed to indicate the need for operator intervention to satisfy goal g (lines 7–10).

If an adaptation triggers an implementation change, adaptation costs as specified in the model are updated according to metrics extracted from the code to improve subsequent adaptation decisions. Metrics, such as changed lines of code, code churn, and time taken, are incorporated to accurately reflect costs of performed adaptation actions.

Similarly, if adaptation decisions lead to changes in view models, according artifacts in the lifecycle model are updated with the cost of the performed adaptation, including implementation changes resulting from model changes.

If architectural decisions are changed, the according cost of change in the lifecycle model includes not only metrics for necessary changes in application architecture, but also view models, as well as related implementation changes.

V. Discussion

To provide a hands-on discussion of the presented approach, we consider three concrete lifecycle use cases related to the scenario in Section II. The first case (C1) is concerned with short-term adaptation of the internal monitoring and adaptation platform of a telecommunications provider. The second case (C2) deals with more coarse-grained evolution of the platform architecture, and the third case (C3) shows how the proposed escalation mechanism is used extend the model to handle previously unforeseen circumstances.

A. Use Case C1: VM Adaptation

For use case C1, we consider the runtime artifacts in Figure 9 for an implementation of variant 1 of the previously introduced scenario application (cf. Figure 2). Deployable *vm1* represents a VM in the "Elastic Pool" handling requests for premium customers. For the given case, we assume that *vm1* is currently exhibiting unusually high RAM usage. The worker pool is managed by deployable *pool1*.

The VM exposes, amongst others, a monitoring capability *m1* reporting general machine health information, such as CPU usage, amount of used RAM, and total RAM. Furthermore, *vm1* offers an adaptation capability *c1* that allows to adjust VM features, such as number of virtual CPUs, available instance storage, and total RAM, at runtime. Worker pool *pool1* exposes an adaptation capability *c2* that allows to start and stop worker VMs. Furthermore, Monitoring state *q1* extracts the current RAM usage from

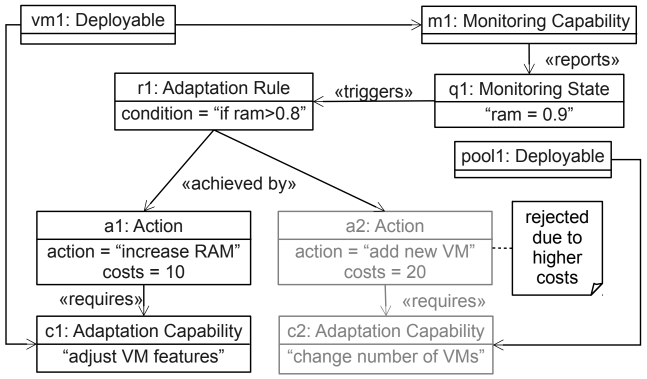


Figure 9. Artifacts of Use Case C1

m1 and asserts the 'current RAM usage too high' status. This state triggers adaptation goal *r1*: "RAM usage should not be too high". Goal *r1* can be achieved by multiple actions, mitigating the encountered problem. In our case, two actions, *a1* and *a2* are available for execution. Action *a1* uses adaptation capability *c1* of *vm1* to increase the amount of RAM available to the worker at runtime, whereas action *a2* uses adaptation capability *c2* of *pool1* to add another worker VM to the pool to spread the work load over more machines. Since *a1* can be performed much quicker than *a2*, we reject *a2* in this case and execute *a1* to mitigate the problem.

The system will subsequently monitor state *q1*, as well as fulfillment of goal *r1* to ensure that the performed adaptation has the desired effect. If the performed adaptation does not lead to the removal of *q1*, and the amount of RAM available to *vm1* cannot be increased further, *c1* becomes inactive, leading to the execution of *a2*.

B. Case C2: Architectural Refinement

In the second use case, we discuss how our approach can be used during application design to document and improve the development of SBSs. The discussion is based on the scenario introduced in Section II. Consider an enterprise that uses the presented approach for all their software projects. During application design for a new customer, stakeholders face the decision of whether to realize communication between the load balancer worker components using remote procedure invocation (Variant 1) or messaging (Variant 2), as illustrated in Figure 2.

Depending on the application requirements, either variant might be suitable. Remote procedure invocation allows for greater control over communication paths and provides for lower absolute latency. On the other hand, messaging reduces coupling between components and enables horizontal scalability independent of the load balancer component.

The presented approach assists application architects by storing artifacts from previously created applications in an

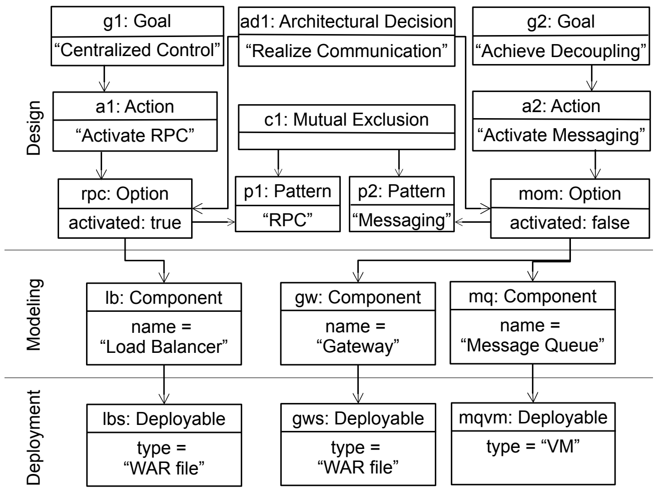


Figure 10. Artifacts of Use Case C2

artifact repository that can be queried for past solutions. The repository acts as an enterprise-wide application development knowledge base documenting experiences gathered in past projects. In our case, architects specify goals *g1* and *g2* for the application to be, as shown in Figure 10. The repository is queried for the created goals and provides actions for using remote procedure invocation (*a1*) as well as messaging (*a2*), along with according patterns and components to be implemented. Furthermore, a constraint *c1* applies to patterns *rpc* and *mom*, stating mutual exclusion. Since goals *g1* and *g2* are conflicting, application architects provide their decision by setting the *importance* attributes of *g1* and *g2* according to application requirements. If decision leads to the execution of either *a1*, action *a2* becomes inactive due to constraint *c1*. Pattern *rpc*, component *lb*, and deployable *lbs* are merged into the current application model from the repository, allowing operators to adjust their properties to the application at hand.

In the context of product line engineering, the presented approach can furthermore be used to significantly improve knowledge transfer and reuse between product variants. During design of a new variant, application architects can reuse 'application slices' from previously implemented variants. If the load balancer component was realized using patterns *rpc* and *mom* in previous variants, the stakeholders' decision will lead not only to the inclusion of relevant model elements, but also the accompanying code artifacts.

C. Case C3: Adaptation Escalation

In the third use case we illustrate the escalation model employed in our approach to enable cross-stage adaptation, as well as incorporating operator intervention. As before, we consider the scenario as described in Section II. We consider the artifacts shown in Figure 11. The scenario application is implemented using variant 1 as shown in Figure 2. For the given case, we assume that the load among the worker VMs is unevenly distributed due to a bug in the load balancer component. For brevity, application artifacts representing worker VMs, pool manager, load balancer, as well as their monitoring and adaptation capabilities are omitted in the figure.

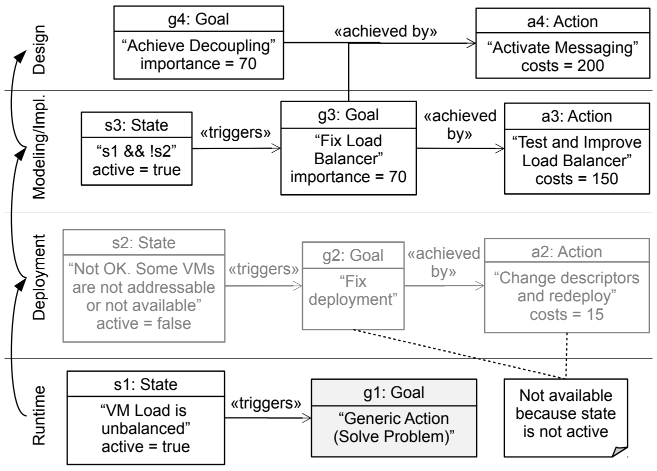


Figure 11. Artifacts of Use Case C3

Worker VMs report their CPU load through monitoring capabilities as described in case C1, and uneven load distribution activates state *s1* in the lower part of Figure 11. We assume that *s1* was put in place by application designers as a precaution to notice a system state that should not occur according to the specification. Hence, there is no mitigation strategy defined for *s1*, i.e., no adaptation goals are specified to execute adaptation actions. A generic goal *g1* is provided by the system to indicate that this state, while not yet handled, should be fixed if it occurs.

Since *g1* does not have any candidate actions associated that could be triggered automatically, the problem is escalated to system operators. Support staff assess the situation and suspect that the problem cannot be solved using runtime adaptations, but a faulty deployment could have caused the problem to appear. The deployment structure is validated using state *s2* that is active if there is a problem with VM deployment. In our case, deployment is valid and the associated mitigating actions need not be executed. Operators now suspect that a bug in the load balancer component is responsible for the problem. A composite state *s3* is created to represent the facts gathered during problem analysis. Furthermore, goal *g3* is created to document the desired system state, along with action *a3* representing the maintenance effort to fix the bug in the load balancer component. Additionally, actions to mitigate the problem are queried from the artifact repository. In our case, action *a4* from the repository is found to also solve the problem, albeit at higher cost. In addition to fulfilling goal *g3*, action *a4* furthermore fulfills goal *g4*, which was not satisfied before due to low importance. However, facing the problem at hand, *a4* is now the most feasible action to execute due to the increased benefits of satisfying *g3* as well as *g4*. If a messaging-based implementation of the load balancer component is already available from previous application variants (as discussed in C2), *a4* can be applied with minimal operator intervention. Otherwise, developers are requested to provide the necessary implementation and relevant application components are re-deployed.

VI. Related Work

The adaptation of service-oriented systems to rapidly uncertain and changing environment and settings has been studied at various levels in the literature. At the component-level, for instance, FRACTAL [19] relates components with a set of control capabilities to allow adaptations of component properties, addition or deletion of components, etc. for supporting dynamic configuration of distributed systems. At the requirements level, Peng et al. [20] address the self-configuration of software systems by introducing a formal reasoning procedure at runtime for supporting dynamic quality trade-off among alternative OR-decomposed goals.

However, existing approaches for self-adaptation of requirements goals and architectural models focus on the adaptation only at one layer. However, in our approach, we explore the possibility of performing adaptations at different layers/phases (Architectural Design, Modeling/Implementation, Deployment/Configuration) for satisfying the same goal. We achieve this by introducing traceability links between artifacts of the different layers and by relating the artifacts to monitoring and adaptation capabilities.

Reconfiguration of software systems at runtime for achieving specific goals has been studied in many contexts with focus on the area of service-oriented architectures. Rainbow framework [21] uses abstract architectural models for monitoring a system's runtime properties and proposes adaptations that can be directly reused at the running system. Irmert et al. [22] perform adaptations in service-oriented component models. Their approach utilizes Aspect-Oriented Programming (AOP) for transparent and atomic replacement of service implementations at runtime. Samimi et al. [23] describe an infrastructure for self-adaptive (autonomic) communication services that improve QoS using dynamic service instantiation and reconfiguration. Yet, these approaches concentrate on the reconfiguration and redeployment of the implementations and do not consider any adaptations at architectural modeling or design layer.

Similar to our approach, Shen et al. [24] propose a quality-driven adaptation approach at three different layers: requirements, design decisions and runtime architecture. In their work, the adaptation plans affect all three layers/phases in a unified manner, that is, the adaptation of the requirements goal model triggers the corresponding design decision deduction and runtime architecture reconfiguration. However, the adaptation plans in our approach can also be performed independently at one or more layers, thus providing more flexibility and alternatives.

VII. Conclusion

The systematic evolution of service-based systems is currently not well supported, particularly when considering integration of the four conceptual lifecycle layers: design, implementation, deployment, and runtime. Our proposed methodology for addressing this problem is to distill the concepts and artifacts from each layer into a generic lifecycle model, which allows for specific adaptation within a layer and at the same time escalation to adjacent layers. Escalation to a layer of higher abstraction (e.g., change of design decision as in Figure 11) is typically followed by downward traversal of the lifecycle phases (e.g., re-generation of code, re-deployment of components, re-initialization of monitoring queries). Technically, the process of adaptation is triggered by monitoring primitives, which can be combined into aggregated information, and are eventually correlated with artifacts from the lifecycle model. The correlation between measurable monitoring metrics and the lifecycle artifacts is the cornerstone for identification of system dependencies and possible adaptation actions.

Modeling the set of goals with associated alternative actions allows to make decisions about the best action to take in specific situations. While the decision for actions on lower layers (deployment, runtime) can be done mostly automatic, more high level decisions (concerning architecture or modeling) are often semi-automatic and require intervention by human experts. To enable the process of automatic selection, we propose using a cost/benefit tradeoff model associated with actions and the respective goals they achieve. Quantifying the benefit (importance) of goals is specific to the application domain, whereas costs of actions can typically be quantified precisely, e.g., man-hours for implementation, or usage fees for elastic Cloud computing resources.

The three case studies, presented to illustrate the feasibility of our approach, highlight runtime adaptation, architecture refinement, as well as adaptation escalation. We argue that the concept of a generic lifecycle model provides a solid basis for extended aspects related to reliability and enforcement of QoS. In our ongoing research we are incorporating fault detection techniques [25] to automatically assert previously unknown fault states when they occur, as well as adaptation policy improvement [26] to improve on the modeled rules and augment system control policies. We also envision that the approach can be integrated with automated testing [27] to identify incompatible configurations of activated artifacts.

Acknowledgement

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 257483 (INDENICA) and from the Austrian Science Fund (FWF) under project reference P23313-N23 (Audit4SOA).

References

[1] B. H. Cheng, R. Lemos, H. Giese *et al.*, "Software engineering for self-adaptive systems." Springer-Verlag, 2009, ch. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26.

[2] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.

[3] W. M. P. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*, 1st ed. Springer, 2011.

[4] TM Forum, "Case study handbook," December 2009.

[5] M. Fiammante, "Dynamic SOA and BPM: From simplified integration to dynamic processes," in *Dynamic SOA and BPM: Best Practices for Business Process Management and SOA Agility*. IBM Press, 2009.

[6] S. M. Glen and J. Andexer, "A practical application of SOA," October 2007. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-soa-practical/

[7] I. Lytra, S. Sobernig, and U. Zdun, "Architectural decision making for service-based platform integration: A qualitative multi-method study," in *IEEE/IFIP Conference on Software Architecture*, 2012, pp. 111–120.

[8] H. Tran, U. Zdun, and S. Dustdar, "View-based and model-driven approach for reducing the development complexity in process-driven SOA," in *International Conference on Business Process and Services Computing*. GI, 2007, pp. 105–124.

[9] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "End-to-end support for QoS-aware service selection, binding, and mediation in VRESCo," *IEEE Trans. Services Comput.*, vol. 3, no. 3, pp. 193–205, 2010.

[10] C. Inzinger, B. Satzger, P. Leitner, W. Hummer, and S. Dustdar, "Model-based adaptation of cloud computing applications," in *International Conference on Model-Driven Engineering and Software Development, Special Session on Model-Driven Software Adaptation*, 2013, pp. 351–355.

[11] P. Kruchten, *The Rational Unified Process: An Introduction*, 3rd ed. Addison-Wesley Professional, 2004.

[12] I. Schnabel and M. Pizka, "Goal-driven software development," in *IEEE/NASA Software Engineering Workshop*, 2006, pp. 59–65.

[13] H. Takeuchi and I. Nonaka, "The new new product development game," *Harvard Business Review*, vol. 64, no. 1, pp. 137–146, 1986.

[14] A. Jansen and J. Bosch, "Software architecture as a set of architectural design decisions," in *Working IEEE/IFIP Conference on Software Architecture*, 2005, pp. 109–120.

[15] O. Zimmermann, U. Zdun, T. Gschwind, and F. Leymann, "Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method," in *Working IEEE/IFIP Conference on Software Architecture*, 2008, pp. 157–166.

[16] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley Professional, 2003, vol. 2.

[17] I. Lytra, H. Tran, and U. Zdun, "Constraint-based consistency checking between design decisions and component models for supporting software architecture evolution," in *European Conference on Software Maintenance and Reengineering*. Springer, 2012, pp. 287–296.

[18] C. Inzinger, B. Satzger, W. Hummer, and S. Dustdar, "Specification and deployment of distributed monitoring and adaptation infrastructures," in *International Workshop on Performance Assessment and Auditing in Service Computing*. Springer, 2013, pp. 167–178.

[19] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The FRACTAL component model and its support in Java," *Softw. Pract. Exper.*, vol. 36, pp. 1257–1284, Sep. 2006.

[20] X. Peng, B. Chen, Y. Yu, and W. Zhao, "Self-tuning of software systems through goal-based feedback loop control," in *IEEE International Requirements Engineering Conference*, 2010, pp. 104–107.

[21] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004.

[22] F. Irmert, T. Fischer, and K. Meyer-Wegener, "Runtime adaptation in a service-oriented component model," in *International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2008, pp. 97–104.

[23] F. Samimi, P. McKinley, S. Sadjadi, C. Tang, J. Shapiro, and Z. Zhou, "Service clouds: Distributed infrastructure for adaptive communication services," *IEEE Trans. Netw. Service Manag.*, vol. 4, no. 2, pp. 84–95, 2007.

[24] L. Shen, X. Peng, and W. Zhao, "Quality-driven self-adaptation: Bridging the gap between requirements and runtime architecture by design decision," in *IEEE Computer Software and Applications Conference*, 2012, pp. 185–194.

[25] C. Inzinger, W. Hummer, B. Satzger, P. Leitner, and S. Dustdar, "Identifying incompatible service implementations using pooled decision trees," in *ACM Symposium on Applied Computing*, 2013, pp. 485–492.

[26] C. Inzinger, B. Satzger, W. Hummer, P. Leitner, and S. Dustdar, "Non-intrusive policy optimization for dependable and adaptive service-oriented systems," in *ACM Symposium on Applied Computing*, 2012, pp. 504–510.

[27] W. Hummer, O. Raz, O. Shehory, P. Leitner, and S. Dustdar, "Testing of data-centric and event-based dynamic service compositions," *Softw. Test. Verif. Reliab.*, 2013, doi:10.1002/stvr.1493.