Model-Based Support for Decision-Making in Architecture Evolution of Complex Software Systems

Konstantinos Plakidas
Software Architecture Research Group, University of Vienna
Vienna, Austria
konstantinos.plakidas@univie.ac.at

Daniel Schall
Siemens Corporate Technology
Vienna, Austria
daniel.schall@siemens.com

Uwe Zdun
Software Architecture Research Group, University of Vienna
Vienna, Austria
uwe.zdun@univie.ac.at

ABSTRACT

Design decision support for software architects in complex industrial software systems, such as software ecosystems and systems-of-systems, which feature extensive reuse of third-party solutions and a variety of deployment options, is still an open challenge. We describe three industrial use cases involving considerable re-architecting, where on-premises solutions were migrated to a cloud-based IoT platforms. Based on these use cases, we analyse the challenges and derive requirements for an architecture knowledge model supporting this process. The presented methodology builds upon existing approaches and proposes a model for the description of extant software applications and the management of domain knowledge. We demonstrate its use to support the evolution and/or composition of software applications in a migration scenario in a systematic and traceable manner.

CCS CONCEPTS

• **Applied computing → Enterprise computing**; • **Software and its engineering → Software creation and management**;

KEYWORDS

Software Architecture Evolution, Software migration, Model-based decision support, Software variability management, Systems-of-systems composition

ACM Reference Format:
Konstantinos Plakidas, Daniel Schall, and Uwe Zdun. 2018. Model-Based Support for Decision-Making in Architecture Evolution of Complex Software Systems. In *12th European Conference on Software Architecture: Companion Proceedings (ECSA '18), September 24–28, 2018, Madrid, Spain.* ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/3241403.3241426

1 INTRODUCTION

Modern industry practice in software engineering has increasingly moved away from monolithic, in-house developed applications to using the offerings provided by open and collaborative platforms such as software ecosystems (SECOs), the cloud, and the Internet of Things (IoT). These feature a complex and dynamic environment of offerings. Their use presents a number of challenges for the software architect. First, the growth, change, and turnover of offerings is so rapid that it quickly exceeds the personal experience of the architect, who loses oversight over what is available. Second, the extensive use of third-party offerings means that architects have to choose from a number of similar or competing solutions without necessarily having an insight into their workings. Third, the emergence of distributed systems or systems-of-systems (SoS) via cloud computing or IoT increase the heterogeneity of the components that have to be integrated, which are furthermore deployable in whole or in part, in different, off-premises locations, according to different service models, subject to different constraints.

It emerges that the management of an application's structure during its lifecycle is a complex task: the software architect needs to collect information, consult experts familiar with the technological foundations, best practices, and limits of available offerings, and make a series of choices regarding trade-offs. Typically, however, knowledge is passed on in an ad-hoc fashion between people, who themselves rely on personal knowledge and experience, resulting in a labor-intensive, error-prone and time-consuming process, which is not guaranteed to provide optimal solutions. While there are many proposals in the field of documenting architecture decisions [8, 9], they have not yet found widespread adoption in practice (yet), and the community is actively researching on how to make them more lightweight, more sustainable, and easier to use [18, 29]. Consequently, software engineering knowledge remains implicitly manifested in applications, and a broader audience cannot reuse this knowledge because it is not systematically captured. The ill effects of this are particularly evident in larger organizations, where applications have to be developed and used by different people with different roles, across different organizational boundaries, and over longer periods of time.

This study proposes a semiformal model for the description of extant software applications, and demonstrates how this model can be used to support decision-making during the evolution and/or composition of software applications in a systematic and traceable manner. The examples we drew on concern the evolution of extant offerings in brownfield environments, but the approach can be extended to greenfield development and used in any environment with a high variety and complexity of alternative solutions.

The main contributions of this work include:

- Description of three industrial scenarios demanding for significant architectural evolution.
- Graph-based software knowledge representation model.
- Description how the model supports decision making in the industrial application scenarios.

The basis of our approach is a graph-based software knowledge representation model (*domain knowledge*) which includes five knowledge areas (*Business Capabilities*, *Software Products*, *Software Architecture*, *Technology*, *Constraints*), whose instances can be used

ECSA '18, September 24–28, 2018, Madrid, Spain

ACM ISBN 978-1-4503-6483-6/18/09…$15.00
https://doi.org/10.1145/3241403.3241426

to define the specific *Software Description* of an application. The graph-based approach was chosen due to its flexibility, extensibility, and comprehensibility. We also provide examples to show how the final model can be used to support the architect in the use cases examined. This model is based on our prior work [23] on a multi-case study (of the systems we also use as use cases in this paper), in which we have studied factors of in-platform evolution and platform migration; this work extends our prior work with a detailed knowledge model derived from the findings of the multi case study.

The remainder of this paper is structured as follows: Section 2 discusses related work in the context of software evolution and migration and Section 3 introduces the industrial use cases and scenarios they represent. Section 4 describes our graph-based knowledge model followed by a discussion of the application of the model in the use cases in Section 5. The paper is concluded in Section 6.

2 RELATED WORK

Modern software systems are expected to evolve and be evolvable to satisfy the changing needs of their users and the changes in the technical environment, and remain economically viable [5, 8, 21]. Software architecture lies at the core of the evolution process. The architecture reflects not only its structure and behaviour, but also its constraints: the architecture specifies a system's possible functionality and the quality attributes it can (or can not) fulfill [3, 10]. Accordingly, correct understanding and representation of the architecture are fundamental for a systematic evolution process [21]. Failure to do so results in "code decay", i.e. a drift away from the original architecture that erodes its performance and qualities [11].

Many studies in the literature deal with topics related to the capture of software knowledge through the documentation of design decisions [9, 17], and the checking of conformity between architecture specifications and implementation (e.g. [16, 27]). The use of patterns, as recurring design problems, is of central importance in the field [17] and has been extended to cover new fields such as cloud-based architectures [12, 19] or microservices [14, 24].

A formal model for conducting architecture evolution and measuring an application's "evolvability" was proposed by [4]. In [28], the authors formed a categorization scheme for changes in software architecture. To detect architectural drifts, Haitzer et al. [16] propose supporting the semi-automated architectural abstraction of models throughout the software life-cycle. Another important topic in the field is that of *variability management*, indicating a software product's ability to be customized depending on a specific context [7, 15]. Furthermore, visual description languages such as UML2 are available to provide a toolset for describing applications.

While valuable on their own, many of these approaches are rather difficult to use in practice: they require much input from the stakeholders and result in a "collection of documents" that must then be acted upon. In summary, they represent an overhead of effort that people usually prove unwilling to invest in, especially when the value of the outcome is not readily apparent to them from the outset. As a result, the knowledge remains in the heads of the architects, rather than being documented, mined, and reused; and the actual evolution process is usually done in an ad-hoc fashion.

Our approach is intended to be a more light-weight alternative, where the discovery and management of offerings is made easier, the decisions are taken during the architecting process based on an immediately relevant set of constraints. As a result, the decision space is limited to manageable proportions by providing only the compatible options, as well as their driving forces and consequences.

3 USE CASES

A typical scenario that occurs in industry today is the migration of monolithic legacy software into a complex environment, such as a cloud platform (cf. [1, 2, 19, 20]). This is a multi-faceted process that entails a number of decisions such as choosing between different service models, such as Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS), moving an application as a whole or in part (hybrid) into the cloud, the use of cloud-native services or third-party solutions, the inclusion of dependencies, or the amount of re-architecting and refactoring to be undertaken to optimize its performance in the cloud. In this section we introduce three cases of real industrial projects where architecture evolution was a major concern. Based on these application scenarios, we derive the requirements for an architectural knowledge model facilitating architecture evolution.

3.1 Scenario 1: Legacy Monolithic Application Migration

The first use case concerns a system developed to perform geospatial analytics for large-scale infrastructure such as oil and gas pipelines, the electrical grid, or water distribution systems [22]. The system followed a 3-tier architecture with presentation, business logic and data storage tiers. Originally, it had been developed as an on-premises solution with SCADA integration in mind. The goal of the scenario was a migration from the on-premises solution, with limited scalability, to a cloud-based solution supporting elastic scaling and large-scale data processing capabilities.

Specifically, the scenario examined was the system's migration into the Siemens MindSphere [25] cloud platform. The legacy application included the open-source GeoServer to handle geospatial data, and the Oracle Database with Oracle Spatial and Graph extension to store original image and geospatial (vector) data. In this case, a decision was taken that GeoServer had to be retained in the target application regardless of any alternatives; and that a cloud-based storage and GUI had to be provided, while some elements of the application had to remain on-premises for legal reasons. This meant a hybrid migration pattern [20], where only part of the application is moved to the cloud.

The most important requirements derived from this scenario were i) the ability to clearly describe the components of the legacy system as well as the cloud platform offerings, ii) to denote deployment limitations (e.g. on data location), and iii) to support the discovery of potential replacement options.

3.2 Scenario 2: Evolution into Microservice-Based Architecture

The second use case concerned a legacy Water Management System (WMS) tasked with optimizing the use of a water delivery system comprising reservoirs, pipes, valves, etc. It used EPANET to model the pipe distribution system and perform the optimization, based on data provided via external sources in XML and through a SCADA interface using the OPC-UA protocol. The application included an orchestration component based on .NET—essentially a stateless

middleware that manages requests to the EPANET DLL via Platform Invocation Services (P/Invoke)—as well as a dashboard to present data and manage the system.

The task at hand involved the migration of WMS into MindSphere, and the refactoring of the application into the microservices architectural style, i.e., "small, single-purpose services that each targets a discrete function. The services are implemented independently of one another, providing a loosely coupled structure that forms a cohesive application based on standardized interfaces that support inter-service communication" [13]. Hence this scenario included not only the migration aspect, but also extensive architecture reconfiguration, splitting up the monolithic legacy application into a modular target application.

The requirements derived from this scenario were i) the inclusion of a comprehensive set of patterns related to the microservice architecture paradigm, ii) the support of the composition from and decomposition of components, and iii) the clear definition of interfaces between components.

3.3 Scenario 3: Software Delivery and Deployment on Edge and Cloud

In the third use case the capabilities of a monitoring system (on-premises SCADA solution) have been extended with cloud connectivity. In particular, the SCADA system has been modelled as an edge platform with analytics capabilities drawn from the R ecosystem. Analytics functions are provided as services that can be deployed either locally at the edge platform or in the cloud. Deployment decisions are continuously evaluated by monitoring quality-of-service (QoS) parameters and platform utilization. Updates in the deployment can be (re-)engineered to prevent overload conditions or unacceptable latencies.

That is, a set of applications, or parts of applications, may have to be deployed on either one or both platforms. The decision depends not only on the functional and non-functional requirements of each application, but also on what resources are actually available in the platform. Furthermore, both factors are dynamic: the requirements of an application can change, especially in the context of the desired more frequent releases, while the resources available in each platform also change over time, as they are utilized by other services and applications. The problem was how to model a dynamic set of deployment options and make them available for automatic assignment to assist engineers in distributing analytics services across cloud-edge platforms.

The requirements derived from this scenario were i) the ability to model non-functional deployment constraints as well as QoS attributes, such as availability, responsiveness, and workload, and ii) describe the offerings of a SECO through metadata such as dependencies, versioning, or usage, to determine their reliability and suitability.

4 MODEL DESCRIPTION

The background, assumptions made, and process used to evolve the model are described in [23]. We found that during the design process, the architect moves constantly between two focus levels: the specific application at hand, with a specific realization and attributes, and a more general realm of options (decision space) that can be employed in building up the architecture of the target application. Therefore we decided to utilize two models: one for documenting the general software engineering *domain knowledge* and one for the *software description* of the individual application. The latter will use elements from the former to describe (instantiate) a specific application. Given that we aim to provide an early architecting capability, we remained at a relatively high level of granularity and did not directly examine the code or implementation aspects. The model aims to enable software architects and project managers to manage concerns such as architectural reconfiguration, balancing non-functional requirements, or choosing from a variety of similar offerings, based on criteria that are often not related to code (cost, licensing, usage patterns, etc.) [20]. If therefore the refactoring requires code changes, our model can flag it as a requirement, but does not examine how to best implement it. Furthermore, the model is intended as a basic representation of the entities and relationships involved in software evolution, not as a stand-alone tool. Additional logic would be required for the handling, summation, and presentation of the available data in an easily comprehensible and user-friendly form, which is the subject of ongoing work.

4.1 Domain Knowledge Model

The *domain knowledge* model is a representation of the "general" software engineering field, including all products, terms, and concepts that are relevant to a software architect. Based on the knowledge required in the use cases, a knowledge repository was populated using literature sources such as pattern books (e.g., [6, 26]) and the descriptions of relevant software products from online sources.

The basic elements of the model are *Instance Elements* and *Type Elements*. The *Type Elements* serve to provide an ontology-like hierarchical structure, while the *Instance Elements* represent the leaves. Following experimentation with various categorizations, we settled on five distinct knowledge sets, each implemented in the model, and provided with types, instances, and relationships particular to it: *Capabilities*, *Applications*, *Architecture*, *Technology*, and *Constraints*.

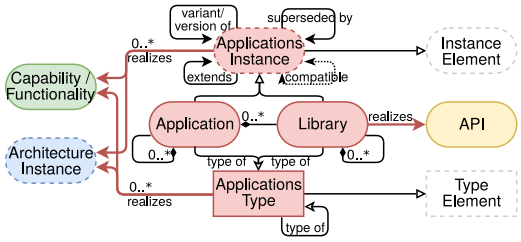

Figure 1: Model of the elements and relationships of the *Applications* set (in red), including its links to other sets (blue for *Architecture*, green for *Capabilities*, and yellow for *API* instances from the *Technology* set)

The *Applications* set (Figure 1) distinguishes between *Applications* proper and *Libraries*: an *Application* is a stand-alone product, and can be composed of both *Application* and *Library* instances, whereas a *Library* can only be used as part of an *Application*, can only be composed of other *Library* instances, and is closely associated with a set of *API* definitions from the *Technology* set. The type hierarchy is represented by *Types*, from the specific to the more abstract (e.g., PostgreSQL $\xrightarrow{\text{TYPE OF}}$ Object-Relational Database $\xrightarrow{\text{TYPE OF}}$ Database $\xrightarrow{\text{TYPE OF}}$ Data Persistence Application). Intra-set

relationships are: COMPATIBLE between two mutually compatible instances; VARIANT, where an application is a variant of a specific baseline (e.g., .NET Core vs. .NET), and EXTENDS for product extensions and plugins; VERSION OF and SUPERSEDED BY are used for versioning purposes. Furthermore, elements can have cross-set relationships, REALIZING specific instances from other sets.

To use the previous example, because Database $\xrightarrow{\text{REALIZES}}$ Data Persistence Capability and Database $\xrightarrow{\text{REALIZES}}$ Data Query Capability, PostgreSQL, as a subtype of Database, will also REALIZE the same capabilities. This allows the use of the *Capabilities* set to search for suitable *Application* instances, as will be shown later. In the same manner, suitable applications that implement specific architecture elements, and libraries that implement specific technologies, can be found.

Applications can be considered as the central set, as is seen from the variety of its internal relationships, and by the role it plays: the other sets exist to provide attributes for the *Applications* instances. The other sets also have connections between them, but *Applications* provides the direct link to the *software description* model, as each *Applications* (*Instances* as well as *Types*) has a corresponding *software description* instance.

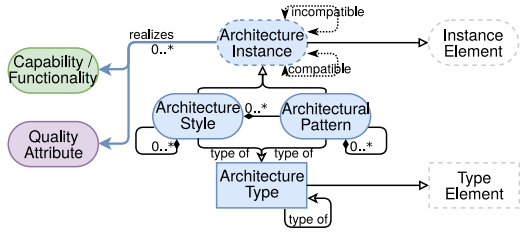

Figure 2: Model of the elements and relationships of the *Architecture* set (in blue), including its links to other sets (in different colours)

The *Architecture* set (Figure 2) is composed of *Architecture Concept* instances, in turn grouped into *Architecture Type* categories. An *Architecture Concept* can be expressed as an *Architectural Pattern* or an *Architectural Style*. In the literature the terms are often found interchangeably, but we have opted to interpret *Style* as a more generic category, such as Service-Oriented Architecture or Client-Server Architecture. Each *Architecture* instance can also REALIZE a specific *Functionality*, or enhance one or more *Quality Attributes* from the *Constraints* set.

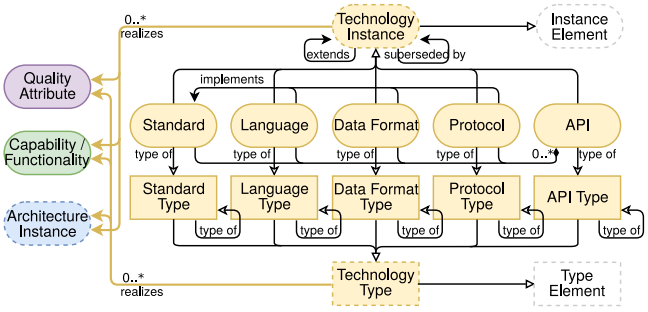

Figure 3: Model of the elements and relationships of the *Technology* set (in yellow), including its links to other sets (in different colours)

The *Technology* set (Figure 3) covers *Standards*, *Languages* and *Data Formats*, *Protocols*, and *API* definitions. Each set element can realize *Capabilities*, *Architecture*, and *Quality Attributes* instances. Again, there is a type hierarchy and additionally an *API* can be composed of other *Technology* instances; a *Standard* can be IMPLEMENTED by a *Language*, *Data Format*, or *Protocol*.

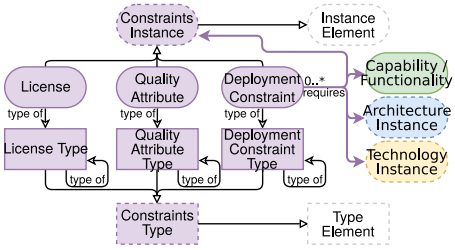

Figure 4: Model of the elements and relationships of the *Constraints* set, including its links to other sets (in different colours)

The *Constraints* set (Figure 4) is divided into three different categories: *Licenses*, *Quality Attributes* such as Non-Functional Requirements (NFRs), and miscellaneous *Deployment Constraints* that may restrict the deployment of an application or application component (e.g., data privacy, hardware requirements, etc.).

Finally, the *Capabilities* set comprises only *Capability* instances, which can in turn be composed of other *Capability* instances. The strength of the composition relationships can vary between "strong" (always included), "medium" (typically included), and "weak" (can be included).

4.2 Software Description Model

Each instance of *software description* makes use of *domain knowledge* instances by combining them in a specific configuration that represents the implementation and attributes of the application in question. Each *Applications* element has a, more or less complete, *software description*. By default, that is inherited from its type or from previous versions, and needs to be further adapted. Mirroring the *domain knowledge* sets, the *software description* model is subdivided into collections of objects:

- *Capabilities* collection: the business capabilities and functionalities that the application realizes.
- *Components* collection: individual software products, either off-the-shelf from the *Applications* set or custom-developed, used in creating the specific application. This collection also includes the links between components, and their (and by extension the application's) interfaces to the external world. Hence the *Components* collection also documents an application's dependencies.
- *Architecture* collection: software architecture concepts such as patterns and styles, and the elements comprising them, that are used to structure the specific application.
- *Implementation Languages*, *Data Input*, *Data Output* and *Data Operations* collections: *Technology* instances used in the application and interaction with its environment.
- *Quality Attributes*, *Licenses*, *Deployment Constraints* collections: supplementary constraints on the application's behaviour (NFRs) and reusability by third parties.

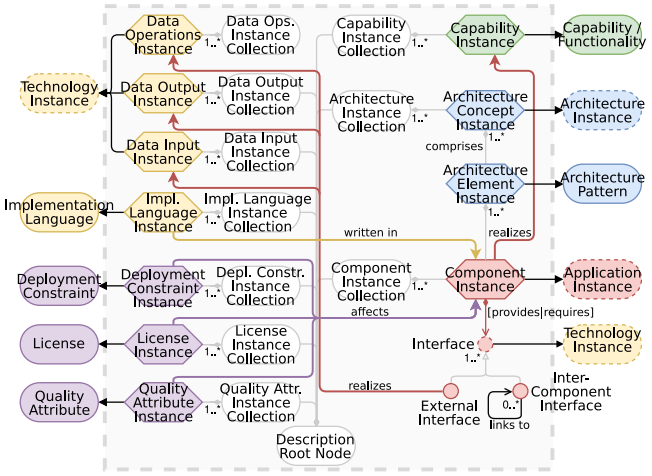

Figure 5: The *software description* Model. *domain knowledge* entities are depicted as ovals, their local instances in a specific *software description* as hexagons. The latter are aggregated into collections. Interfaces are shown as circles. The boundary between the two models is shown as a striped grey line.

Since each *Component Instance* is linked to its own *software description*, any software product can be composed of other software products. The external attributes of each component (*Component Instance*) are available for the composite application, and can be linked to specific attributes of its *software description*. For instance, a database, which on its own REALIZES the "Data Persistence" *Capability*, can fulfill the same *Capability Instance* when used as a component.

The degree of detail and completeness of the *software description* can vary. While completeness is desired, it is not always necessary or required: third-party offerings in a SECO, for example, are usually reused as a unitary package, and their internals (i.e., the *Components* and *Architecture* collections) may be unknown. As long as the *software description* provides a product's *external* interfaces and associations with all other *domain knowledge* elements (e.g., compatible technologies, functionalities realized, licenses, etc.), the product itself can be treated as a "black box" or a placeholder while still allowing its reuse as a component. This allows the development to proceed even when the exact implementation, in terms of a component's own *Component Instances* and *Architecture Instances* has not been defined yet, or when it is not our job to define it, for instance in a typical scenario where a generic application for interacting with a specific system can be defined (*compliance-by-design*), leaving the details of its implementation for the various vendors to decide.

4.3 Software Templates

A major feature that emerged during the testing of the models were templates that provide complex pre-defined arrangements of model elements. This emerged particularly through the scenario described in Sec. 3.2, where we were confronted with the need to perform a heavily architecture-centric evolution; thus we imported the comprehensive set of patterns that are related to the microservice paradigm described at [24] into our model, describing their interrelations, constraints, and linking them to elements

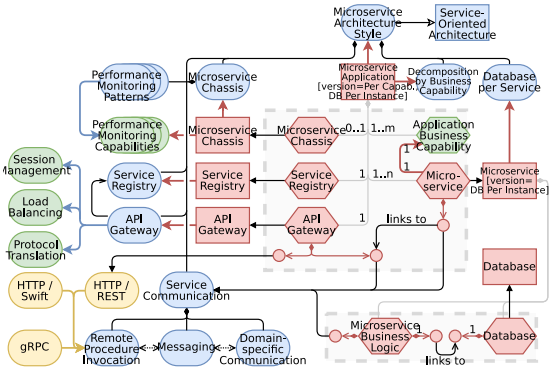

Figure 6: Example of a template for a microservice-based application: the application has a number of microservices that each fulfill a specific business capability, and have their own database each. The options for the use of a "Microservice Chassis" component and the choice of the microservices' communication pattern(s) have been left open. For ease of presentation, the collections and root node of the *software description* are not shown.

from other sets. However, the correct use of this knowledge proved too complex for practitioners unfamiliar with microservices. As a result, we turned to defining various templates for commonly occurring pattern combinations. An example is shown in Figure 6, for a microservice-based application using the "Database per Service" and "Decomposition by Business Capability" *Architecture Patterns*. The template provides a certain type, number, and structure of *Component Instances*, leaving the rest optional. For example, in the microservice type which implements the "Database per Service" pattern, the business logic (an otherwise undefined placeholder) must be connected to a database (placeholder of type "Database"). Similarly, in the Microservice Application type, we can see how specific components can be tied to certain attributes. In this case, each Microservice *Component Instance* realizes exactly one *Capability*, and the API Gateway needs to be accessible via HTTP/REST.

5 EMPLOYING THE MODELS

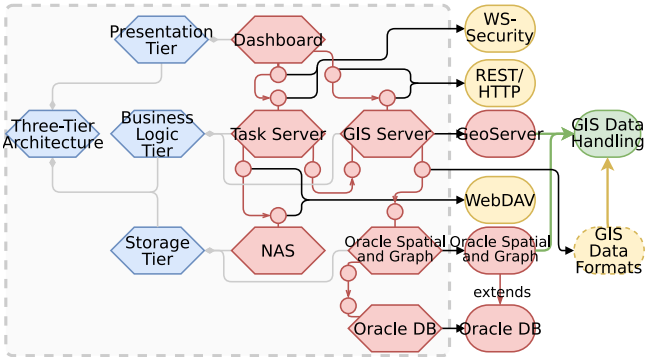

Figure 7: Simplified *software description* of the *Legacy Application* from Sec. 3.2.

We will use the scenario from Sec. 3.1 to demonstrate the use of the models in an actual migration context. Similar models were developed for the other two scenarios, but not included here for space

reasons. First we used the models to build the *software description* of the *Legacy Application* ($\mathcal{L}$)—a simplified instance of which is shown in Figure 7—as well as describe the *Target Environment* ($\mathcal{E}$). The scenario parameters led to some preliminary decisions:

- The *Capabilities* are the same in our *Target Application* ($\mathcal{T}$) as in $\mathcal{L}$, as no new functionality is to be added in this process. A new *Capability Instance* would typically require the introduction of a new *Component Instance* that would REALIZE it, so the migration begins with the initial assumption that the *Component Instances* of $\mathcal{L}$ and $\mathcal{T}$ are expected to be approximately the same, or at least of the same function and type.
- Some of $\mathcal{L}$'s components, however, have the *Deployment Constraint* "On-premises Software". This requirement clashes with the global *Deployment Constraints* of $\mathcal{E}$ (and hence of $\mathcal{T}$), which from its Type "Cloud Platform" includes the requirement "Cloud Compatibility". As a result, $\mathcal{T}$ will be composed of a part of $\mathcal{L}$ remaining on-premises ($\mathcal{R}$), and the part that will be migrated ($\mathcal{M}$).
- The decision to migrate to the cloud also implies the decision that the components of $\mathcal{T}$ will have to be individually accessible via REST/HTTP.
- The constraints of the particular $\mathcal{E}$ include the restriction of using only "Free and Open-Source Software" (FOSS) licensed products.

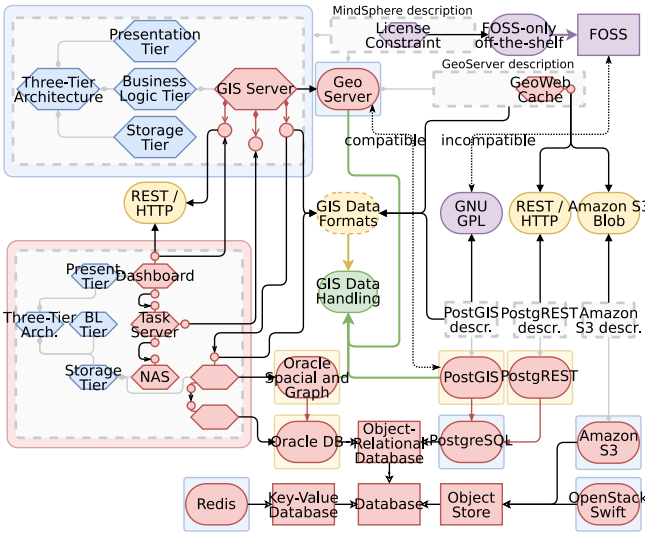

Figure 8: Simplified depiction of the first step in the migration process: the move of GeoServer from the *Legacy* (highlighted red) environment to the *Target Environment* (highlighted blue). This begins the formulation of $\mathcal{M}$, and shows the areas where adaptation of GeoServer may be required. Also shown are some of the relationships in the *domain knowledge* used in the present example.

This sets up the context for the first step in the migration process: the move of GeoServer into $\mathcal{E}$, and the step-by-step construction of $\mathcal{M}$ around it. Figure 8 shows the consequences of that step: the component is moved with its operational context, which involves its links to other components in $\mathcal{L}$, its role as a member of a Business Logic tier in a Three-Tier Architecture (which leads to the instantiation of a stub of the latter in $\mathcal{M}$), and the deployment attributes (red background) carried over from $\mathcal{L}$. Each of these results in a set of possible mismatches: the architecture may be incompatible with $\mathcal{M}$; the links to components in $\mathcal{L}$ have to be made compatible with the fact that the communication now takes place between two different deployment environments, or be replaced one by one with analogous solutions in $\mathcal{E}$ (Application Instances on blue background) or from off-the-shelf products (yellow background); and the component's deployment attributes may have to be matched with the requirements of $\mathcal{E}$ (blue background).

As a specific example of the adaptation process, we will use the discovery of a suitable cloud-deployed database component to link with GeoServer and implement $\mathcal{M}$'s Storage Tier. A simple matching query showed that our previous database, Oracle DB (extended with Oracle Spatial and Graph), was not among $\mathcal{E}$'s offerings, meaning that we had to find an alternative. From GeoServer's documentation, we had a relatively rich *software description* of links to various database offerings.

Oracle DB is of the *Application Type* "Object-Relational Database", so we first look for equivalent solutions among $\mathcal{E}$'s offerings that were of the same type as Oracle DB (? $\xrightarrow{\text{TYPE OF}}$ () $\xleftarrow{\text{TYPE OF}}$ Oracle DB), which returned PostgreSQL. In order for PostgreSQL to occupy the slot held by OracleDB + Oracle Spatial and Graph, it must fulfill the same role and requirements, such as connecting with GeoServer and supporting the storage and handling of GIS Data formats (its compatibility with $\mathcal{E}$ is a given).

A query on the *Technology* links between GeoServer and PostgreSQL resulted in two possible solutions, shown in Figure 8: the PostGIS and PostgREST plugins. The query results show both the explicit compatibility (COMPATIBLE relationship), known from their documentation, between PostGIS and GeoServer, as well as the implicit compatibility through the use of common *Technology Instances*. This means that even in the absence of an explicit documentation, a link is still suggested through set matching. The PostgreSQL + PostgREST combination could not be used to replace Oracle DB because neither provides support for GIS data; PostGIS fulfilled that requirement, but failed the $\mathcal{E}$ requirement that only FOSS third-party products can be deployed there, as it has a GNU GPL license.

Since the previous combinations could not be used, we moved one abstraction level up and looked for less strictly similar solutions, going to the generic "Database" type. Of the database solutions that are hosted in $\mathcal{E}$ and hence immediately available to us, only the "Object Store" solutions were compatible with the kind of data we wanted to persist. From GeoServer's documentation, furthermore, we knew that the GeoWebCache component is compatible with Amazon S3 blob, hence we chose that as our solution. This is a typical example of how the matching works: if an identical component is not found, we iterate up along the hierarchy tree(s) of the legacy component to find the nearest similar solution, and check whether it matches our already existing components and satisfies our constraints.

We consider the query-and-match approach to be simple to understand and effective in discovering potential solutions through the *domain knowledge*, particularly if the matching can be automated and carried out in the background. As seen in this case, the process permits filtering out incompatible offerings based on the

constraints and features included in $\mathcal{T}$. The type hierarchies and sets allow the discovery of connections as well as incompatibilities between software products that are not explicitly documented, by querying along the typologies and the knowledge sets to bridge an incompatibility. In our case studies, the usual depth of querying that provided immediately usable solutions were 2–4 elements. Finally, the trial-and-error approach creates a decision tree that can be traversed and altered, and a clearly defined decision context.

6 DISCUSSION

Based on the migration of three applications to cloud-based IoT platforms in industry, we have derived a model for easing software architecture migration decisions with the following main benefits. It provides a centrally managed knowledge repository, and a consistent but also easily extensible reference model. It provides a framework that links software products, software architecture aspects, business requirements/constraints, and technologies. This is valuable in cases like a software ecosystem or cloud platform, where hundreds if not thousands of offerings are available. As seen in the scenarios, the structure of the model allows the interplay between the specific *software description* and the general *domain knowledge*, providing various axes of approach to solve problems: through the technology, architecture, business capabilities, etc. The links between views provide a means to filter out incompatible offerings through different constraint sets, while also discovering solutions that are not explicitly documented. The model offers a key advantage in its extensibility. New nodes and domains/views can be added easily, while maintaining the same query mechanism as in the existing views, and additional attributes and metadata can easily be added and queried.

The model can not replace the architect. Throughout the scenarios, a number of design decisions were taken by the architects based on factors that do not derive from factors present in the model, but represent other concerns, such as familiarity with a product. We are also aware of the fact that the model itself is ill suited to be worked with directly, as populating its knowledge database is time-consuming, and the number of parameters involved in any decision grow very quickly; it is therefore our plan to implement and validate a web-based decision support tool that will abstract the use of the model. The tool will offer a wizard and guide the user through, for example, migration processes, allowing for the tracking of the decision-making process. This will also enable data-driven machine learning approaches that will enable recommender approaches for specific contexts.

ACKNOWLEDGMENTS

This work was partially supported by Austrian Research Promotion Agency (FFG) project DECO (no. 864707) and Austrian Science Fund (FWF) project ADDCompliance.

REFERENCES

[1] Amazon. 2010. New Whitepapers on Cloud Migration: Migrating Your Existing Applications to the AWS Cloud. https://aws.amazon.com/blogs/aws/new-whitepaper-migrating-your-existing-applications-to-the-aws-cloud/. (November 2010).

[2] Amazon. 2015. A Practical Guide to Cloud Migration: Migrating Services to AWS. https://aws.amazon.com/blogs/publicsector/a-practical-guide-to-cloud-migration/. (December 2015).

[3] Len Bass, Paul Clements, and Rick Kazman. 2003. *Software Architecture in Practice* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[4] Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. 2012. Software architecture evolution through evolvability analysis. *Journal of Systems and Software* 85, 11 (2012), 2574–2592. https://doi.org/10.1016/j.jss.2012.05.085

[5] Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. 2012. A systematic review of software architecture evolution research. *Information and Software Technology* 54, 1 (2012), 16 – 40. https://doi.org/10.1016/j.infsof.2011.06.002

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. John Wiley & Sons.

[7] Rafael Capilla, Jan Bosch, and Kyo Chul Kang. 2013. *Systems and Software Variability Management*. Springer. https://doi.org/10.1007/978-3-642-36583-6_20

[8] Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar. 2016. 10 years of software architecture knowledge management: Practice and future. *Journal of Systems and Software* 116, Supplement C (2016), 191 – 205. https://doi.org/10.1016/j.jss.2015.08.054

[9] R. Capilla, F. Nava, and J. C. Duenas. 2007. Modeling and Documenting the Evolution of Architectural Design Decisions. In *Sharing and Reusing Architectural Knowledge - Architecture, Rationale, and Design Intent, 2007. SHARK/ADI '07: ICSE Workshops 2007. Second Workshop on*. https://doi.org/10.1109/SHARK-ADI.2007.9

[10] Paul C. Clements, Rick Kazman, and Mark H. Klein. 2001. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, Boston, MA, USA.

[11] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. 2001. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering* 27, 1 (Jan 2001), 1–12. https://doi.org/10.1109/32.895984

[12] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer.

[13] Martin Fowler. 2014. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html. (March 2014).

[14] Arun Gupta. 2015. Microservice Design Patterns. http://blog.arungupta.me/microservice-design-patterns/. (April 2015).

[15] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. 2001. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '01)*. IEEE Computer Society, 45–. https://doi.org/10.1109/WICSA.2001.948406

[16] Thomas Haitzer and Uwe Zdun. 2014. Semi-automated architectural abstraction specifications for supporting software evolution. *Science of Computer Programming* 90, Part B (2014), 135–160. https://doi.org/10.1016/j.scico.2013.10.004 Special Issue on Component-Based Software Engineering and Software Architecture.

[17] Neil B. Harrison, Paris Avgeriou, and Uwe Zdun. 2007. Using Patterns to Capture Architectural Decisions. *IEEE Softw.* 24, 4 (July 2007), 38–45. https://doi.org/10.1109/MS.2007.124

[18] G. Hohpe, I. Ozkaya, U. Zdun, and O. Zimmermann. 2016. The Software Architect's Role in the Digital Age. *IEEE Software* 33, 6 (Nov 2016), 30–39. https://doi.org/10.1109/MS.2016.137

[19] Pooyan Jamshidi, Claus Pahl, Samuel Chinenyeze, and Xiaodong Liu. 2015. *Cloud Migration Patterns: A Multi-cloud Service Architecture Perspective*. Springer International Publishing, 6–19. https://doi.org/10.1007/978-3-319-22885-3_2

[20] Pooyan Jamshidi, Claus Pahl, and Nabor C. Mendonça. 2017. Pattern-based multi-cloud architecture migration. *Software: Practice and Experience* 47, 9 (2017), 1159–1184. https://doi.org/10.1002/spe.2442

[21] Nenad Medvidovic, Richard N. Taylor, and David S. Rosenblum. 1998. An Architecture-Based Approach to Software Evolution. In *In Proceedings of the International Workshop on the Principles of Software Evolution*. 11–15.

[22] Wilma Mert and Sebastian Webel. 2016. From Big Data to Smart Data: Do-it-all Drones. https://www.siemens.com/innovation/en/home/pictures-of-the-future/digitalization-and-software/from-big-data-to-smart-data-smart-drones.html. (2016).

[23] Konstantinos Plakidas, Daniel Schall, and Uwe Zdun. 2018. Software Migration and Architecture Evolution with Industrial Platforms: A Multi-Case Study. In *Proceedings of the 12th European Conference on Software Architecture*. Springer, Madrid, Spain.

[24] Chris Richardson. 2014–2018. Microservices.io. http://microservices.io/. (2014–2018).

[25] Siemens. 2015. MindSphere: Siemens Cloud for Industry. https://www.siemens.com/customer-magazine/en/home/industry/digitalization-in-machine-building/mindsphere-siemens-cloud-for-industry.html. (November 2015).

[26] Markus Völter, Michael Kircher, and Uwe Zdun. 2004. *Remoting Patterns: Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. John Wiley & Sons, Hoboken, NJ, USA. http://eprints.cs.univie.ac.at/2380/

[27] Rainer Weinreich and Georg Buchgeher. 2012. Towards Supporting the Software Architecture Life Cycle. *J. Syst. Softw.* 85, 3 (Mar 2012), 546–561. https://doi.org/10.1016/j.jss.2011.05.036

[28] Byron J. Williams and Jeffrey C. Carver. 2010. Characterizing software architecture changes: A systematic review. *Information and Software Technology* 52, 1 (2010), 31–51. https://doi.org/10.1016/j.infsof.2009.07.002

[29] Uwe Zdun, Rafael Capilla, Huy Tran, and Olaf Zimmermann. 2013. Sustainable Architectural Design Decisions. *IEEE Softw.* 30, 6 (Nov. 2013), 46–53. https://doi.org/10.1109/MS.2013.97