A Modeling Method for Systematic Architecture Reconstruction of Microservice-Based Software Systems

Florian Rademacher[1(✉)], Sabine Sachweh[1], and Albert Zündorf[2]

[1] IDiAL Institute, University of Applied Sciences and Arts Dortmund, Otto-Hahn-Straße 27, 44227 Dortmund, Germany
{florian.rademacher,sabine.sachweh}@fh-dortmund.de

[2] Department of Computer Science and Electrical Engineering, University of Kassel, Wilhelmshöher Allee 73, 34121 Kassel, Germany
zuendorf@uni-kassel.de

Abstract. Microservice Architecture (MSA) is an approach to architecting service-based software systems, which aims for decreasing service coupling to enable independent service development and deployment. Consequently, the adoption of MSA is expected to particularly benefit the scalability, maintainability, and reliability of monolithic systems. However, MSA adoption also increases architectural complexity in service design, implementation, and operation. As a result, Software Architecture Reconstruction (SAR) of microservice architectures is aggravated. This paper presents a modeling method that systematizes SAR of microservice architectures with the goal to facilitate its execution. The method yields reconstruction models for certain architecture viewpoints in MSA to enable efficient architecture analysis. We validate the method's applicability by means of a case study architecture and the assessment of its risk in technical debt using derived reconstruction models.

Keywords: Microservice architecture · Software Architecture Reconstruction · Model-driven engineering · Modeling languages

1 Introduction

Microservice Architecture (MSA) is a novel approach to architecting service-based software systems that puts a strong emphasis on *service-specific independence* [11]. MSA promotes to (i) tailor services to exactly one, distinct capability; (ii) shift responsibilities in a service's design, development, and deployment to a single team composed of members with heterogeneous professional skills; and (iii) keep services executable, testable, and deployable in isolation [10,11].

MSA is expected to benefit quality attributes like scalability, maintainability, and reliability [11]. Thus, it is frequently used to refactor monolithic systems for which these quality attributes decreased critically [16].

© Springer Nature Switzerland AG 2020
S. Nurcan et al. (Eds.): BPMDS 2020/EMMSAD 2020, LNBIP 387, pp. 311–326, 2020.
https://doi.org/10.1007/978-3-030-49418-6_21

However, MSA adoption increases architectural complexity significantly. For example, MSA architects and developers need to make sure that microservices do not become too fine-grained to lower network load [4]. Additionally, MSA allows for choosing different technologies per technical concern and microservice, which can increase learning curves for developers and the risk for technical debt [15]. Moreover, MSA requires a sophisticated operation infrastructure to enable independent service deployment and DevOps practices [17], as well as the provisioning of components for, e.g., service discovery and monitoring [2].

The different degrees in complexity aggravate *software architecture reconstruction* (SAR) [3] of MSA-based software systems [1]. While SAR is key to architecture verification, conformance checking, and trade-off analysis [3], research on SAR of microservice architectures is still formative [1,6].

In this paper, we present a modeling method that systematizes SAR of microservice architectures with the goal to guide its structured execution. The method builds upon our previous research on model-driven MSA engineering, in which we developed a set of modeling languages for the specification of microservice architectures [12,13] based on *architecture viewpoints* [3]. Our modeling method exploits these languages to capture reconstructed architecture information in *reconstruction models*. They aim to facilitate architecture analysis in the context of MSA. We validate the applicability of our SAR modeling method by means of a case study microservice architecture and the assessment of its risk in technical debt [14] leveraging the derived reconstruction models.

The remainder of the paper is organized as follows. Section 2 presents background information on SAR and our languages for model-driven MSA engineering. Section 3 introduces our method for systematic SAR of microservice architectures. In Sect. 4, we apply the method to a case study architecture and assess its risk in technical debt using the reconstruction models. Section 5 discusses our approach. Section 6 presents related work and Sect. 7 concludes the paper.

2 Background

This section presents background information on SAR and an overview of our modeling languages for viewpoint-based, model-driven MSA engineering.

2.1 Software Architecture Reconstruction

SAR is an iterative reverse engineering process that derives a representation of a software architecture from artifacts like documentation or source code [3]. It aims to document architecture implementations, which lack thorough documentation, and enable subsequent architecture analysis. SAR consists of four phases [3]:

1. *Raw view extraction*: Gathers architecture information from architecture-related artifacts. Each set of artifact-specific information can be considered a *view* that represents certain architecture elements and their relations [3].
2. *Database construction*: Transforms views into a canonical representation and stores them in a structured form like a database.

3. *View fusion and manipulation*: Combines views to improve accuracy of reconstructed information. For instance, a domain view may be associated with a component view to document runtime processing of domain data.
4. *Architecture analysis*: Aims for answering hypotheses about architecture implementations from reconstructed architecture information.

2.2 Viewpoint-Based Modeling of Microservice Architectures

In our previous works, we developed a set of modeling languages for model-driven MSA engineering [12,13]. Each language focuses on an architecture viewpoint in MSA and thus addresses the concerns of certain MSA stakeholder groups [3]. In the following, we provide an overview of our modeling languages per viewpoint.

Domain Viewpoint. This viewpoint addresses the concerns of domain experts and service developers in MSA engineering. Its Domain Data Modeling Language [13] enables both stakeholder groups to collaboratively construct *domain models* and augment them with patterns from Domain-driven Design (DDD) [7,11].

Technology Viewpoint. This viewpoint focuses on service developers and operators. Its Technology Modeling Language [12] allows for constructing *technology models* that prescribe available technologies for microservice implementation and operation. In addition, generic *technology aspects* may be modeled to augment, e.g., microservices with technology-specific access means and configuration.

The viewpoint also comprises the Technology Mapping Language. *Mapping models* extend domain and service models with technology information.

Service Viewpoint. The viewpoint's Service Modeling Language [13] targets the Dev perspective in DevOps-based MSA teams [10]. It enables service developers to construct *service models* that specify microservices, interfaces, and endpoints.

Operation Viewpoint. The Operation Modeling Language [13] targets the Ops perspective [10]. It enables service operators to construct *operation models* that describe service deployment and infrastructure for, e.g., service discovery and monitoring [2].

The modeling languages integrate an import mechanism to connect viewpoints' models and create coherent architecture descriptions. For example, service models may import domain models to relate domain concepts with microservices. Moreover, operation models may import service models to express service deployment and infrastructure usage.

3 A Modeling Method for Systematic Reconstruction of Microservice Architecture

It is the goal of our modeling method to systematize SAR of microservice architectures. To this end, we explore the application of our viewpoint-based languages (cf. Subsect. 2.2), which were originally developed to enable model-driven MSA engineering [13], for realizing the SAR process described in Subsect. 2.1.

Figure 1 shows our SAR modeling method in a UML activity diagram. The sequence of its six activities follows the relationships between MSA viewpoints (cf. Subsect. 2.2). Each activity targets certain phases of the SAR process (cf. Subsect. 2.1) and is described in the following subsections together with example reconstruction models expressed in corresponding modeling languages.

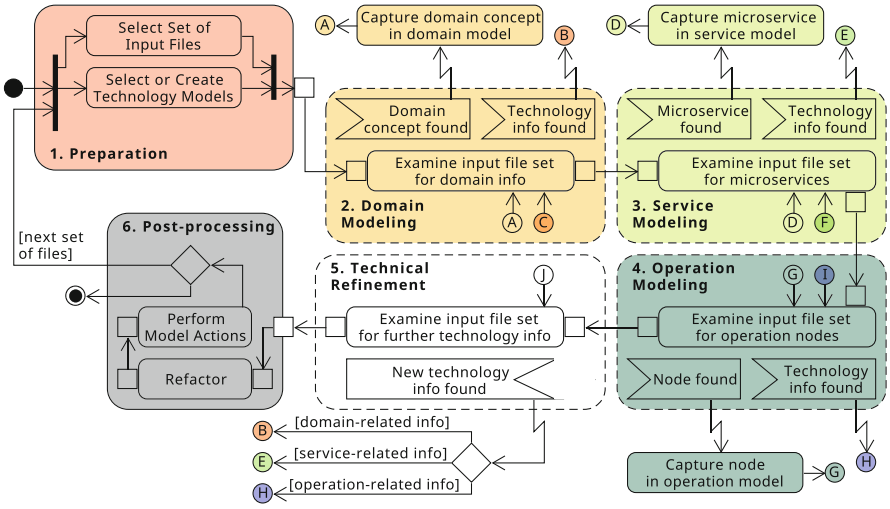

Fig. 1. Definition of our SAR modeling method in a UML activity diagram.

3.1 Activity 1: Preparation

Each instance of the SAR modeling method starts with the Preparation activity (cf. Fig. 1). In its first action, a set of input files, from which the examined microservice architecture shall be reconstructed, is selected. Such files may contain, e.g., documentation, source code, build scripts, or configuration values [1].

In the second action, technology models (cf. Subsect. 2.2) are selected or created. In case the technologies employed by the examined architecture are known, already existing technology models, e.g., constructed in previous method instances, may be reused. If no such models exist or the technology stack is unknown, empty technology models are created. A reasonable default is the creation of three technology models for domain-, service-, and operation-related technologies to be discovered (cf. Subsects. 3.2 to 3.4).

The Preparation activity contributes to Phase 1 of the SAR process as they identify the sources to extract architecture information from (cf. Subsect. 2.1).

3.2 Activity 2: Domain Modeling

This activity examines the selected input file set for domain concepts (cf. Fig. 1). In object-oriented programming languages like Java, domain concepts may be

realized by POJOs[1], i.e., classes that implement concepts of the application domain independent of external frameworks. Discovered domain concepts are captured in reconstruction domain models via our Domain Data Modeling Language (cf. Subsect. 2.2). Listing 1 shows a reconstruction domain model excerpt.

Listing 1. Excerpt of a reconstruction domain model (cf. Subsect. 2.2).

```
1 // Reconstruction domain model "customerCore.data"
2 context customer { structure Address <valueObject> {
3     string streetAddress, string postalCode, string city } }
```

The reconstruction domain model excerpt captures a reconstructed domain concept called `Address` as a structure with three string fields, i.e., `streetAddress`, `postalCode`, and `city`. The structure carries the semantics of a DDD Value Object [7]. Value Objects are typically immutable and lack a domain-specific identity. Hence, they may act as value containers for data exchange.

The `Address` structure belongs to the `customer` Bounded Context. In DDD, a Bounded Context is a means to cluster domain concepts and constrain their scope [7]. Bounded Contexts are crucial to MSA, since each microservice should be responsible for exactly one context [11]. Thus, when the input file set of an instance of our SAR modeling method (cf. Subsect. 3.1) belongs to a single service, the Domain Modeling activity should yield a domain model with only one context. In case the input files concern several services, ambiguous assignment of discovered domain concepts to contexts hints at wrong service tailoring.

During the examination of input files for domain concepts, technology-related information, e.g., for mapping concept instances to database tables, may be discovered. In this case, our modeling method delegates to a *technology modeling sub-activity* via activity edge connector "B" (cf. Fig. 1). It handles the occurrence of technology-related information in domain concepts and is shown in Fig. 2.

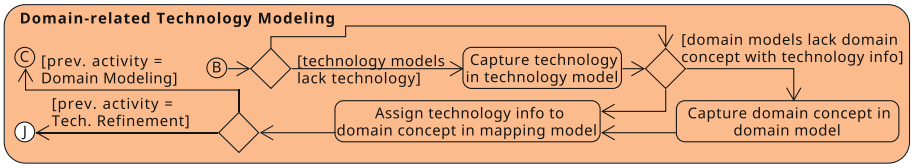

Fig. 2. Domain-related technology modeling sub-activity

Starting at connector "B", the sub-activity checks if the discovered technology information was already captured in a technology model. Otherwise, it is added to a suitable existing technology model (cf. Subsect. 3.6) or to the domain-related technology model created in Activity 1 (cf. Subsect. 3.1). Next, the domain concept itself is captured in a domain model if it was not yet, which may happen, e.g., for Java code where annotations like `@Table` from the Java Persistence

[1] https://www.martinfowler.com/bliki/POJO.html.

API (JPA)[2] are placed before class definitions. Finally, the discovered technology information is assigned to the captured domain concept within a mapping model (cf. Subsect. 2.2). Listing 2 shows a reconstruction domain-related technology model and mapping model derived during the Domain Modeling SAR activity.

Listing 2. Reconstruction domain-related technology model and reconstruction mapping model (cf. Subsect. 2.2).

```
1 // Reconstruction domain-related technology model "domainTech.technology"
2 technology domainTech {service aspects{ aspect Table for types { string name; } }}
3 // Reconstruction mapping model "customerCore.mapping"
4 import technology from "domainTech.technology" as domainTech
5 @technology(domainTech)
6 type customer::Address { aspects { domainTech::_aspects.Table("addresses"); } }
```

Line 2 specifies an aspect in a technology model [12]. The aspect reflects the discovered `@Table` annotation. The mapping model in Lines 4 to 6 imports the technology model and assigns the aspect to the `Address` data structure (cf. Listing 1) to prescribe address storage in a database table called "addresses".

The Domain Modeling activity and its successor Activities 3 to 5 (cf. Subsects. 3.3 to 3.5) cover SAR Phases 2 and 3 (cf. Subsect. 2.1). Discovered architecture information is transformed into canonical forms, i.e., models (Phase 2), which are then combined via imports to reflect architectural relations (Phase 3).

3.3 Activity 3: Service Modeling

The Service Modeling activity (cf. Fig. 1) examines the input file set for microservices and related information, which are then to be captured in service models (cf. Subsect. 2.2). In Java-based microservice architectures such information may be found, e.g., in classes that employ annotations for web-based data binding like `@RestController` and `@GetMapping` from the Spring[3] framework. Docker Compose[4] and build scripts also support microservice identification [1].

Similarly to Activity 2 (cf. Subsect. 3.2), discovered technology information is handled in a dedicated sub-activity. This *service-related technology modeling sub-activity* is entered via activity edge connector "E" (cf. Fig. 1). It proceeds analogously to the domain-related technology modeling sub-activity of Activity 2 (cf. Fig. 2), but captures newly discovered microservices in service models and returns to the current method instance via edge connector "F" (cf. Fig. 1).

Listing 3 shows reconstruction models that result during the Service Modeling activity, i.e., a service-related technology model, a service model, and a mapping model (cf. Subsect. 2.2).

[2] https://jakarta.ee/specifications/platform/8/apidocs/javax/persistence/Table.html.
[3] https://spring.io.
[4] https://docs.docker.com/compose.

Listing 3. A reconstruction service-related technology model, a reconstruction service model, and a reconstruction mapping model (cf. Subsect. 2.2).

```
1  // Reconstruction service-related technology model "serviceTech.technology"
2  technology serviceTech {
3    protocols { sync http data formats json; }
4    service aspects { aspect PutMapping for operations; } }
5  // Reconstruction service model "customerCore.services"
6  import datatypes from "customerCore.data" as domainData
7  functional microservice com.example.CustomerCore { interface management {
8    changeAddress(sync id : long, sync address : domainData::customer.Address); } }
9  // Reconstruction mapping model "customerCore.mapping"
10 import technology from "serviceTech.technology" as serviceTech
11 import microservices from "customerCore.services" as services
12 @technology(serviceTech)
13 services::com.example.CustomerCore {
14   protocols { sync: serviceTech::_protocols.http }
15   endpoints { serviceTech::_protocols.http: "/customers"; }
16   operation management.changeAddress {
17     aspects { serviceTech::_aspects.PutMapping; } } }
```

The technology model in Lines 2 to 4 comprises discovered protocols and technology aspects [12]. Line 3 captures the HTTP protocol and JSON data format. Line 4 defines the `PutMapping` aspect for the eponymous Spring annotation. Lines 6 to 8 show a service model, which imports the domain model in Listing 1 to refer to the `Address` domain concept. Lines 7 to 8 capture the discovered `CustomerCore` service. Its `management` interface clusters the `changeAddress` operation, whose `address` parameter is typed with the `Address` concept.

Lines 10 to 17 of Listing 3 show a mapping model. According to the examined input files, it specifies that the `CustomerCore` service (i) uses the HTTP protocol from the service-related technology model (Line 14); (ii) has an HTTP endpoint in the form of a URI path (Line 15); and (iii) enables invocation of the `changeAddress` operation via Spring's `@PutMapping` annotation (Lines 16 and 17).

3.4 Activity 4: Operation Modeling

This activity captures discovered operation nodes (cf. Fig. 1) in reconstruction operation models (cf. Subsect. 2.2). Operation nodes may represent *containers* for service deployment [10] or provide infrastructure capabilities to the examined architecture [2]. From the input file set, containers and infrastructure nodes may be discovered in Dockerfiles[5], or Docker Compose or build files [1], respectively.

The Operation Modeling activity invokes a sub-activity when operation-related technology is discovered (cf. activity edge connector "H" in Fig. 1). The sub-activity proceeds analogously to the domain-related technology modeling sub-activity (cf. Fig. 2), but captures newly discovered operation nodes in operation models. It returns to a method instance via connector "I" (cf. Fig. 1).

Listing 4 shows reconstruction models derived during the execution of the Operation Modeling SAR activity.

[5] https://docs.docker.com/engine/reference/builder/.

Listing 4. Reconstruction operation-related technology model and reconstruction operation model (cf. Subsect. 2.2).

```
1  // Reconstruction operation-related technology model "operationTech.technology"
2  technology operationTech {
3      protocols { sync http data formats json; }
4      deployment technologies {
5          Docker { operation environments = "openjdk:8-jre" default; } }
6      operation aspects {
7          aspect Dockerfile<singleval> for containers {
8              selector(technology = Docker); string contents <mandatory>; } } }
9  // Reconstruction operation model "customerCore.operation"
10 import microservices from "customerCore.services" as customerServices
11 import technology from "operationTech.technology" as opTech
12 @technology(opTech)
13 container CC_Container deployment technology opTech::_deployment.Docker
14     deploys customerServices::com.example.CustomerCore {
15         aspects { opTech::_aspects.Dockerfile("FROM openjdk:8-jre AS build ...");
               }
16         default values { basic endpoints {
17             opTech::_protocols.http: "http://localhost:8110"; } } }
```

Lines 2 to 8 of Listing 4 capture discovered technology information in an operation-related technology model. Line 3 specifies the HTTP protocol with the JSON format. Lines 4 to 5 model the discovered deployment technology Docker[6]. Its default operation environment (Line 5) corresponds to the Docker image discovered as being used to execute microservices. Lines 6 to 8 define the `Dockerfile` aspect. It can be used in combination with the Docker deployment technology to capture reconstructed Dockerfile contents in operation models.

Lines 10 to 17 of Listing 4 show a reconstruction operation model for the `CustomerCore` microservice. The service's reconstruction model (cf. Listing 3) is imported in Line 10. The `CC_Container` uses the Docker deployment technology (Line 13) from the operation-related technology model to deploy the service (Line 14). In Line 15, the `Dockerfile` aspect captures discovered Dockerfile contents, and Lines 16 and 17 determine a discovered container endpoint.

3.5 Activity 5: Technical Refinement

This activity focuses on discovering technology information (cf. Fig. 1), which was yet not captured in Activities 2 to 4 (cf. Subsects. 3.2 to 3.4). For example, the Spring framework allows for keeping microservice configuration separate from source code in distinct configuration files. Thus, these files may not have been examined in Activity 3 and are hence explicitly targeted by the Technical Refinement activity. In the event of discovering a yet not reconstructed technology information, the technology modeling sub-activity corresponding to the type of the new information is invoked via activity edge connectors "B", "E", or "H" (cf. Fig. 1 and the descriptions of the sub-activities in Subsects. 3.2 to 3.4).

The Technical Refinement SAR activity focuses on extending previously captured reconstruction models. Thus, it specifically targets Phase 3 of the SAR process (cf. Subsect. 2.1), i.e., the manipulation of derived architecture models.

[6] https://www.docker.com.

3.6 Activity 6: Post-processing

The Post-processing activity comprises two SAR actions (cf. Fig. 1). The Refactor action is concerned with refactoring reconstruction models and thus contributes to SAR Phase 3 (cf. Subsect. 2.1). For example, when conducting Activities 2 and 3 (cf. Subsects. 3.2 and 3.3) it is convenient to collect all discovered Bounded Contexts and their domain concepts, as well as all discovered microservices, in a single domain and a single service model, respectively. However, when adopting DDD in MSA engineering, domain and service models should be aligned to Bounded Contexts [11]. That is, a domain model should specify a single context and a related service model should only comprise microservices being responsible for concepts from that context (cf. Listings 1 and 3). Another Refactor task is to derive reusable technology models from domain-, service-, and operation-related technology models (cf. Subsect. 3.1). For instance, both service- and operation-related technology models in Listings 3 and 4 define the HTTP protocol. Both protocol specifications may therefore be merged into a technology model dedicated to clustering protocol specifications only.

The Post-processing activity concludes with executing actions, e.g., for architecture verification, conformance checking, or trade-off analysis [3], on reconstruction models. Hence, it covers Phase 4 of the SAR process (cf. Subsect. 2.1). Subsection 4.2 illustrates the processing of reconstruction models to assess indicators for the risk in technical debt of reconstructed microservice architectures.

4 Validation

In the following, we validate the applicability of our SAR modeling method (cf. Sect. 3) on a case study microservice architecture (cf. Subsect. 4.1). Moreover, we illustrate the usage of the reconstruction models in the Post-processing activity of our method (cf. Subsect. 3.6) on the example of assessing certain indicators for the risk in technical debt of the reconstructed architecture (cf. Subsect. 4.2).

4.1 Executing the Modeling Method on a Case Study Architecture

We validated the applicability of our SAR modeling method with a case study microservice architecture called "Lakeside Mutual" (LM)[7]. LM realizes an application for a fictitious insurance company. For the validation of our modeling method, we focused on LM's backend microservices, because they implement LM's domain concepts and business logic. Table 1 describes the capabilities of the examined microservices according to their documentation on GitHub (See footnote 7).

The following paragraphs summarize, per activity (cf. Sect. 3), the results from executing our SAR modeling method on the LM architecture. To enable reproducibility, we provide a comprehensive validation package on GitHub[8]. It includes the examined source code and the derived reconstruction models.

[7] https://github.com/Microservice-API-Patterns/LakesideMutual.
[8] https://github.com/SeelabFhdo/emmsad2020.

Table 1. Overview of the backend microservices of the LM case study architecture.

#	Service name	Capabilities
1	Customer Core	Manages LM customer data. The service provides REST endpoints [8] to interact with Services 2, 3, and 4
2	Customer Management Backend	Enables employees of LM's customer service to interact with customers
3	Customer Self-Service Backend	Allows customers for registering to an LM web portal, change their address, and view their insurance policy
4	Policy Management Backend	Provides management functionalities to LM employees regarding customers' insurance policies

Activity 1: Preparation. We used the files in the source code folders of LM, which correspond to the examined microservices (cf. Table 1), and the files on the top-level folder hierarchy of LM's repository (See footnote 7), e.g., "docker-compose.yml", as input file set. Together, the set comprised 160 files with 8858 lines of code (LOC). Moreover, we created empty technology models for Activities 2, 3, and 4 (cf. Subsect. 3.1), because we were not aware of the technologies employed by LM.

Activity 2: Domain Modeling. We created a reconstruction domain model with a Bounded Context for each LM backend microservice (cf. Subsect. 3.2 and Table 1). The domain concepts in the contexts were reconstructed from Java classes found in the corresponding services' source code folders. Whenever recognizable from their classes, DDD information were added to domain concepts. For example, classes that represent Data Transfer Objects (DTOs) [5] were modeled as DDD Value Objects, since in MSA they are used to prescribe data exchange [13]. In total, we reconstructed 99 domain concepts from the input files.

We also discovered that LM uses JPA for database mapping of domain concepts and Spring for DTO serialization. Hence, technology aspects were created in the domain-related technology model (cf. Subsect. 3.2). They reflect, e.g., JPA's `@Table` annotation and Spring's `@ResourceParam`[9] annotation. Aspects were then assigned to domain concepts in mapping models (cf. Listing 2).

Activity 3: Service Modeling. We created a reconstruction service model for each examined microservice (cf. Subsect. 3.3 and Table 1). Microservice elements were reconstructed from Java classes that employed Spring annotations for web controllers and mapping of HTTP methods (cf. Subsect. 3.3). For all four services, we discovered 13 interfaces with 39 operations.

Based on the Spring annotations for HTTP mapping, we reconstructed 16 operations with an explicit REST endpoint (cf. Listing 3). The annotations

[9] https://docs.spring.io/spring-hateoas/docs/0.25.3.BUILD-SNAPSHOT/api.

were captured in the service-related technology model and assigned to modeled microservices via mapping models (cf. Listing 3).

Activity 4: Operation Modeling. This activity discovered that each microservice source code folder of LM exhibits a Dockerfile (cf. Subsect. 3.4). Hence, we created a reconstruction operation model for each LM backend service (cf. Table 1) and specified a Docker container for each reconstructed microservice (cf. Listing 4). To capture Dockerfiles' contents, we modeled a dedicated `Dockerfile` technology aspect. From the source code of Service 4, we also reconstructed an ActiveMQ[10] broker as infrastructure node and the AMQP[11] protocol, which was added to the operation-related technology model (cf. Subsect. 3.4).

Activity 5: Technical Refinement. In this activity, we reconstructed information from yet unconsidered input files (cf. Subsect. 3.5). For example, from Spring-related configuration files like "application.properties" we extracted information about URIs and ports of microservice containers (cf. Listing 4). Moreover, LM's "docker-compose.yml" file provided us with information about service interactions. As a result, we extended the reconstruction models of Services 2, 3, and 4 (cf. Table 1) to *require* Service 1 for their operation. For this purpose, our Service Modeling Language defines the `required microservices` statement [13].

Activity 6: Post-processing. We refactored the domain-, service-, and operation-related technology models created during Activities 2 to 4 (cf. Subsect. 3.6). To this end, we first merged all three models and removed duplicate information. Next, we split the merged technology model into six models, each dedicated to a certain technology. For example, the "java" technology model only clusters Java- and Spring-related information, while the "activemq" and "docker" technology models only focus on the eponymous technologies. The refactored technology models can thus be reused in future instances of the SAR modeling method.

Next, we processed the reconstruction models for assessing certain indicators of LM's architecture concerning its risk in technical debt (cf. Subsect. 4.2).

4.2 Post-processing Example: Technical Debt Assessment

Toledo et al. discovered indicators for MSA-specific *architectural technical debt* (ATD), which can be examined by analyzing service communication characteristics [14]. We illustrate the processing of reconstruction models derived by our SAR modeling method (cf. Subsect. 4.1) to assess these indicators and thus a part of LM's risk in technical debt. The following paragraphs describe, per ATD type [14], our findings from processing LM's reconstruction models.

[10] https://activemq.apache.org.
[11] https://www.amqp.org.

Too Many Point-to-Point (PtP) Connections. PtP connections between microservices are identifiable from reconstruction models by two characteristics. First, an infrastructure node realizes a *communication layer* [14] used only by a small subset of services. Second, services not using the node require other services. LM defines a communication layer with an ActiveMQ node used by Service 4 only (cf. Activity 4 in Subsect. 4.1). It is thus likely that Services 2 to 4, which require Service 1 (cf. Activity 5 in Subsect. 4.1), interact with it via PtP connections.

Business Logic Inside Communication Layer. Reconstruction domain and mapping models may indicate this ATD type. For instance, the Domain Data Modeling Language (cf. Subsect. 2.2) allows for declaring function signatures in data structures. Thus, functions conveying the semantics of data format conversions hint at this ATD type [14]. These functions usually take a single input parameter typed by a Value Object that reflects a DTO (cf. Activity 2 in Subsect. 4.1) and return an instance of another data structure within the Bounded Context of the function. Moreover, this ATD type is also indicated by protocol assignments in mapping models (cf. Listing 3). Microservices, that convert requests for use by other services, exhibit endpoints, whose protocols differ from the majority of services in the same Bounded Context. Our analysis of the reconstruction models showed that LM does not exhibit conversion domain functions or microservices.

No Standardized Communication Model. This ATD type is identifiable from reconstruction domain models (cf. Subsect. 3.2). They capture *business-related communication models* [14]. For example, data structures, which cannot be unambiguously assigned to a single Bounded Context, violate the principle of a *canonical domain model* [14]. We did not find such violations for LM.

Reconstruction domain models also allow for efficient analysis of the consistency of *shared domain concepts* [11]. For instance, the domain models of Services 2 to 4 (cf. Table 1) all specify the Value Object `CustomerProfileDto` as a DTO for the `CustomerProfileEntity` of Service 1. LM thus exhibits the risk that the Value Objects evolve differently from the Entity [7], although they are meant to be shareable representations of it. Our analysis of the three versions of `CustomerProfileDto` showed, however, that they all exhibit the same structure. Hence, they could be refactored into a single shared domain model.

Weak Source Code and Knowledge Management. Our SAR modeling method does not directly support in assessing this ATD type. However, reconstruction models provide a well-defined means for documenting views on microservice architectures (cf. Subsect. 2.1). Consequently, they can accompany *centralized MSA documentation* [14] as they capture architecture knowledge in a concise format.

Different Middleware Technologies for Service Communication. Reconstruction technology and mapping models facilitate identification of this ATD type, because they capture discovered communication technologies and their usage (cf. Listing 3). LM employs different means for synchronous and asynchronous

communication, i.e., REST and ActiveMQ (cf. Activities 3 and 4 in Subsect. 4.1). We consider LM's risk in this ATD type to only be slightly increased, as in MSA it is common to employ at most one protocol for each communication kind [11].

However, our analysis of reconstruction mapping models showed that more REST operations are invokable via an HTTP method (26) than explicit REST endpoints were specified (16). Such inconsistencies in services' communication specifications are likely to cause communication failures at runtime.

5 Discussion

For the validation of our SAR modeling method (cf. Sect. 4), we executed it manually on the input file set. We then ensured the correctness of the reconstruction models by comparing them with LM's documentation and double-checking their consistency with LM's source code. Consequently, we perceive our method to be basically applicable on microservice architectures. Nonetheless, a current threat to validity is the increased error-proneness given the manual execution of the method. However, this weakness may be mitigated by employing automated source code analysis techniques, particularly in SAR Activities 2 to 4 (cf. Subsects. 3.2 to 3.4). For example, in case of Java-based microservice architectures, class bodies and employed annotations, as well as Dockerfiles in general, represent valuable analysis targets (cf. Sect. 4).

The input file set selected in Activity 1 of our SAR modeling method (cf. Subsect. 3.1) depends on the availability of artifacts in the targeted microservice architecture. For instance, due to the structure of the case study architecture, the input file set for the method's validation mainly consisted of Java files (cf. Sect. 4). Hence, the reconstruction effort was relatively high, because all LOC needed to be examined. However, source code files that reflect domain concepts or service implementations may also be replaced, e.g., by concise models of database structures or API documentation. Like the SAR process (cf. Subsect. 2.1), our modeling method does not constrain input file types.

In its current form, the SAR modeling method directly aligns its activities to the viewpoints being addressed by our languages for model-driven MSA engineering and their relationships (cf. Fig. 1 and Subsect. 2.2). As a result, the method does not take the perspective of stakeholders like business analysts or project managers into account, yet. To this end, the set of SAR activities would need to be extended with modeling approaches tailored to stakeholders, who do not directly participate in software engineering in the context of MSA. Further research is necessary to identify the concerns of these stakeholders and derive corresponding SAR activities.

Since our method anticipates reconstruction of technology information, the degree of abstraction in reconstruction models may be comparatively close to that of source code. However, due to the usage of mapping models, reconstruction domain and service models are basically technology-agnostic (cf. Subsects. 3.2 and 3.3). Thus, the execution of sub-activities, which capture technology information, may be omitted in Activities 2 and 3, depending on the goal of the

conducted SAR process (cf. Subsect. 2.1) and technology information being irrelevant to its achievement.

6 Related Work

Alshuqayran et al. [1] conduct an empirical study on eight open source microservice architectures to derive a metamodel for SAR in MSA. They also analyze a set of heterogeneous input files that contain, e.g., Java source code, build scripts, and configuration files (cf. Sects. 3 and 4). The derived metamodel is similar to the ones of our Service and Operation Modeling Languages [13]. However, it does not support the reconstruction of domain concepts. Furthermore, technologies like `Asynchronous Message Bus` are fixed metamodel concepts, while with our Technology Modeling Language [12] they can flexibly be integrated in reconstruction models as they occur in input files. In addition, Alshuqayran et al. do not present a concrete syntax for their metamodel, nor do they specify its systematic usage in a SAR process like our modeling method.

MicroART [9] is a tool for reconstructing microservice architectures. It extracts service-related information, e.g., services' names, ports, and developers from source code repositories. Moreover, it performs a runtime analysis of log files in order to determine containers, network interfaces, and service interaction relationships. From the gathered information, MicroART instantiates a model from a specifically designed metamodel. Like our approach, MicroART is model-based. On the contrary, it does not consider the Domain, Operation, and Technology viewpoints (cf. Subsect. 2.2) when gathering architecture information. Furthermore, a systematic method and concrete syntax for facilitating architecture analyses is not presented.

Zdun et al. introduce an approach towards assessing MSA conformance [18]. Therefore, existing microservice architectures are reconstructed leveraging a formal model with MSA-specific component and connector types. MSA conformance of reconstructed architectures is then assessed via metrics and constraints defined by the relationships between these types. Like for our SAR modeling method, reconstructed formal models also need to be derived manually from existing architecture implementations. However, no modeling language with MSA-specific abstractions is employed to facilitate the creation of the formal models. Moreover, a systematic reconstruction method is not presented and domain-specific information is not considered.

7 Conclusion and Future Work

In this paper, we presented a modeling method for systematic Software Architecture Reconstruction (SAR) of software systems based on Microservice Architecture (MSA). The method employs a set of modeling languages for model-driven MSA engineering to capture reconstructed information in viewpoint-specific architecture models. Consequently, method instances yield domain, technology,

service, and operation reconstruction models for examined microservice architectures. These models aim to facilitate architecture analysis in the context of MSA. We validated our SAR modeling method with a case study microservice architecture and showed the applicability of derived reconstruction models on the example of assessing certain indicators of the case study's risk in technical debt.

In future works, we plan to investigate the extension of our SAR modeling method with automation capabilities. First, the derivation of the reconstruction models may be facilitated by automated source code analysis. Second, post-processing of reconstruction models would benefit from static model analysis in order to automatically gather metrics like the diversity of communication technologies, their usage by microservices, or the existence of duplicate domain concepts. Moreover, we are currently working on a code generator to produce source code and configuration files from reconstruction models. With the generator, architecture design and refactoring based on reconstruction models would become feasible, because architecture models and architecture implementation could be automatically kept consistent.

References

1. Alshuqayran, N., Ali, N., Evans, R.: Towards micro service architecture recovery: an empirical study. In: 2018 IEEE International Conference on Software Architecture (ICSA), pp. 47–56 (2018)
2. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables devops: migration to a cloud-native architecture. IEEE Softw. **33**(3), 42–52 (2016)
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 3rd edn. Addison-Wesley, Boston (2013)
4. Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A.: Microservices in industry: insights into technologies, characteristics, and software quality. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 187–195 (2019)
5. Daigneau, R.: Service Design Patterns. Addison-Wesley, Boston (2012)
6. Di Francesco, P., Malavolta, I., Lago, P.: Research on architecting microservices: trends, focus, and potential for industrial adoption. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 21–30. IEEE (2017)
7. Evans, E.: Domain-Driven Design. Addison-Wesley, Boston (2004)
8. Fielding, R.: Representational state transfer. Ph.D. thesis (2000)
9. Granchelli, G., Cardarelli, M., Francesco, P.D., Malavolta, I., Iovino, L., Salle, A.D.: Towards recovering the software architecture of microservice-based systems. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 46–53 (2017)
10. Nadareishvili, I., Mitra, R., Mclarty, M., Amundsen, M.: Microservice Architecture. O'Reilly Media, Sebastopol (2016)
11. Newman, S.: Building Microservices. O'Reilly Media, Sebastopol (2015)
12. Rademacher, F., Sachweh, S., Zündorf, A.: Aspect-oriented modeling of technology heterogeneity in microservice architecture. In: 2019 IEEE International Conference on Software Architecture (ICSA), pp. 21–30. IEEE (2019)

13. Rademacher, F., Sorgalla, J., Wizenty, P., Sachweh, S., Zündorf, A.: Graphical and textual model-driven microservice development. In: Bucchiarone, A., et al. (eds.) Microservices, pp. 147–179. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-31646-4_7
14. Soares de Toledo, S., Martini, A., Przybyszewska, A., Sjøberg, D.I.K.: Architectural technical debt in microservices: a case study in a large company. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), pp. 78–87. IEEE (2019)
15. Taibi, D., Lenarduzzi, V.: On the definition of microservice bad smells. IEEE Softw. **35**(3), 56–62 (2018)
16. Taibi, D., Lenarduzzi, V., Pahl, C.: Processes, motivations, and issues for migrating to microservices architectures: an empirical investigation. IEEE Cloud Comput. **5**, 22–32 (2017)
17. Taibi, D., Lenarduzzi, V., Pahl, C.: Continuous architecting with microservices and DevOps: a systematic mapping study. In: Muñoz, V.M., Ferguson, D., Helfert, M., Pahl, C. (eds.) CLOSER 2018. CCIS, vol. 1073, pp. 126–151. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29193-8_7
18. Zdun, U., Navarro, E., Leymann, F.: Ensuring and assessing architecture conformance to microservice decomposition patterns. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) ICSOC 2017. LNCS, vol. 10601, pp. 411–429. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69035-3_29