2019 IEEE International Conference on Software Architecture (ICSA)

Aspect-oriented Modeling of Technology Heterogeneity in Microservice Architecture

Florian Rademacher, Sabine Sachweh
IDiAL Institute
University of Applied Sciences and Arts Dortmund
Otto-Hahn-Straße 23, 44227 Dortmund, Germany
{florian.rademacher,sabine.sachweh}@fh-dortmund.de

Albert Zündorf
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 a novel approach for the development and deployment of service-based software systems. MSA puts a strong emphasis on loose coupling and high cohesion of services. This increases service-specific independence, which is considered to result in a better scalability, adaptability, and quality of software architectures.**

Another perceived benefit of adopting MSA is technology heterogeneity. Microservice teams are free to employ the technologies they deem to be most appropriate for service implementation and operation. However, technology heterogeneity increases the number of technologies in a microservice architecture with the risk to introduce technical debt and steeper learning curves for team members.

In this paper, we present an approach to streamline the usage of different technologies in MSA-based software systems. It employs Aspect-oriented Modeling to make technology decisions in microservice architectures explicit and enable reasoning about them. Therefore, a set of languages for model-driven microservice development is extended with means to define, modularize, and apply MSA technology aspects. The usage and characteristics of our approach are shown and discussed in the context of a case study from the mobility domain.

***Index Terms*—Service-oriented systems engineering, Model-driven development, Software architecture, Software design**

I. Introduction

Microservice Architecture (MSA) [1] is a novel approach for the development and deployment of service-based software systems [2]. It relies on *services* as functional components of the resulting software architecture, and puts a strong emphasis on loose coupling and high cohesion of services [1]. MSA focuses on decomposing the software architecture and its business domain along distinct functional and infrastructural capabilities. Therefore, each *microservice* is expected to be responsible for providing exactly one, distinct capability to the software architecture by also considering all phases of service engineering—from design over development to deployment. A microservice team is hence not only responsible for its services' implementations, but also for their build scripts, deployment descriptors, and operation configurations [3].

Based on the resulting increase of *service-specific independence*, employing MSA for the realization of distributed software systems is considered to yield several technical as well as organizational benefits. First, the system's scalability is increased. As every microservice can be deployed and run separately, it can also be scaled independently [1]. Second, the system's adaptability is increased. Provided that its interface remains stable, each microservice can be replaced with an alternative, possibly competing implementation [1]. This also fosters *permissionless innovation* [4]. Third, the system's overall resilience is increased, because failures are generally expected to happen at any point in runtime. Therefore, services need to be as robust as possible to prevent failure cascades [5]. Fourth, service quality and team productivity are expected to increase. That is, because microservices foster isolated testability, small team sizes, and directed, efficient team communication along service boundaries [3].

Moreover, MSA promotes *technology heterogeneity* [1]. Teams may use the technologies they believe to be most appropriate for realizing their microservices. They, for instance, have the freedom to align technology choices to (i) quality requirements, such as operation and network performance [6]; (ii) implementation requirements, such as programming languages that are specialized on implementing microservices and MSA communication paradigms [7]; or (iii) deployment-specific requirements [5]. Next to increased scalability and resilience, technology heterogeneity is considered to be one of the key benefits of MSA [1]. However, it also increases the potential number of technologies being involved in MSA engineering with the risk to introduce technical debt and steeper learning curves for team members [8].

In this paper, we present an approach to streamline and facilitate the usage of different technologies in MSA engineering. It aims to enable reasoning about technology heterogeneity by explicitly capturing technological decisions related to microservice development and deployment in models. Therefore, our approach is based on combining two orthogonal software development paradigms, i.e., Model-driven Development (MDD) [9] and Aspect-oriented Software Development (AOSD) [10]. Our contribution is threefold. First, we identify technology variation points of MSA-based software systems on the basis of a case study application. Starting from these variation points, common technology aspects of microservice architectures are derived. Second, we extend a set of languages for viewpoint-specific microservice modeling with means to express, cluster, and apply technology aspects. Third, we present an approach for the modeling of custom technology aspects.

978-1-7281-0528-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSA.2019.00011

The remainder of the paper is structured as follows. Section II introduces a case study to illustrate the relevance of technology heterogeneity in MSA and infer a set of variation points at which technologies in MSA-based software systems frequently differ. Section III provides background information on the combination of MDD and AOSD as the conceptual foundation of our approach. Section IV introduces a set of viewpoint-specific MSA modeling languages. They are extended in Section V to model, cluster, and apply custom as well as common technology aspects that are based on the technology variation points derived in Section II. Section VI demonstrates and discusses the application of our approach in the context of the case study. Section VII presents related work. Section VIII concludes the paper and outlines future work.

II. Motivating Case Study

In this section, we describe the case study application, which illustrates the relevance of technology heterogeneity in MSA. Therefore, we first introduce its domain context (cf. Subsection II-A). Next, we present its architecture and infer MSA technology variation points (cf. Subsections II-B and II-C).

A. Context

The case study application originates from a research project in the mobility domain. Within the project we developed a monolithic enterprise application for parking space accounting [11]. It receives data about vehicles that enter or exit a parking space. These data comprise vehicle types (car or truck), speeds, lengths, and locations. For data gathering, low-power radio modules are integrated into delineators at the entrances and exits of parking spaces. A group of six delineators forms a mesh of radio connections. In case a vehicle passes the mesh, the mentioned data are inferred for it based on characteristic variations in the radio connections. Table I lists selected capabilities, which the application provides to its users.

TABLE I
Selected Capabilities of the Case Study Application

#	Title	Description
C.1	Current Allocation	Users can request the current allocation of a parking space per vehicle type.
C.2	Search for Free Space	Users can leverage a mobile app to search for the next free parking space for their type of vehicle and within a given distance.
C.3	Allocations over Time	Parking space operators can request allocations of their parking spaces for a time period.
C.4	Parking Space Management	Operators can manage parking spaces and reset vehicle counters for calibration purposes.
C.5	Data Export	Operators can decide to export a parking space's data, e.g., name, location, and current allocation, to a marketplace for mobility data[1].

During the project it turned out that the monolithic architecture would not scale when extending its area of application across the project's model region. Further analyses showed that the main bottlenecks were the central database and the implementations of the application components. They could not sufficiently be parallelized due to technical reasons. Moreover, the handling of incoming sensor data was inefficient. First, it constituted a one-dimensional data flow. Incoming data was preprocessed and stored in the database. After that, the application components got informed that new data were received in order to update their internal data models. The update sequence in terms of capabilities was C.1, C.2, C.3, C.5 (cf. Table I). Second, the data flow was based on synchronous communication. For example, the component for capability C.1 first had to confirm the update of its data model before C.2 was requested to do so. To overcome the mentioned limitations and achieve the required scalability, we eventually decided to refactor the monolithic architecture to MSA.

B. Refactored Microservice Architecture

Fig. 1 shows the refactored, MSA-based architecture. Each capability listed in Table I is realized as a microservice. Therefore, the central database was functionally decomposed and coherent parts were assigned to the respective microservices [1]. The arrows in Fig. 1 show the data flows of the refactored architecture. The "API Gateway Service" acts as a scalable entry point to the application and exposes only relevant interfaces to external clients [5]. It receives sensor data from delineators via HTTP in a custom data format. Moreover, it enables users to interact with the application via a web portal and mobile app. Data exchange with these frontends happens resource-based on the basis of HTTP. The gateway service is also responsible for providing marketplaces with information. The "Service Discovery" is the architecture-internal registry for microservice instances to discover each other by logical instead of physical endpoints [5].

Within the architecture, asynchronous communication is employed to increase scalability. For example, the gateway service forwards incoming sensor data asynchronously to the "Current Allocation" and "Allocations over Time" microservices. Thus, sensor data processing is, in contrast to the monolithic implementation, now parallelized. Furthermore, updates of parking space information are sent asynchronously by the "Parking Space Management" service to other microservices as they occur. The management service hence resolves the one-dimensional data flow of the monolith and does not have to wait for each service to update its internal data model (cf. Subsection II-A).

C. Inferred Technology Variation Points

The case study architecture exhibits several *variation points* (VPs) regarding the employed technologies. Similarly to Software Product Line Engineering (SPLE) [12], we consider a VP as a place in a basic system design model where variations are possible in the resulting implementation. Based on that, we define a *technology variation point* (TVP) in the context of MSA as a VP at which different technologies may be chosen to

[1]https://www.mdm-portal.de/en

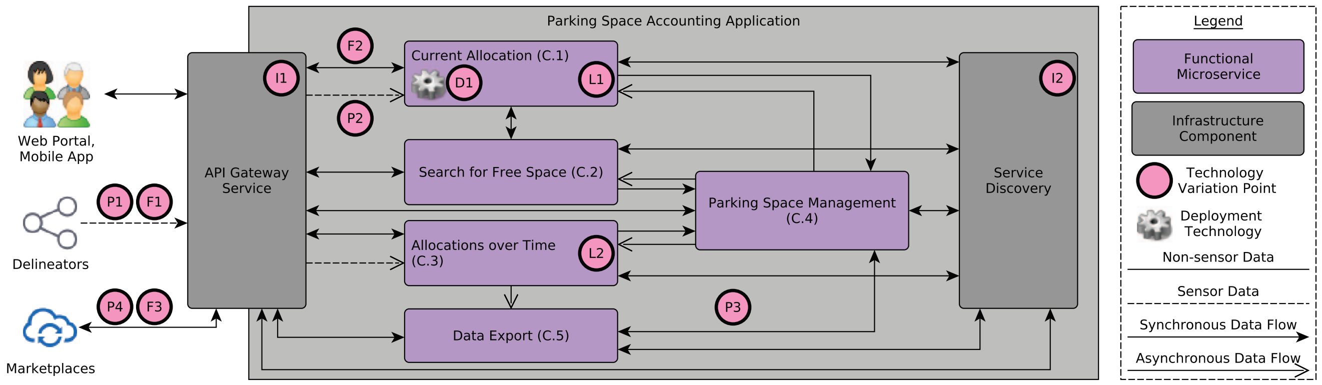

Fig. 1. Architecture overview of the refactored, MSA-based case study application.

realize the architecture. Thus, TVPs are the places in an MSA-based software system at which technology heterogeneity is accepted by intent.

Fig. 1 depicts the TVPs of the refactored case study application as colored circles. Each letter refers to a certain kind of TVP and numeric suffixes identify different TVP *variants* [12]. To keep the figure concise, however, each possible variant is shown only once. For example, variant "P3" applies to all synchronous data flows of the "Parking Space Management" service.

Table II elucidates the inferred TVPs and their variants.

TABLE II
MICROSERVICE ARCHITECTURE TVPS INFERRED FROM THE CASE STUDY

TVP	Description
Deployment Technology	The *containerization* of microservices increases deployment efficiency and speeds up feature delivery [3]. While the case study microservices are deployed in Docker containers[2] (variant **D1** in Fig. 1), alternative container technologies like rkt and LXC currently emerge [13].
Data **F**ormat	The case study application needs to support three different formats for data encoding. Delineators encapsulate sensor data in a custom format (variant **F1** in Fig. 1), while architecture-internal microservice interactions leverage JSON (**F2**). The communication with marketplaces employs XML (**F3**) for data encoding.
Infrastructure Technology	Infrastructure technology provides microservices with mechanisms like API consolidation (variant **I1** in Fig. 1) and service discovery (**I2**). However, there are certainly more, e.g., configuration providers [5], which are omitted in Fig. 1 for brevity. All infrastructure components of the refactored case study application were realized on the basis of the Spring Cloud framework[3].
Programming **L**anguage	MSA's technology heterogeneity fosters the usage of the most appropriate programming language for service implementation [7]. All microservices of the case study were realized in Java (variant **L1** in Fig. 1), except for "Allocations over Time". It was written in Python (**L2**) to leverage NumPy[4] for efficient time series processing.
Protocol	The case study employs four different protocol technologies. Delineators send sensor data via HTTP (variant **P1** in Fig. 1). AMQP[5] (**P2**) is employed for asynchronous and REST (**P3**) for synchronous, resource-based microservice communication. Data transmission to marketplaces is based on DATEX II [14] and hence SOAP [15] (**P4**) is used as communication protocol.

III. Aspect-oriented Modeling

Our approach towards coping with MSA's technology heterogeneity is based on Aspect-oriented Modeling (AOM) [10]. AOM is a combination of MDD and AOSD. MDD employs models as means of abstraction in the software engineering process. Its adoption introduces benefits such as facilitated reasoning about a system's architecture and behavior, and automatic code generation [9]. AOSD focuses on the modularization of concerns that crosscut a system's business functionality, e.g., logging, performance, and security, in order to separate them from functional components [10]. AOSD introduces benefits such as separated analysis of crosscutting concerns and flexible integration of concern-specific code.

In the context of AOSD, crosscutting concerns are abstracted in *aspects* [16]. An aspect comprises a portion of aspect-related code, e.g., a logger's invocation, called *advice*. *Join points* [10] represent the semantic elements of a system or implementation language, e.g., method bodies, to which advices can be applied. An aspect may define several *pointcuts* to constrain advice application to a certain join point by considering the join point's concrete peculiarity. For instance, a logging advice could be constrained to be applicable only to methods whose names exhibit a certain prefix. The process of applying advices to join points is called *weaving* [16]. AOM leverages modeling languages to provide means for aspect definition and modularization as well as join point and pointcut specification [17]. It aims to combine the benefits of MDD and AOSD to, e.g., enable flexible weaving of advices into functional models and automatically derive implementations that comprise woven advice code. In the following, we consider the TVPs and their variants described in Table II as *technology aspects* and *technology advices*.

We decided to employ AOM for capturing and organizing technology decisions in MSA-based software systems due to experiences from our previous work in which we developed modeling languages to enable MDD of MSA (cf. Section IV).

[2]https://www.docker.com
[3]https://www.spring.io/projects/spring-cloud
[4]https://www.numpy.org
[5]https://www.amqp.org

Initially, the languages included technology-specific concepts to, e.g., specify concrete protocols for microservice interactions. However, with this approach the languages needed to be continuously extended with new concepts as soon as new technologies arose. Given MSA's high degree of technology heterogeneity, this would eventually result in complicated languages with lots of concepts. In addition, for each new considered technology, model validation and code generation mechanisms needed to be adapted. By contrast, AOM allows us to separate technology-dependent information from functional models. First, this promotes dedicated reasoning about technology decisions in MSA-based software systems. Second, it keeps the modeling languages concise and stable. Third, the languages need not comprise model validation and code generation mechanisms for a variety of technologies. Instead, those mechanisms can be developed, evolved, and provided together with the specifications of the MSA technologies they support (cf. Section V).

IV. Viewpoint-specific Microservice Architecture Modeling Languages

To make the paper self-contained, this section briefly introduces our previous work on enabling MDD of MSA. It consists of three modeling languages, each of which addresses a different viewpoint in MSA engineering. They will be adapted in Section V to include means for technology aspect and advice definition, modularization, and application (cf. Section III). A detailed description of the languages and their concepts can be found in [18].

Each modeling language addresses the concerns of a specific role in a DevOps-based microservice team. Next to "service developer" and "service operator", "domain expert" is considered as an additional supportive role [3]. Each role has a different *modeling viewpoint* [19] on an MSA-based software system and therefore a dedicated modeling language to construct models that constitute *views* from the role's viewpoint on the system. The introduction of explicit modeling viewpoints allows to decompose the considered system's complexity into specialized role- and concern-specific modeling tasks [17]. Thus, each role in a microservice team is only confronted with modeling the information being of specific relevance for it. This facilitates the application of MDD for MSA engineering and generally increases modeling efficiency [17].

Models of different viewpoints can be composed into a coherent microservice architecture. For instance, service operators construct deployment models for microservices being modeled by service developers. Based on the models and their compositions, sophisticated MDD techniques like model validation and automatic generation of, e.g., microservice code and deployment descriptors, can be employed.

Fig. 2 shows the concepts of the viewpoint-specific MSA modeling languages and their composition relationships. The latter are depicted as arrows pointing from one viewpoint language to those concepts of another viewpoint language on which the composition of both languages is based.

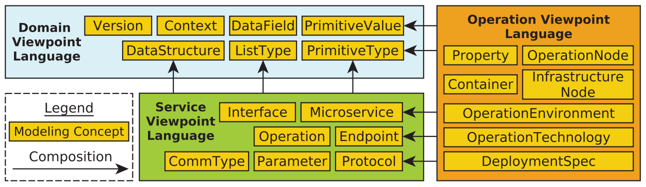

Fig. 2. Concepts and composition relationships of the viewpoint-specific MSA modeling languages.

A. Domain Viewpoint Language

The Domain Viewpoint Language enables domain experts and service developers to capture static domain concepts in data structures and list types (cf. Fig. 2). A data structure clusters a set of named data fields. A data field's type may either be (i) primitive; (ii) a previously defined data structure; or (iii) a previously defined list type, which describes sequences of primitive or structured values. Modeled domain concepts can be clustered in contexts that correspond to *bounded contexts* and thus determine the functional responsibilities of derived microservices [1]. Contexts, data structures, and list types may further be organized in versions.

B. Service Viewpoint Language

The Service Viewpoint Language enables service developers to model microservices, their interfaces and operations (cf. Fig. 2). It also allows to assign logical endpoints to services and their structural components. For instance, when using REST for service interaction (cf. Subsection II-C), endpoints correspond to the path fragments of the URI at which consumers can invoke services. The specification of protocols in a service interaction is possible with the `Protocol` modeling concept. A protocol specification has a synchronous or asynchronous communication type (`CommType`).

The Service Viewpoint Language refers to modeling concepts of the Domain Viewpoint Language (cf. Fig. 2). The type of an operation parameter may be either a primitive type, a data structure, or list type. Such *viewpoint compositions*, in which one viewpoint refers to another viewpoint's concepts, are realized via an import mechanism. That is, if a service model imports a domain model, data structures and list types from the latter are referenceable in the service model.

C. Operation Viewpoint Language

The Operation Viewpoint Language supports service operators in modeling microservice deployment and operation. Therefore, microservices being imported from service models are assigned to operation nodes (cf. Fig. 2). An operation node is either a container or an infrastructure node. Containers exhibit service deployment semantics, i.e., assigned microservices are to be run on containers. Infrastructure nodes provide generic, technical capabilities to assigned services for, e.g., API consolidation and service discovery (cf. Section II).

Services are assigned to operation nodes via the `DeploymentSpec` concept. It associates a service with a protocol-specific endpoint to, e.g., specify the scheme and authority fragments of a URI used for REST interactions. Deployment specifications may also encompass properties with primitive values. They model deployment configuration parameters like the minimum and maximum microservice instances on a container. For each operation node, an operation technology and environment need to be specified.

D. Case Study Example

Listing 1 illustrates the syntax and usage of the modeling languages. It comprises excerpts of viewpoint models for the "Allocations over Time" service (cf. Subsection II-B).

Listing 1
EXAMPLE MODELS IN EACH OF THE THREE MODELING LANGUAGES

```
1  // Domain model expressed in the Domain Viewpoint Language
2  // (model file: "case_study.data")
3  structure VehicleCount {
4    string vehicleType, int count, date timestamp }
5  list VehicleCounts { VehicleCount vehicleCount }
6  // Service model expressed in the Service Viewpoint
7  // Language (model file: "case_study.services")
8  import datatypes from "case_study.data" as domain
9  functional microservice com.example.AllocationsOverTime
10 (sync = REST/JSON, async = AMQP/JSON) {
11   interface AllocationsOverTime {
12     getAllocations(sync in parkingSpaceId : long,
13       sync in beginTimestamp : date,
14       sync in endTimestamp : date,
15       sync out allocations : domain::VehicleCounts);}}
16 // Operation model expressed in the Operation Viewpoint
17 // Language (model file: "case_study.operation")
18 import microservices from "case_study.services" as services
19 docker container AllocationsOverTime image python
20 deploys services::com.example.AllocationsOverTime
21   services::com.example.AllocationsOverTime {
22     basic endpoints {
23       REST/JSON: "http://example.com:8080";
24       AMQP/JSON: "amqp://example.com:5672"; } } }
```

Lines 3 to 5 show an excerpt of the domain model. It defines the "VehicleCount" data structure and the "VehicleCounts" list type to count occurrences of a vehicle type per timestamp.

Lines 8 to 15 comprise an excerpt of the service model for the "Allocations over Time" microservice. Line 10 specifies the protocols and data formats for synchronous and asynchronous interactions with the service. Effectively, the modeled protocol and data format combinations "REST/JSON" and "AMQP/JSON" represent advices (cf. Section III) for the technology aspects of the Protocol and Data Format TVPs (cf. Table II). Lines 12 to 15 define the operation "getAllocations", which is invoked when a parking space operator requests the allocations of a parking space. It has four parameters. The incoming parameters "parkingSpaceId", "beginTimestamp", and "endTimestamp" determine the parking space, whose allocations are requested, and the considered time period. The outgoing parameter "allocations" returns the calculated allocations as an instance of the "VehicleCounts" type from the imported domain model (cf. line 8). As all parameters are synchronous, they implicitly employ "REST/JSON" for service interactions (cf. line 10).

Lines 18 to 24 contain the operation model. It describes the container deployment of the "Allocations over Time" service. The `docker` and `python` keywords in line 19 specify an advice "Docker/python" for the technology aspect of the Deployment Technology VP (cf. Table II). Thus, the container will employ Docker as operation technology and Python as operation environment to run the service. Lines 22 to 24 determine physical endpoints for the deployment.

V. Extending the Viewpoint-specific MSA Modeling Languages with Technology Aspects

This section describes the AOM-based extension of the modeling languages (cf. Section IV) with means to define, modularize, and apply MSA technology aspects and their advices. Subsection V-A introduces a dedicated modeling language for aspect and advice definition. Subsection V-B shows how the Service Viewpoint Language was adapted to enable aspect and advice application. Consequently, technology-specific concepts and keywords could be removed from the language. Subsection V-C describes the implementation of our approach.

A. Technology Modeling Language

The Technology Modeling Language allows for the model-based definition and modularization of MSA technology aspects and advices. It aims to make technology decisions in microservice architectures explicit and facilitate reasoning about them. To this end, the language comprises concepts to model each MSA TVP in Table II in the form of a *built-in technology aspect* (cf. Section III). Additionally, the language provides means to declare *custom technology aspects*. Advices may then be specified for both kinds of aspects. *Technology models* constructed with the language cluster technology aspects and their advices. These models may be flexibly integrated into service and operation models (cf. Subsection V-B) to apply modeled aspects and advices at prescribed join points (cf. Section III).

Fig. 3 shows the *metamodel* [9] of the Technology Modeling Language. It formalizes the language's modeling concepts, their structures, and relationships in a UML class diagram [9]. Coherent concepts related to the modeling of built-in and custom technology aspects are colored differently.

Modeling of Built-in Technology Aspects

The metamodel comprises modeling concepts to express the TVPs in Table II as built-in technology aspects and specify advices for them (cf. Section III). The `Protocol` and `DataFormat` concepts (cf. Fig. 3) may be leveraged to specify advices for the eponymous TVPs. A protocol has a `name` and is specific to a communication type (`commType`). It may further be marked as being the `default` protocol for the respective communication type. A protocol has several data formats assigned, of which one is the `defaultFormat`. In case no concrete protocol and data format advice is specified for a microservice (cf. Subsection V-B), the defaults for the service's communication type are implicitly assigned, e.g., "REST/JSON" for synchronous communication.

To express advices for the technology aspects of the Deployment and Infrastructure Technology VPs (cf. Table II), the metamodel defines the `InfrastructureTechnology` and `DeploymentTechnology` concepts, which inherit from `OperationTechnology`. An operation technology

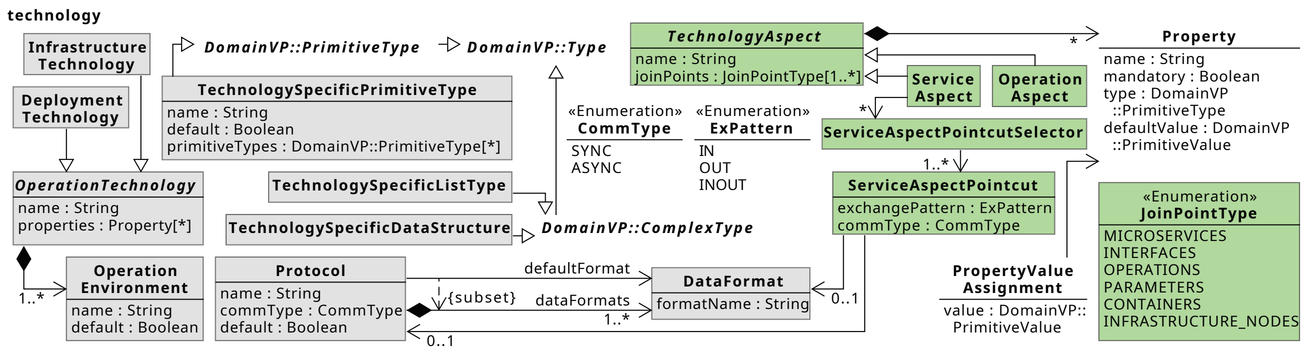

Fig. 3. Metamodel of the Technology Modeling Language with modeling concepts that reflect built-in aspects (gray) and custom aspects (green).

advice has a `name`, e.g., "Docker" (cf. Subsection IV-D), and an optional set of properties. The `Property` concept corresponds to the eponymous concept of the Operation Viewpoint Language (cf. Subsection IV-C). An operation technology advice also has to determine at least one `OperationEnvironment`, e.g., "python" as a Docker container's image.

The metamodel covers the aspect of the Programming Language TVP (cf. Table II) in that it enables advice specification for technology-specific primitive, structure, and list types. Python, for example, exhibits a native `list` type, which can be modeled as an advice with the metamodel's `TechnologySpecificListType` concept.

Based on the `primitiveTypes` property, an advice for a `TechnologySpecificPrimitiveType` (cf. Fig. 3) may constitute a *type synonym*. Type synonyms map native primitive types of a programming language to one or more built-in primitive types of the viewpoint-specific modeling languages' type system, which itself is based on Java [18]. For instance, as opposed to Java, Python comprises only one primitive type for floating point numbers. Thus, a type synonym `float` could be defined in a technology model for the Python programming language as a synonym for the modeling languages' `float` and `double` primitive types (cf. Subsection VI-A). Synonyms for technology-specific primitive types may be marked as `default` (cf. Fig. 3). If a technology model with default type synonyms is assigned to a microservice, all operation parameters, which are typed with a built-in primitive type from the modeling languages' type system, are implicitly considered to be typed with the technology-specific type synonym (cf. Subsection VI-B).

Modeling of Custom Technology Aspects

A technology model may comprise custom aspects. They reflect technology-related concerns beyond the scope of the built-in aspects or substantiate semantics of built-in aspects' advices. For example, a service to which a "REST" advice for the built-in `Protocol` aspect (cf. Fig. 3) was applied, could be substantiated by specifying HTTP methods and statuses for its operations and parameters (cf. Subsections VI-A and VI-B).

A custom aspect can be modeled as a `ServiceAspect` or `OperationAspect` (cf. Fig. 3). Both aspect kinds are distinguished by their prescribed join points, which in the metamodel are clustered in the `JoinPointType` enumeration. Service aspects may be applied to microservices, interfaces, operations, and parameters. The possible join points of operation aspects comprise containers and infrastructure nodes. Custom aspects may cluster a `Property` set to prescribe advices' possible values, e.g., a concrete HTTP status code. A property may be marked as `mandatory`, i.e., an advice value must be assigned when applying the aspect.

The `ServiceAspectPointcut` and `ServiceAspectPointcutSelector` modeling concepts (cf. Fig. 3) enable the specification of pointcuts for service aspects (cf. Section III). Possible pointcuts comprise exchange patterns, communication types, protocols, and data formats. Pointcut selectors cluster several pointcuts and specify concrete values for them. A service aspect and its advices are only applicable to those join points for which all pointcut values of at least one of its selectors match. This mechanism, for instance, allows to constrain the applicability of a custom aspect for HTTP methods to operations that exhibit a "REST" advice for the built-in `Protocol` aspect (cf. Subsection VI-A).

B. Technology Model Integration

In the following, we describe the adaptation of the viewpoint-specific MSA modeling languages in order to apply technology aspects and advices specified in technology models. Due to space constraints, we only describe the adaptation of the Service Viewpoint Language (cf. Subsection IV-B). Fig. 4 shows the extensions of its metamodel [18], which are related to aspect and advice application.

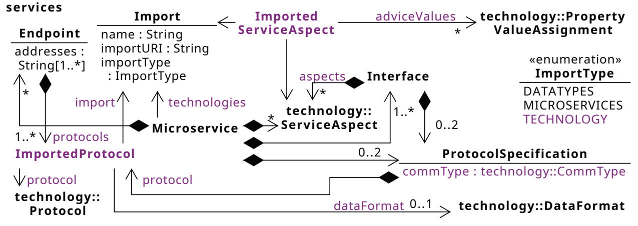

Fig. 4. Extensions (in purple) of the Service Viewpoint Language's metamodel to enable the application of technology aspects and advices in service models.

The import mechanism of the Service Viewpoint Language was extended to enable the import of technology models. To this end, the TECHNOLOGY import type was added to the Service Viewpoint Language's metamodel, so that modeled Imports can refer to technology model files (cf. Fig. 4). The import of a technology model makes the defined technology aspects and advices available in the importing service model. They can then be applied at the prescribed join points of the modeled microservices (cf. Subsection V-A) on the basis of the new technologies association (cf. Fig. 4). It allows to assign more than one technology model to a microservice, which enables a fine-grained modularization of technology aspects and advices, as well as the reuse of technology models (cf. Section VI). However, only one of a microservice's technology models can define technology-specific types. This model represents the service's implementation language.

The metamodel of the Service Viewpoint Language was further extended with the ImportedProtocol concept (cf. Fig. 4). It provides a means to apply advices for the built-in technology aspects of the Protocol and Data Format TVPs (cf. Subsection V-A) corresponding to the value of the new commType property of the ProtocolSpecification concept. The applied advices determine the protocols, which are employed for synchronous and asynchronous interactions with the respective microservice. The application of protocol advices is transitive. For example, if the "REST" advice is applied to determine the synchronous protocol of a microservice, it is also implicitly applied to all modeled Interfaces of the service that lack a protocol specification.

The ImportedServiceAspect concept (cf. Fig. 4) allows for applying custom technology aspects to microservices, interfaces, operations, and parameters (cf. Subsection V-A). Due to space constraints, we omitted the extensions of the Operation and Parameter concepts in Fig. 4.

Our approach to modularize MSA technology aspects and advices in specialized technology models enabled us to remove technology-specific concepts and keywords like AMQP, REST, JSON, docker, and python (cf. Subsection IV-D) from the Service and Operation Viewpoint Languages. This facilitates the languages' usage and makes it more consistent. Moreover, the specification of technological characteristics in service and operation models is now uniformly possible via dedicated, generic modeling concepts.

C. Implementation

Like the metamodels of the viewpoint-specific MSA modeling languages [18], the metamodel of the Technology Modeling Language was implemented with Xcore[6]. Xcore is a metamodel specification language. Starting from the metamodel, the Xtext framework[7] was used to implement the language's textual syntax and editor, as well as its import, scoping, and validation mechanisms. Thus, the language integrates seamlessly with the viewpoint languages on the basis of Eclipse. It uses the same tooling infrastructure, and can import existing service and operation models from Eclipse workspaces.

[6]https://wiki.eclipse.org/Xcore

[7]https://www.eclipse.org/Xtext

The Service and Operation Viewpoint Language were extended with additional grammar rules and functionality to integrate technology models. Listing 2 shows in lines 1 to 5 the grammar rule of the Service Viewpoint Language that defines the concrete syntax for the ImportedServiceAspect concept (cf. Fig. 4). Custom technology aspects and their advices are applied to join points in the form of annotations, i.e., by specifying the aspect's name prefixed with an "@". Lines 6 to 8 show the grammar parts for assigning an imported technology model and custom aspects to a microservice.

Listing 2
ADAPTED XTEXT GRAMMAR OF THE SERVICE VIEWPOINT LANGUAGE

```
ImportedServiceAspect returns ImportedServiceAspect:                1
  '@' ^import=[Import] '::'                                          2
  importedAspect=[technology::ServiceAspect|QualifiedName]           3
    ('(' adviceValues+=PropertyValueAssignment                       4
    (',' adviceValues+=PropertyValueAssignment)* ')')? ;             5
Microservice returns Microservice:                                   6
  ('@' 'technology' '(' technologies+=[Import] ')')*                 7
  aspects+=ImportedServiceAspect* ...;                               8
```

The source code of the Technology Modeling Language and the viewpoint languages can be found on GitHub[8].

VI. APPLYING THE APPROACH TO THE CASE STUDY

This section illustrates and discusses the application of our approach in the context of the case study (cf. Section II).

A. Modeling of Aspects and Advices for the Case Study

To employ our approach, technology models need to be constructed first leveraging the Technology Modeling Language (cf. Subsection V-A). Listing 3 shows two technology models for the case study and illustrates the usage of the language.

Listing 3
TECHNOLOGY MODELS FOR THE CASE STUDY

```
// Content of technology model file "python.technology"
technology python {
  types {
    primitive type Int based on byte, int, short default;
    primitive type Long based on long default;
    primitive type Float based on float, double default;
    primitive type Bool based on boolean default;
    primitive type Str based on string, char default;
    primitive type Unicode based on string;
    primitive type Datetime based on date default;
    primitive type Complex; list type List; }
  deployment technologies { docker {
    operation environments = "python" default; } } }
// Content of technology model file "shared.technology"
technology shared {
  protocols {
    sync rest data formats json default with format json;
    async amqp data formats json default with format json;}
  service aspects {
    aspect get for operations {selector (protocol = rest);}
    aspect HttpStatus for parameters {
      selector(protocol = rest, exchange_pattern = out);
      int status mandatory; } }
  infrastructure technologies { serviceDiscovery {
    operation environments = "eureka" default; } } }
```

[8]https://www.github.com/SeelabFhdo/ddmm

Lines 2 to 13 comprise a technology model for Python. It defines Python-specific type synonyms (cf. Subsection V-A) in lines 3 to 10. For example, "Str" is a default type synonym for the modeling languages' built-in primitive types `string` and `char`. Thus, when the technology model is assigned to a microservice, all occurrences of `string` and `char` in the parameters of the service's operations are implicitly mapped to "Str" (cf. Subsection IV-B). However, with "Unicode" there also exists an additional type synonym for `string`. This results in the type checker of the modeling languages to consider Python's "Str" and "Unicode" types to be compatible, because both are synonyms for `string`. The "Complex" and "List" types defined in line 11 model Python-specific types without counterparts in the modeling languages' type system.

In lines 12 and 13 of Listing 3 the Python technology model defines a "docker" advice for the technology aspect of the Deployment Technology VP (cf. Table II). It has a default operation environment called "python". Thus, Docker containers to which the technology model is assigned in an operation model (cf. Subsection IV-C), implicitly use "python" as their image. By intent, the Technology Modeling Language does not support versioning of technologies. Instead, it is assumed that all technologies being captured in a technology model are compatible with each other. For instance, the "python" image must support the modeled version of the Python language.

Lines 15 to 25 in Listing 3 comprise a technology model that may be shared across programming languages. It defines generic protocols, service aspects and their advices, and infrastructure technologies. In lines 17 and 18 the advices "rest/json" and "amqp/json" for the technology aspects of the Protocol and Data Format TVPs of the case study (cf. Section II) are defined. Lines 20 to 23 declare two custom technology aspects "get" and "HttpStatus" (cf. Subsection V-A). The former is applicable to operations that use REST for service interactions (cf. the pointcut selector in line 20) and corresponds to the GET HTTP method. An operation annotated with the aspect is hence to be invoked with an HTTP GET request. The "HttpStatus" aspect allows for assigning HTTP status codes to outgoing parameters of a REST operation (cf. the pointcut selector in line 22). Therefore, its mandatory "status" advice must exhibit a status code value (cf. line 23).

Lines 24 and 25 define "serviceDiscovery" as an advice for the built-in aspect of the Infrastructure Technology VP (cf. Section II). It enables the definition of an infrastructure node (cf. Subsection IV-C) for service discovery. Eureka[9] is employed as the default discovery technology.

B. Applying the Modeled Aspects and Advices

Modelers can import the technology models shown in Listing 3 into microservice and operation models being expressed with the Service and Operation Viewpoint Languages (cf. Subsection V-B). After their import, the technology models can be assigned to modeled microservices and operation nodes to specify their technologies. Those assignments also allow modelers for applying custom aspects and advices being defined in the assigned technology models at the prescribed join points of the annotated microservices and operation nodes (cf. Subsection V-A). Listing 4 illustrates the import of technology models, their assignment to modeled services and nodes, and the application of aspects and advices. Therefore, it adapts the case study's service and operation models (cf. Listing 1).

Listing 4
APPLICATION OF TECHNOLOGY ASPECTS AND ADVICES

```
1  // Service model "case_study.services" from Listing 1
2  // applying technology aspects from Listing 3
3  import technology from "python.technology" as python
4  import technology from "shared.technology" as shared
5  import datatypes from "case_study.data" as domain
6  @technology(python) @technology(shared)
7  functional microservice com.example.AllocationsOverTime {
8    interface AllocationsOverTime {
9      @shared::_aspects.get getAllocations(
10       ...
11       sync in endTimestamp : date,
12       @shared::_aspects.HttpStatus(200)
13       sync out allocations : domain::VehicleCounts);}}
14 // Operation model "case_study.operation" from Listing 1
15 // applying technology aspects from Listing 3
16 import technology from "python.technology" as python
17 import technology from "shared.technology" as shared
18 import microservices from "case_study.services" as services
19 @technology(python) @technology(shared)
20 container AllocationsOverTime
21 deployment technology python::_deployment.docker
22 deploys services::com.example.AllocationsOverTime {
23   services::com.example.AllocationsOverTime {
24     basic endpoints {
25     shared::_protocols.rest: "http://example.com:8080";}}}
26 @technology(shared)
27 Discovery is shared::_infrastructure.serviceDiscovery
28   used by services::com.example.AllocationsOverTime {}
```

The adapted service model in lines 3 to 13 first imports the technology models from Listing 3. Both are assigned in line 6 to the "AllocationsOverTime" microservice. This implicitly results in mapping the parameters of the "getAllocations" operation to the type synonyms of the Python technology model (cf. Subsection V-A). For instance, "endTimestamp" is implicitly treated as being of type "Datetime" (cf. line 10 in Listing 3). To express that it is invokable by an HTTP GET request, the "getAllocations" operation is annotated in line 9 of Listing 4 with the custom "get" aspect from the shared technology model (cf. line 20 in Listing 3). The application of the "HttpStatus" aspect in line 12 results in the operation to send the HTTP status code 200 when returning "allocations".

The adapted operation model in lines 16 to 28 of Listing 4 first imports the technology models. They are assigned in line 19 to the "AllocationsOverTime" container. This allows the usage of the "docker" advice in line 21 as the container's deployment technology. In line 25 an endpoint is specified for the imported "rest" protocol advice. Lines 26 to 28 model the case study's service discovery (cf. Section II) via the "serviceDiscovery" advice from the shared technology model.

All models for the case study can be found on GitHub[8].

C. Discussion

We discuss our approach for the model-based definition and application of MSA technology aspects and advices w.r.t. the AOM evaluation criteria in [10].

[9]https://www.github.com/Netflix/eureka

1) Language: According to [10], the presented approach can be considered a *general-purpose approach* to AOM. That is, it provides means to express more than one aspect. Next to the built-in technology aspects for the TVPs in Table II, the Technology Modeling Language also enables the modeling of custom technology aspects (cf. Subsection V-A) to, e.g., substantiate the semantics of built-in aspects (cf. Subsection VI-A). Custom technology aspects represent a *lightweight extension mechanism* [10], i.e., they can be applied to elements of *base models* without altering the underlying metamodels. Thus, the metamodels of the adapted viewpoint languages (cf. Subsection V-B) remain stable when applying custom technology aspects in service or operation models.

The assignment of technology models to microservices or operation nodes (cf. Subsection VI-B) represents a *refining extension* [10] to their defining service and operation models. This characteristic facilitates the exchange of service and node technologies, which, in case the default type synonyms, protocols, and operation technologies are sufficient, merely corresponds to an exchange of assigned technology models. For example, the programming language and deployment technology of the "AllocationsOverTime" microservice could be flexibly exchanged by replacing the technologies in the annotations in lines 6 and 19 of Listing 4 with references to alternative technology models. In addition, the generic aspect mechanism enabled us to remove technology-related keywords from the viewpoint languages (cf. Subsection V-B).

2) Concern Composition: Technology aspects and advices are distinguished from the *base elements* they are applied to, e.g., services and nodes. Hence, the concern composition in our approach is *asymmetric* [10]. It happens via annotations on the base elements that start with an "@" followed by an aspect and, possibly, an advice (cf. Subsection V-C).

The composition technique of our approach is *pointcut-advice* [10]. Pointcut selectors allow for more fine-grained control of the join points to which a custom technology aspect and its advices are applicable (cf. Subsection V-A). Compared to alternative approaches like *compositor*, which enables to implement the composition process in specialized composition programs [10], pointcut-advice composition is sufficient in the context of the viewpoint-specific MSA modeling languages (cf. Section IV). That is, because we delegate the composition process to technology-specific code generators (see below) and aim for reducing the learning curve of our approach. Service and operation modelers need not have to learn the basics of AOSD, AOM, and a specialized composition language.

3) Tool Support: Modeling support [10] for our approach is provided on the basis of an Eclipse plugin (cf. Subsection V-C). It seamlessly integrates with the IDE and comprises an editor for the Technology Modeling Language (cf. Section V-A) with syntax highlighting and validation. In addition, the plugin provides a scoping mechanism so that service and operation models can be augmented with imported technology aspects and their advices (cf. Subsection VI-B).

However, the existing code generators, that produce microservice code and deployment descriptors from service and operation models [18], also need to be adapted to support the weaving of technology advices (cf. Section III). To this end, we decompose the current code generation infrastructure to cope with different technology models. For instance, the part of the Python code generator that is responsible for producing code for the Eureka service discovery technology (cf. Subsection VI-A) is now realized as a separate generator that may be reused across service implementation technologies. It generates the service discovery for all microservices of the architecture that uses it, based on an input operation model such as the one in Listing 4. Analogously, the generation of Docker deployment descriptors is decomposed and can be reused for all microservices that are deployed with Docker. Alternative container technologies (cf. Section II) may be covered by specific code generators.

VII. Related Work

We present work related to abstracting technology and applying AOM in service-based software architectures.

In [20], an approach is presented to abstract technology in service-oriented software systems. It enables model-based refinement of architectural styles at different levels of platform abstraction. For instance, an architectural style for the business level of an application comprises interfaces of business components, but omits infrastructure components like service discoveries. These are considered, instead, in the architectural style for the implementation level. Architectural styles are formalized as graph transformation systems, which allows the iterative refinement of a concrete model of a style on a higher level of abstraction, e.g., the business level, to a model of a style on a lower level of abstraction, e.g., the implementation level. The approach corresponds to OMG's Model-driven Architecture (MDA) [19] methodology, in which Platform-independent Models (PIMs) are transformed into Platform-specific Models (PSMs) based on technology-related decision making. In our approach, we use viewpoints, instead of architectural styles, and import mechanisms, instead of graph transformations, to express and connect different levels of platform abstraction (cf. Section IV). However, there is, by intent, no strict distinction between PIMs and PSMs. Instead, technology and hence platform-dependency is treated and made explicit in microservice and operation models as crosscutting concern in the sense of AOM (cf. Section V). This is possible, because the target groups of our approach are service developers and operators, who are regularly and directly confronted with heterogeneous technologies.

In [21], an AOM-based architecture for building context-aware service-based applications is defined. It involves several metamodels to address different AOM characteristics. The ContextAspect metamodel combines AOM concepts like pointcuts and aspects with concepts for describing context awareness. Conceptually, this metamodel corresponds to that of our Technology Modeling Language (cf. Subsection V-A), but exhibits a different concern domain, i.e., context awareness instead of technology. The ContextAspect metamodel is accompanied by the Weaving metamodel [21], which enables to influence the process of aspect composition. Our approach does not comprise a separate metamodel or other mechanism for that purpose (cf. Subsection VI-C). Instead, weaving is delegated to code generators, which is possible because with technology they focus on the same concern as the technology models. This also leads to the *composition phase* and *transformation phase* of [21] to coincide in our approach.

In [22], the modeling concept of *microservice ambients* as adaptable specifications of functional service boundaries is introduced. The adaptation of a microservice's boundary and hence its granularity is considered to be a crosscutting concern. A microservice ambient is specified in an architecture description language on a lower abstraction level than our modeling languages. Furthermore, the ambients focus on runtime, while our modeling languages focus on design time. It could therefore be beneficial to integrate both approaches. With the custom aspect mechanism of our Technology Modeling Language it would be possible to specify certain characteristics of a microservice ambient, e.g., the elements being monitored by an ambient for runtime adaptation [22]. Our code generators could then derive service code, which comprises a completed version of the *ambient template* [22]. This would support the correct specification of microservice ambients.

VIII. Conclusion and Future Work

In this paper, we presented an approach to cope with technology heterogeneity in microservice-based software systems. To this end, Aspect-oriented Modeling is employed to capture technology decisions in microservice architectures in the form of aspects and advices (cf. Section III). The approach is centered around a Technology Modeling Language (cf. Subsection V-A). It allows to specify advices for built-in, microservice-related technology aspects, as well to define custom technology aspects to, e.g., substantiate the semantics of built-in aspects. The resulting technology models modularize the aspects and advices. Thus, they make technology decisions in microservice architectures explicit and enable reasoning about technology heterogeneity. A set of viewpoint-specific microservice modeling languages from our previous work was adapted to enable the application of technology aspects and advices within microservice and operation models (cf. Subsection V-B). Consequently, technology-specific concepts and keywords could be removed from the languages, which facilitates their usage and makes it more consistent.

We are currently working on decomposing the code generation infrastructure as described in Subsection VI-C. One of our goals is to enable the reuse of code generators, even for service-specific technologies. For example, the interpretation of protocols is typically realized in language-dependent frameworks, and code generators depend on the usage patterns of such frameworks. In this context, the question arises, how such usage patterns and their implementations can be made reusable across code generators. After the decomposition of the code generation infrastructure, we plan to evaluate our approach in the context of industrial case studies. Therefore, we take the evaluation criteria mentioned in Subsection VI-C as basis to validate the generality of our approach. Additionally, we aim to analyze the efficiency of the decomposed code generation infrastructure. Next to the expected increase in development productivity, we are particularly interested to what extent the separation of the technology concern from functional models facilitates the technological migration of microservices in combination with technology-specific code generators.

References

[1] S. Newman, *Building Microservices*. O'Reilly Media, 2015.

[2] O. Zimmermann, "Microservices tenets," *Computer Science - Research and Development*, vol. 32, no. 3, pp. 301–310, 2017.

[3] I. Nadareishvili, R. Mitra, M. Mclarty, and M. Amundsen, *Microservice Architecture*. O'Reilly Media, 2016.

[4] T. Killalea, "The hidden dividends of microservices," *Communications of the ACM*, vol. 59, no. 8, pp. 42–45, 2016.

[5] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[6] N. Kratzke and P.-C. Quint, "Investigation of impacts on network performance in the advance of a microservice design," in *Proc. of the Sixth Int. Conf. on Cloud Computing and Services Science (CLOSER)*. Springer, 2016, pp. 187–208.

[7] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[8] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, May 2018.

[9] B. Combemale, R. B. France, J.-M. Jézéquel, B. Rumpe, J. Steel, and D. Vojtisek, *Engineering Modeling Languages*. CRC Press, 2017.

[10] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, and G. Kappel, "A survey on aspect-oriented modeling approaches," University of Vienna, Tech. Rep., 2007.

[11] F. Rademacher, M. Lammert, M. Khan, and S. Sachweh, "Towards a model-based architecture for road traffic management systems," in *Information and Software Technologies*. Springer, 2016, pp. 650–662.

[12] F. van der Linden, K. Schmid, and E. Rommes, *The Product Line Engineering Approach*. Springer, 2007, pp. 3–20.

[13] P. Jamshidi, C. Pahl, N. C. Mendona, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, May 2018.

[14] European Committee for Standardization, "Datex ii data exchange specifications," no. CEN/TS 16157-1:2011, 2011.

[15] N. Mitra and Y. Lafon, *SOAP Version 1.2*. W3C, 2007.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Object-Oriented Programming*. Springer, 1997, pp. 220–242.

[17] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," *Proc. of the 2007 Workshop on Future of Software Engineering (FOSE)*, 2007.

[18] F. Rademacher, J. Sorgalla, S. Sachweh, and A. Zündorf, "Viewpoint-specific model-driven microservice development with interlinked modeling languages," in *Proc. of the 13th Int. Conf. on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, in press.

[19] Object Management Group, *Model Driven Architecture (MDA) Guide*, OMG Std., Rev. 2.0, 2014.

[20] L. Baresi, R. Heckel, S. Thöne, and D. Varró, "Style-based modeling and refinement of service-oriented architectures," *Software & Systems Modeling*, vol. 5, no. 2, pp. 187–207, 2006.

[21] B. Boudaa, S. Hammoudi, L. A. Mebarki, A. Bouguessa, and M. A. Chikh, "An aspect-oriented model-driven approach for building adaptable context-aware service-based applications," *Science of Computer Programming*, vol. 136, pp. 17–42, 2017.

[22] S. Hassan, N. Ali, and R. Bahsoon, "Microservice ambients: An architectural meta-modelling approach for microservice granularity," in *Proc. of the Int. Conf. on Software Architecture (ICSA)*. IEEE, 2017, pp. 1–10.