# A Non-Intrusive Approach to Extend Microservice Modeling Languages with Architecture Pattern Support

Florian Rademacher

IDiAL Institute, University of Applied Sciences and Arts Dortmund, Germany
florian.rademacher@fh-dortmund.de

**Abstract**

The adoption of Microservice Architecture (MSA) is expected to increase software quality attributes like scalability, maintainability, and reliability. In particular, monolithic architectures are expected to benefit from a migration to MSA. However, it also introduces additional complexity in architecture design, implementation, and operation. To cope with this complexity, we investigate the application of Model-driven Engineering to MSA engineering. Therefore, we developed a set of textual modeling languages, that addresses the viewpoints and concerns of different stakeholder groups in MSA engineering, in recent works.

By intent, our modeling languages only integrate concepts and keywords for basic building blocks of MSA. In particular, we do not provide built-in concepts for architecture patterns to increase languages' learnability and stability. Instead, we implemented an aspect mechanism to flexibly augment model elements with metadata.

In this paper, we present an approach based on this aspect mechanism to non-intrusively extend our modeling languages with architecture pattern support. Therefore, we illustrate the specification of aspects and the implementation of aspect constraints to ensure pattern-compliance. We illustrate our approach by means of the Event Sourcing and Command Query Responsibility Segregation patterns, both becoming increasingly popular in MSA engineering.

**Keywords:** Microservice Architecture, Model-driven Engineering, Modeling Languages, Architecture Modeling, Event Sourcing, Command Query Responsibility Segregation

## 1 Introduction

The adoption of Microservice Architecture (MSA) is expected to increase software quality attributes like scalability, maintainability, and reliability [5]. However, the migration to MSA, particularly from monolithic architectures, introduces additional complexity regarding architecture design, implementation, and operation. Typical challenges in these areas comprise, e.g., API versioning, service tailoring, and dealing with technology heterogeneity [26].

To support MSA stakeholders in coping with such challenges, we investigate the application of Model-driven Engineering (MDE) [3] to MSA engineering. In recent works, we therefore developed the Language Ecosystem for Modeling Microservice Architecture (LEMMA) [20, 23]. LEMMA comprises a set of textual *architecture modeling languages* [25] and means to facilitate processing of models constructed with these languages, e.g., for architecture analysis [22] or generation of microservice code [21]. Table 1 provides a brief overview of LEMMA's modeling languages. Their implementation can be found on GitHub[1].

LEMMA's modeling languages provide built-in concepts and keywords for basic building blocks of MSA, e.g., `microservice`, `interface`, and `operation` [23]. By intent, however, they

[1] https://github.com/SeelabFhdo/lemma

Table 1: Overview of LEMMA modeling languages and their relationship to stakeholders in MSA engineering [20, 23].

| **Stakeholders** | **Modeling Languages** |
|---|---|
| Domain experts, Service developers | *Domain Data Modeling Language* [23]: Enables stakeholders to construct *domain models* with domain-specific data structures, lists, and enumerations, including the assignment of patterns from Domain-driven Design (DDD) [7]. |
| Service developers, Service operators | *Technology Modeling Language* [20]: Allows for constructing *technology models* that prescribe available types of microservice implementation languages, communication protocols, and operation technologies. In addition, generic *technology aspects* can be modeled to augment, e.g., data structures and microservices modeled in LEMMA with technology-specific information like database mappings or endpoint URIs.<br>*Technology Mapping Language*: Enables the construction of *mapping models*, which assign technology-specific information to elements captured in domain and service models. Domain and service models can therefore be kept technology-agnostic and reusable across technological alternatives in order to cope with MSA's technology heterogeneity [17]. |
| Service developers | *Service Modeling Language* [23]: Enables the construction of *service models* that specify microservices, their interfaces and operations. Domain models may be imported for operation parameter typing. |
| Service operators | *Operation Modeling Language* [23]: Allows for constructing *operation models*. They import service models to describe microservice deployment and infrastructure, e.g., for data storage, API provisioning, service discovery and monitoring [1], including its usage by services. |

do not integrate dedicated modeling concepts and keywords for patterns like Circuit Breaker or Command Query Responsibility Segregation (CQRS) [15, 24]. That is, to foster the learnability of the languages and the comprehensibility of constructed models [2]. Furthermore, by keeping the languages free from pattern-specific concepts and keywords, we aim to increase their stability. The emergence of a new pattern thus does not require the languages to be adapted accordingly.

On the other hand, LEMMA's Technology Modeling Language (cf. Table 1) integrates the *technology aspect* concept. It can be used to flexibly augment elements in LEMMA domain and service models with custom, yet technology-specific metadata. A technology aspect may, for instance, represent a certain annotation from microservice implementation frameworks such as Spring[2]. To this end, technology aspects are first defined in LEMMA technology models. Next, technology models are imported into service models for aspect application. Alternatively, a mapping model may be constructed with LEMMA's Technology Mapping Language to keep service models technology-agnostic (cf. Table 1). Therefore, the mapping model imports relevant technology, domain, and service models, and applies technology aspects to target elements from domain or service models as required.

In this paper, we present first insights into exploring the applicability of the technology aspect mechanism for enriching LEMMA models with architecture patterns. Our main goal is

[2] https://www.spring.io

to investigate the non-intrusive and flexible extension of LEMMA with architecture patterns, without the need for altering any of its modeling languages (cf. Table 1). Thus, learnability, comprehensibility, and stability of LEMMA would be preserved, by still guaranteeing its modifiability to cope with technical progress in the field of MSA. More specifically, the contributions of our paper revolve around the following research questions (RQs):

**RQ 1** *Which steps are required to exploit LEMMA's aspect mechanism and model processing framework to integrate architecture patterns into LEMMA models and check for their correct application?*

**RQ 2** *Which artifacts need to be provided to MSA stakeholders (cf. Table 1) to enable the enrichment of their LEMMA models with architecture patterns?*

**RQ 3** *To what extent is LEMMA capable of architecture pattern integration in terms of the overall complexity of the integration process?*

To gather initial answers to the RQs, we leverage the Event Sourcing and CQRS patterns [24, 15] as objects of study for non-intrusive pattern integration in LEMMA. Both patterns are frequent in event-driven microservice architectures [19].

The remainder of the paper is organized as follows. Section 2 provides background information on Event Sourcing and CQRS. Section 3 presents an approach to extend LEMMA with modeling support for both patterns. Section 4 discusses the approach in the light of our RQs. Section 5 presents related work and Section 6 concludes the paper.

# 2 Background

The Event Sourcing and CQRS patterns are expected to positively impact the scalability and modifiability [9] of a microservice architecture [24]. That is, because both patterns are centered around event-driven and thus asynchronous interaction between microservices [15]. To make the paper self-contained, the following Subsections 2.1 and 2.2 provide a brief overview of both patterns.

## 2.1 Event Sourcing

The Event Sourcing pattern considers the current state of a *domain object* [7] within a software architecture to have emerged from a sequence of *domain events* [24]. In this context, a concrete domain event is a typically immutable domain object that conveys information about activities, which occurred within the domain model of an application or a parts thereof [8]. For example, a domain concept `Account`, which represents a bank account in a microservice architecture located in the Financial Service domain [13], may exhibit a data field called `balance`. It stores the current balance of a concrete bank account and its value is the result of a sequence of past domain events, e.g., for depositing and withdrawing money.

A technical building block for the realization of Event Sourcing is the Event Store, i.e., a message broker to distribute domain events in a publish-subscribe fashion and persist them in their order of occurrence to enable traceability of the current state of a domain object [24]. Next to increasing the decoupling, and thus scalability and modifiability, of services within a microservice architecture, the Event Store also allows for the implementation of history and auditing mechanisms.

### 2.2 CQRS

The CQRS pattern separates actions that change domain objects' states from those that read them [24]. A CQRS-enabled microservice is therefore considered to have a *command side*, *query side*, or both. The command side handles write actions to domain objects, while the query side is responsible for read actions. Technically, the command and read sides may either be interfaces, which cluster corresponding write or read operations, of the same microservice. On the other hand, to further increase scalability, the command and query sides may also be realized as distinct microservices to enable independent scalability, e.g., when read actions are much more frequent than write actions. In either case, the command side informs query sides about state changes of domain objects by means of events. Thus, CQRS can directly be integrated with Event Sourcing (cf. Subsection 2.1).

The adoption of CQRS yields several benefits [24]. First, the pattern allows for applying the most appropriate means for read actions. In the example of an `Account` domain object it may make sense to store its master data in a relational database management system (DBMS). However, to realize functionality that, e.g., enables account holders to efficiently search through references, parts of the `Account's` transactions may be stored in a DBMS being optimized for full-text search. A query side microservice dedicated to full-text reference search would then be responsible for (i) setting up the optimized DBMS; (ii) keeping it consistent with newly occurred transactions received as events; and (iii) provide access to its entries for full-text search.

Another benefit of CQRS is the possibility to provide additional query sides as needed, given the loose coupling between command and query sides induced by their event-based interaction.

## 3 Non-Intrusive Extension of LEMMA with Support for Event Sourcing and CQRS

In the following, we present a non-intrusive approach to extend LEMMA with support for the Event Sourcing and CQRS patterns (cf. Section 2). In a first step, pattern-specific technology models (cf. Table 1) are defined (cf. Subsection 3.1). They capture the underlying concepts of the patterns as technology aspects, which can then be applied to elements in LEMMA domain and service models (cf. Table 1). Next, model validators for the pattern-specific technology models are provided (cf. Subsection 3.2). The validators are based on LEMMA's model processing framework and can thus check models for the correct application of technology aspects that reflect pattern concepts. The section is concluded by illustrating the non-intrusive extension of LEMMA with support for Event Sourcing and CQRS leveraging pattern-specific technology models and the corresponding model validators (cf. Subsection 3.3).

### 3.1 Technology Models

LEMMA's Technology Modeling Language can be used to define technology aspects being applicable to certain elements in LEMMA domain and service models (cf. Table 1). While originally being developed to augment these models with technology-specific information, e.g., built-in types from programming languages and annotations or configuration options predefined by microservice implementation frameworks, LEMMA's aspect mechanism provides a means to assign arbitrary metadata to model elements. Consequently, we exploit it to capture initial underlying concepts of the Event Sourcing and CQRS patterns as shown in Subsubsections 3.1.1 and 3.1.2.

### 3.1.1 Event Sourcing Technology Model

Listing 1 shows an initial LEMMA technology model for the Event Sourcing pattern.

Listing 1: LEMMA technology model for the Event Sourcing pattern.

```
1  // Model file name: EventSourcing.technology
2  technology EventSourcing {
3    service aspects {
4      aspect Producer<singleval> for operations {
5        string handlerName<mandatory>;
6      }
7
8      aspect Consumer<singleval> for operations {
9        string handlerName<mandatory>;
10       boolean groupEventsOnly = false;
11     }
12
13     aspect EventGroup<singleval> for types {
14       string name<mandatory>;
15     }
16   }
17 }
```

The model specifies three service aspects for the `EventSourcing` technology. They are based on the concepts of the Event Sourcing pattern as described by Richardson [24]. The `Producer` aspect in Lines 4 to 6 identifies *event producers*. The `for operations` statement constrains the applicability of the aspect to microservice operations specified within a LEMMA service model and the `singleval` keyword allows the aspect to occur exactly once per operation. In addition, the aspect comprises the `string`-typed `handlerName` property. It is `mandatory` and thus has to be specified each time the aspect is applied to an operation. Semantically, the property identifies the name of a handler, which is responsible for sending an event. Such a handler could be, for example, a Java class that reflects a Spring `Service`[3] and is responsible for sending event objects via a Kafka[4] broker.

The `Consumer` aspect specified in Lines 8 to 11 of Listing 1 identifies *event consumers*. Similar to `Producer`, the `Consumer` aspect takes a `handlerName` as mandatory property. Consumer handlers are responsible for event receiving and processing. Furthermore, a consumer may be constrained to the handling of event groups (`groupEventsOnly` property). Event groups can be defined via the `EventGroup` aspect (Lines 13 to 15). The `for types` statement enables the application of the aspect to domain-specific types contained in LEMMA domain models (cf. Table 1). Its mandatory `name` property holds the name of an event group, which clusters several events, e.g., to unify their processing by the same event handler.

### 3.1.2 CQRS Technology Model

Listing 2 contains a preliminary technology model for the CQRS pattern.

Listing 2: LEMMA technology model for the CQRS pattern.

```
1  // Model file name: Cqrs.technology
2  technology CQRS {
3    service aspects {
4      aspect CommandSide for microservices {
5        string logicalService;
6      }
7
```

[3] https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/stereotype/Service.html

[4] https://kafka.apache.org

```
8       aspect QuerySide for microservices {
9         string logicalService;
10      }
11
12      aspect CommandSide for interfaces;
13      aspect QuerySide for interfaces;
14    }
15  }
```

The CQRS technology model specifies the `CommandSide` and `QuerySide` aspects. Both aspects are applicable to microservices (`for microservices` statement) as well as interfaces (`for interfaces` statement) defined within LEMMA service models (cf. Table 1). However, the microservice-enabled versions of the aspects (cf. Lines 4 to 10) specify the additional `logicalService` property. It can be used to indicate that a command and one or more query sides belong to the same *logical microservice* when they are realized by different physical microservices to increase scalability [24].

## 3.2 Model Validators

The initial technology models presented in Subsection 3.1 capture the underlying concepts of the Event Sourcing and CQRS patterns in the form of technology aspects. However, LEMMA's Technology Modeling Language only allows for constraining (i) the types of target elements that can be augmented by aspects (e.g., `for operations` and `for types` statement); (ii) the number of aspect occurrences at model elements (`singleval` keyword); and (iii) the existence of aspect properties (`mandatory` keyword). As a result, constraints that exceed those conditions are not directly verifiable by LEMMA's modeling languages albeit them being crucial to model the correctness of pattern applications. For example, the application of the `Producer` aspect of the Event Sourcing technology model (cf. Subsubsection 3.1.1) is only sensible for microservice operations that define at least one asynchronous outgoing parameter [23]. In case of CQRS, it should be guaranteed, for instance, that events sent by a command side microservice are type-compatible to events received by any query side microservice of the same logical microservice (cf. Subsection 2.2 and Subsubsection 3.1.2).

To circumvent the lack of expressivity in LEMMA's Technology Modeling Language for more sophisticated constraints on aspect application, LEMMA's model processing framework can be used. This framework aims to facilitate the implementation of static analyzers and code generators [3] for LEMMA models by abstracting from the complexity of Eclipse technologies like Ecore[5], Xcore[6], and Xtext[7], on which LEMMA is based [23].

The following Subsubsections 3.2.1 and 3.2.2 list preliminary constraints for the underlying concepts of the Event Sourcing and CQRS patterns. Consequently, the constraints target the pattern-specific technology models and aspects presented in Subsection 3.1. Subsubsection 3.2.3 describes the implementation of the constraints by means of LEMMA's model processing framework.

### 3.2.1 Event Sourcing Constraints

Table 2 lists initial constraints for the technology model of the Event Sourcing pattern (cf. Subsubsection 3.1.1).

[5] https://wiki.eclipse.org/Ecore
[6] https://wiki.eclipse.org/Xcore
[7] https://www.eclipse.org/Xtext

Table 2: Constraints in the context of the technology model for the Event Sourcing pattern.

| # | Constraint Summary | Model Type | Failure Type |
|---|---|---|---|
| C.1 | Only domain events may be clustered in groups | Mapping Model | Error |
| **Description:** The `EventGroup` aspect may only be applied to domain-specific types with the `domainEvent` feature [23]. | | | |
| C.2 | Domain event consumers must specify asynchronous incoming parameters | Service Model | Error |
| **Description:** The `Consumer` aspect may only be applied to microservice operations with at least one asynchronous incoming parameter. Otherwise, the consumption of events is not possible. | | | |
| C.3 | Domain event producers must specify asynchronous outgoing parameters | Service Model | Error |
| **Description:** The `Producer` aspect may only be applied to microservice operations with at least one asynchronous outgoing parameter. Otherwise, the production of events is not possible. | | | |
| C.4 | Event group consumers must only specify asynchronous incoming parameters, whose types are domain events | Service Model | Error |
| **Description:** If the `groupEventsOnly` property of the `Consumer` aspect is set, augmented microservice operations must only exhibit asynchronous incoming parameters with the `domainEvent` feature. That is, because the `EventGroup` aspect is only applicable to domain events (see above). | | | |

### 3.2.2 CQRS Constraints

Similar to Table 2, Table 3 lists initial constraints for the technology model of the CQRS pattern (cf. Subsubsection 3.1.2).

Table 3: Constraints in the context of the technology model for the CQRS pattern.

| # | Constraint Summary | Model Type | Failure Type |
|---|---|---|---|
| C.5 | Existence of operations with asynchronous outgoing parameters in command side | Service Model | Warning |
| **Description:** The command side of a CQRS microservice identified by the `CommandSide` aspect should be able to asynchronously communicate state changes in domain objects to query sides [24]. | | | |
| C.6 | Existence of operations with asynchronous incoming parameters in query side | Service Model | Warning |
| **Description:** The query side of a CQRS microservice identified by the `QuerySide` aspect should be able to asynchronously receive state changes in domain objects from a command side. | | | |
| C.7 | Query side interface requires command side interface within the same microservice | Service Model | Warning |

Table 3: Constraints in the context of the technology model for the CQRS pattern (continued).

| # | Constraint Summary | Model Type | Failure Type |
|---|---|---|---|
| **Description:** In case the `QuerySide` aspect is applied to an interface, there should be a command side interface within the same microservice for the query side interface to receive updates on state changes of domain objects. | | | |
| C.8 | Query side microservice requires command side microservice of same logical microservice | Service Model | Warning |
| **Description:** If the `QuerySide` aspect is applied to a microservice, the query side microservice should require a command side microservice [23], which belongs to the same logical microservice, to receive updates on state changes of domain objects. | | | |
| C.9 | Query side requires compatibility of receiving event parameters with events being sent by command side | Service Model | Warning |
| **Description:** A `QuerySide` interface or microservice modeled in LEMMA should exhibit at least one operation, whose asynchronous incoming parameters are type-compatible with the asynchronous outgoing parameters of an operation of the `CommandSide` interface or microservice. Otherwise, state changes may not be received by query sides leading to inconsistent query results [24]. | | | |

### 3.2.3 Model Validator Implementation

We employed LEMMA's model processing framework to realize two validators for the constraints listed in Tables 2 and 3. Each validator targets exactly one pattern.

LEMMA's model processing framework abstracts from MDE technologies like Ecore, Xcore, and Xtext as much as possible. It aims to facilitate model processor implementation by technology-savvy stakeholders of MSA engineering, e.g., service developers, without experience in the MDE field. To this end, LEMMA's model processing framework relies on well-known programming patterns, e.g., Inversion of Control (IoC) [10], and integrates built-in, yet extensible *model processing phases* [14], e.g., for model validation or code generation. Leveraging the framework, model processors may (i) directly be implemented as standalone executable Java archives; (ii) automatically parse LEMMA models and gain access to the resulting Abstract Syntax Trees (ASTs); and (iii) structure model processing in subsequent phases and rely on annotation-driven IoC to enrich phases with processor-specific logic. The framework is written in Kotlin[8], and thus fully compatible with Java and the Java Virtual Machine. Its implementation can be found on GitHub[9].

Next to built-in model processing phases for model validation and code generation, LEMMA's model processing framework also includes a phase called Live Validation. While the Model Validation phase prints warnings and errors issued by model processors to the standard output, the Live Validation phase is capable of interacting with a running Eclipse IDE to show validation messages directly within the model editor. Consequently, modelers can construct LEMMA models with interactive validation support and directly fix processor-specific model issues as they occur. Technically, the Live Validation phase is based on the Language Server Protocol

[8] https://kotlinlang.org
[9] https://github.com/SeelabFhdo/lemma/tree/master/de.fhdo.lemma.model_processing

(LSP)[10] and its Eclipse implementation[11]. More specifically, each LEMMA model processor also represents an LSP client. Model validations realized by model processors in their Model Validation phase are automatically executed during Live Validation without any additional implementation effort required.

## 3.3 Extending LEMMA with Support for Event Sourcing and CQRS

In the following, we illustrate the extension of LEMMA with support for the Event Sourcing and CQRS patterns as prescribed by the pattern-related technology models (cf. Subsection 3.1) and constraints (cf. Subsection 3.2). To this end, we provide a holistic example for LEMMA modeling in the Financial Service domain (cf. Section 2). Subsubsection 3.3.1 describes the corresponding LEMMA domain model. Subsubsection 3.3.2 presents the LEMMA service models, which build upon the pattern-specific technology models. Subsubsection 3.3.3 illustrates the implementation and usage of the pattern-specific model validators to ensure pattern-compliance in LEMMA models.

### 3.3.1 LEMMA Domain Model

Listing 3 shows the LEMMA domain model for the example from the Financial Service domain. It is expressed in LEMMA's Domain Data Modeling Language (cf. Table 1) and employs patterns from DDD [7, 8] to model bank accounts, commands, and domain events.

Listing 3: Example LEMMA domain model for the DDD Bounded Context `Account`.

```
// Model file name: Account.data
context Account {
  structure Account<aggregate, entity> {
    long id<identifier>,
    string owner,
    double balance
  }

  structure CreateAccountCommand<valueObject> {
    immutable string owner,
    immutable double initialBalance
  }

  structure DepositMoneyCommand<valueObject> {
    immutable long accountId,
    immutable double amount
  }

  structure AccountCreatedEvent<valueObject, domainEvent> {
    immutable long id,
    immutable string owner,
    immutable double initialBalance
  }

  structure MoneyDepositedEvent<valueObject, domainEvent> {
    immutable long accountId,
    immutable double amount,
    immutable double newBalance
  }
}
```

The domain model defines the `Account` context. Semantically, LEMMA's `context` keyword corresponds to the Bounded Context pattern from DDD [7]. In MSA, bounded contexts may be used to cluster coherent, domain-specific data of a microservice [17, 6]. The `Account` context

[10] https://microsoft.github.io/language-server-protocol
[11] https://projects.eclipse.org/projects/technology.lsp4j

in Listing 3 comprises the structured `Account` domain concept, which is modeled as a DDD Aggregate and Entity [7] (cf. the `aggregate` and `entity` keywords in Line 3). The domain concept comprises three data fields. The `id` field acts as the `identifier` of an `Account` instance. The `owner` field identifies the owner of a bank account and the `balance` field holds the current account balance.

Lines 9 to 17 of Listing 3 specify two DDD Value Objects [7], each of which represents a command in the sense of the CQRS pattern (cf. Subsection 2.2). `CreateAccountCommand` (cf. Lines 9 to 12) signals the creation of a new bank account, including its `owner` and `initialBalance`. Both fields are `immutable` and receive a value only once upon creation of a new bank account [23]. Immutability is a a common characteristic of DDD Value Objects [7]. `DepositMoneyCommand` (cf. Lines 14 to 17) is a DDD Value Object and CQRS command for depositing money in a bank account. Therefore, it clusters the `accountId` and deposited `amount`.

Lines 19 to 29 of Listing 3 define two DDD Value Objects and domain events [8]. The `AccountCreatedEvent` (cf. Lines 19 to 23) communicates the creation of a new bank account, including its `id`, `owner`, and `initialBalance`. The `MoneyDepositedEvent` (cf. Lines 25 to 29) signals the deposit of money for a bank account. It conveys the `accountId` and the deposited `amount`. Furthermore, it considers the `newBalance` of the account, whose calculation is in the responsibility of the `DepositMoneyCommand` receiver (cf. Subsubsection 3.3.2).

### 3.3.2 LEMMA Service Models

Starting from the domain model (cf. Subsubsection 3.3.1), LEMMA service models may now be specified. Listing 4 shows the command side microservice of the logical CQRS-enabled `AccountMicroservice`, which is responsible for the `Account` bounded context in Listing 3. The command side microservice is expressed in LEMMA's Service Modeling Language (cf. Table 1).

Listing 4: Excerpt of the example LEMMA service model for the command side of the logical `AccountMicroservice`.

```
1  // Model file name: Account.services
2  import datatypes from "Account.data" as Account
3  import technology from "EventSourcing.technology" as EventSourcing
4  import technology from "Cqrs.technology" as CQRS
5
6  @technology(EventSourcing)
7  @technology(CQRS)
8  @CQRS::_aspects.CommandSide(logicalService="AccountMicroservice")
9  functional microservice org.example.AccountCommand {
10   interface CommandSide {
11     ---
12     API endpoint for creating a new account
13     @required command Command object to specify the values of the new account
14     ---
15     createAccount(
16       sync in command : Account::Account.CreateAccountCommand,
17       sync out accountId : long
18     );
19
20     ---
21     API endpoint for money deposit
22     @required command Deposit command object
23     ---
24     depositMoney(
25       sync in command : Account::Account.DepositMoneyCommand
26     );
27
28     @EventSourcing::_aspects.Producer(handlerName="AccountEventProducer")
29     sendAccountCreatedEvent(
30       async out event : Account::Account.AccountCreatedEvent
31     );
```

```
32
33      @EventSourcing::_aspects.Producer(handlerName="AccountEventProducer")
34      sendMoneyDepositedEvent(
35        async out event : Account::Account.MoneyDepositedEvent
36      );
37    }
38  }
```

In Lines 2 to 4, the service model first imports the "Account.data" domain model (cf. Listing 3), and the Event Sourcing as well CQRS technology models (cf. Listings 1 and 2). Next, Lines 6 to 7 use LEMMA's built-in `@technology` annotation to assign the imported technologies to the following `org.example.AccountCommand` microservice. In Line 8, the `CommandSide` aspect from the CQRS technology model is assigned to the service. Consequently, it is semantically recognizable as the command side of the logical `AccountMicroservice` (cf. Subsubsection 3.1.2).

In Lines 11 to 18 of Listing 4, the `createAccount` microservice operation is specified [23]. It expects an instance of the `CreateAccountCommand` from the `Account` domain model (cf. Subsubsection 3.3.1) in the form of the synchronous `command` parameter and returns the `accountId` of the newly created bank account after having executed the command. The `depositMoney` operation in Lines 20 to 26 is responsible for handling the `DepositMoneyCommand` from the `Account` domain model.

The events subsequent to the `CreateAccountCommand` and `DepositMoneyCommand` are sent by means of the microservice operations `sendAccountCreatedEvent` (cf. Lines 28 to 31 in Listing 4) and `sendMoneyDepositedEvent` (cf. Lines 33 to 36). Therefore, both operations specify an asynchronous outgoing parameter `event` [23] being typed by the corresponding domain event from the `Account` domain model (cf. Subsubsection 3.3.1). The `Producer` aspect from the Event Sourcing technology model identifies both operations as domain event producers (cf. Subsubsection 3.1.1).

Listing 5 models a query side microservice `org.example.AccountQuery` for the logical `AccountMicroservice`. The query side model is part of the same LEMMA service model file as the command side model (cf. Listing 4).

Listing 5: Excerpt of the example LEMMA service model for the query side of the logical `AccountMicroservice` (continuation of Listing 4).

```
1   // Model file name: Account.services
2   ...
3
4   @technology(EventSourcing)
5   @technology(CQRS)
6   @CQRS::_aspects.QuerySide(logicalService="AccountMicroservice")
7   functional microservice org.example.AccountQuery {
8     required microservices {
9       AccountCommand
10    }
11
12    interface QuerySide {
13      ...
14
15      @EventSourcing::_aspects.Consumer(handlerName="AccountEventConsumer")
16      receiveAccountCreatedEvent(
17        async in event : Account::Account.AccountCreatedEvent
18      );
19
20      @EventSourcing::_aspects.Consumer(handlerName="AccountEventConsumer")
21      receiveMoneyDepositedEvent(
22        async in event : Account::Account.MoneyDepositedEvent
23      );
24    }
```

```
25 }
```

Similarly to the command side microservice in Listing 4, the `org.example.AccountQuery` microservice is augmented with the Event Sourcing and CQRS technology models. Moreover, the service is preceded with the `QuerySide` aspect from the CQRS technology model (cf. Subsubsection 3.1.2). The value of the `logicalService` property identifies `AccountQuery` as a query side microservice for the command side microservice `AccountCommand` (cf. Listing 4). In addition, `AccountQuery` specifies `AccountCommand` as a required microservice [23] (cf. Lines 8 to 10 in Listing 5). That is, because, following the CQRS pattern, `AccountQuery` depends on the command side microservice `AccountCommand` to receive updates to domain objects in the form of events (cf. Subsection 2.2).

In Lines 15 to 18, Listing 5 models the `receiveAccountCreatedEvent` operation. It is the receiving counterpart to the command side's `sendAccountCreatedEvent` operation (cf. Lines 28 to 31 in Listing 4), because it defines an asynchronous incoming `event` parameter typed with the `AccountCreatedEvent` domain event from the `Account` domain model (cf. Subsubsection 3.3.1). Furthermore, the operation exhibits the `Consumer` aspect from the Event Sourcing technology model to identify itself as an event consumer (cf. Subsubsection 3.1.1). Similarly, the `receiveMoneyDepositedEvent` operation in Lines 20 to 23 of Listing 5 is the receiving counterpart of the `sendMoneyDepositedEvent` operation in Lines 33 to 36 of Listing 4.

### 3.3.3 Model Validators

In the following, we briefly describe the implementation of model validators for the constraints listed in Subsubsections 3.2.1 and 3.2.2. The model validators can be used to check the service models from Subsubsection 3.3.2 for compliance with the Event Sourcing and CQRS patterns.

We developed a model validator for each of the two pattern-specific technology models (cf. Subsection 3.1) by means of LEMMA's model processing framework. That is, the Event Sourcing Validator focuses on the Event Sourcing pattern and the CQRS Validator is capable of checking service models for CQRS compliance. To illustrate the implementation of LEMMA model validators, each of the following Listings 6 to 8 shows an excerpt from the Kotlin implementation of the Event Sourcing Validator. Listing 6 comprises the entry point of the validator.

Listing 6: Entry point of the Event Sourcing Validator.

```
1 class EventSourcingValidator : AbstractModelProcessor("org.example.lemma.event_sourcing")
2
3 fun main(args: Array<String>) {
4   EventSourcingValidator().run(args)
5 }
```

A LEMMA model validator inherits from the `AbstractModelProcessor` class from LEMMA's model processing framework. Its constructor receives a single string value, which identifies the Java package for the framework to lookup implementation units of model processing phases following the IoC pattern (cf. Subsubsection 3.2.3). In Line 4, control over the model validator's flow of execution is delegated to the model processing framework by invoking the `run` method and passing possible command-line parameters (`args`) to it.

Each LEMMA model processor requires a *language description provider*. That is, because LEMMA's model processing framework relies on infrastructure components provided by Xcore and Xtext (cf. Subsection 3.2) to parse LEMMA model files. However, since the infrastructure components are derived by Xcore and Xtext specific to a certain modeling language, the

model processing framework needs to access them at runtime. This access is enabled by model processor implementations via language description providers.

Listing 7 shows the language description provider for the Event Sourcing Validator, which needs to parse mapping and service models in order to validate them (cf. Subsubsection 3.2.1).

Listing 7: Language description provider of the Event Sourcing Validator.

```
1  @LanguageDescriptionProvider
2  class DescriptionProvider : LanguageDescriptionProviderI {
3    override fun getLanguageDescription(forLanguageNamespace: String) : LanguageDescription? {
4      return when(forLanguageNamespace) {
5        MappingPackage.eNS_URI -> MAPPING_DSL_LANGUAGE_DESCRIPTION
6        ServicePackage.eNS_URI -> SERVICE_DSL_LANGUAGE_DESCRIPTION
7        else -> null
8      }
9    }
10 }
11
12 val MAPPING_DSL_LANGUAGE_DESCRIPTION = XtextLanguageDescription(
13   MappingPackage.eINSTANCE,
14   MappingDslStandaloneSetup()
15 )
16
17 val SERVICE_DSL_LANGUAGE_DESCRIPTION = XtextLanguageDescription(
18   ServicePackage.eINSTANCE,
19   ServiceDslStandaloneSetup()
20 )
```

At model processor runtime, a language description provider is identified from the Java implementation package of the model processor (cf. Listing 6) as a class with the `@LanguageDescriptionProvider` annotation, which also implements the `LanguageDescriptionProviderI` interface. The interface prescribes the `getLanguageDescription` method to be implemented. It must return a `LanguageDescription` instance, which encapsulates the required Xcore and Xtext infrastructure components, per supported model type of a model processor. The model type is identified by its Ecore namespace (`eNS_URI`). In the excerpt in Listing 7, two language descriptions are returned by the language description provider. `MAPPING_DSL_LANGUAGE_DESCRIPTION` (cf. Lines 12 to 15) covers mapping models (cf. Table 1) and `SERVICE_DSL_LANGUAGE_DESCRIPTION` targets service models.

Starting from the model processor entry point and language description provider implementations (cf. Listings 6 and 7), the model processor is enabled to parse LEMMA mapping and service models, and handle them within the considered model processing phases (cf. Subsubsection 3.2.3). Listing 8 shows an excerpt from the model validator for the Event Sourcing technology model (cf. Subsubsection 3.1.1).

Listing 8: Constraint validation within the Event Sourcing Validator.

```
1  @SourceModelValidator(
2    validationMode = SourceModelValidationMode.XTEXT,
3    supportedFileExtensions = ["services"]
4  )
5  class ServiceModelSourceValidator : AbstractXtextSourceModelValidator() {
6    @Check
7    private fun checkProducer(operation: Operation) {
8      val eventSourcingAlias = operation.interface.microservice
9        .findAliasForTechnology("EventSourcing") ?: return
10     if (operation.hasServiceAspect(eventSourcingAlias, "Producer") &&
11       !operation.hasResultParameters(CommunicationType.ASYNCHRONOUS))
12       error("The Producer aspect may only be applied to operations with a result parameter",
13         ServicePackage.Literals.OPERATION__NAME)
14   }
15
16   ...
```

```
17 }
```

A model validator in LEMMA's model processing framework is identified at runtime based on the `@SourceModelValidator` annotation. Furthermore, it must inherit from the `AbstractXtextSourceModelValidator`, in case the `validationMode` is set to `XTEXT`, which is required to parse ASTs from LEMMA models. The validation of parsed models then follows Xtext's Custom Validation mechanism[12]. That is, validations of model elements are realized within methods that are annotated with `@Check` and specify a single parameter. If the type of the parameter corresponds to a concept from the modeling language used to construct a parsed model, the validation method will be invoked for each instance of the concept in the parsed AST.

For example, the `checkProducer` method in Lines 6 to 14 of Listing 8 realizes the validation of constraint C.3 (cf. Table 2). Therefore, it operates on the `Operation` concept from the Service Modeling Language [23]. In Lines 8 and 9, `checkProducer` retrieves the alias of the import [23] of the Event Sourcing technology model. In Lines 10 and 11, the passed microservice `Operation` instance is checked for the existence of the `Producer` aspect from the Event Sourcing technology model, as well as asynchronous outgoing parameters. If the aspect is present but no result parameters were modeled, an error message is displayed to the modeler. As described in Subsubsection 3.2.3, depending on how the Event Sourcing Validator was invoked the error message is either written to standard output or displayed within the Eclipse editor for the Service Modeling Language. Figure 1 shows the display of the validation error in Eclipse, in case a violation of constraint C.3 is detected by the Event Sourcing Validator during Live Validation[13].

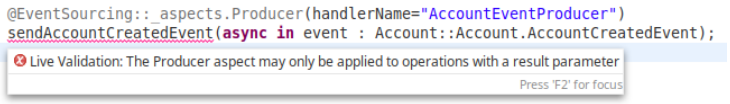


Figure 1: Example of a constraint violation error detected by the Event Sourcing Validator for constraint C.3 (cf. Table 2). The error is displayed directly in Eclipse at modeling time by means of the Live Validation model processing phase (cf. Subsubsection 3.2.3).

# 4 Discussion

This section discusses our initial results from investigating the non-intrusive extension of LEMMA with modeling support for architecture patterns. We structure our discussion based on the RQs posed in Section 1.

## 4.1 Research Question 1: Pattern Integration Steps

RQ 1 focuses on the steps being required to extend LEMMA with architecture pattern support. As described in Section 3, the integration process is twofold. First, a technology model needs to be specified for each architecture pattern (cf. Subsection 3.1). The model captures the semantic concepts of the architecture pattern, and enables their assignment to LEMMA domain

[12] https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#validation

[13] To cause the error for the sake of illustration, the `out` keyword in Line 30 of Listing 4 was replaced with LEMMA's `in` keyword [23].

and service models (cf. Table 1). Given the flexibility of LEMMA's Technology Modeling Language [20], upfront reasoning about the degree of technology-alignment of the captured pattern is sensible. For example, the initial Event Sourcing and CQRS technology models (cf. Subsubsections 3.1.1 and 3.1.2) are technology-agnostic. Consequently, a third technology model, which clusters technology for the required event-driven communication, is needed. Such a model could, for example, target a persisting event broker such as Apache Kafka[14]. Alternatively, an existing Kafka technology model could be extended with Event Sourcing and CQRS pattern concepts in the form of technology aspects. While this approach mixes technology-specific with technology-agnostic aspects, and constrains the applicability of the resulting technology model to Kafka, it would lower domain and service modeling complexity, because there exists only one technology model, which clusters all relevant aspects at once.

The second step in the pattern integration process is the provisioning of model validators (cf. Subsection 3.2). As illustrated in Subsubsection 3.3.3, each technology model should be accompanied by exactly one model validator in the form a standalone executable Java archive based on LEMMA's model processing framework. The validation of a pattern's application for conceptual compliance is then realized in dedicated validation methods. For the sake of concern separation, each validation method should focus on a single modeling language concept and pattern aspect. Given the IoC approach of the framework (cf. Subsubsection 3.2.3), new validation methods can be added at a single, well-defined place, i.e., the model validator class (cf. Listing 8). The invocation of validation methods is then triggered automatically by LEMMA's model processing framework. Another benefit of the framework's adoption is its integration with Eclipse via LSP. As a result, validation errors in models are directly displayed at modeling time, and may immediately and interactively be fixed as the modeler types.

Since both integration steps do not require an adaption of any LEMMA modeling language, the overall integration process is non-intrusive. Consequently, LEMMA may be extended with support for new architecture patterns as required.

## 4.2 Research Question 2: Required Pattern Integration Artifacts

The required artifacts for architecture pattern integration with LEMMA follow directly from the steps of the integration process (cf. Subsection 4.1). Hence, LEMMA modelers need to be provided with a technology model and executable Java archive of the corresponding model validator. Both artifacts may be shared across microservice teams, e.g., by means of a *shared repository* [17]. Concerning the model validator, its code should also be made accessible to transparently enable testing and fixing of issues.

## 4.3 Research Question 3: Complexity of the Pattern Integration Process

In its current form, the process to non-intrusively extend LEMMA with support for architecture patterns is not trivial. First, it requires upfront analysis of targeted architecture patterns and their adoption in the microservice development process in order to construct sufficient technology models. Second, knowledge of LEMMA's modeling languages and their concepts is mandatory. Otherwise, the implementation of model validators will not be feasible, because validation methods are aligned to modeling languages' concepts (cf. Subsubsection 3.3.3). However, LEMMA's model processing framework reliefs implementers from direct interaction

[14] https://kafka.apache.org

with Xcore and Xtext infrastructure components, e.g., for parsing, validation triggering, and LSP client realization.

Nonetheless, the target group for extending LEMMA with architecture pattern support are technology-savvy stakeholders in MSA engineering like microservice developers and software architects. Once technology models and corresponding model validators have been realized, they may be shared across the MSA development organization and we expect extensions to be straightforward based on LEMMA's model processing framework. However, a central action in our future work is to aim for lowering the complexity of the extension process (cf. Section 6).

# 5 Related Work

Kallel et al. [11] present an approach to formalize architecture patterns as constraints in the Object Constraint Language (OCL) [18], and use them to check for the correctness of patterns' application at design time and runtime. For design time checking, UML-based architecture models are validated for constraint satisfaction and thus pattern-compliance. For runtime checking, Java meta-programs are generated from pattern-specific OCL constraints. These meta-programs rely on Aspect-oriented Programming (AOP) [12] to automatically validate Java objects at runtime for pattern-compliance. By contrast to our approach, the usage of OCL for constraint-based architecture pattern specification has two advantages. First, OCL is a dedicated language for constraint expression and thus provides concise language concepts optimized for, e.g., AST traversal and filtering, as well as the definition of invariants. Second, OCL models can be flexibly extended, e.g., when further pattern constraints become apparent or new concepts need to be supported for an already formalized pattern. In the context of LEMMA, OCL models could thus replace the manual implementation of model validators for constraint checking (cf. Subsection 3.2). Instead, each pattern-specific technology model could be accompanied by an OCL model that guides technology aspect application for correct pattern usage. Code for validation methods (cf. Listing 8) could then be automatically derived from OCL constraints, e.g., by means of Eclipse OCL[15]. By contrast to the approach of Kallel et al. [11], constraint violations could then be fixed interactively in Eclipse using the built-in Live Validation phase of LEMMA's model processing framework.

JDL[16] and MicroDSL [27] are modeling languages for MSA, which integrate technology and pattern elements on the language level. JDL, for example, includes the `openshiftNamespace` and `gatewayType` keywords to enable modeling in the context of OpenShift[17] or specify API Gateways [16]. Similarly, MicroDSL provides keywords for the Load Balancer and Circuit Breaker patterns [15]. While the consideration of technologies and patterns on the language level fosters a modeling language's expressivity, its learnability and stability decrease. That is, because the languages come with an increased set of syntactic constructs and semantics [2]. For example, in JDL each technology- or pattern-specific keyword has its own set of valid values that need to be known in advance to construct valid models. Moreover, each new technology or pattern requires a language extension. In addition, in case a technology or pattern is not supported by the languages, users are required to reach out to languages' developers for its integration or extend the languages themselves. Our Technology Modeling Language (cf. Table 1), on the other hand, provides a means for non-intrusive language extension. Technology models can be constructed by MSA modelers without the need to extend any of LEMMA's modeling languages (cf. Subsection 3.1). In addition, technologies and patterns can be constrained to the

[15] https://projects.eclipse.org/projects/modeling.mdt.ocl
[16] https://www.jhipster.tech/jdl
[17] https://www.openshift.com

necessary minimum, keeping the syntactic core of LEMMA's modeling language comparatively concise. Moreover, LEMMA's model processing framework (cf. Subsection 3.2) supports the subsequent implementation of model validators to ensure correctness of language extensions formalized in technology models.

Cuadrado and Molina [4] present a phasing mechanism in the context of rule-based model transformation languages such as ATL[18]. Like in LEMMA's model processing framework, the phase concept is used to modularize coherent parts of a model transformation into logical stages (cf. Subsubsection 3.2.3). To systematize the introduction of a phasing mechanism into rule-based modeling languages, Cuadrado and Molina describe the relevant concepts as an abstract, language-agnostic syntax, which shares similarities with the design concepts of the model processing framework. First, a phase may define several parameters required for its execution. Second, phases may be prevented from executing either explicitly by the caller or implicitly because a precondition, like an expected parameter value, does not hold. Third, phases may be dependent on each other and thus imply a certain execution order. By contrast to our approach, Cuadrado and Molina distinguish between primitive and composite phases. A primitive phase defines production rules of a transformation, while a composite phase organizes several primitive or other composite phases into logical units. LEMMA's model processing framework does not enable phase composition besides specifying phase ordering. Instead, we consider all model processing phases as primitive phases. We expect the missing layer of indirection, i.e., the composition of phases from others, to facilitate the implementation of model processors for MSA stakeholders without experience in MDE or model processing. Moreover, Cuadrado and Molina do not present a framework to practically leverage their phasing mechanism.

# 6 Conclusion and Future Work

This paper presented a non-intrusive approach to extend our Language Ecosystem for Modeling Microservice Architecture (LEMMA) with architecture pattern support. The approach leverages LEMMA's aspect mechanism to capture pattern concepts in the form of aspects within dedicated models. Starting from these models, LEMMA domain and service models may be flexibly augmented with pattern-specific aspects. The pattern-compliant application of aspects is then validated by pattern-specific model validators realized with LEMMA's model processing framework. A model validator may be executed as a standalone application or connect with model editors leveraging the Language Server Protocol to enable immediate and interactive pattern validation and error fixing. We illustrated our approach on the example of the Event Sourcing and Command Query Responsibility Segregation (CQRS) patterns. Furthermore, we discussed it w.r.t. the required steps and artifacts for extending LEMMA with architecture pattern support, as well as its complexity.

In future works, we aim to lower the complexity of our approach by relying on the Object Constraint Language to specify constraints for pattern-specific aspects and automatically generate code for pattern-specific model validators based on LEMMA's model processing framework. Moreover, we plan to further investigate the initial aspect models for the Event Sourcing and CQRS patterns concerning their completeness and applicability. Additionally, we plan to formalize further architecture patterns for their usage with LEMMA.

[18] https://www.eclipse.org/atl

# References


[1] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to cloud-native architectures using microservices: An experience report. In Antonio Celesti and Philipp Leitner, editors, *Advances in Service-Oriented and Cloud Computing*, pages 201–215. Springer, 2016.

[2] Ankica Barišic, Vasco Amaral, Miguel Goulão, and Bruno Barroca. *Evaluating the Usability of Domain-Specific Languages*, pages 2120–2141. Software Design and Development: Concepts, Methodologies, Tools, and Applications. IGI Global, 2014.

[3] Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, Bernhard Rumpe, Jim Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. CRC Press, first edition, 2017.

[4] Jesús Sánchez Cuadrado and Jesús García Molina. Modularization of model transformations through a phasing mechanism. *Software & Systems Modeling*, 8(3):325–345, Jul 2009. Springer.

[5] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. Research on architecting microservices: Trends, focus, and potential for industrial adoption. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 21–30. IEEE, 2017.

[6] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. In Manuel Mazzara and Bertrand Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer, Cham, 2017.

[7] Eric Evans. *Domain-Driven Design*. Addison-Wesley, first edition, 2004.

[8] Eric Evans. *Domain-Driven Design Reference*. Dog Ear Publishing, first edition, 2015.

[9] ISO/IEC. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. Standard ISO/IEC 25010:2011(E), International Organization for Standardization/International Electrotechnical Commission, 2011.

[10] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988. SIGS Publications.

[11] Sahar Kallel, Chouki Tibermacine, Slim Kallel, Ahmed Hadj Kacem, and Christophe Dony. Specification and automatic checking of architecture constraints on object oriented programs. *Information and Software Technology*, 101:16–31, 2018.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, 1997. Springer.

[13] Holger Knoche and Wilhelm Hasselbring. Drivers and barriers for microservice adoption – a survey among professionals in germany. *Enterprise Modelling and Information Systems Architectures*, 14(1):1–35, 2019. German Informatics Society.

[14] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Model-transformation design patterns. *IEEE Transactions on Software Engineering*, 40(12):1224–1259, 2014. IEEE.

[15] Gastón Márquez, Mónica M. Villegas, and Hernán Astudillo. A pattern language for scalable microservices-based systems. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, ECSA '18, pages 24:1–24:7. ACM, 2018.

[16] Gastón Márquez and Hernán Astudillo. Actual use of architectural patterns in microservices-based open source projects. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 31–40. IEEE, 2018.

[17] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, first edition, 2015.

[18] OMG. Object constraint language version 2.4. Standard, Object Management Group, 2014.

[19] Felipe Osses, Gastón Márquez, and Hernán Astudillo. Exploration of academic and industrial evidence about architectural tactics and patterns in microservices. In *Proceedings of the 40th*

*International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 256—257. ACM, 2018.

[20] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Aspect-oriented modeling of technology heterogeneity in microservice architecture. In *2019 IEEE International Conference on Software Architecture (ICSA)*, pages 21–30. IEEE, 2019.

[21] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Deriving microservice code from underspecified domain models using devops-enabled modeling languages and model transformations. In *Proc. of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2020)*. To appear, IEEE, 2020.

[22] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. A modeling method for systematic architecture reconstruction of microservice-based software systems. In Selmin Nurcan, Iris Reinhartz-Berger, Pnina Soffer, and Jelena Zdravkovic, editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 311–326. Springer, 2020.

[23] Florian Rademacher, Jonas Sorgalla, Philip Wizenty, Sabine Sachweh, and Albert Zündorf. Graphical and textual model-driven microservice development. In *Microservices: Science and Engineering*, pages 147–179. Springer, 2020.

[24] Chris Richardson. *Microservices Patterns*. Manning Publications, 2019.

[25] Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Alfonso Pierantonio. Developing next generation ADLs through MDE techniques. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 85–94. IEEE, 2010.

[26] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018. Elsevier.

[27] Branko Terzić, Vladimir Dimitrieski, Slavica Kordić, Gordana Milosavljević, and Ivan Luković. Development and evaluation of microbuilder: a model-driven tool for the specification of rest microservice software architectures. *Enterprise Information Systems*, 12(8-9):1034–1057, 2018. Taylor & Francis.