Model-Driven Code Generation for Microservices: Service Models

Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher

Model-Driven Code Generation for Microservices: Service Models

Saverio Giallorenzo
Università di Bologna, Italy
INRIA, Sophia Antopolis, France

Fabrizio Montesi
University of Southern Denmark, Odense, Denmark

Marco Peressotti
University of Southern Denmark, Odense, Denmark

Florian Rademacher
Software Engineering, RWTH Aachen University, Germany

Abstract

We formally define and implement a translation of domain and service models expressed in the LEMMA modelling ecosystem for microservice architectures to source code in the Jolie microservice programming language. Specifically, our work extends previous efforts on the generation of Jolie code to the inclusion of the LEMMA service modelling layer.

We also contribute an implementation of our translation, given as an extension of the LEMMA2Jolie tool, which enables the practical application of our encoding. As a result, LEMMA2Jolie now supports a software development process whereby microservice architectures can first be designed by microservice developers in collaboration with domain experts in LEMMA, and then be automatically translated into Jolie APIs. Our tool can thus be used to enhance productivity and improve design adherence.

2012 ACM Subject Classification Software and its engineering → Software development methods; Applied computing → Service-oriented architectures; Software and its engineering → Model-driven software engineering

Keywords and phrases Microservices, Model-driven Engineering, Code Generation, Jolie APIs

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.6

Supplementary Material *Software (Source Code)*: `https://github.com/jolie/lemma2jolie` archived at `swh:1:dir:66cd1f01a83c6b26f7b22edc76291cf7ed7cc460`

Funding *Fabrizio Montesi*: Work partially supported by Villum Fonden, grants no. 29518 and 50079, and the Independent Research Fund Denmark, grant no. 0135-00219.

1 Introduction

Microservice Architecture (MSA) has risen to be a popular approach [24], but it also presents challenges related to design, development, and operation [5, 35]. To tackle design and development, researchers in software engineering and programming languages have proposed linguistic approaches to MSA, which feature high-level abstractions aimed at making microservice concerns more visible.

Model-Driven Engineering (MDE) [2] is a popular method for designing service architectures [1]. MDE can be applied to MSA by means of modelling languages such as MicroBuilder, MDSL, LEMMA, and JHipster [37, 39, 30, 15]. LEMMA, in particular, has been validated in real-world applications [36, 31]. On the side of development, Ballerina and Jolie [25, 22] are programming languages oriented towards services and their coordination. Jolie's abstractions have been found to improve productivity in industry [13], and LEMMA's support for Domain-Driven Design has been validated in real-world applications [36, 31].

In recent work, Giallorenzo et al. [10] observed that the metamodels of LEMMA and Jolie have numerous contact points. This motivated the quest for integrating the two tools and their approaches, which in the long term could bring (quoting from [10])

> "*an ecosystem that coherently combines MDE and programming abstractions to offer a tower of abstractions [18] that supports a step-by-step refinement process from the abstract specification of a microservice architecture to its implementation*".

In other words, the objective is building a toolset that allows for (i) designing an MSA using the principles of MDE, and then (ii) seamlessly switching to implementing the design with a programming language that offers dedicated linguistic support for coding microservices. Achieving this objective requires integrating three elements of the metamodels of both LEMMA and Jolie [10]:

1. *Application Programming Interfaces* (API), describing what functionalities (and their data types) a microservice offers to its clients;
2. *Access Points*, capturing where and how clients can interact with a microservice's API;
3. *Behaviours*, defining the internal business logic of a microservice.

In [9], we started addressing the first element, by presenting an encoding, and a tool built on such encoding, that translates a large fragment of LEMMA's Domain Data Modelling Language (DDML) to Jolie types and interfaces. However, this encoding ignored the important aspects of modelling services, and in particular their interfaces in terms of operations and their associated communication patterns (e.g., synchronous vs asynchronous data provision). In this paper, we aim to bridge this gap and obtain the first prototype of an API generator from LEMMA service models.

Since the API is the layer the other two build upon, in this paper we focus on concretising the relationship between LEMMA and Jolie API layers. To this end, we extend previous work focused on a formal encoding from a large fragment of LEMMA's Domain Data Modelling Language (DDML) to Jolie types and interfaces [9]

Our key contribution is extending the encoding in [9] to a significant fragment of LEMMA's Service Modelling Language (SML); the one used for defining a set of microservices with their interfaces, operations, and accompanying communication patterns. Our extended encoding supports the systematic translation of LEMMA domain models – which, following Domain-Driven Design (DDD) [6] principles, capture domain-specific types including operation signatures – to Jolie APIs. As a second contribution, we extend the tool presented in [9], called LEMMA2Jolie, to accept both DDML and SML models and translate these into Jolie APIs, following the extended version of the encoding presented in this paper.

Taken together, these contributions constitute a new milestone on the roadmap traced in [10] for building a conceptual and technical bridge between the communities of programming languages and MDE on microservices. Specifically, our previous work made domain information from microservices' design actionable [9]. Here, we build upon our previous work [9] and move forward by adding support for the Service Viewpoint in MSA engineering [30]. While domain modelling is essential to most software systems and independent of the implemented architectural style, service modelling is essential to MSA, as it reifies the foundational concepts of information hiding and component interfacing. Therefore, this contribution completes previous work on APIs [9], and is pivotal for future activities that address the remaining elements, i.e., Access Points and Behaviours.

The remainder of the paper is organised as follows. Section 2 presents modelling concepts from LEMMA's DDML and SML and the relevant elements of the Jolie APIs required by the encoding, which we present in Section 3. Section 4 describes the implementation of LEMMA2Jolie and illustrates it with an example. Section 5 presents related work and a concluding discussion.

$$
\begin{array}{lcl}
CTX & ::= & \textbf{context}\ id\ \{\overline{CT}\} \\
CT & ::= & STR \mid COL \mid ENM \\
STR & ::= & \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\ \overline{OPS}\} \\
STRF & ::= & \textbf{aggregate} \mid \textbf{domainEvent} \mid \textbf{entity} \mid \textbf{factory} \\
 & \mid & \textbf{service} \mid \textbf{repository} \mid \textbf{specification} \mid \textbf{valueObject} \\
FLD & ::= & id\ id\ [\langle\overline{FLDF}\rangle] \mid S\ id\ [\langle\overline{FLDF}\rangle] \\
FLDF & ::= & \textbf{identifier} \mid \textbf{part} \\
OPS & ::= & \textbf{procedure}\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD}) \mid \textbf{function}\ (id \mid S)\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD}) \\
OPSF & ::= & \textbf{closure} \mid \textbf{identifier} \mid \textbf{sideEffectFree} \mid \textbf{validator} \\
COL & ::= & \textbf{collection}\ id\ \{(S \mid id)\} \\
ENM & ::= & \textbf{enum}\ id\ \{\overline{id}\} \\
S & ::= & \textbf{int} \mid \textbf{string} \mid \textbf{unspecified} \mid \ldots
\end{array}
$$

Figure 1 Simplified grammar of LEMMA's DDML [9]. Greyed out features are out of the scope of this paper and subject to future work.

2 Background

This section describes and exemplifies domain and service modelling with LEMMA, and the development of microservice APIs with Jolie.

2.1 LEMMA Domain Modelling Concepts

LEMMA's DDML supports domain experts and service developers in the construction of models that capture domain-specific types of microservices. We include the core grammar of this language in Figure 1 (grayed elements are not relevant for the translation presented in this work).[1]

LEMMA's DDML captures the foundational DDD concepts for MSA design. DDD's Bounded Context pattern [6] marks the boundaries of coherent domain concepts, thereby defining their scope and applicability [24]. A LEMMA domain model defines named bounded **context**s (rule CTX in Figure 1). A **context** may specify domain concepts in the form of complex types (CT), which are either structures (STR), collections (COL), or enumerations (ENM).

A **structure** gathers a set of data fields (FLD) each associated with a type that can be either a complex type from the same bounded context (id) or a built-in primitive type, e.g., **int** or **string** (S). LEMMA support continuous domain exploration by allowing the construction of underspecified models by means of the keyword **unspecified**. This concise solution provides domain experts and developers with a light-weight facility for refining models as they gain new domain knowledge [29]. **structure**s can comprise operation signatures (OPS) to reify domain-specific behaviour. An operation is either a **procedure** without a return type, or a **function** with a complex or primitive return type.

[1] The complete grammar can be found at `https://github.com/SeelabFhdo/lemma/blob/main/de.fhdo.lemma.data.datadsl/src/de/fhdo/lemma/data/DataDsl.xtext`.

$$
\begin{array}{lcl}
SVR & ::= & \textbf{microservice}\ id\ \{\overline{IF}\} \\
IF & ::= & \textbf{interface}\ id\ \{\overline{IOP}\} \\
IOP & ::= & id\ (\overline{PAR}) \\
PAR & ::= & SYN\ DIR\ id : id \\
SYN & ::= & \textbf{sync} \mid \textbf{async} \\
DIR & ::= & \textbf{in} \mid \textbf{out}
\end{array}
$$

Figure 2 Simplified grammar of LEMMA's SML.

LEMMA's DDML supports the assignment of DDD patterns, called *features*, to structured domain concepts and their components. For instance, the **entity** feature (rule $STRF$ in Figure 1) expresses that a structure comprises a notion of domain-specific identity. The **identifier** feature then marks the data fields ($FLDF$) or operations ($OPSF$) of an **entity** which determine its identity.

The DDML also enables the modelling of **collection**s (rule COL in Figure 1), which represent sequences of primitives (S) or complex (id) values, as well as **enum**erations (ENM), which gather sets of predefined literals.

The following listing shows an example of a LEMMA domain model constructed with the grammar of the DDML [31].

```
context BookingManagement {
 structure ParkingSpaceBooking⟨entity⟩ {
  long bookingID⟨identifier⟩,
  double priceInEuro,
  function double priceInDollars
 }
}
```
LEMMA

The domain model defines the bounded **context** *BookingManagement* and its **structure**d domain concept *ParkingSpaceBooking*. It is a DDD **entity** whose *bookingID* field holds the **identifier** of an entity instance. The entity also clusters the field *priceInEuro* to store the price of a parking space booking, and the **function** signature *priceInDollars* for currency conversion of a booking's price.

2.2 LEMMA Service Modelling Concepts

We report in Figure 2 the (simplified) grammar of LEMMA's SML. Following the rules, we see that a LEMMA SML model can contain one or more **microservice**s, each associated with a name (id) and a collection of **interface**s. Each **interface** encloses a collection of operations, each identified by an id and a collection of PARameters. These parameters define the messaging pattern of the operation by associating each parameter – $id : id$, where the first id from the left is the name of the parameter and the second one is the name of its type (cf. Section 2.1) – with its timing of reception/transmission. Indeed, each parameter can either be **sync**hronous or **async**hronous and either be part of an **in**bound or an **out**bound message. We illustrate the matter with an example

$$
\begin{array}{lcl}
I & ::= & \textbf{interface } id \textbf{ \{}[\textbf{RequestResponse } \overline{id(TP_1)(TP_2)}][\textbf{OneWay } \overline{id(TP)}]\textbf{\}} \\
TP & ::= & id \mid B \\
TD & ::= & \textbf{type } id : \ T \\
T & ::= & B \ [\textbf{\{}\overline{id \ C : \ T}\textbf{\}}] \mid \textbf{undefined} \\
C & ::= & \textbf{[}min, max\textbf{]} \mid \ * \mid \textbf{?} \\
B & ::= & \textbf{int}[(R)] \mid \textbf{string}[(R)] \mid \textbf{void} \mid \ \ldots \\
R & ::= & \textbf{range(}\textbf{[}min, max\textbf{]}\textbf{)} \mid \textbf{length(}\textbf{[}min, max\textbf{]}\textbf{)} \mid \textbf{enum(}...\textbf{)} \mid \ \ldots
\end{array}
$$

Figure 3 Simplified syntax of Jolie APIs (types and interfaces).

```
interface Sample {
    op(sync in a : int, async in b : int, sync out c : int, async out d : int)
}
```
LEMMA

Above, we defined an **interface** called `Sample` which contains a single operation, *op*. The operation has four parameters. Starting from the leftmost, we find the parameter `a`, which is **sync**hronous and **in**bound. This means that `a` is part of the messages that *op* receives upon invocation. On the contrary, `b` is an **async**hronous **in**bound message, which means that it can reach *op* at any time between the invocation of *op* and its termination. Looking at the **out**bound parameters, we have `c` which is **sync**hronous, meaning that it is part of the message *op* sends when it terminated; `d`, on the contrary, is an **async**hronous **out**bound parameter, which *op* can transmit at any time between its invocation and its termination.

2.3 Jolie Types and Interfaces

Jolie interfaces and types define the functionalities of a microservice and the data types associated with those functionalities i.e., the API of a microservice. Figure 3 shows a simplified variant of the grammar of Jolie APIs, taken from [22] and updated to Jolie 1.10 (the latest major release at the time of writing).An **interface** is a collection of named operations (**RequestResponse**),where the sender delivers its message of type TP_1 and waits for the receiver to reply with a response of type TP_2 – although Jolie also supports **oneWay**s, where the sender delivers its message to the receiver, without waiting for the latter to process it (fire-and-forget), we omit them here because they are not used in the encoding (cf. Section 3). Operations have types describing the shape of the data structures they can exchange, which can either define custom, named types (id) or basic ones (B) (**int**egers, **string**s, etc.).

Jolie **type** definitions (TD) have a tree-shaped structure. At their root, we find a basic type (B) – which can include a refinement (R) to express constraints that further restrict the possible inhabitants of the type [7]. The possible branches of a **type** are a set of nodes, where each node associates a name (id) with an array with a range length (C) and a type T.

Jolie data types and interfaces are technology agnostic: they model Data Transfer Objects (DTOs) built on native types generally available in most architectures [4].

Based on the grammar in Figure 3, the following listing shows the Jolie equivalent of the example LEMMA domain model from Section 2.1.

```jolie
///@beginCtx(BookingManagement)
///@entity
type ParkingSpaceBooking {
 ///@identifier
 bookingID: long
 priceInEuro: double
}
interface ParkingSpaceBooking_interface {
 RequestResponse:
  priceInDollars(ParkingSpaceBooking)(double)
}
///@endCtx
```
Jolie

Structured LEMMA domain concepts like *ParkingSpaceBooking* and their data fields, e.g., *bookingID*, are directly translatable to corresponding Jolie **type**s.

To map LEMMA DDD information to Jolie, we use Jolie documentation comments (*///*) together with an *@*-sign followed by the DDD feature name, e.g., *entity* or *identifier*. This approach enables to preserve semantic DDD information for which Jolie currently does not support native language constructs. The comments serve as documentation to the programmer who will implement the API. In the future, we plan on leveraging these special comments also in automatic tools (see Section 5).

LEMMA operation signatures are expressible as **RequestResponse** operations within a Jolie **interface** for the LEMMA domain concept that defines the signatures. For example, we mapped the domain concept *ParkingSpaceBooking* and its operation signature *priceInDollars* to the Jolie interface *ParkingSpaceBooking_interface* with the operation *priceInDollars*.

The following listing shows the Jolie equivalent of the example LEMMA service model from Section 2.2.

```jolie
///@interface(Sample)
///@operationTypes(Sample.op)
type op_in {
 a : int
}
type op_out {
 c : int
}
type op_in_b {
 token:Token
 data : int
}
interface Sample {
 RequestResponse:
  op_in(op_in)(Token)
  op_out_d(Token)(int)
  op_out(Token)(op_out)
 OneWay:
  op_in_b(op_in_b)
}
```
Jolie

The operation *op* defined by the interface *Sample* contains asynchronous input and output parameters which do not have a direct equivalent in Jolie and thus need to be encoded. We propose to implement *op* into a series of request-response and one-way operations correlated

$$
\begin{array}{lcl}
[\![\textbf{context } id\ \{\overline{CT}\}]\!]^{\mathrm{C}} & = & ///module(id) \\
 & & \overline{[\![CT]\!]^{\mathrm{C}}} \\
[\![\textbf{structure } id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\ \overline{OPS}\}]\!]^{\mathrm{O}} & = & \overline{[///@STRF]}\ \textbf{interface } id\_interface\ \{\overline{[\![OPS]\!]^{\mathrm{O}}_{id}}\} \\
[\![\textbf{procedure } id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})]\!]^{\mathrm{O}}_{id_s} & = & \textbf{RequestResponse}:\ \overline{[///@OPSF]}\ id(id\_type)(id_s) \\
[\![\textbf{function } (S \mid id_r)\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})]\!]^{\mathrm{O}}_{id_s} & = & \textbf{RequestResponse}:\ \overline{[///@OPSF]}\ id(id\_type)((\overline{[\![S]\!]^{\mathrm{S}} \mid id_r})) \\
[\![\textbf{structure } id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\ \overline{OPS}\}]\!]^{\mathrm{C}} & = & \textbf{type } [\![\textbf{structure } id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\}]\!]^{\mathrm{S}} \\
 & & \overline{[\![OPS]\!]^{\mathrm{C}}_{id}}\ [\![\textbf{structure } id\ [\langle\overline{STRF}\rangle]\ \{\overline{OPS}\}]\!]^{\mathrm{O}}_{id} \\
[\![\textbf{procedure } id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})]\!]^{\mathrm{C}}_{id_s} & = & \textbf{type } id\_type:\ \textbf{void}\ \{self?:\ id_s\ \overline{[\![FLD]\!]^{\mathrm{S}}}\} \\
[\![\textbf{function } (id_r \mid S)\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})]\!]^{\mathrm{C}}_{id_s} & = & \textbf{type } id\_type:\ \textbf{void}\ \{self?:\ id_s\ \overline{[\![FLD]\!]^{\mathrm{S}}}\} \\
[\![\textbf{collection } id\ \{(S \mid id_r)\}]\!]^{\mathrm{C}} & = & \textbf{type } id:\ \textbf{void}\ \{[\![\textbf{collection } id\ \{(S \mid id_r)\}]\!]^{\mathrm{S}}\} \\
[\![\textbf{enum } id\ \{\overline{id}\}]\!]^{\mathrm{C}} & = & \textbf{type } [\![\textbf{enum } id\ \{\overline{id}\}]\!]^{\mathrm{S}} \\
[\![\textbf{structure } id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\}]\!]^{\mathrm{S}} & = & \overline{[///@STRF]}\ id:\ \textbf{void}\ \{\overline{[\![FLD]\!]^{\mathrm{S}}}\} \\
[\![S\ id\ [\langle\overline{FLDF}\rangle]]\!]^{\mathrm{S}} & = & \overline{[///@FLDF]}\ id:\ [\![S]\!]^{\mathrm{S}} \\
[\![id_r\ id\ [\langle\overline{FLDF}\rangle]]\!]^{\mathrm{S}} & = & \overline{[///@FLDF]}\ id:\ id_r \\
[\![\textbf{collection } id\ \{S\}]\!]^{\mathrm{S}} & = & id*:\ [\![S]\!]^{\mathrm{S}} \\
[\![\textbf{collection } id\ \{id_r\}]\!]^{\mathrm{S}} & = & id*:\ id_r \\
[\![\textbf{enum } id\ \{\overline{id}\}]\!]^{\mathrm{S}} & = & id:\ \textbf{string}(enum(\overline{"id"})) \\
[\![\textbf{int}]\!]^{\mathrm{S}} & = & \textbf{int} \\
[\![\textbf{unspecified}]\!]^{\mathrm{S}} & = & \textbf{undefined}
\end{array}
$$

Figure 4 Salient parts of the Jolie encoding for LEMMA's domain modelling concepts [9].

by a *Token*. The first is the request-response *op_in* which takes the synchronous inputs of *op* and returns the token to be used to invoke the operation to provide and retrieve the remaining parameters of the operation. The asynchronous input *b* is provided to the implementation of *op* by means of the one-way operation *op_in_b* which takes as argument the token provided by *op_in* and the value for *b*. The asynchronous output *d* is retrieved by invoking *op_out_d* with the given token and the synchronous output *c* by invoking *op_out*. This encoding leverages Jolie's behavioural language which allows the definition of sophisticated interactions among a client and a service within the same session.

3 Encoding LEMMA Domain and Service Models as Jolie APIs

In this section we extend the encoding from LEMMA Domain Models to Jolie APIs presented in [9] (Section 3.1) to support also Service Models (Section 3.2).

3.1 Encoding LEMMA Domain Models [9]

We recap the description of the encoding from LEMMA domain models to Jolie from [9].

The encoding of LEMMA domain models is reported in Figure 4 and consists of three encoders: the *context* encoder $[\![\cdot]\!]^{\mathrm{C}}$ walks through the structure of LEMMA domain models to generate Jolie APIs using the encoders for *operations* ($[\![\cdot]\!]^{\mathrm{O}}$) and for *structures* ($[\![\cdot]\!]^{\mathrm{S}}$), respectively.

The operations encoder $[\![\cdot]\!]^{O}$ generates Jolie interfaces based on **procedure**s and **function**s in the given models by translating structure-specific operations into Jolie operations. Because Jolie separates data from code that can operate on it (operations) the encoding needs to decouple **procedure**s and **function**s from their defining structures as illustrated in Section 2.3 by the mapping of the LEMMA domain concept *ParkingSpaceBooking* and its operation signature *priceInDollars* to the Jolie interface *ParkingSpaceBooking_interface* with the operation *priceInDollars*.

Given a structure X, we extend the signature of its **procedure**s with a parameter for representing the structure they act on and a return type X for the new state of the structure, essentially turning them into functions that transform the enclosing structure. For instance, we regard a procedure with signature $(Y \times \cdots \times Z)$ in X as a function with type $X \times Y \times \cdots \times Z \to X$. This approach is not new and can be found also in modern languages like Rust [17, 38] and Python [27]. The operation synthesised by the $[\![\cdot]\!]^{O}$ encoder accepts the *id_type* generated by the $[\![\cdot]\!]^{C}$ encoder that, in turn, has a *self* leaf carrying the enclosing data structure (id_s). The encoding of **function**s follows a similar path. Note that, when encoding *self* leaves, we do not impose the constraint of providing one such instance (represented by the **?** cardinality), but rather allow clients to provide it (and leave the check of its presence to the API implementer).

The main encoder $[\![\cdot]\!]^{C}$ and the structure encoder $[\![\cdot]\!]^{S}$ transform LEMMA types into Jolie types. **context**s translate into modules and, similarly to other DDD features, using pairs of ///@*beginCtx*(*context_name*) and ///@*endCtx* Joliedoc comment annotations. All the other constructs translate into **type**s and their subparts. When translating **procedure**s and **function**s, the two encoders follow the complementary scheme of $[\![\cdot]\!]^{O}$ and synthesise the types for the generated operations. The other rules are straightforward.

3.2 Encoding LEMMA Service Models

The encoding of LEMMA service models is reported in Figure 5. The microservice interface encoder $(\!|\cdot|\!)^{MI}$ translates the interfaces of a microservice into Jolie interfaces using the encoders $(\!|\cdot|\!)^{RR}$ and $(\!|\cdot|\!)^{OW}$ to translate its operations and $(\!|\cdot|\!)^{OT}$ to generate the types required by them. The encoding assumes that each microservice works within a single context and fixes a type Token for data used to correlate invocations to Jolie operations that implement the same LEMMA operation (as discussed in Section 2.3), e.g., a UUID.

The type encoder $(\!|\cdot|\!)^{OT}$ generates for each operation (i) a type collecting all its synchronous input parameters, (ii) a type for all its synchronous output parameters and (iii) at type for each of its asynchronous input parameters (to pair them with the token). Asynchronous output parameters do not require dedicated types.

The operation encoder $(\!|\cdot|\!)^{RR}$ generates the request-response operations required to implement a LEMMA operation. If the LEMMA operation has only synchronous parameters, then it can be directly implemented as a single Jolie operation (similarly to procedure and functions of LEMMA's DDML). If an operation has asynchronous parameters, then it is encoded using multiple operations: (i) one to accept the synchronous inputs which is invoked first and provides the token used by the subsequent operations; (ii) one for retrieving each asynchronous output given a token; and (iii) one for awaiting the end of the implemented operation and retrieve all the synchronous outputs. Asynchronous inputs are provided using one-way operations generated by the encoder $(\!|\cdot|\!)^{OW}$.

$$
\begin{aligned}
(\!|\textbf{microservice } ms\ \{\overline{IF}\}|\!)^{\mathrm{MI}} &= \overline{(\!|IF|\!)^{\mathrm{MI}}_{ms}} \\
(\!|\textbf{interface } if\ \{\overline{IOP}\}|\!)^{\mathrm{MI}}_{ms} &= \begin{array}{l} ///@interface(ms.if) \\ \overline{(\!|IOP|\!)^{\mathrm{OT}}_{if}} \\ \textbf{interface } if\ \{ \\ \quad \textbf{RequestResponse}: \overline{(\!|IOP|\!)^{\mathrm{RR}}_{if}} \\ \quad \textbf{OneWay}: \overline{(\!|IOP|\!)^{\mathrm{OW}}_{if}} \\ \} \end{array} \\
\left(\!\left| op\left(\begin{array}{c} \overline{\textbf{sync in } id_{SI} : id'_{SI}}, \\ \overline{\textbf{async in } id_{AI} : id'_{AI}}, \\ \overline{\textbf{async out } id_{AO} : id'_{AO}}, \\ \overline{\textbf{sync out } id_{SO} : id'_{SO}} \end{array}\right)\right| \!\right|^{\mathrm{OT}}_{if} &= \begin{array}{l} ///@operationTypes(if.op) \\ \textbf{type } op\_in\{\overline{id_{SI} : id'_{SI}}\} \\ \textbf{type } op\_out\{\overline{id_{SO} : id'_{SO}}\} \\ \overline{\textbf{type } op\_in\_id_{AI}\{\text{token} : \text{Token}, \text{data} : id'_{AI}\}} \end{array} \\
\left(\!\left| op\left(\begin{array}{c} \overline{\textbf{sync in } id_{SI} : id'_{SI}}, \\ \overline{\textbf{sync out } id_{SO} : id'_{SO}} \end{array}\right)\right| \!\right|^{\mathrm{RR}}_{if} &= \begin{array}{l} ///@operation(if.op) \\ op(op\_in)(op\_out) \end{array} \\
\left(\!\left| op\left(\begin{array}{c} \overline{\textbf{sync in } id_{SI} : id'_{SI}}, \\ \overline{\textbf{async in } id_{AI} : id'_{AI}}, \\ \overline{\textbf{async out } id_{AO} : id'_{AO}}, \\ \overline{\textbf{sync out } id_{SO} : id'_{SO}} \end{array}\right)\right| \!\right|^{\mathrm{RR}}_{if} &= \begin{array}{l} ///@operation(if.op) \\ op\_in(op\_in)(\text{Token}) \\ \overline{op\_out\_id_{AO}(\text{Token})(id_{AO})} \\ op\_out(\text{Token})(op\_out) \end{array} \\
\left(\!\left| op\left(\begin{array}{c} \overline{\textbf{sync in } id_{SI} : id'_{SI}}, \\ \overline{\textbf{async in } id_{AI} : id'_{AI}}, \\ \overline{\textbf{async out } id_{AO} : id'_{AO}}, \\ \overline{\textbf{sync out } id_{SO} : id'_{SO}} \end{array}\right)\right| \!\right|^{\mathrm{OW}}_{if} &= \begin{array}{l} ///@operation(if.op) \\ \overline{op\_out\_id_{AI}(op\_in\_id_{AI})} \end{array}
\end{aligned}
$$

Figure 5 Jolie encoding for LEMMA's service modelling concepts.

4 LEMMA2Jolie and Example

4.1 LEMMA2Jolie

We implement our extended encoding (cf. Section 3) by including the parsing of SML models and the new rules of the encoding presented here into LEMMA2Jolie. LEMMA2Jolie is a tool that transforms LEMMA models into Jolie code and that was initially presented in [9] where we have shown the feasibility of producing Jolie code from LEMMA domain models. The additions to LEMMA2Jolie described in this paper target LEMMA service models and are relatively straightforward. Specifically, we integrate the parsing of the SML models, which generate an in-memory object graph containing types and service information. Then, these run through an execution engine for templates, that transforms the in-memory representation into Jolie code that the tool outputs in file format. We provide the extended version of LEMMA2Jolie in a permanent repository on Software Heritage[2].

[2] `https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/jolie/lemma2jolie`

4.2 Example

We exemplify the encoding of LEMMA service and domain models in Jolie APIs based on the Food to Go (FTGO) case study by Richardson [33]. FTGO consists of six microservices that realise the backend of a web application for online food ordering from local restaurants. The microservices are responsible for accounting, consumer handling, delivery management, kitchen management, order handling, and restaurant organization. In the following, we focus on FTGO's Order microservice which is responsible for handling food orders. The following listing shows an excerpt of the LEMMA domain model for the Order microservice[3].

```
context API {
 structure CreateOrderRequest⟨valueObject⟩ {
  immutable long consumerId,
  immutable long restaurantId,
  immutable LineItems lineItems
 }

 structure LineItem {
  string menuItemId,
  int quantity
 }

 collection LineItems { LineItem i }

 structure CreateOrderResponse⟨valueObject⟩ {
  immutable long orderId
 }

 structure GetOrderResponse⟨valueObject⟩ {
  immutable long orderId,
  immutable string state,
  immutable double orderTotal
 }
}
```

LEMMA

The domain model defines the *API* bounded context. It comprises five domain concepts:

- *CreateOrderRequest:* This domain concept is a DDD **valueObject** and as such responsible for encapsulating data that is shared between software components [6]. The data fields of value objects are usually **immutable** because they receive a value exactly once for data transmission. The Order microservice enables clients to communicate information relevant to food order placing using the *CreateOrderRequest* concept.
- *LineItem:* This domain concept models a single line item of some food order. Therefore, it identified the item on the available menu and its ordered quantity.
- *LineItems:* The *LineItems* concept gathers all line items of a food order. *CreateOrderRequest* concept relies on it to communicate a consumer's order to the selected restaurant.
- *CreateOrderResponse:* The Order microservice replies to *CreateOrderRequesta* with this **valueObject**. It clusters the identifier of the created order.

[3] The complete model can be found at `https://archive.softwareheritage.org/browse/revision/d4447fe8bfcaa319e540ed89d160d8fe817e128f/?origin_url=https://github.com/jolie/lemma2jolie&path=sample-2.data&revision=d4447fe8bfcaa319e540ed89d160d8fe817e128f`

- *GetOrderResponse:* Using this domain concept, the Order microservice provides clients with information about the state of a certain order, e.g., "accepted'" or "cancelled", and its total costs.

The following listing shows an example LEMMA service model for the Order microservice[4].

```
import datatypes from ''sample−2.data'' as Domain

functional microservice org.example.OrderService {
 interface Orders {
  createOrder(
   sync in request : Domain::API.CreateOrderRequest,
   sync out response : Domain::API.CreateOrderResponse
  );

  getOrder(
   sync in orderId : long,
   sync out response : Domain::API.GetOrderResponse
  );

  monitorOrder(
   sync in orderId : long,
   async out response : Domain::API.GetOrderResponse
  );
 }
}
```

LEMMA

The model **import**s the above domain model including the *API* bounded context. LEMMA's import mechanism allows the composition of models for different viewpoints on a microservice architecture by enabling inter-model references [30]. The purpose of these references depends on the composed model kinds. For a service model that imports a domain model as shown in the listing, inter-model references support typing of microservice operation parameters with modelled domain concepts (see below). Import statements in LEMMA start with the **import** keyword followed by a keyword that identifies the kinds of imported elements, e.g., **datatypes** for domain concepts that are to be used as types for microservice operation parameters. After the **from** keyword, modellers specify the path to the imported model, i.e., "sample-2.data" in the listing[5], and a shorthand alias after the **as** keyword. Thus, in the service model, modellers can refer to the elements of the above domain model located in the file "sample-2.data" by the alias *Domain*.

After the **import** statement, the service model defines the **functional microservice** *org.example.OrderService*. In LEMMA's SML, microservices must have at least one qualifying naming level like "org.example" to allow the semantic clustering of services [30]). The *OrderService* consists of a single **interface** called *Orders* that gathers the following operations:

[4] The actual model can be found at `https://archive.softwareheritage.org/browse/content/sha1_git:267211533f271c8140166b3acc3729906baf3126/?origin_url=https://github.com/SeelabFhdo/lemma&path=examples/food-to-go/Order/Order.services`

[5] Please note that the file "sample-2.data" comprises the previous domain model and that the filename indicates the fact that the file clusters the LEMMA model code for the second usage example of LEMMA2Jolie in its repository at `https://archive.softwareheritage.org/browse/revision/d4447fe8bfcaa319e540ed89d160d8fe817e128f/?origin_url=https://github.com/jolie/lemma2jolie`.

- *createOrder:* This operation supports the creation of food orders in FTGO. To this end, it expects the **sync**hronously **in**coming parameter *request* whose type is the domain concept *CreateOrderRequest* imported from the *API* bounded context of the above domain model. The **sync**hronously **out**going parameter *response* then represents the result of food order creation as reified by the imported concept *CreateOrderResponse*.
- *getOrder:* This operation enables the retrieval of information about placed food orders. Therefore, it requires the identifier of an order and informs callers about its state by means of the *GetOrderResponse* concept from the imported domain model. As for *createOrder*, *getOrder* interacts with clients in a fully synchronous fashion.
- *monitorOrder:* Similar to *getOrder*, this operation provides callers with information about food orders. However, it expects continuous querying for this information leveraging the **async**hronously **out**going parameter *response*. Thus, the operation can be used, e.g., by mobile apps to display notifications about order state changes without the need for re-establishing synchronous HTTP connections[6].

Based on our encoding (Sect. 3), LEMMA2Jolie produces the following Jolie code from the LEMMA service model and the imported domain model.

```
///@beginCtx(API)
///@valueObject
type CreateOrderRequest {
 consumerId: long
 restaurantId: long
 lineItems: LineItems
}
type LineItem {
 menuItemId: string
 quantity: int
}
type LineItems {
 i*: LineItem
}
///@valueObject
type CreateOrderResponse {
 orderId: long
}
///@valueObject
type GetOrderResponse {
 orderId: long
 state: string
 orderTotal: double
}
///@endCtx
```

Jolie

The code between the Joliedoc comments *///@beginCtx(API)* and *///@endCtx(API)* represents the result of our encoding for LEMMA's domain modelling concepts (Sect. 2). The code following the Joliedoc comment *///@interface(org.example.OrderService.Orders)*, on the other hand, adheres to the novel encoding for LEMMA's service modelling concepts (Sect. 3).

[6] Note that *monitorOrder* is not part of the Order microservice's original interface [33]. Instead, we included this operation for illustration purposes.

That is, all synchronously typed parameters of the *createOrder*, *getOrder*, and *monitorOrder* receive a dedicated type per direction (*createOrder_in* and *createOrder_out*, *getOrder_in* and *getOrder_out*, and *monitorOrder_in*) given LEMMA's semantics of communicating synchronous data in coherent data transfer objects [4]. By contrast, each asynchronously typed parameter (*response* of *monitorOrder*) is mapped to a dedicated type (*monitorOrder_out_response*) to enable clients the sending and receipt of asynchronous data at arbitrary and decoupled points in operations' runtime.

```
///@interface(org.example.OrderService.Orders)
///@operationTypes(org.example.OrderService.Orders.createOrder)
type createOrder_in {
 request : CreateOrderRequest
}
type createOrder_out {
 response : CreateOrderResponse
}
///@operationTypes(org.example.OrderService.Orders.getOrder)
type getOrder_in {
 orderId : long
}
type getOrder_out {
 response : GetOrderResponse
}
///@operationTypes(org.example.OrderService.Orders.monitorOrder)
type monitorOrder_in {
 orderId : long
}
type monitorOrder_out_response {
 response : GetOrderResponse
}
interface org_example_OrderService_Orders {
 RequestResponse:
  createOrder(createOrder_in)(createOrder_out),
  getOrder(getOrder_in)(getOrder_out),
  monitorOrder_in(monitorOrder_in)(Token),
  monitorOrder_out_response(Token)(monitorOrder_out_response)
}
```

Jolie

As described in Sect. 3, interfaces of modelled LEMMA microservices are encoded as Jolie **interface**s. That is, from the *OrderService*'s *Orders* interface, LEMMA2Jolie produces the Jolie interface *org_example_OrderService_Orders* with four **RequestResponse** operations. *createOrder* and *getOrder* map to the eponymous, fully synchronous operations in the LEMMA service model for the *OrderService*. On the other hand, *monitorOrder_in* and monitorOrder_out_response reify different parts of the modelled *monitorOrder* operation. *monitorOrder_in* is the synchronous trigger for the execution of the *monitorOrder* logic. The result of the trigger's invocation is a *Token* that identifies the execution of the triggered *monitorOrder* instance. monitorOrder_out_response then requires the *Token* to provide clients with the instance's data that was modelled by the asynchronous *response* parameter.

5 Discussion, Related Work, and Conclusion

The use of MDE in both industrial and academic contexts, along with its effective support for developing intricate software systems, has led to the creation of numerous tools similar to LEMMA2Jolie [34, 16, 39, 37, 15]. These tools act as code generators within the conceptual framework of MDE [2] and generate artefacts for the engineering of MSA. They accomplish this task through models built using specific modelling languages.

Compared to LEMMA2Jolie, most of the related alternatives focus on Java as the target technology [34, 37, 15], rather than service-oriented programming languages. Contrarily, LEMMA2Jolie focuses on Jolie, which has been introduced to reduce the semantic gap between microservice concepts and implementation languages. Jolie's APIs are by design technology-agnostic and support their implementation with different transport protocols and technologies (e.g., Jolie, Java, JavaScript) [22, 20, 19]. Additionally, the modelling languages supported by the mentioned proposals and the resulting generated code only address single concerns in MSA engineering, such as domain modelling [34, 16] or service API implementation and provisioning [39, 37, 15]. In contrast, LEMMA's modelling languages provide an integrated solution for multi-concern modelling in MSA engineering by offering modelling languages for various microservice architecture viewpoints [30].

As described in Section 3 and Section 4, the encoding we specify and its implementation demonstrate the practicality of combining the LEMMA and Jolie ecosystems. There are several areas for future exploration, including extending the findings to other programming languages, examining the maturity of LEMMA2Jolie, formally proving the correctness of the encoding, and expanding the integration in different directions.

Interesting future work includes assessing the practical usefulness of LEMMA2Jolie. We mention a few possibilities, inspired also by best practices found in previous research on modelling languages [36, 31]. The first is to conduct controlled user experiments with practitioners, for example in order to evaluate how LEMMA2Jolie contributes to improving quality and productivity. Second, we could recruit practitioners to use LEMMA2Jolie, in order to evaluate their experience with using it and the result of their efforts. Finally, we could use LEMMA2Jolie to recreate existing microservice architectures written in Jolie and then compare the existing and obtained codebases in qualitative and quantitative terms [13, 11, 3]. Some of these architectures [11, 3] follow the API patterns recently identified in [39], and checking whether these patterns can be faithfully captured in LEMMA2Jolie could extend our knowledge on the connection between API patterns and MDE for microservices [31, 32].

To provide correctness guarantees of the encoding, we must first establish a formalisation of the semantics of both LEMMA's DDML and SML and Jolie APIs, and then prove that the encoding generates Jolie APIs that maintain the semantics of the input DDML and SML models. This effort is currently underway, as portions of Jolie have already been formalised [12, 21, 8, 22], and LEMMA implements context conditions [14] to restrict the proper formation of DDML models concerning their intended semantics [30].

We also intend to expand LEMMA2Jolie with capabilities for round-trip engineering (RTE). RTE accounts for the bidirectional synchronisation between models and generated code [26]. In the context of LEMMA2Jolie, RTE would further strengthen the collaboration between domain experts, who capture relevant application concepts in non-technical DDML models, and microservice developers, who adapt generated Jolie code to their needs and leverage RTE to reflect these changes back to the model-level for an efficient communication with domain experts.

The extension of LEMMA2Jolie presented in this paper forms the basis for future support of Access Point and Behaviour derivation from LEMMA models (Section 1). To this end, LEMMA2Jolie would have to consider further languages of LEMMA, e.g., the Technology Modeling Language [28] and Operation Modeling Language [30], in the translation towards Jolie code. Further along this direction, we plan to investigate the integration with [23] to automatically decompose the Jolie codebase generated by LEMMA2Jolie and synthesise suitable cloud deployment configurations.

References

1 David Ameller, Xavier Burgués, Oriol Collell, Dolors Costal, Xavier Franch, and Mike P. Papazoglou. Development of service-oriented architectures using model-driven development: A mapping study. *Information and Software Technology*, 62:42–66, 2015. Elsevier. doi:10.1016/J.INFSOF.2015.02.006.

2 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, 2017.

3 Luís Cruz-Filipe, Sofia Kostopoulou, Fabrizio Montesi, and Jonas Vistrup. μxl: Explainable lead generation with microservices and hypothetical answers. In Florian Rademacher and Jacopo Soldani, editors, *Service-Oriented and Cloud Computing - 10th IFIP WG 6.12 European Conference, ESOCC 2023, Larnaca, Cyprus, October 24-25, 2023, Proceedings*, Lecture Notes in Computer Science. Springer, 2023. doi:10.1007/978-3-031-46235-1_1.

4 Robert Daigneau. *Service Design Patterns*. Addison-Wesley, 2012.

5 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, 2017. doi:10.1007/978-3-319-67425-4_12.

6 Eric Evans. *Domain-Driven Design*. Addison-Wesley, 2004.

7 Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. of the 1991 Conf. on Programming Language Design and Implementation*, pages 268–277, 1991. doi:10.1145/113445.113468.

8 Saverio Giallorenzo, Fabrizio Montesi, and Maurizio Gabbrielli. Applied choreographies. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 21–40. Springer, 2018. doi:10.1007/978-3-319-92612-4_2.

9 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, and Florian Rademacher. Model-driven generation of microservice interfaces: From LEMMA domain models to jolie apis. In Maurice H. ter Beek and Marjan Sirjani, editors, *Coordination Models and Languages - 24th IFIP WG 6.1 International Conference, COORDINATION 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, volume 13271 of *Lecture Notes in Computer Science*, pages 223–240. Springer, 2022. doi:10.1007/978-3-031-08143-9_13.

10 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh. Jolie and LEMMA: Model-driven engineering and programming languages meet on microservices. In *Coordination Models and Languages*, pages 276–284. Springer, 2021. doi:10.1007/978-3-030-78142-2_17.

11 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Narongrit Unwerawattana. Jot: A jolie framework for testing microservices. In Sung-Shik Jongmans and Antónia Lopes, editors, *Coordination Models and Languages*, volume 13908 of *Lecture Notes in Computer Science*, pages 172–191. Springer, 2023. doi:10.1007/978-3-031-35361-1_10.

12 Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. Sock: a calculus for service oriented computing. In *International Conference on Service-Oriented Computing*, pages 327–338. Springer, 2006. doi:10.1007/11948148_27.

13 Claudio Guidi and Balint Maschio. A jolie based platform for speeding-up the digitalization of system integration processes, 2019. URL: https://www.conf-micro.services/2019/papers/Microservices_2019_paper_6.pdf.

14 David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, oct 2004. IEEE. doi:10.1109/MC.2004.172.

15 JHipster. Jhipster domain language (jdl), 2022-14-02. URL: https://www.jhipster.tech/jdl.

16 Stefan Kapferer and Olaf Zimmermann. Domain-specific language and tools for strategic Domain-driven Design, context mapping and bounded context modeling. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD*, pages 299–306. INSTICC, SciTePress, 2020. doi:10.5220/0008910502990306.

17 Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

18 Robin Milner. The tower of informatic models. *From semantics to Computer Science*, 2009.

19 Fabrizio Montesi. Jolie: a Service-oriented Programming Language. Master's thesis, University of Bologna, Department of Computer Science, 2010. URL: http://amslaurea.cib.unibo.it/1226/.

20 Fabrizio Montesi. Process-aware web programming with Jolie. *Sci. Comput. Program.*, 130:69–96, 2016. doi:10.1016/J.SCICO.2016.05.002.

21 Fabrizio Montesi and Marco Carbone. Programming services with correlation sets. In Gerti Kappel, Zakaria Maamar, and Hamid R. Motahari Nezhad, editors, *Service-Oriented Computing - 9th International Conference, ICSOC 2011, Paphos, Cyprus, December 5-8, 2011 Proceedings*, volume 7084 of *Lecture Notes in Computer Science*, pages 125–141. Springer, 2011. doi:10.1007/978-3-642-25535-9_9.

22 Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with Jolie. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 81–107. Springer, 2014. doi:10.1007/978-1-4614-7518-7_4.

23 Fabrizio Montesi, Marco Peressotti, and Valentino Picotti. Sliceable monolith: Monolith first, microservices later. In Barbara Carminati, Carl K. Chang, Ernesto Daminai, Shuigung Deng, Wei Tan, Zhongjie Wang, Robert Ward, and Jia Zhang, editors, *IEEE International Conference on Services Computing, SCC 2021, Chicago, IL, USA, September 5-10, 2021*, pages 364–366. IEEE, 2021. doi:10.1109/SCC53864.2021.00050.

24 Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015. URL: https://www.worldcat.org/oclc/904463848.

25 Andy Oram. *Ballerina: A Language for Network-Distributed Applications*. O'Reilly, 2019.

26 Richard F. Paige, Nicholas Matragkas, and Louis M. Rose. Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272–280, 2016. doi:10.1016/J.JSS.2015.08.047.

27 Python Software Foundation. The python language reference, 2021.

28 Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Aspect-oriented modeling of technology heterogeneity in Microservice Architecture. In *2019 IEEE Int. Conf. on Software Architecture (ICSA)*, pages 21–30. IEEE, 2019. doi:10.1109/ICSA.2019.00011.

29 Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Deriving microservice code from underspecified domain models using DevOps-enabled modeling languages and model transformations. In *2020 46th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA)*, pages 229–236. IEEE, 2020. doi:10.1109/SEAA51224.2020.00047.

30 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. doi:10.1007/978-3-030-31646-4_7.

31 Florian Rademacher, Jonas Sorgalla, Philip Wizenty, and Simon Trebbau. Towards holistic modeling of microservice architectures using LEMMA. In *Companion Proc. of the 15th Europ.*

Conf. on Software Architecture. CEUR-WS, 2021. URL: https://ceur-ws.org/Vol-2978/mde4sa-paper2.pdf.

32 Florian Rademacher, Jonas Sorgalla, Philip Wizenty, and Simon Trebbau. Towards an extensible approach for generative microservice development and deployment using lemma. In Patrizia Scandurra, Matthias Galster, Raffaela Mirandola, and Danny Weyns, editors, *Software Architecture*, pages 257–280, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-15116-3_12.

33 Chris Richardson. *Microservices Patterns*. Manning Publications, first edition, 2019.

34 Sculptor Team. Sculptor–generating Java code from DDD-inspired textual DSL, 2022-14-02. URL: https://www.sculptorgenerator.org.

35 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. doi:10.1016/J.JSS.2018.09.082.

36 Jonas Sorgalla, Philip Wizenty, Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Applying model-driven engineering to stimulate the adoption of devops processes in small and medium-sized development organizations. *SN Computer Science*, 2(6):459, 2021. doi:10.1007/S42979-021-00825-Z.

37 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. doi:10.1080/17517575.2018.1460766.

38 The Rust Foundation. The rust reference, 2021.

39 Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley, Boston, 2023.