Model-Driven Generation of Microservice Interfaces: From LEMMA Domain Models to Jolie APIs

Saverio Giallorenzo[1,2], Fabrizio Montesi[3], Marco Peressotti[3], and Florian Rademacher[4(✉)]

[1] Università di Bologna, Bologna, Italy
saverio.giallorenzo2@unibo.it
[2] INRIA, Sophia Antipolis, France
[3] University of Southern Denmark, Odense, Denmark
{fmontesi,peressotti}@imada.sdu.dk
[4] University of Applied Sciences and Arts Dortmund, Dortmund, Germany
florian.rademacher@fh-dortmund.de

Abstract. We formally define and implement a translation from domain models in the LEMMA modelling framework to microservice APIs in the Jolie programming language. Our tool enables a software development process whereby microservice architectures can first be designed with the leading method of Domain-Driven Design (DDD), and then corresponding data types and service interfaces (APIs) in Jolie are automatically generated. Developers can extend and use these APIs as guides in order to produce compliant implementations. Our tool thus contributes to enhancing productivity and improving the design adherence of microservices.

1 Introduction

Microservice Architecture (MSA) is one of the current leading patterns in distributed software architectures [22]. While widely adopted, MSA comes with specific challenges regarding architecture design, development, and operation [5,29]. To cope with this complexity, researchers in software engineering and programming languages started proposing linguistic approaches to MSA: language frameworks that ease the design and development of MSAs with high-level constructs that make microservice concerns in the two different stages syntactically manifest.

Concerning development, Ballerina and Jolie are examples of programming languages [21,23] with new linguistic abstractions for effectively programming the configuration and coordination of microservices. Concerning design, Model-Driven Engineering (MDE) [3] has gained relevance as a method for the specification of service architectures [1], crystallised in MDE-for-MSA modelling

Work partially supported by Independent Research Fund Denmark, grant no. 0135-00219.

languages such as MicroBuilder, MDSL, LEMMA, and JHipster [15,16,26,32]. Jolie's abstractions have been found to offer a productivity boost in industry [13]. LEMMA provides linguistic support for the application of concepts from Domain-Driven Design [6,26], and has been validated in real-world use cases [27,30].

Recently, it has been observed that the metamodels of LEMMA's modelling languages and the Jolie programming language have enough contact points to consider their integration [11]. In the long term, such an integration could bring (quoting from [11])

> "*an ecosystem that coherently combines MDE and programming abstractions to offer a tower of abstractions* [19] *that supports a step-by-step refinement process from the abstract specification of a microservice architecture to its implementation*".

The aim is to provide a toolchain that enables people to apply MDE to the design of microservices in LEMMA, and then seamlessly switch to a programming language with dedicated support for microservices like Jolie in order to develop an implementation of the design. To this end, three important parts of the metamodels of LEMMA and Jolie need to be covered and integrated [11]:

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 the API;
3. *Behaviours*, defining the internal business logic of a microservice.

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 contribute a formal encoding between a meaningful subset of LEMMA's Domain Data Modelling Language (DDML) and Jolie types and interfaces. This encoding enables systematic translation of LEMMA domain models, which, following Domain-Driven Design (DDD) [6] principles, capture domain-specific types including operation signatures, to Jolie APIs. Our second contribution, LEMMA2Jolie implements our encoding as a code generator that allows automatic translation of LEMMA domain models to Jolie APIs. Specifically, LEMMA2Jolie not only shows the encoding's feasibility and practicability, but also constitutes a crucial contribution towards improving the adoption of DDD in microservice design, which in practice is often perceived complex given the lack of formal guidelines on how to map DDD domain models to microservice code [2]. We have evaluated LEMMA2Jolie in the context of a nontrivial microservice architecture that had previously been used to validate LEMMA [27], which covers all the aspects of the formal encoding. The generated Jolie code is as expected, in the sense that it is faithful to the formal encoding and the model defined in LEMMA. We use snippets of this code to exemplify our method throughout the paper.

In general, LEMMA2Jolie is a concrete proof that the work started in [11] constitutes a bridge between the two communities of programming language

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

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

and MDE research, converging on linguistic approaches to MSA—for instance, one can take our insights and apply them to integrate other MSA modelling and programming languages.

The remainder of the paper is organised as follows. Section 2 introduces and exemplifies the encoding between LEMMA's DDML and Jolie APIs. Section 3 describes the architecture and implementation of LEMMA2Jolie. Section 4 presents future work and concludes the paper.

2 Encoding LEMMA Domain Modelling Concepts in Jolie

This section describes and exemplifies domain modelling with LEMMA (cf. Sect. 2.1), and the development of types and interfaces with Jolie (cf. Sect. 2.2). Next, it reports a formal encoding from LEMMA domain models to Jolie APIs and illustrates its application (cf. Sects. 2.3 and 2.4).

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. Figure 1 shows the core rules of the DDML grammar[1].

The DDML follows DDD to capture domain concepts. DDD's Bounded Context pattern [6] is crucial in MSA design as it makes the boundaries of coherent domain concepts explicit, thereby defining their scope and applicability [22]. A

[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.

LEMMA domain model defines named bounded **context**s (rule CTX in Fig. 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). The type of a data field is either a complex type from the same bounded context (*id*) or a built-in primitive type, e.g., **int** or **string** (S). The **unspecified** keyword enables continuous domain exploration according to DDD [6]. That is, it supports the construction of underspecified models and their subsequent refinement as one gains new domain knowledge [25]. Next to fields, **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.

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 Fig. 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. For compactness, we defer the detailed presentation of the considered DDD features to Sect. 2.4, when discussing their relationship with our encoding to Jolie.

The DDML also enables the modelling of **collection**s (rule COL in Fig. 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 [27].

```
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 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 2 shows a simplified variant of the grammar of Jolie APIs, taken from [21] 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,

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

Fig. 2. Simplified syntax of Jolie APIs (types and interfaces)

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. Sect. 2.3). 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 [9]. 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 Fig. 2, the following listing shows the Jolie equivalent of the example LEMMA domain model from Sect. 2.1.

```
///@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. It is followed by (i) the string *beginCtx* and the parenthesised name of a modelled bounded context, e.g., *BookingManagement*; (ii) the DDD feature name, e.g., *entity*; or (iii) the string *endCtx* to conclude a bounded context. 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 Sects. 2.4 and 4).

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 *ParkingSpace-Booking_interface* with the operation *priceInDollars*.

2.3 Encoding LEMMA Domain Models as Jolie APIs

In the following, we report an encoding from LEMMA domain models to Jolie APIs that formalises and extends the mapping exemplified in Sect. 2.2. Figure 3 shows the encoding.

The encoding is split in three encoders: the *main* encoder $[\![\cdot]\!]$ walks through the structure of LEMMA domain models to generate Jolie APIs using the encoders for *operations* ($(\!(\cdot)\!)$) and for *structures* ($\ulcorner\!\ulcorner \cdot \urcorner\!\urcorner$), respectively.

The operations encoder $(\!(\cdot)\!)$ generates Jolie interfaces based on **procedure**s and **function**s in the given models by translating structure-specific operations into Jolie operations. This translation requires some care. On one hand, LEMMA's **procedure**s and **function**s recall object methods in the sense that they operate on data stored in their defining structure. On the other hand, Jolie separates data from code that can operate on it (operations). Therefore, the encoding needs to decouple **procedure**s and **function**s from their defining structures as illustrated in Sect. 2.2 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 [18,33] and Python [24]. The operation synthesised by the $(\!(\cdot)\!)$ encoder accepts the *id_type* generated by the $[\![\cdot]\!]$ 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 $[\![\cdot]\!]$ and the structure encoder $\ulcorner\!\ulcorner \cdot \urcorner\!\urcorner$ transform LEMMA types into Jolie types. **context**s translate into 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 $\llparenthesis\,\cdot\,\rrparenthesis$ and synthesise the types for the generated operations. The other rules are straightforward.

$$
\begin{aligned}
&[\![\textbf{context}\ id\ \textbf{\{}\overline{CT}\textbf{\}}]\!] && = ///@beginCtx(id)\\
& && \quad \overline{[\![CT]\!]}\\
& && \quad ///@endCtx\\
&\llparenthesis\textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \textbf{\{}\overline{FLD}\ \overline{OPS}\textbf{\}}\rrparenthesis && = [\overline{///@STRF}]\ \textbf{interface}\ id\_interface\ \textbf{\{}\overline{\llparenthesis OPS\rrparenthesis_{id}}\textbf{\}}\\
&\llparenthesis\textbf{procedure}\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})\rrparenthesis_{id_s} && = \textbf{RequestResponse}:\ [\overline{///@OPSF}]\ id(id\_type)(id_s)\\
&\llparenthesis\textbf{function}\ (S \mid id_r)\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})\rrparenthesis_{id_s} && = \textbf{RequestResponse}:\ [\overline{///@OPSF}]\ id(id\_type)((\lceil\!\lceil S\rceil\!\rceil \mid id_r))\\
&[\![\textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \textbf{\{}\overline{FLD}\ \overline{OPS}\textbf{\}}]\!] && = \textbf{type}\ \lceil\!\lceil\textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \textbf{\{}\overline{FLD}\textbf{\}}\rceil\!\rceil\\
& && \quad \overline{[\![OPS]\!]_{id}}\ \llparenthesis\textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \textbf{\{}\overline{OPS}\textbf{\}}\rrparenthesis_{id}\\
&[\![\textbf{procedure}\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})]\!]_{id_s} && = \textbf{type}\ id\_type:\ \textbf{void}\ \textbf{\{}self?:\ id_s\ \overline{\lceil\!\lceil FLD\rceil\!\rceil}\textbf{\}}\\
&[\![\textbf{function}\ (id_r \mid S)\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})]\!]_{id_s} && = \textbf{type}\ id\_type:\ \textbf{void}\ \textbf{\{}self?:\ id_s\ \overline{\lceil\!\lceil FLD\rceil\!\rceil}\textbf{\}}\\
&[\![\textbf{collection}\ id\ \textbf{\{}(S \mid id_r)\textbf{\}}]\!] && = \textbf{type}\ id:\ \textbf{void}\ \textbf{\{}\lceil\!\lceil\textbf{collection}\ id\ \textbf{\{}(S \mid id_r)\textbf{\}}\rceil\!\rceil\textbf{\}}\\
&[\![\textbf{enum}\ id\ \textbf{\{}\overline{id}\textbf{\}}]\!] && = \textbf{type}\ \lceil\!\lceil\textbf{enum}\ id\ \textbf{\{}\overline{id}\textbf{\}}\rceil\!\rceil\\
&\lceil\!\lceil\textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \textbf{\{}\overline{FLD}\textbf{\}}\rceil\!\rceil && = [\overline{///@STRF}]\ id:\ \textbf{void}\ \textbf{\{}\overline{\lceil\!\lceil FLD\rceil\!\rceil}\textbf{\}}\\
&\lceil\!\lceil S\ id\ [\langle\overline{FLDF}\rangle]\rceil\!\rceil && = [\overline{///@FLDF}]\ id:\ \lceil\!\lceil S\rceil\!\rceil\\
&\lceil\!\lceil id_r\ id\ [\langle\overline{FLDF}\rangle]\rceil\!\rceil && = [\overline{///@FLDF}]\ id:\ id_r\\
&\lceil\!\lceil\textbf{collection}\ id\ \textbf{\{}S\textbf{\}}\rceil\!\rceil && = id*:\ \lceil\!\lceil S\rceil\!\rceil\\
&\lceil\!\lceil\textbf{collection}\ id\ \textbf{\{}id_r\textbf{\}}\rceil\!\rceil && = id*:\ id_r\\
&\lceil\!\lceil\textbf{enum}\ id\ \textbf{\{}\overline{id}\textbf{\}}\rceil\!\rceil && = id:\ \textbf{string}(enum(\overline{``id"}))\\
&\lceil\!\lceil\textbf{int}\rceil\!\rceil && = \textbf{int}\\
&\lceil\!\lceil\textbf{unspecified}\rceil\!\rceil && = \textbf{undefined}
\end{aligned}
$$

Fig. 3. Salient parts of the Jolie encoding for LEMMA's domain modelling concepts.

2.4 Applying the Encoding

This subsection illustrates the application of the encoding from Sect. 2.3 using the Booking Management Microservice (BMM) of a microservice-based Park and Charge Platform (PACP) modelled with LEMMA [27]. The PACP enables drivers of electric vehicles to offer their charging stations for use by others. Its BMM manages the corresponding bookings based on domain concepts that were designed following DDD principles [6] and expressed in LEMMA's DDML.

In the following paragraphs, unless indicated, the encoded Jolie APIs respect the DDD constraints expressed by the considered features.

Aggregate and Part. In DDD, aggregates prescribe object graphs, whose parts must maintain a consistent state [6]. Aggregates are always loaded from and stored to a database in a consistent state and within one transaction. A

DDD aggregate consists of at least an entity or value object (see below). The following left listing shows the *PSB* aggregate in the LEMMA domain model for the BMM.

```
structure PSB
⟨ aggregate ⟩ {
 TimeSlot timeSlot ⟨ part ⟩,
 double priceInEuro
}
structure TimeSlot { . . . }
                                    LEMMA
```

```
///@aggregate
type PSB {
 ///@part
 timeSlot: TimeSlot
 priceInEuro: double
}
type TimeSlot { . . . }             Jolie
```

PSB is a **structure**d domain concept with the **aggregate** feature (cf. Sect. 2.1) and it clusters the field *timeSlot*, which has a structured type and is a **part** of the aggregate. Notice that for this domain model, LEMMA's DDML would emit warnings, because (i) a DDD aggregate must specify a root entity; and (ii) a part should either be an entity or value object [6]. We extend the *PSB* aggregate below to gradually fix these issues, thereby explaining the semantics of DDD entities and value objects.

In the Jolie encoding (on the right), we have as many **type** definitions as we have **structure**s in the LEMMA model.

Entity and Identifier. Instances of DDD entities are distinguishable by a domain-specific identity [6], e.g., a unique ID. The following left listing extends the *PSB* aggregate with the **entity** feature and an **identifier** field.

```
structure PSB
⟨ aggregate, entity ⟩ {
 long bookingID ⟨ identifier ⟩,
 TimeSlot timeSlot ⟨ part ⟩,
 double priceInEuro
}
                                    LEMMA
```

```
///@aggregate
///@entity
type PSB {
 ///@identifier
 bookingID: long
 ///@part
 timeSlot: TimeSlot
 priceInEuro: double
}                                   Jolie
```

LEMMA's DDML requires the **entity** feature on an aggregate to signal that its fields prescribe the structure of its root entity. The **identifier** feature can be used to mark those fields that determine the identity of an entity instance. In the example above, the value of *bookingID* is marked to identify *PSB*s.

The Jolie encoding of **entity** and **identifier** fields is straightforward.

Next to fields, DDML supports the **identifier** feature on a single **function** of an entity to enable identity calculation at runtime. To illustrate this approach, the following listing models the *bookingID* of the *PSB* root entity as a function.

```
structure PSB ⟨ entity ⟩ {
 function long bookingID
 ⟨ identifier ⟩ (),
 ...
}
                                   LEMMA
```

```
///@entity
type PSB { ... }
type bookingID_type { self?: PSB }
interface PSB_interface {
 RequestResponse:
  ///@identifier
  bookingID(bookingID_type)(long)
}
                                   Jolie
```

Following our encoding (cf. Sect. 2.3), we create the Jolie **type** *bookingID_type* for the *bookingID* **identifier function**. The **type**'s *self* leaf enables implementers to access the fields of the *PSB* and define how to compute the identifier.

Factory. DDD factories make the creation of objects with complex consistency requirements explicit [6]. LEMMA's DDML considers factories to constitute **function**s that return instances of aggregates, entities, or value objects. The following left listing illustrates the usage of factories by specifying the **factory** function *create* as part of the *PSB* aggregate. This function shall create *PSB* instances for a given time slot *timeSlot* and a *priceInEuro*.

```
structure PSB ⟨ ... ⟩{
 TimeSlot timeSlot,
 double priceInEuro,
 function PSB create⟨factory⟩(
  TimeSlot timeSlot,
  double priceInEuro
 )


}
                                   LEMMA
```

```
type PSB { ... }
///@factory
type create_type {
 timeSlot: TimeSlot
 priceInEuro: double
}
interface PSBFactory_interface {
 RequestResponse:
  create(create_type)(PSB)
}
                                   Jolie
```

As opposed to the encoding for LEMMA **identifier function**s (see above), we do not encode a *self* leaf in Jolie **type**s such as *create_type* for LEMMA **factory function**s. Since the semantics of factories is that of generating an instance of the enclosing **structure**, it would not make sense to pass to it one of those instances as a *self* leaf. Consequently, we could include a rule in Fig. 3 which avoids the generation of said *self* leaf (this is more an issue of minimality of the generated code, since we set the leaf as optional (**?**)). Additionally, we can enforce a check on Jolie operations like *create* following immediately after *///@factory*-commented **type**s by making sure their input **type**s do not contain the produced **type**, e.g., *PSB*. Complementary, we can also check that the response type of Jolie-encoded factory operations coincides with the produced **type**.

Specification and Validator. DDD specifications are domain concepts that make business rules, policies, or consistency specifications for aggregates explicit [6]. A specification must comprise one or more validators, which are functions with a boolean return type that reify the specification's predicates.

LEMMA's DDML provides the features **specification** and **validator** to mark structures as specifications and identify their validators. Below we extend the BMM's domain model with the *BookingExpiration* specification. Its *isExpired* validator returns **true** if a parking space booking *PSB* instance has expired.

```
structure PSB ⟨ ... ⟩ { ... }
structure BookingExpiration
⟨ specification ⟩ {
 function boolean isExpired
 ⟨ validator ⟩ (PSB p)
}
                                   LEMMA
```

```
type PSB { ... }
///@specification
type isExpired_type { p: PSB }
interface BookingExpiration_interface {
 RequestResponse:
  ///@validator
  isExpired(isExpired_type)(bool)
}
                                   Jolie
```

Since the specification is a field-less **structure**, we do not create a corresponding **type** *BookingExpiration* as it would be empty. Instead, and as per our encoding (cf. Sect. 2.3), we create the ///@*specification*-annotated **type** *isExpired_type* for the *isExpired* **validator** within the **interface** *BookingExpiration_interface*. From the point of view of the consistency of the annotations, following the namespace convention from Fig. 3, we can check that the ///@*validator* actually accepts the related **structure**. To do this, we follow the "breadcrumbs" left by our encoders. First, we find a ///@*validator*-commented **RequestResponse** (e.g., *isExpired*) and we make sure its response type is **bool**. Then, we follow the request type (e.g., *isExpired_type*) to make sure that: *i*) the ///@*validator* has an associated ///@*specification* (e.g., *isExpired_type*) **type** and *ii*) the **type** has one leaf, which is the **structure** the **validator** validates.

Value Object. As opposed to entities, DDD value objects cluster data and logic, which are not dependent on objects' identity [6]. Thus, value objects serve as DTOs for data exchange between microservices [22]. In asynchronous communication scenarios, value objects can model *domain events* emitted by a bounded context during runtime [7]. For example, all PACP microservices interact with each other via domain events [27].

LEMMA's DDML supports the **valueObject** and **domainEvent** features to mark structured domain concepts as value objects and possibly as domain events. The following left listing illustrates the usage of the **valueObject** feature.

```
context BookingManagement {
 structure PSB ⟨ . . . ⟩ {
  TimeSlot timeSlot,
  double priceInEuro
 }
 structure PSB_VO⟨ valueObject ⟩ {
  TimeSlot timeSlot,
  double price,
  string currency
 }
 structure TimeSlot⟨ valueObject ⟩ {
  . . .
 }
}
                                   LEMMA
```

```
///@beginCtx(BookingManagement)
type PSB {
 timeSlot: TimeSlot
 priceInEuro: double
}
///@valueObject
type PSB_VO {
 timeSlot: TimeSlot
 price: double
 currency: string
}
///@valueObject
type TimeSlot { . . . }
///@endCtx
                                   Jolie
```

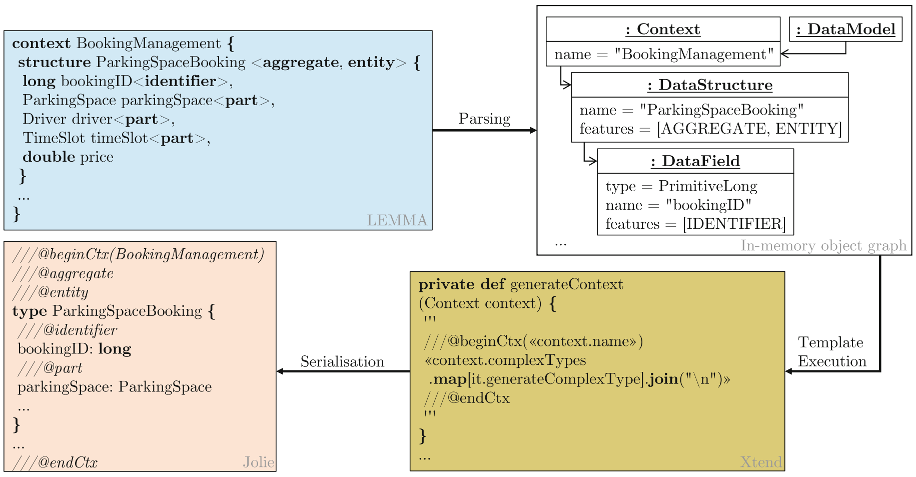
Above, we extend the BMM's domain model with the *PSB_VO* value object: a DTO for the *PSB* aggregate that slightly changes it type to make its representation more general. Namely, *PSB_VO* separates the *currency* from the value of the *price*. The *timeSlot* field remains the same, but we make sure it is also a **valueObject**.

The LEMMA domain model also shows the definition of bounded **context**s in the DDML. All three structures *PSB*, *PSB_VO*, and *TimeSlot* are enclosed by the *BookingManagement* **context** on which the BMM operates exclusively.

The encoding from LEMMA to Jolie follows Fig. 3 without exceptions. Notice, in particular, the "opening" ///@*beginCtx*(*BookingManagement*) and "closing" ///@*endCtx* comments for the context. With those comments, we are declaring that the types (and interfaces) that appear between them belong to the context *BookingManagement*. In LEMMA, contexts indicate a boundary within which (complex) types belonging in the same context can co-exist and interact (e.g., by being part of the inputs and output of **procedure**s and **function**s). Then, as seen above, **valueObject**s exist to allow data to cross boundaries, by defining data types (e.g., **structure**s) purposed to act as DTOs.

While the encoding from LEMMA's DDML ensures that, at the API level, the coherence defined by **context**s and **valueObject**s is preserved, e.g., there exists no **type** with leaves whose types belong in different contexts nor **interface**s belonging in a context that accept types from another context, unless ///@*valueObject*s. However, behaviours that users write can arbitrarily combine data structures and operators and possibly break the coherence of contexts.

In the future, we would like to devise static checks able to enforce the coherence of LEMMA's DDML contexts also in behaviours, e.g., by tracing the contexts in which values belong—from the types of the operations that generated them, via receptions—and prohibit mixing values that belong in different contexts (e.g., by forbidding to use them with operations belonging in different contexts, although their types might be compatible). This static check would also handle the exception of values whose types are annotated as ///@*valueObject*s, which are the only ones allowed to be used in a mixed way (i.e., in operations that take or produce ///@*valueObject*-annotated types.).



```
context BookingManagement {
 structure ParkingSpaceBooking <aggregate, entity> {
  long bookingID<identifier>,
  ParkingSpace parkingSpace<part>,
  Driver driver<part>,
  TimeSlot timeSlot<part>,
  double price
 }
 ...
}
```

```
: Context
name = "BookingManagement"

: DataModel

: DataStructure
name = "ParkingSpaceBooking"
features = [AGGREGATE, ENTITY]

: DataField
type = PrimitiveLong
name = "bookingID"
features = [IDENTIFIER]
...
```

```
private def generateContext
(Context context) {
 '''
 ///@beginCtx(«context.name»)
 «context.complexTypes
  .map[it.generateComplexType].join("\n")»
 ///@endCtx
 '''
}
...
```

```
///@beginCtx(BookingManagement)
///@aggregate
///@entity
type ParkingSpaceBooking {
 ///@identifier
 bookingID: long
 ///@part
 parkingSpace: ParkingSpace
 ...
}
...
///@endCtx
```

Fig. 4. LEMMA2Jolie phases to generate Jolie APIs from LEMMA domain models.

Additional Features. Our encoding captures the **repository** and **closure** features of LEMMA's DDML (cf. Fig. 1). Checks regarding the **sideEffectFree** feature follow the same considerations of the **valueObject** feature: we need to inspect a service behaviour to make sure its does not modify the values obtained from ///@*sideEffectFree*-commented operations. **service**s are a generalisation of the **specification** feature, where we have a structure that contains only **functions** and **procedure**s—LEMMA further specialises **service**s into, **domainServices**, **infrastructureService**s, **applicationService**s, which are subject to future works.

3 LEMMA2Jolie: A Code Generator to Derive Jolie APIs from LEMMA Domain Models

This section presents our LEMMA2Jolie tool, which implements the encoding presented in Sect. 2. In the sense of MDE, LEMMA2Jolie is a *model-to-text transformation* [3] that generates Jolie APIs from LEMMA domain models.

Architecture. As depicted in Fig. 4, LEMMA2Jolie consists of three phases to derive Jolie APIs from LEMMA domain models.

In the Parsing phase, LEMMA2Jolie instantiates an in-memory object graph conforming to the metamodel of the DDML [26] from a given LEMMA domain model. The object graph allows systematic traversal of the model elements to map them to the corresponding Jolie code (cf. Sect. 2.3) in the following Template Execution phase. As the phase name indicates, LEMMA2Jolie relies on *template-based code generation* [3] to transform in-memory LEMMA domain models to

Jolie. That is, we prescribe the target blocks of a Jolie program as strings involving static Jolie statements and dynamic variables which are evaluated at runtime to complement the prescribed target blocks with context-dependent information, e.g., the name of a bounded context in a specific LEMMA domain model. After template execution, the Serialisation phase stores the evaluated templates to physical files with valid Jolie code.

Implementation Overview. We implemented `LEMMA2Jolie` in Xtend[2], which is a Java dialect that integrates a sophisticated templating language (see below). Furthermore, `LEMMA2Jolie` relies on LEMMA's Java-based Model Processing Framework[3], which aims to facilitate the development of model processors such as code generators. To this end, the framework provides built-in support for parsing models constructed with languages that are based on the Eclipse Modelling Framework [31]—as is the case for all LEMMA modelling languages including the DDML. Additionally, the framework prescribes a certain workflow for model processing and enables implementers to integrate with it using Java annotations.

Listing 1 describes the implementation of `LEMMA2Jolie`'s code generation module which integrates with the Code Generation phase of LEMMA's Model Processing Framework. The module is responsible for template execution and the eventual serialisation of Jolie code.

A code generation module is a Java class with the `@CodeGenerationModule` annotation that extends the `AbstractCodeGenerationModule` class (Lines 1 and 2). LEMMA's Model Processing Framework delegates to a code generation module after it parsed an input model in the modelling language supported by the module. To specify the supported language, a code generation module overrides the inherited `getLanguageNamespace` method to return the language's namespace, which in the case of `LEMMA2Jolie` is that of LEMMA's DDML (Line 4).

The entrypoint for code generation is the `execute` method of a code generation module. It can access the in-memory object graph of a parsed model via the `resource` attribute. Lines 6 to 13 show the `execute` method of `LEMMA2Jolie`'s code generation module. In Line 7, we retrieve the root of the model as an instance of the `DataModel` concept of the DDML's metamodel (cf. Fig. 4). Next, we call the template method `generateContext` (see Listing 1) for each parsed `Context` instance and gather the generated Jolie code in the `generatedContexts` variable (Line 8). In Lines 9 and 10, we determine the path of the generated Jolie file, which will be created in the given target folder and with the same base name as the input LEMMA domain model but with Jolie's extension "ol". Line 11 triggers the serialisation of the generated Jolie code via the inherited `withCharset` method.

Lines 15 to 19 show the implementation of the template method `generateContext`. It expects an instance of the metamodel concept `Context` as input

[2] https://www.eclipse.org/xtend

[3] https://github.com/SeelabFhdo/lemma/tree/main/de.fhdo.lemma.model_processing

Listing 1. Xtend excerpt of LEMMA2Jolie's code generation module.

```
 1 @CodeGenerationModule(name="main")
 2 class GenerationModule extends AbstractCodeGenerationModule {
 3   ...
 4  override getLanguageNamespace() { return DataPackage.eNS_URI }
 5
 6  override execute(...) {
 7   val model = resource.contents.get(0) as DataModel
 8   val generatedContexts = model.contexts.map[it.generateContext]
 9   val baseFileName = FilenameUtils.getBaseName(modelFile)
10   val targetFile = '''«targetFolder»«File.separator»«baseFileName».ol'''
11   return withCharset(#{targetFile -> generatedContexts.join("\n")},
12    StandardCharsets.UTF_8.name)
13  }
14
15  private def generateContext(Context context) {'''
16   ///@beginCtx(«context.name»)
17   «context.complexTypes.map[it.generateComplexType].join("\n")»
18   ///@endCtx
19  ''' }
20
21  private def dispatch generateComplexType(DataStructure structure) {'''
22   «structure.generateType»
23   «IF !structure.operations.empty»
24    «structure.generateInterface»
25   «ENDIF»
26  ''' }
27 }
```

(cf. Fig. 4) and represents the starting point of each template execution since bounded contexts are the top-level elements in LEMMA domain models. An Xtend template is realized between a pair of three consecutive apostrophes within which it is whitespace-sensitive and preserves indentation. Within opening and closing guillemets, Xtend templates enable access to variables and computing operations, whose evaluation shall replace a certain template portion. Consequently, the expression `«context.name»` in the template string in Line 16 is at runtime replaced by the name of the bounded context passed to `generateContext`. For a bounded context with name "BookingManagement", Line 16 of the template will thus result in the generated Jolie code `///@beginCtx(BookingManagement)` (cf. Fig. 4).

To foster its overview and maintainability, we decomposed our template for Jolie APIs into several template methods following the specification of our encoding (cf. Sect. 2.3). As a result, the generation of Jolie code covering the internals of modelled bounded contexts happens in overloaded methods called `generateComplexType`. Each of these methods derives Jolie code for a certain kind of LEMMA complex type, i.e., data structure, list, or enumeration. In Line 17, the template delegates to the version of `generateComplexType` for LEMMA data structures. Following our encoding, the method implements a template to map data structures to Jolie types (Line 22) and interfaces in case the LEMMA data structure exhibits operation signatures (Lines 23 to 25).

The LEMMA2Jolie source code is available as a Software Heritage archive [10]. In addition, we provide a publicly downloadable video illustrating LEMMA2-Jolie's practical capabilities[4].

4 Related and Future Work

Related Work. The maturity of MDE in research and practice as well as its ability to effectively support the engineering of complex software systems [8] has fostered the development of a variety of tools similar to LEMMA2Jolie [15–17,28,32]. That is, they constitute code generators in the sense of MDE [3] and are capable to generate artefacts relevant to MSA engineering. For this purpose, the tools process models constructed in a certain modelling language.

By contrast to LEMMA2Jolie, the majority of related code generators focuses on Java as target technology [15,28,32] and thus not on a programming language specifically tailored to the challenges of microservice implementation. Reducing the semantic gap between the concepts of microservices and implementation languages is the reason for which new service-oriented languages like Ballerina and Jolie have been developed. Furthermore, the modelling languages supported by related tools and hence the generated code address only single concerns in MSA engineering, i.e., domain modelling [17,28] or the implementation and provisioning of service APIs [15,16,32]. By contrast, LEMMA's modelling languages offer an integrated solution to multi-concern modelling in MSA engineering, by providing modelling languages for various viewpoints on microservice architectures [26].

Future Work. The specified encoding (cf. Sect 2.3) and its implementation (cf. Sect. 3) show the feasibility to integrate the LEMMA and Jolie ecosystems. Future works include extending our results to other languages, studying the maturity of LEMMA2Jolie, proving formal guarantees on the correctness of the encoding, and extending the presented integration in several ways.

We plan to evaluate the maturity and stability of LEMMA2Jolie by investigating its application in real-world use cases [27,30].

To obtain correctness guarantees on our encoding, first we would need to formalise the semantics of LEMMA's DDML and of Jolie APIs, and then prove that the encoding generates Jolie APIs that preserve the semantics of input DDML models. This work is in progress, e.g., parts of Jolie have been already formalised [12,20,21] and LEMMA implements context conditions [14] to constrain the well-formedness of DDML models w.r.t. their intended semantics [26].

We also aim to investigate the possibility of round-trip engineering (RTE), i.e., the bidirectional synchronisation of changes between LEMMA models and Jolie code. This would enable, for example, domain experts and microservice

[4] https://bit.ly/3rTGysX

developers to interact by using their views of interest (model vs implementation) but without risking that they fall out of sync. While domain experts could continue to capture domain knowledge about a microservice architecture in conceptual DDD domain models, developers could adapt data types and APIs derived from those models using Jolie as their primary language. Based on RTE, changes in Jolie code could then automatically be reflected in DDD domain models and vice versa, with the option to immediately resolve potential conflicts in domain understanding. Furthermore, we see potential for LEMMA2Jolie to cover all phases in MSA engineering, from domain-driven service design to implementation and deployment. For example, we would like to extend LEMMA2Jolie to deal also with the definition of access points (communication endpoints that define how APIs can be accessed), behaviours (implementations of services written in Jolie that accompany LEMMA models), and the generation of deployment configurations (e.g., configuration of infrastructural services, containerisation, and deployment plans for Kubernetes). This potential is specifically fostered by both LEMMA and Jolie constituting *language-based approaches to MSA engineering*, which facilitates their integration. For example, we could extend LEMMA to include Jolie implementation code in service models.

References

1. Ameller, D., Burgués, X., Collell, O., Costal, D., Franch, X., Papazoglou, M.P.: Development of service-oriented architectures using model-driven development: a mapping study. Inf. Softw. Technol. **62**, 42–66 (2015)
2. Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A.: Microservices in industry: insights into technologies, characteristics, and software quality. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), pp. 187–195. IEEE (2019). https://doi.org/10.1109/ICSA-C.2019.00041
3. Combemale, B., France, R.B., Jézéquel, J.-M., Rumpe, B., Steel, J., Vojtisek, D.: Engineering Modeling Languages: Turning Domain Knowledge into Tools. CRC Press (2017)
4. Daigneau, R.: Service Design Patterns. Addison-Wesley (2012)
5. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
6. Evans, E.: Domain-Driven Design. Addison-Wesley (2004)
7. Evans, E.: Domain-Driven Design Reference. Dog Ear Publishing (2015)
8. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: 2007 Future of Software Engineering, pp. 37–54. IEEE (2007)
9. Freeman, T., Pfenning, F.: Refinement types for ML. In: Proceedings of the 1991 Conference on Programming Language Design and Implementation, pp. 268–277 (1991)
10. [SW] Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., LEMMA2Jolie: a tool to generate Jolie APIs from LEMMA domain models 2022. Università di Bologna et al. vcs: https://github.com/frademacher/lemma2jolie. SWHID: https://swh:1:dir:05b245d8a132648eefffd8aaac5cc35ae945637b;origin=github.com/frademacher/lemma2jolie;visit=swh:1:snp:5f1f9cd4eca3af22d943302e8ab593c92b1d59ef;anchor=swh:1:rev:bae07adfaa0acdf7841c8295cc62d03e894a6bc1

11. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S.: Jolie and LEMMA: model-driven engineering and programming languages meet on microservices. In: Damiani, F., Dardha, O. (eds.) COORDINATION 2021. LNCS, vol. 12717, pp. 276–284. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78142-2_17
12. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: a calculus for service oriented computing. In: International Conference on Service-Oriented Computing, pp. 327–338 (2006)
13. Guidi, C., Maschio, B.: A Jolie based platform for speeding-up the digitalization of system integration processes. In: Proceedings of the Second International Conference on Microservices (Microservices 2019) (2019). https://www.conf-micro.services/2019/papers/Microservices_2019_paper_6.pdf
14. Harel, D., Rumpe, B.: Meaningful modeling: what's the semantics of "semantics"? Computer **37**(10), 64–72 (2004). https://doi.org/10.1109/MC.2004.172
15. JHipster: JHipster Domain Language (JDL), 14 February 2022. https://www.jhipster.tech/jdl
16. Kapferer, S., Zimmermann, O.: Domain-driven service design. In: Dustdar, S. (ed.) SummerSOC 2020. CCIS, vol. 1310, pp. 189–208. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64846-6_11
17. Kapferer, S., Zimmermann, O.: 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, pp. 299–306. SciTePress (2020). https://doi.org/10.5220/0008910502990306
18. Klabnik, S., Nichols, C.: The Rust Programming Language (Covers Rust 2018). No Starch Press (2019)
19. Milner, R.: The tower of informatic models. From semantics to Computer Science (2009)
20. Montesi, F., Carbone, M.: Programming services with correlation sets. In: Kappel, G., Maamar, Z., Motahari-Nezhad, H.R. (eds.) ICSOC 2011. LNCS, vol. 7084, pp. 125–141. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25535-9_9
21. Montesi, F., Guidi, C., Zavattaro, G.: Service-oriented programming with Jolie. In: Web Services Foundations. In: Bouguettaya, A., Sheng, Q.Z., Daniel, F. (eds.) Web Services Foundations, pp. 81–107. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-7518-7_4
22. Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly (2015)
23. Oram, A.: Ballerina: A Language for Network-Distributed Applications. O'Reilly (2019)
24. Python Software Foundation: The Python Language Reference (2021). https://docs.python.org/3/reference/index.html
25. Rademacher, F., Sachweh, S., Zündorf, A.: Deriving microservice code from underspecified domain models using DevOps-enabled modeling languages and model transformations. In: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 229–236. IEEE (2020)
26. Rademacher, F., Sorgalla, J., Wizenty, P., Sachweh, S., Zündorf, A.: Graphical and textual model-driven microservice development. In: Microservices, pp. 147–179. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-31646-4_7

27. Rademacher, F., Sorgalla, J., Wizenty, P., Trebbau, S.: Towards holistic modeling of microservice architectures using LEMMA. In: Companion Proceedings of the 15th European Conference on Software Architecture. CEUR-WS (2021)
28. Sculptor Team: Sculptor-Generating Java code from DDD-inspired textual DSL, 14 February 2022. https://www.sculptorgenerator.org
29. Soldani, J., Tamburri, D.A., Heuvel, W.-J.V.D.: The pains and gains of microservices: a systematic grey literature review. J. Syst. Softw. **146**, 215–232 (2018)
30. Sorgalla, J., Wizenty, P., Rademacher, F., Sachweh, S., Zündorf, A.: Applying model-driven engineering to stimulate the adoption of DevOps processes in small and medium-sized development organizations. SN Comput. Sci. **2**(6), 1–25 (2021). https://doi.org/10.1007/s42979-021-00825-z
31. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley (2008)
32. Terzić, B., Dimitrieski, V., Kordić, S., Milosavljević, G., Luković, I.: Development and evaluation of MicroBuilder: a model-driven tool for the specification of REST microservice software architectures. Enterprise Inf. Syst. **12**(8–9), 1034–1057 (2018)
33. The Rust Foundation: The Rust Reference (2021). https://doc.rust-lang.org/reference/