Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

LEMMA2Jolie: A tool to generate microservice APIs from domain models

(Article begins on next page)

14 January 2025

LEMMA2Jolie: A Tool to Generate Microservice APIs from Domain Models

Saverio Giallorenzo[a], Fabrizio Montesi[b], Marco Peressotti[b], Florian Rademacher[c]

[a]Università di Bologna, Italy and INRIA, France
[b]University of Southern Denmark
[c]University of Applied Sciences and Arts Dortmund

Abstract

We introduce LEMMA2Jolie, a tool for translating domain models of microservice architectures given in LEMMA into concrete APIs of microservices in the Jolie programming language. Our tool combines the state of the art for the design and implementation of microservices: developers can use Domain-Driven Design (DDD) for the construction of the domain models of a microservice architecture, and then automatically transition to a service-oriented programming language that provides native linguistic support for implementing the behaviour of each microservice.

Keywords: Microservices, Model-Driven Engineering, Domain-Driven Design, LEMMA, Jolie

Metadata

See Table 1.

1. Motivation and Significance

Microservice Architecture (MSA) is an approach towards the service-oriented design, development, and operation of software systems based on *microservices*: small and reusable services, which can be composed by using message passing [28, 9]. Microservices can greatly enhance the scalability and maintainability of software systems. They are often contrasted to *monoliths*,

---

Email addresses: `saverio.giallorenzo@gmail.com` (Saverio Giallorenzo),
`fmontesi@imada.sdu.dk` (Fabrizio Montesi), `peressotti@imada.sdu.dk` (Marco
Peressotti), `florian.rademacher@fh-dortmund.de` (Florian Rademacher)

Nr.	Code metadata description	Please fill in this column
C1	Current code version	v1
C2	Permanent link to code/repository used for this code version	`https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/jolie/lemma2jolie`
C3	Permanent link to Reproducible Capsule	`https://archive.softwareheritage.org/browse/origin/directory/?origin_url=https://github.com/jolie/lemma2jolie`
C4	Legal Code License	MIT
C5	Code versioning system used	`git`
C6	Software code languages, tools, and services used	Docker, Jolie, Kotlin, LEMMA Model Processing Framework, Shellscript, Xtend
C7	Compilation requirements, operating environments and dependencies	• **Local installation:** Java 11 (compiler and operating environment), JAR files in `libs` directory (local dependencies), Maven 3.6.3 (remote dependency management according to file `pom.xml` and build management); the `install.sh` script can be used to perform a complete local installation • **Docker-based installation:** Docker >= 20.10.18; the `docker-build.sh` script can be used to build the Docker image on local hardware
C8	If available, link to developer documentation/manual	`https://archive.softwareheritage.org/browse/content/sha1_git:014ea173786c83e149004bad1f4721a548e07d7e/`
C9	Support email for questions	florian.rademacher@fh-dortmund.de

Table 1: Code metadata

applications whose modules cannot be implemented, executed, and deployed independently [9]. However, the adoption of microservices also introduces complexity, a notable example being the structuring of a software architecture into multiple services [40].

To cope with this complexity, researchers in software engineering and programming languages recently started to investigate linguistic approaches to microservices. Such approaches propose language frameworks with syntactic constructs that cover microservice-specific concerns on a high level of abstraction. Consequently, these concerns become explicitly addressable, which facilitates the design, development, and operation of MSAs. Here in particular, we are interested in LEMMA[1] [36] and Jolie[2] [24]. LEMMA applies Model-Driven Engineering (MDE) [6] to provide a set of integrated modelling languages and model processors that, among others, support the high-level specification and implementation of microservice domain models following Domain-Driven Design (DDD) [11]. Jolie, on the other hand, is a programming language focusing on the development of microservices, including their configuration and coordination.

This paper presents LEMMA2Jolie: a tool that integrates LEMMA and Jolie by mapping domain models expressed in LEMMA's Domain Data Modelling Language (DDML) to executable Jolie code, based on the encoding specified in [16]. LEMMA2Jolie brings the following contributions to the field of microservices:

- Support for the refinement of microservices' abstract specifications to implementations by combining MDE and programming abstractions.

- Improvement of DDD adoption in microservice design—in practice, this is frequently perceived complex, given the lack of formal guidelines on how to map domain models to microservice code [2].

- Increase of domain models' value by elevating them from documentation to implementation artefacts.

- Fostering the collaboration of domain experts, who can capture domain knowledge using LEMMA's DDML, and microservice developers, who can readily integrate Jolie programs generated from LEMMA domain models with their microservice implementations.

LEMMA2Jolie is an open-source command line application that takes an input file containing a valid LEMMA domain model and, according to the

---

[1] https://github.com/SeelabFhdo/lemma
[2] https://jolie-lang.org

3

specified encoding [16], generates an output file with a corresponding Jolie program. To keep LEMMA2Jolie portable, it is implemented as a standalone executable Java application that can also be run in Docker[3] containers.

In the context of microservices, there are other approaches to the application of MDE, like MicroBuilder [43], MDSL [20], and JHipster [18], and other programming languages, like Ballerina [30]. We chose LEMMA and Jolie as basis for our work because their independently-developed metamodels are known to be similar [14], which not only aids our implementation but also strengthens our confidence in the foundations for LEMMA2Jolie. Additionally, LEMMA has been validated in real-world use cases [41, 37] and Jolie's abstractions have been found to offer a productivity boost in industry [17]. Nevertheless, LEMMA2Jolie is a concrete example of how research on MDE and programming languages for microservices can benefit from each other in general.

2. Software Description

In the following, we describe LEMMA2Jolie's architecture (cf. Sect. 2.1) and functionalities (cf. Sect. 2.2).

2.1. Software Architecture

Figure 1 shows the architecture of LEMMA2Jolie as a UML class diagram [29].

LEMMA provides a model processing framework (MPF) [34] that aims to facilitate the development of model processors, e.g. code generators or static analysers, by MSA engineers without a strong background in MDE. Among others, the MPF has built-in support for parsing models that were constructed with languages based on the Eclipse Modelling Framework [42]—as is the case for all LEMMA modelling languages including the DDML. Additionally, the MPF prescribes a workflow that consists of steps which are frequent in practical model processing. Due to its popularity in MSA engineering [38, 2], the MPF focuses on the Java Virtual Machine (JVM) and is written in Kotlin[4]. Furthermore, like the Spring framework[5], which is popular in JVM-based microservice development [2], the MPF adopts annotation-based Inversion of Control (IoC) [19]. This approach enables MSA engineers, who want develop model processors, to integrate with the MPF's model processing

---

[3]https://www.docker.com
[4]https://www.kotlinlang.org
[5]https://www.spring.io

Figure 1: Architecture of LEMMA2Jolie depicted as a UML class diagram [29]. Highlighted classes are part of LEMMA2Jolie. All other classes originate from LEMMA's model processing framework.

workflow via Java annotations, and readily focus on the actual processing logic rather than boilerplate code for model parsing, model validation, or code generation.

LEMMA2Jolie is based on LEMMA's MPF. Consequently, the source code of our tool mainly revolves around its actual task, i.e., the translation of LEMMA domain models into Jolie programs, with the MPF abstracting most of the complexity in model processing. As a result, in the implementation of LEMMA2Jolie, we leveraged the mentioned structure and focussed our efforts on the meaningful modules that make up the logic of the tool: the three the classes highlighted in grey in Fig. 1, which we describe below.

The Lemma2Jolie class provides the entrypoint of our LEMMA2Jolie by implementing a static `main` method as expected by the JVM. The class also extends the MPF's `AbstractModelProcessor` class and invokes its `run` method from `main`, thereby delegating all further execution to the MPF.

As a next step, the MPF will parse the commandline arguments from the `args` array, and then scan the Java package hierarchy of the model processor following annotation-based IoC for a class that acts as *language description*

provider and implements the `LanguageDescriptionProviderI` interface. At model processor runtime, the MPF queries the language description provider whenever it requires information about the language of a given model, e.g., for parsing purposes, by invoking the providers `getLanguageDescription` method. This method must return an instance of the `LanguageDescription` class that comprises an implementation of the `loadModel` method (cf. Fig. 1). This method is expected to return an instance of the `Resource` class, which is EMF's for persistent documents including models [42].

With the `LanguageDescriptionProviderI` interface and `LanguageDescription` class, the MPF abstracts from concrete modelling languages and enables to implement custom loaders for EMF-based modelling techniques. Since LEMMA's modelling languages are based on the Xtext framework[6] the MPF already integrates a specialised language description with the `XtextLanguageDescription` class (cf. Fig. 1). LEMMA2Jolie's language description provider in the `LangDescriptionProvider` class relies on this specialised language description to provide the MPF with all information necessary to parse LEMMA domain models. More precisely, the `getLanguageDescription` method of `LangDescriptionProvider` will return an `XtextLanguageDescription` instance covering LEMMA's DDML whenever the `namespaceOrExtension` parameter exhibits the value "data", which indicates that the MPF received as commandline argument the path to a LEMMA domain model with the file extension ".data" (cf. Sect. 2.3).

LEMMA's MPF applies the Phased Construction pattern of model transformation design [21] to systematize the processing of input models in consecutive phases. `AbstractModelProcessingPhase` is the base class of all MPF phases (cf. Fig. 1). A concrete phase must implement the `process` method, which receives phase-specific commandline arguments at runtime. To facilitate the implementation of model processors, the MPF prescribes a workflow of certain phases, e.g., for model parsing and code generation, and thus provides built-in phases such as `SourceModelParsingPhase` and `CodeGenerationPhase` (cf. Fig. 1). A phase can specify return values in the form of `ReturnParameter` instances and also express the expectation of certain return values from previous phases leveraging the `ExpectedReturnParameter` class. The MPF ensures that phases return values matching the `name` and `type` of non-`optional` `ReturnParameter` instances so that subsequent phases have guaranteed access to expected values via the `PhaseHeap`, which is a generic means for phases to share data.

In MDE terms, LEMMA2Jolie is a code generator that produces Jolie code

---

[6] https://www.github.com/xtext

from LEMMA domain models. Hence, it integrates with the MPF's phase for code generation whose execution is controlled by the `CodeGenerationPhase` class. This class relies on the notion of *code generation module* to enable implementers the organisation of code generation steps, e.g., by the kinds of processed models or produced artefacts. The code generation phase scans the Java package hierarchy of MPF-model processors for implementations of the `AbstractCodeGenerationModule` class (cf. Fig. 1). A concrete code generation module specifies a `name`, and receives the path to the `model-File` it shall process as well as the corresponding EMF `resource` of the parsed model. The code generation phase will only invoke those modules on a given model file for which the return value of `getLanguageNamespace` matches the namespace of the parsed model as identified by the language the model was constructed with. For the DDML and LEMMA domain models constructed with it, only code generation modules targeting the namespace `de.fhdo.lemma.data` are applicable. A code generation module is expected to implement the `execute` method and return a Java `Map` instance[7] that determines per file path the content and character set of the generated file.

The `GenerationModule` class realises the code generation module of LEMMA2Jolie (cf. Fig. 1). Each method of the module with the prefix `generate` is responsible for mapping a certain kind of LEMMA domain model element to corresponding Jolie code following the specified encoding [16]. For example, `generateComplexType` maps a domain concept specified in a LEMMA domain model [36] to the corresponding Jolie type. The `generateInterface` method, on the other hand, produces code for Jolie interfaces from a given LEMMA structure type. According to the specified encoding [16], each LEMMA structure receives a Jolie interface consisting of request-response operations derived from LEMMA domain operations by the `generateOperation` method of the `GenerationModule` class (cf. Fig. 1).

2.2. Software Functionalities

We describe the functionalities of LEMMA2Jolie along with its usage workflow shown in Fig. 2 as a UML activity diagram [29].

LEMMA2Jolie's usage workflow consists of the following steps:

S1. Domain Model Construction: This step covers the construction of the LEMMA domain model to be processed by LEMMA2Jolie. This may be a new model or an existing model that is adapted to cope with newly discovered facts about the application domain [11]. For guided

---

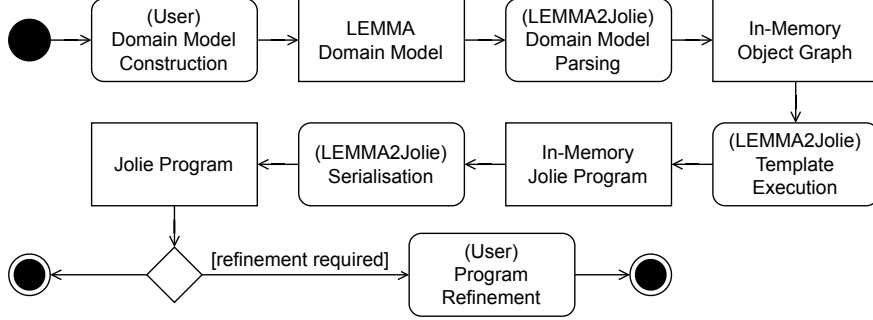[7] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Map.html

7

Figure 2: Usage workflow of LEMMA2Jolie as a UML activity diagram [29].

domain model construction including syntax highlighting, code completion, and interactive model validation, the user may apply LEMMA's Eclipse plugin for the DDML [33]. While LEMMA2Jolie does not assume a certain kind of user for domain model construction, this step is usually performed by domain experts, possibly in collaboration with microservice developers [36].

S2. Domain Model Parsing: LEMMA2Jolie performs this step after the user invoked the tool on the previously constructed domain model. More precisely, the `SourceModelParsingPhase` class of LEMMA's MPF (cf. Sect. 2.1) accounts for this step, which will result in an in-memory object graph of the parsed model. Specifically, this graph is an instance of the DDML metamodel [36, 14].

S3. Template Execution: LEMMA2Jolie relies on *template-based code generation* [6] to transform LEMMA domain models into Jolie programs. A template is a specification consisting of static Jolie statements and dynamic variables pointing to concepts from the metamodel of LEMMA's DDML. At runtime, LEMMA2Jolie evaluates each template and replaces the dynamic variables with actual values from the in-memory object graph resulting from Step S2. For example, the template that maps a bounded context in a LEMMA domain model to the corresponding Jolie code [16] is located in the `generateContext` method of LEMMA2Jolie's `GenerationModule` class (cf. Sect. 2.1) and starts with the statement `///@beginCtx(«context.name»)`. The guillemets enclose the dynamic part of the template to be evaluated at runtime. All other parts of the statement are static and thus invariant at runtime (cf. Sect. 2.3). The Template Execution step produces an in-memory representation of the target Jolie program as a single string consisting of Jolie statements resulting from template evaluations.

8

S4. Serialisation: This step stores the in-memory representation of the target Jolie program in a file with the extension ".ol" as expected by the Jolie interpreter [24]. The file path and name can be specified by the user upon invocation of LEMMA2Jolie in Step S1.

S5. Program Refinement: In this optional step, the user, usually a service developer, can refine the serialised Jolie program, e.g., to add additional documentation. For this purpose, the user can rely on the Jolie extension for Visual Studio Code[8]. Like the Eclipse plugins for LEMMA's modelling languages, the extension supports Jolie programming with syntax highlighting, code completion, and interactive validation.

2.3. Sample Code Snippets

In the following, we describe the implementation of LEMMA2Jolie's core components which account for the realisation of the workflow steps that exhibit the `LEMMA2Jolie` annotation [29] in the UML activity diagram in Fig. 2. We orient the description towards the ordering of the annotated activities. LEMMA2Jolie was written in the Xtend Java dialect[9] and its complete source code can be found in the accompanying code repository.

2.3.1. Domain Model Parsing

As described in Sect. 2.1, LEMMA's MPF is capable of parsing domain models expressed in the DDML. In order to delegate parsing to the MPF, a model processor like LEMMA2Jolie must provide a language description provider and specify the Java package to scan for that provide following annotation-based IoC. Listing 1 shows the implementation of the `Lemma2-Jolie` class. It is the programmatic entrypoint of LEMMA2Jolie and specifies the Java package for the MPF to search for annotated classes.

Listing 1: Programmatic entrypoint of LEMMA2Jolie written in Xtend.

```
1  class Lemma2Jolie extends AbstractModelProcessor {
2    new() { super("lemma2jolie") }
3    def static void main(String[] args) { new Lemma2Jolie().run(args) }
4  }
```

In Line 2, the class configures the MPF to search for annotated classes in the "lemma2jolie" Java package by passing the package name to the constructor of the `AbstractModelProcessor` class (cf. Sect. 2.1). Line 3 implements the entrypoint method of LEMMA2Jolie and delegates execution to

---

[8]https://github.com/jolie/vscode-jolie
[9]https://www.eclipse.org/xtend

the MPF by calling the `run` method inherited from the `AbstractModelProcessor`.

The execution of `run` results in the MPF to first parse the commandline arguments in the `args` array and then scan the "lemma2jolie" Java package for a language description provider. Listing 2 shows an excerpt of the `LangDescriptionProvider` class, which implements LEMMA2Jolie's language description provider.

Listing 2: Xtend excerpt of LEMMA2Jolie's language description provider.

```
1  @LanguageDescriptionProvider
2  class LangDescriptionProvider implements LanguageDescriptionProviderI {
3   override getLanguageDescription(..., String namespaceOrExtension) {
4    return switch (namespaceOrExtension) {
5     case "data": new XtextLanguageDescription(DataPackage.eINSTANCE,
6      new DataDslStandaloneSetup)
7     ...
8    }
9   }
10 }
```

The MPF identifies `LangDescriptionProvider` to constitute LEMMA-2Jolie's language description provider by the `@LanguageDescriptionProvider` annotation (cf. Line 1). As described in Sect. 2.1, the class's `getLanguageDescription` method returns an `XtextLanguageDescription` whenever LEMMA2Jolie received a LEMMA domain model file, recognized by the ".data" extension, as input model (cf. Lines 4 to 6).

2.3.2. Template Execution

Listing 3 shows selected code generation templates of LEMMA2Jolie.

Listing 3: Selected Xtend templates of LEMMA2Jolie.

```
1  @CodeGenerationModule(name="main", ...)
2  class GenerationModule extends AbstractCodeGenerationModule {
3   ...
4
5   private def generateContext(Context context) {'''
6    ///@beginCtx(«context.name»)
7    «context.complexTypes.map[it.generateComplexType].join("\n")»
8    ///@endCtx
9   '''}
10
11  private def dispatch generateComplexType(DataStructure structure) {'''
12   «structure.generateType»
13   «IF !structure.operations.empty»
14    «structure.generateInterface»
15   «ENDIF»
16  '''}
17
18   ...
19 }
```

All code generation templates are located in methods with the `gener-ate` prefix in LEMMA2Jolie's code generation module implemented by the `GenerationModule` class (cf. Sect. 2.1). To make a code generation module recognizable at runtime by the MPF, its class must be augmented with the `@CodeGenerationModule` annotation as shown in Line 1. Lines 5 to 16 illustrate the realisation of code generation templates in Xtend by the `generateContext` and `generateComplexType` methods. An Xtend template is enclosed by three consecutive apostrophes (''''). Within a template, Xtend treats all strings outside a pair of guillemets as static template parts. For example, the string "///@beginCtx(" in Line 6 is a static part, whereas the following statement «context.name» constitutes a dynamic part that Xtend replaces at runtime with the value in the `name` attribute of the `context` parameter (cf. Line 5), i.e., the name of the processed bounded context from the given LEMMA domain model [36].

For each data structure in a bounded context of a LEMMA domain model, `generateContext` delegates to `generateComplexType` to generate the Jolie code for the structure (cf. Line 7). According to the previously specified encoding [16], a LEMMA data structure will be mapped to a Jolie type by invoking the `generateType` method of the `GenerationModule` class (cf. Line 12 and Fig. 1). Furthermore, in case the structure does not only define data field but also operations, LEMMA2Jolie will derive a Jolie interface for the structure [16] (cf. Lines 13 to 15).

2.3.3. Serialisation

Next to template execution, the `GenerationModule` class is also responsible for providing the MPF with the generated Jolie code to serialise as files on the user's hard drive. Listing 4 shows the corresponding excerpt from the implementation of the `GenerationModule` class.

The `execute` method is the entrypoint of MPF code generation modules, from which the in-memory object graph of the parsed model (cf. Fig. 2) is accessible via the inherited `resource` attribute. Line 7 of LEMMA2Jolie's code generation module retrieves the root of the parsed LEMMA domain model as an instance of the `DataModel` concept of the DDML's metamodel [36]. Next, the module calls the template method `generateContext` (cf. Listing 3) for each parsed `Context` instance under the domain model root and gathers the generated Jolie code as a list of strings in the `generatedContexts` variable (cf. Line 8).

Finally, the module determines the path of the file for the generated Jolie code which will be created in the given target folder and with the same base name as the input LEMMA domain model but with the extension "ol" (cf. Lines 9 and 10). The serialisation of the generated Jolie code is triggered

Listing 4: Xtend excerpt of LEMMA2Jolie's code generation module responsible for Jolie code serialisation.

```
1  @CodeGenerationModule(name="main", ...)
2  class GenerationModule extends AbstractCodeGenerationModule {
3   ...
4   override getLanguageNamespace() { return DataPackage.eNS_URI }
5
6   override execute(String[] phaseArguments, String[] moduleArguments) {
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  /* cf.  Listing 3 */
16  private def generateContext(Context context) { ... }
17  private def dispatch generateComplexType(DataStructure structure) { ... }
18 }
```

by invoking the inherited MPF method `withCharset`. The method expects a map of file paths and contents, and the target character set as argument. For LEMMA2Jolie's code generation module, the first argument associates the previously assembled path of the file for the generated Jolie code with the generated code concatenated in a string separated by line breaks (cf. Line 11). The second argument of `withCharset` determines the character set of the generated code, i.e., UTF-8 (cf. Line 12).

3. Illustrative Examples

We exemplify the main functions of LEMMA2Jolie (cf. Sect. 2.2) by an excerpt of a case study for booking parking spaces with charging stations for electric vehicles. The case study was introduced by a previous paper [37] and also used to illustrate our encoding of LEMMA domain models into Jolie programs [16]. The complete case study, including a description of its processing with LEMMA2Jolie, can be found in the accompanying code repository. Additionally, we provide a video showing the practical usage of LEMMA2Jolie on the case study[10].
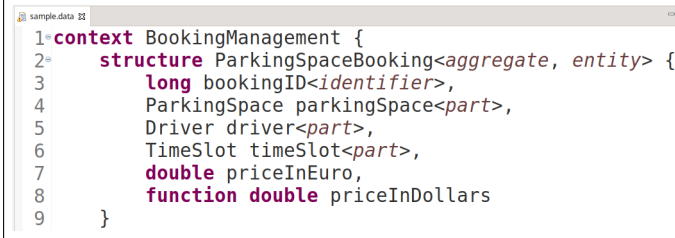
Next, we describe LEMMA2Jolie's application on the selected excerpt of the case study following the structure of the workflow in Fig. 2.

3.1. Domain Model Construction

Figure 3 exemplifies the construction of a LEMMA domain model using the Eclipse plugin for the DDML. The complete model can be found in the

---

[10]https://doi.org/10.5281/zenodo.7547046

accompanying code repository


```
sample.data ⅗
1  context BookingManagement {
2      structure ParkingSpaceBooking<aggregate, entity> {
3          long bookingID<identifier>,
4          ParkingSpace parkingSpace<part>,
5          Driver driver<part>,
6          TimeSlot timeSlot<part>,
7          double priceInEuro,
8          function double priceInDollars
9      }
```

Figure 3: Construction of a LEMMA domain model using LEMMA's Eclipse plugin for the DDML.

The domain model specifies the `BookingManagement` bounded context[11, 16]. The context comprises the `ParkingSpaceBooking` structure which is a DDD aggregate and entity. As a result, instances of the structure resemble object graphs that must maintain a consistent state when being fetched from and stored to a database [11].

The structure consists of the following data fields:

- `bookingID`: The field is of LEMMA's primitive type `long` [36] and enables to distinguish `ParkingSpaceBooking` instances.

- `parkingSpace`: The field allows storing information about the booked parking space together with a `ParkingSpaceBooking` instance. Therefore, `parkingSpace` is modelled as an aggregate *part* that, according to DDD, is only valid and accessible from the root of its enclosing aggregate [11].

- `driver`, `timeSlot`: These fields enable storing information about the driver of an electric vehicle who made a certain parking booking and the time slot for which this booking is valid together with the booking. As for the same reason as `parkingSpace`, both fields are modelled as parts of the `ParkingSpaceBooking` aggregate.

- `priceInEuro`: The field is of LEMMA's primitive type `double` and receives the price for the parking space booking in the Euro currency.

Next to the data fields, the structure also specifies the signature of the `priceInDollars` function which is responsible for converting the value in the `priceInEuro` field to the Dollar currency.

3.2. LEMMA2Jolie

The domain model in Fig. 3 can be transformed into the corresponding Jolie program by executing LEMMA2Jolie from its Java archive or Docker image. To this end, the user invokes LEMMA2Jolie from the commandline as shown in Listing 5.

Listing 5: Execution of LEMMA2Jolie with (a) Java and (b) Docker.

(a)

```
java -jar lemma2jolie.jar \
 -s /home/user/sample.data \
 -t /home/user
```

(b)

```
docker run \
 -u `id -u`:`id -g` \
 -v /home/user:/home/user \
 lemma2jolie:latest \
  -s /home/user/sample.data \
  -t /home/user
```

3.3. Program Refinement

The execution of LEMMA2Jolie on the LEMMA domain model (cf. Listing 5) in the file `sample.data` (cf. Fig. 3) results in a Jolie program file `sample.ol`. As depicted in Fig. 4, this file can be refined, e.g., by a microservice developer, using the Jolie extension for Visual Studio Code. We provide the complete source code of the `sample.ol` file in the accompanying code repository

Following the specified encoding [16], LEMMA2Jolie transformed the `BookingManagement` bounded context introduced by the LEMMA domain model (cf. Fig. 3) into a Jolie documentation comment starting with `@beginCtx` (cf. Line 1). In addition, the tool mapped the `ParkingSpaceBooking` structure to a Jolie type consisting of fields the correspond to those in the input LEMMA structure (cf. Lines 4 to 14). The `priceInDollars` function was however transformed into a request-response operation of the Jolie interface for the `ParkingSpaceBooking` structure (cf. Lines 18 to 21). The operation expects an instance of the `priceInDollars_type` (cf. Lines 15 to 17) as input, thereby, following the encapsulation principle of object-oriented programming, allowing implementers to access the `ParkingSpaceBooking` instance for which the operation was invoked.

4. Conclusion: Impact and Future Plans

LEMMA2Jolie demonstrates how the ecosystems of Model-Driven Engineering (MDE), Domain-Driven Design (DDD), and service-oriented programming can be successfully integrated, using in particular LEMMA and Jolie. The immediate benefit is providing an automatic transition from DDD (in

```
sample.ol  ×
home > user > sample.ol
 1   ///@beginCtx(BookingManagement)
 2   ///@aggregate
 3   ///@entity
 4   type ParkingSpaceBooking {
 5       ///@identifier
 6       bookingID: long
 7       ///@part
 8       parkingSpace: ParkingSpace
 9       ///@part
10       driver: Driver
11       ///@part
12       timeSlot: TimeSlot
13       priceInEuro: double
14   }
15   type priceInDollars_type {
16       self?: ParkingSpaceBooking
17   }
18   interface ParkingSpaceBooking_interface {
19       RequestResponse:
20           priceInDollars(priceInDollars_type)(double)
21   }
```

Figure 4: Refinement of a Jolie program, which was derived from a LEMMA domain model by LEMMA2Jolie, using the Jolie extension for Visual Studio Code.

LEMMA) to the implementation of each service (in Jolie), including informative annotations in the generated Jolie code that come from the domain models.

In general, our approach paves the way for the future exploration of synergies between the abstraction facilities of MDE/DDD and service-oriented programming languages. We mention five interesting directions for future investigations.

First, LEMMA2Jolie could be extended to cover all phases of MSA engineering, from domain-driven service design to implementation and deployment. For example, we plan to extend LEMMA2Jolie to cover also the deployment configuration of MSAs. A promising option is leveraging the toolchain presented in [25] for automatically slicing a codebase consisting of many Jolie services into separate codebases, each with its own configuration for containerisation, and a deployment configuration for a container orchestrator (e.g., Docker Swarm, Kubernetes). In the same spirit, we plan on integrating LEMMA, Jolie, and deployment languages more closely, by supporting development environments where the languages can be used together with a unified view. This is in line with corresponding research on software language composition [10, 8, 3]. For example, we could use quasi-quotation

15

mechanisms to enable layering the languages and expose information about the domain model to the implementation in Jolie and the deployment configurations such that mismatches across these boundaries could be detected.

Second, LEMMA2Jolie offers the possibility to study approaches to round-trip engineering (RTE) [39], i.e., the bidirectional synchronisation of LEMMA models and Jolie code. RTE support for LEMMA2Jolie would improve interaction between domain experts and microservice developers. Each stakeholder would use their views of interest (model vs implementation) to understand, build, and modify a given architecture and LEMMA2Jolie would keep those views consistent. In this way, LEMMA2Jolie would support RTE by reflecting changes done by developers on the Jolie codebase on its companion domain models and, vice versa, keep the codebase on par when experts change the domain models.

Third, we envision that LEMMA2Jolie could also stimulate research concerning the specification of coordinated microservice behaviour following the paradigm of Choreographic Programming (CP) [22, 23]. CP formally models how distributed components coordinate with each other via communication as artifacts called *choreographies*, which are then compiled to correct-by-construction implementations of communication behaviours for the components. CP guarantees quality attributes such as compliance (the components interact as expected) and deadlock-freedom [4, 7, 13, 23]. Choreography, in the sense of distributed coordination, is the de-facto best practice for the composition of microservices [28]. CP facilitates the writing of choreographies and their correct implementation, giving a high potential for impact. However, while there exist early works on the application of CP to microservices in Jolie [4, 12], there are still no tools that enable domain experts to participate in the specification of domain aspects in choreographies. This could be achieved, e.g., by providing abstract textual or graphical representations of choreographies that correspond to choreographies in CP. In this spirit, MSDL (Microservice Domain-Specific Language) includes a textual language for expressing integration flows, but it is not integrated with CP [45].

Fourth, there is potential in exploring the connection between architectural and API patterns for microservices in MDE and their implementation in Jolie. For example, the application of reusable infrastructural components (like circuit breakers) and patterns for API control has been investigated in both the realms of MDE and Jolie [45, 31, 32, 26, 27, 5]. While these aspects are interesting for both communities, they have never been integrated to support, e.g., the application of infrastructural and API management components starting from domain models.

Finally, following the approach of Software Architecture Reconstruction [1] (SAR), LEMMA supports the model-based reconstruction of microservice

architectures for which no documentation exists or whose documentation is outdated [35]. Currently, we are working on automating model-based SAR with LEMMA [44]. In combination with LEMMA2Jolie, automated SAR with LEMMA provides an opportunity for studying the semi-automated migration of microservices from general-purpose programming languages, such as Java or C# [38], to Jolie with the aim of raising service implementation quality by relying on service-oriented primitives that, e.g., ensure the correctness of concurrent service execution.

Acknowledgements

References

[1] Len Bass, Paul Clements and Rick Kazman. *Software Architecture in Practice*. Third. Addison-Wesley, 2013.

[2] Justus Bogner et al. "Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, Mar. 2019, pp. 187–195. DOI: 10.1109/ICSA-C.2019.00041.

[3] Arvid Butting et al. "Modeling Language Variability with Reusable Language Components". In: SPLC '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 65–75.

[4] Marco Carbone and Fabrizio Montesi. "Deadlock-freedom-by-design: multiparty asynchronous global programming". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 263–274. DOI: 10.1145/2429069.2429101.

[5] Ramaswamy Chandramouli. *Security Strategies for Microservices-based Application Systems*. Available at https://doi.org/10.6028/NIST.SP.800-204. National Institute of Standards and Technology, 2019. DOI: https://doi.org/10.6028/NIST.SP.800-204.

[6] Benoit Combemale et al. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. CRC Press, 2017.

[7] Mila Dalla Preda et al. "Dynamic Choreographies: Theory And Implementation". In: *Logical Methods in Computer Science* 13.2 (2017). DOI: 10.23638/LMCS-13(2:1)2017.

[8]   J. Deantoni. "Modeling the Behavioral Semantics of Heterogeneous Languages and their Coordination". In: *2016 Architecture-Centric Virtual Integration (ACVI)*. 2016, pp. 12–18.

[9]   Nicola Dragoni et al. "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Springer, 2017, pp. 195–216. ISBN: 978-3-319-67425-4.

[10]   Sebastian Erdweg, Paolo G. Giarrusso and Tillmann Rendel. "Language Composition Untangled". In: LDTA '12. Tallinn, Estonia: Association for Computing Machinery, 2012. ISBN: 9781450315364.

[11]   Eric Evans. *Domain-Driven Design*. Addison-Wesley, 2004.

[12]   Saverio Giallorenzo, Ivan Lanese and Daniel Russo. "ChIP: A Choreographic Integration Process". In: *On the Move to Meaningful Internet Systems. OTM 2018 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part II*. Springer, 2018, pp. 22–40. DOI: `10.1007/978-3-030-02671-4\_2`.

[13]   Saverio Giallorenzo, Fabrizio Montesi and Marco Peressotti. "Object-Oriented Choreographic Programming". In: arXiv, 2022. DOI: `10.48550/ARXIV.2005.09520`. URL: `https://arxiv.org/abs/2005.09520`.

[14]   Saverio Giallorenzo et al. "Jolie and LEMMA: Model-Driven Engineering and Programming Languages Meet on Microservices". In: *Coordination Models and Languages*. Springer, 2021, pp. 276–284. ISBN: 978-3-030-78142-2.

[15]   [SW] Saverio Giallorenzo et al., *LEMMA2Jolie: A Tool to Generate Jolie APIs from LEMMA Domain Models* 2022. Università di Bologna et al. VCS: `https://github.com/jolie/lemma2jolie`, SWHID: ⟨`swh:1:dir:bb764182f97dc9bff420347092356eac38e0d051;origin=https://github.com/jolie/lemma2jolie;visit=swh:1:snp:73e0bba679965a3f7fa712eca7b161f2fac3c673;anchor=swh:1:rev:4011e45ecb5a233acbe29853dc47606159767a78`⟩.

[16]   Saverio Giallorenzo et al. "Model-Driven Generation of Microservice Interfaces: From LEMMA Domain Models to Jolie APIs". In: *Coordination Models and Languages*. Ed. by Maurice H. ter Beek and Marjan Sirjani. Cham: Springer Nature Switzerland, 2022, pp. 223–240. ISBN: 978-3-031-08143-9.

[17]   Claudio Guidi and Balint Maschio. "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`.

[18]   JHipster. *JHipster Domain Language (JDL)*. 2022-14-02. URL: `https://www.jhipster.tech/jdl`.

[19]   Ralph E. Johnson and Brian Foote. "Designing Reusable Classes". In: *Journal of Object-Oriented Programming* 1.2 (1988). SIGS Publications, pp. 22–35.

[20]   Stefan Kapferer and Olaf Zimmermann. "Domain-Driven Service Design". In: *Service-Oriented Computing*. Springer, 2020, pp. 189–208.

[21]   Kevin Lano and Shekoufeh Kolahdouz-Rahimi. "Model-Transformation Design Patterns". In: *IEEE Transactions on Software Engineering* 40.12 (2014). IEEE, pp. 1224–1259.

[22]   Fabrizio Montesi. "Choreographic Programming". PhD thesis. IT University of Copenhagen, 2013.

[23]   Fabrizio Montesi. "Introduction to Choreographies". Accepted for publication by Cambridge University Press. 2022.

[24]   Fabrizio Montesi, Claudio Guidi and Gianluigi Zavattaro. "Service-Oriented Programming with Jolie". In: *Web Services Foundations*. Ed. by Athman Bouguettaya, Quan Z. Sheng and Florian Daniel. Springer, 2014, pp. 81–107. DOI: `10.1007/978-1-4614-7518-7\_4`. URL: `https://doi.org/10.1007/978-1-4614-7518-7%5C_4`.

[25]   Fabrizio Montesi, Marco Peressotti and Valentino Picotti. "Sliceable Monolith: Monolith First, Microservices Later". In: *IEEE International Conference on Services Computing, SCC 2021, Chicago, IL, USA, September 5-10, 2021*. Ed. by Barbara Carminati et al. IEEE, 2021, pp. 364–366. DOI: `10.1109/SCC53864.2021.00050`. URL: `https://doi.org/10.1109/SCC53864.2021.00050`.

[26]   Fabrizio Montesi and Janine Weber. "Circuit Breakers, Discovery, and API Gateways in Microservices". In: *CoRR* abs/1609.05830 (2016). arXiv: `1609.05830`. URL: `http://arxiv.org/abs/1609.05830`.

[27]   Fabrizio Montesi and Janine Weber. "From the decorator pattern to circuit breakers in microservices". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*. Ed. by Hisham M. Haddad, Roger L. Wainwright and Richard Chbeir. ACM, 2018, pp. 1733–1735. DOI: `10.1145/3167132.3167427`. URL: `https://doi.org/10.1145/3167132.3167427`.

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

[29]   OMG. *OMG Unified Modeling Language (OMG UML) Version 2.5.1*. Standard formal/17-12-05. Object Management Group, 2017.

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

[31]   Mila Dalla Preda et al. "Interface-Based Service Composition with Aggregation". In: *Service-Oriented and Cloud Computing - First European Conference, ESOCC 2012, Bertinoro, Italy, September 19-21, 2012. Proceedings*. Ed. by Flavio De Paoli, Ernesto Pimentel and Gianluigi Zavattaro. Vol. 7592. Lecture Notes in Computer Science. Springer, 2012, pp. 48–63. DOI: `10.1007/978-3-642-33427-6\_4`. URL: `https://doi.org/10.1007/978-3-642-33427-6%5C_4`.

[32]   Mila Dalla Preda et al. "Service integration via target-transparent mediation". In: *2012 Fifth IEEE International Conference on Service-Oriented Computing and Applications (SOCA), Taipei, Taiwan, December 17-19, 2012*. IEEE Computer Society, 2012, pp. 1–5. DOI: `10.1109/SOCA.2012.6449432`. URL: `https://doi.org/10.1109/SOCA.2012.6449432`.

[33]   [SW] Florian Rademacher, *Eclipse Plugin for LEMMA's Domain Data Modelling Language* 2022. University of Applied Sciences and Arts Dortmund. VCS: `https://github.com/SeelabFhdo/lemma`, SWHID: ⟨`swh:1:dir:78f5d6179d9f84fa847d07c609cac4e43eb93e19;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:d29f6f5ad7f519755932db9fa22ff237c3beab90;anchor=swh:1:rev:13cac5a32db49c362244d5b9f6545d28757edbee;path=/de.fhdo.lemma.data.datadsl/`⟩.

[34]   [SW] Florian Rademacher, *LEMMA Model Processing Framework* 2022. University of Applied Sciences and Arts Dortmund. VCS: `https://github.com/SeelabFhdo/lemma`, SWHID: ⟨`swh:1:dir:5280a90287fb45c1bfea4d217497399fc8b345d6;origin=https://github.com/SeelabFhdo/lemma;visit=swh:1:snp:d29f6f5ad7f519755932db9fa22ff2`

`37c3beab90;anchor=swh:1:rev:13cac5a32db49c362244d5b9f6545d`
`28757edbee;path=/de.fhdo.lemma.model_processing/⟩`.

[35] Florian Rademacher, Sabine Sachweh and Albert Zündorf. "A Modeling Method for Systematic Architecture Reconstruction of Microservice-Based Software Systems". In: *Enterprise, Business-Process and Information Systems Modeling.* Springer, 2020, pp. 311–326. ISBN: 978-3-030-49418-6.

[36] Florian Rademacher et al. "Graphical and Textual Model-Driven Microservice Development". In: *Microservices: Science and Engineering.* Springer, 2020, pp. 147–179. ISBN: 978-3-030-31646-4.

[37] Florian Rademacher et al. "Towards an Extensible Approach for Generative Microservice Development and Deployment Using LEMMA". In: *Software Architecture.* Ed. by Patrizia Scandurra et al. Cham: Springer International Publishing, 2022, pp. 257–280. ISBN: 978-3-031-15116-3.

[38] Gerald Schermann, Jürgen Cito and Philipp Leitner. "All the Services Large and Micro: Revisiting Industrial Practice in Services Computing". In: *Service-Oriented Computing – ICSOC 2015 Workshops.* Ed. by Alex Norta et al. Berlin, Heidelberg: Springer, 2016, pp. 36–47. ISBN: 978-3-662-50539-7.

[39] Bran Selic. "The pragmatics of model-driven development". In: *IEEE Software* 20.5 (Sept. 2003). IEEE, pp. 19–25. ISSN: 1937-4194. DOI: `10.1109/MS.2003.1231146`.

[40] Jacopo Soldani, Damian Andrew Tamburri and Willem-Jan Van Den Heuvel. "The pains and gains of microservices: A Systematic grey literature review". In: *Journal of Systems and Software* 146 (2018). Elsevier, pp. 215–232. ISSN: 0164-1212.

[41] Jonas Sorgalla et al. "Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations". In: *SN Computer Science* 2.6 (2021), p. 459. ISSN: 2661-8907.

[42] Dave Steinberg et al. *EMF: Eclipse Modeling Framework.* Second. Addison-Wesley, 2008.

[43] Branko Terzić et al. "Development and evaluation of MicroBuilder: a Model-Driven tool for the specification of REST Microservice Software Architectures". In: *Enterprise Information Systems* 12.8-9 (2018). Taylor & Francis, pp. 1034–1057.

[44] Philip Wizenty and Florian Rademacher. "Towards Viewpoint-Based Microservice Architecture Reconstruction". In: *Fourth International Conference on Microservices (Microservices 2022)*. URL: https://www.conf-micro.services/2022/papers/paper_16.pdf.

[45] Olaf Zimmermann et al. *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley Professional, 2022. ISBN: 978-0137670109.

LEMMA2Jolie's source code is permanently available at Software Heritage under the persistent identifier (SWHID) `swh:1:dir:bb764182f97dc9bff4 20347092356eac38e0d051;origin=https://github.com/jolie/lemma2j olie;visit=swh:1:snp:73e0bba679965a3f7fa712eca7b161f2fac3c673;a nchor=swh:1:rev:4011e45ecb5a233acbe29853dc47606159767a78` [15].