LEMMA2Jolie: Model-Driven Generation of Microservice Interfaces

Saverio Giallorenzo[1],

Fabrizio Montesi[2], Marco Peressotti[2], and Florian Rademacher[3]

[1] Università di Bologna and INRIA saverio.giallorenzo@gmail.com
[2] University of Southern Denmark {fmontesi,peressotti}@imada.sdu.dk
[3] University of Applied Sciences and Arts Dortmund florian.rademacher@fh-dortmund.de

1 Introduction

Microservice Architecture (MSA) is one of the current leading patterns in distributed software architectures [14]. While widely adopted, MSA comes with specific challenges on architecture design, development, and operation [4, 20]. 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 microservice architectures with high-level constructs that make microservice concerns in the two different stages syntactically manifest.

Regarding development, Ballerina and Jolie are examples of programming languages [15, 13] with new linguistic abstractions for effectively programming the configuration and coordination of microservices. Concerning design, Model-Driven Engineering (MDE) [2] has gained relevance as a method for the specification of service architectures [1], crystallised in MDE-for-MSA frameworks such as MicroBuilder, MDSL, LEMMA, and JHipster [22, 11, 18, 10]. Guidi and Maschio [9] recently reported how Jolie's abstractions offer a productivity boost in industry. LEMMA provides linguistic support for the application of concepts from Domain-Driven Design (DDD) [5, 18], and has been validated in real-world use cases [21, 19].

In a recent paper [8], we observed that the metamodels of LEMMA's modelling languages and the Jolie programming language have enough contact points to consider their integration.

One of the practical aims of such an integration is to provide a toolchain that supports the transition from MDE-based MSA models, e.g., expressed in LEMMA, to compliant implementations in languages with dedicated support for microservices, like Jolie.

A fundamental piece of said toolchain, which we propose to present at Microservices 2022, is a tool, called LEMMA2Jolie[1], able to convert LEMMA domain models into Jolie APIs. To illustrate how LEMMA2Jolie works, we introduce the core concepts of LEMMA's Domain Data Modelling Language (DDML) in Section 2 and the Jolie API layer in Section 3, followed by the formal encoding in Section 4 which LEMMA2Jolie implements to generate Jolie APIs from a DDML model. Notably, the encoding enables the systematic translation of LEMMA domain models—which, following DDD principles, capture domain-specific types (including operation signatures)—into Jolie APIs.

2 LEMMA Domain Modelling Concepts

LEMMA's DDML enables domain experts and microservice developers to capture domain-specific types of microservices in domain models [18]. Figure 1 shows the core rules of the DDML grammar[2].

[1] Source code available at https://github.com/frademacher/lemma2jolie.

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

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

Figure 1: Simplified grammar of LEMMA's DDML.

The DDML follows DDD to model domain concepts. DDD's Bounded Context pattern [5] is crucial in MSA design as it makes the boundaries of coherent domain concepts explicit, thereby defining their scope and applicability [14]. A LEMMA domain model defines named bounded **context**s (rule CTX in Figure 1). A **context** may specify domain concepts in the form of complex types (CT), which are either structures (STR), collections (COL), or enumerations (ENM).

A **structure** gathers a set of data fields (FLD). 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 [5]. That is, it supports the construction of underspecified models and their subsequent refinement as one gains new domain knowledge [17]. 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 Figure 1) expresses that a structure comprises a notion of domain-specific identity. The **identifier** feature then marks the data fields ($FLDF$) or operations ($OPSF$) of an **entity** which determine its identity. For a detailed presentation of the considered DDD features we refer to [7].

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

The LEMMA listing in Figure 2 shows an example of a LEMMA DDML model [19]. The 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.

The **greyed-out** terms in Figure 1 are LEMMA DDML features we do not consider in this work and leave for future extensions.

LEMMA	Jolie
context BookingManagement **{**	///*@beginCtx(BookingManagement)*
	///*@entity*
structure ParkingSpaceBooking⟨**entity**⟩ **{**	**type** ParkingSpaceBooking **{**
long bookingID⟨**identifier**⟩,	///*@identifier*
double priceInEuro,	bookingID: **long**
	priceInEuro: **double**
	}
function double priceInDollars	**interface** ParkingSpaceBooking_interface **{**
	RequestResponse:
	priceInDollars (ParkingSpaceBooking)(**double**)
}	**}**
}	///*@endCtx*

Figure 2: An example LEMMA Domain Model (taken from [19]) and its encoding as Jolie API.

3 Jolie Types and Interfaces

Jolie interfaces and types define the functionalities of a microservice and the data types associated with those functionalities i.e., the API of a microservice. Figure 3 shows a simplified grammar of Jolie APIs, taken from [13] and updated to Jolie 1.10 (the latest major release at the time of writing).

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

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

An **interface** is a collection of named operations (**RequestResponse**), where the sender delivers its message of type TP_1 and waits for the receiver to reply with a response of type TP_2—although Jolie also supports **OneWay**s, where the sender delivers its message to the receiver, without waiting for the latter to process it (fire-and-forget), we omit them here because they are not used in the encoding (cf. Section 4). 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 [6]. 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 [3].

Based on the grammar in Figure 3, the Jolie listing in Figure 2 (on the right) shows the equivalent of the example LEMMA domain model (on the left) and works as a preview example of the logic behind our encoding, presented in Section 4. 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. LEMMA operation signatures are expressible as **RequestResponse** operations within a Jolie **interface** for the LEMMA domain concept that defines the signatures. For example, we mapped the domain concept *ParkingSpaceBooking* and its operation signature *priceInDollars* to the Jolie interface *ParkingSpaceBooking_ interface* with the operation *priceInDollars*.

4 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 Section 3. Figure 4 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 are similar in nature to methods of OOP, since they operate on data stored in their defining structure. On the other hand, Jolie does not support objects in the OOP sense but rather 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 Section 3 by the mapping of the LEMMA domain concept *ParkingSpaceBooking* and its operation signature *priceInDollars* to the Jolie interface*ParkingSpaceBooking_ interface* with the operation *priceInDollars*.

Given a structure X, we extend the signature of its **procedure**s with a parameter for representing the structure they act on and a return type X for the new state of the structure, essentially turning them into functions that transform the enclosing structure. For instance, we regard a procedure with signature $(Y \times \cdots \times Z)$ in X as a function with type $X \times Y \times \cdots \times Z \to X$. This approach is not new and can be found also in modern languages like Rust [12, 23] and Python [16]. 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 $(\!(\cdot)\!)$ and synthesise the types for the generated operations. The other rules are straightforward.

$$
\begin{array}{lcl}
\llbracket \textbf{context}\ id\ \{\overline{CT}\}\rrbracket & = & ///@beginCtx(id) \\
 & & \llbracket \overline{CT} \rrbracket \\
 & & ///@endCtx \\
\langle\!\langle \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\ \overline{OPS}\}\rangle\!\rangle & = & [\overline{///@STRF}]\ \textbf{interface}\ id\_interface\ \{\overline{\langle\!\langle OPS \rangle\!\rangle_{id}}\} \\
\langle\!\langle \textbf{procedure}\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})\rangle\!\rangle_{id_s} & = & \textbf{RequestResponse}\colon [\overline{///@OPSF}]\ id(id\_type)(id_s) \\
\langle\!\langle \textbf{function}\ (S \mid id_r)\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})\rangle\!\rangle_{id_s} & = & \textbf{RequestResponse}\colon [\overline{///@OPSF}]\ id(id\_type)((\lceil S\rceil \mid id_r)) \\
\llbracket \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\ \overline{OPS}\}\rrbracket & = & \textbf{type}\ \lceil \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\}\rceil \\
 & & \overline{\llbracket OPS \rrbracket_{id}}\ \langle\!\langle \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{OPS}\}\rangle\!\rangle_{id} \\
\llbracket \textbf{procedure}\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})\rrbracket_{id_s} & = & \textbf{type}\ id\_type\colon \textbf{void}\ \{self?\colon id_s\ \overline{\lceil FLD\rceil}\} \\
\llbracket \textbf{function}\ (id_r \mid S)\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})\rrbracket_{id_s} & = & \textbf{type}\ id\_type\colon \textbf{void}\ \{self?\colon id_s\ \overline{\lceil FLD\rceil}\} \\
\llbracket \textbf{collection}\ id\ \{(S \mid id_r)\}\rrbracket & = & \textbf{type}\ id\colon \textbf{void}\ \{\lceil \textbf{collection}\ id\ \{(S \mid id_r)\}\rceil\} \\
\llbracket \textbf{enum}\ id\ \{\overline{id}\}\rrbracket & = & \textbf{type}\ \lceil \textbf{enum}\ id\ \{\overline{id}\}\rceil \\
\lceil \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\}\rceil & = & [\overline{///@STRF}]\ id\colon \textbf{void}\ \{\overline{\lceil FLD\rceil}\} \\
\lceil S\ id\ [\langle\overline{FLDF}\rangle]\rceil & = & [\overline{///@FLDF}]\ id\colon \lceil S\rceil \\
\lceil id_r\ id\ [\langle\overline{FLDF}\rangle]\rceil & = & [\overline{///@FLDF}]\ id\colon id_r \\
\lceil \textbf{collection}\ id\ \{S\}\rceil & = & id{*}\colon \lceil S\rceil \\
\lceil \textbf{collection}\ id\ \{id_r\}\rceil & = & id{*}\colon id_r \\
\lceil \textbf{enum}\ id\ \{\overline{id}\}\rceil & = & id\colon \textbf{string}(enum(\overline{``id"})) \\
\lceil \textbf{int}\rceil & = & \textbf{int} \\
\lceil \textbf{unspecified}\rceil & = & \textbf{undefined}
\end{array}
$$

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

5 Conclusion

This extended abstract summarises our presentation proposal to Microservices 2022, concerning our first step towards the integration of two *language-based approaches to Microservice Architecture (MSA) engineering*, namely the MSA-focused modelling ecosystem LEMMA and the microservice programming language Jolie.

In particular, as done in this document, we intend to briefly introduce LEMMA's Domain Data Modelling Language (DDML; cf. Section 2) and Jolie data types and interfaces (Jolie APIs; cf. Section 3), followed by the presentation of the general rules we followed to define our formal encoding (cf. Section 4). Then, we will present our implementation of the encoding as a code generator, called LEMMA2Jolie, which is applicable in MSA engineering practice to translate LEMMA domain models into Jolie APIs.

The encoding/tool provides a basis to enable a software development process whereby microservice architectures can first be designed with the leading method of Domain-Driven Design using LEMMA's DDML, and then corresponding Jolie APIs are automatically generated. In the presentation, we will detail how programmers can use LEMMA2Jolie to transit from microservice-specific domain models expressed as LEMMA's DDML models into Jolie APIs, which they can then extend and use as guides to produce compliant implementations.

References

[1] D. Ameller et al. Development of service-oriented architectures using model-driven development: A mapping study. *Information and Software Technology*, 62:42–66, 2015. Elsevier.

[2] B. Combemale, R. B. France, J.-M. Jézéquel, B. Rumpe, J. Steel, and D. Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. CRC Press, 2017.

[3] R. Daigneau. *Service Design Patterns*. Addison-Wesley, 2012.

[4] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: Yesterday, today, and tomorrow. In M. Mazzara and B. Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[5] E. Evans. *Domain-Driven Design*. Addison-Wesley, 2004.

[6] T. Freeman and F. Pfenning. Refinement types for ML. In *Proc. of the 1991 Conf. on Programming Language Design and Implementation*, pages 268–277, 1991.

[7] S. Giallorenzo, F. Montesi, M. Peressotti, and F. Rademacher. Model-Driven Generation of Microservice Interfaces: From LEMMA Domain Models to Jolie APIs, 2022.

[8] S. Giallorenzo, F. Montesi, M. Peressotti, F. Rademacher, and S. Sachweh. Jolie and LEMMA: Model-driven engineering and programming languages meet on microservices. In *Coordination Models and Languages*, pages 276–284. Springer, 2021.

[9] C. Guidi and B. 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.

[10] JHipster. Jhipster domain language (jdl), 2022-14-02.

[11] S. Kapferer and O. Zimmermann. Domain-driven service design. In *Service-Oriented Computing*, pages 189–208. Springer, 2020.

[12] S. Klabnik and C. Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[13] F. Montesi, C. Guidi, and G. Zavattaro. Service-oriented programming with jolie. In A. Bouguettaya, Q. Z. Sheng, and F. Daniel, editors, *Web Services Foundations*, pages 81–107. Springer, 2014.

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

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

[16] Python Software Foundation. *The Python Language Reference*. 2021.

[17] F. Rademacher, S. Sachweh, and A. Zündorf. Deriving microservice code from underspecified domain models using DevOps-enabled modeling languages and model transformations. In *2020 46th Euromicro Conf. on Software Engineering and Advanced Applications (SEAA)*, pages 229–236. IEEE, 2020.

[18] F. Rademacher, J. Sorgalla, P. Wizenty, S. Sachweh, and A. Zündorf. Graphical and textual model-driven microservice development. In *Microservices: Science and Engineering*, pages 147–179. Springer, 2020.

[19] F. Rademacher, J. Sorgalla, P. Wizenty, and S. Trebbau. Towards holistic modeling of microservice architectures using LEMMA. In *Companion Proc. of the 15th Europ. Conf. on Software Architecture*. CEUR-WS, 2021.

[20] J. Soldani, D. A. Tamburri, and W.-J. V. D. Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018. Elsevier.

[21] J. Sorgalla, P. Wizenty, F. Rademacher, S. Sachweh, and A. Zündorf. Applying model-driven engineering to stimulate the adoption of devops processes in small and medium-sized development organizations. *SN Computer Science*, 2(6):459, 2021.

[22] B. Terzić, V. Dimitrieski, S. Kordić, G. Milosavljević, and I. Luković. Development and evaluation of MicroBuilder: a model-driven tool for the specification of REST microservice software architectures. *Enterprise Information Systems*, 12(8-9):1034–1057, 2018. Taylor & Francis.

[23] The Rust Foundation. *The Rust Reference*. 2021.