Jolie & LEMMA: Model-Driven Engineering and Programming Languages Meet on Microservices*

Saverio Giallorenzo[1][0000−0002−3658−6395], Fabrizio Montesi[2][0000−0003−4666−901X], Marco Peressotti[2][0000−0002−0243−0480], Florian Rademacher[3][0000−0003−0784−9245], and Sabine Sachweh[3]

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

Abstract In microservices, Model-Driven Engineering (MDE) has emerged as a powerful methodology for architectural design. Independently, the community of programming languages has investigated new linguistic abstractions for effective microservice development. Here, we present the first preliminary study of how the two approaches can cross-pollinate, taking the LEMMA framework and the Jolie programming language as respective representatives. We establish a common ground for comparing the two technologies in terms of metamodels, discuss practical enhancements that can be derived from the comparison, and present some directions for future work that arise from our new viewpoint.

1 Introduction

In microservices, applications emerge as compositions of independently-executable components (*microservices*, or briefly, services), which communicate via message passing [12]. Building microservice systems poses a series of challenges for both design and development, which has motivated two prolific strands of research.

On the side of design, Model-Driven Engineering (MDE) [17] has become a prominent methodology for the specification of service architectures [2]. Frameworks such as LEMMA, MicroBuilder, and MDSL offer modelling languages to design service components that abstract from concrete implementations [32,33,21].

On the side of development, new linguistic abstractions for programming languages are emerging as powerful tools to effectively express the configuration and coordination of microservices. Ballerina and Jolie are examples of such languages [29,27]. In particular, Jolie incorporates ideas from process calculi to ease the programming of workflows and it offers “polyglot” constructs to integrate services written in foreign languages (e.g., Java) [25,27].

So far, results on microservices by the MDE and programming communities have evolved prolifically, yet separately. This is unfortunate since previous research showed great potential in combining programming language and MDE

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

techniques [13,6,11]. In part, we deem this phenomenon due to the few opportunities the two communities have to interact. Case in point, the authors come from the different two communities and met only recently, at the last two editions of the International Conference on Microservices (an event organised specifically to bridge sub-communities of traditional fields that share an interest in microservices). Seminars from both parts evidenced that MDE methodologies and programming languages for microservices share a common conceptual foundation that has never been properly made precise nor leveraged [16,30].

This article is the first step towards bridging conceptually MDE frameworks and programming languages for microservices. As grounding, we take LEMMA [31,32] and Jolie [27] as respective representatives of the two approaches.

The main challenge is that MDE frameworks come with specifications—like LEMMA's metamodels [31,32]—distant from those given for programming languages—some parts of Jolie are described by using process calculi [19,26], and for others there is a reference implementation [24,1]. To address this, we develop the first conceptual metamodel of the Jolie language, drawing from our experience with its formalisations [19,26,10] and reference implementation [24,1].

Having metamodels for both Jolie (from this paper) and LEMMA (from [31,32]) allows for comparing them. We identify some key shared concepts and differences. Interestingly, the differences are complementary perspectives on common concerns, providing fertile ground for future evolutions of both approaches: we sketch extensions of LEMMA induced by Jolie, and vice versa.

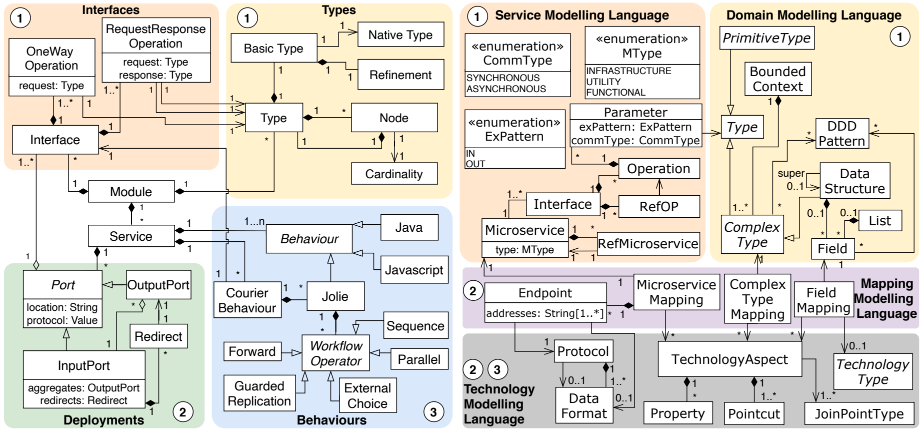
The common footing we establish brings us closer to an ecosystem that coherently combines MDE and programming abstractions to offer a tower of abstractions [23] that supports a step-by-step refinement process from the abstract specification of a microservice architecture (MSA) to its implementation.

2 A Structured Comparison of Jolie and LEMMA

The conceptual metamodels of Jolie (new in this article) and LEMMA (a simplification of the metamodel in [32]) are respectively displayed in Figure 1a and Figure 1b, in UML format. As a basis for comparison, we classify their elements in the three categories commonly found in characterisation and specifications of (micro)services in [14,3,35]: Application Programming Interfaces (APIs) ① and Access Points ②, which, combined, define the public contract of a microservice, and the private internal behaviour ③ that a microservice enacts. We proceed by explaining the metamodels and our comparison following these categories.

2.1 Application Programming Interfaces (APIs)

APIs—originally introduced to provide hardware independence to programs [8]—specify *what* functionalities a microservice offers to clients [12]. Besides loosing coupling, APIs contribute to technology agnosticism, especially when minimising the assumptions made on the technologies used to implement behaviours.



(a) (b)

Figure 1: Core meta-models of Jolie (a) and LEMMA (b).

Jolie conceptualises APIs into Interfaces. An Interface is a collection of Operations, each having its own name and being either: a OneWay operation, where the sender delivers its message to the service but does not wait for it to be processed by the service's behaviour; or a RequestResponse operation, where the sender delivers its message and waits for the receiving service's behaviour to reply with a response. Operations include types for the data structures that can be exchanged through them. A Jolie Type is a tree-shaped data type made of two components: (i) a Basic Type that describes the type of the root of the tree and (ii) a set of Nodes that define the fields of the data structure. Basic Types include a Native Type (primitives like boolean, integer, char, string) and a Refinement that specifies further restrictions on the native type [18]. Nodes are arrays with specified ranges of lengths (Cardinality). Jolie data types, and thus interfaces, are technology agnostic: they model Data Transfer Objects that build on native types generally available in most architectures [9].

LEMMA captures APIs as characterising components of a given Microservice though its Service Modelling Language [32]. Conceptually, a Microservice is a composition of Interfaces, each clustering one or more Operations. LEMMA distinguishes three types of microservices. Functional and utility ones realise domain-specific business logic and reusable generic functionality, respectively. Infrastructure microservices provide technical capabilities, e.g., for service discovery [4]. In LEMMA, a microservice operation is a collection of Parameters, each defined by an exchange pattern (either incoming or outgoing), a communication type (synchronous or asynchronous), and a Type, expressed in the Domain Modelling language. Types can specify some Domain-Driven Design (DDD) semantics in the form of DDD patterns, e.g., the Entity pattern [15] which defines the identifying traits of the Type's inhabitants, e.g., a Person with a name and birthdate but uniquely identified by its social security number.

From the above descriptions—also remarked with the colours of the partitions in Figure 1a and Figure 1b, tagged with ①—APIs are captured similarly in Jolie and LEMMA: they both attribute a paradigm to each operation, either request-response/synchronous or notification/asynchronous, although Jolie at the level of operations and LEMMA at the level of parameters. Types in the two models differ, but, besides LEMMA's DDD semantics, the differences are mostly technical. We exploit the vicinity of views on APIs between Jolie and LEMMA to propose in Section 3 an extension of Jolie that captures DDD patterns of LEMMA's Types. At the conceptual level, Jolie and LEMMA interpret API design from different perspectives. Jolie defines APIs as reusable artefacts, separately from services (a service can then refer to API definitions). In LEMMA, APIs are part of a service definition. This difference makes for an interesting point for building a reference metamodel for microservices, as discussed in Section 3.

2.2 Access Points

When a microservice implements an API, it must make a technological commitment on *where* and *how* its clients can interact with the API. Access points fulfil this need, complementing the public APIs of a microservice with the specification and configuration of the technologies used to (i) format data (how data are structured/marshalled for transmission, e.g., JSON); and (ii) transmit data (where microservices can contact each other and how data are transported among them, e.g., an IP address). Access points are the main elements that increase coupling between microservices, as providers expect clients to include in their technology stacks the technologies used at providers' access points.

Jolie integrates the Port concept (cf. Figure 1a) to support access point definition and configuration. A Jolie Port determines the *location* of an access point in the form of a URI [5] and associates it with a *protocol*. Furthermore, a Port clusters one or more Jolie Interfaces, which define the operations available at that access point (and also complete the public contract of the given microservice).

Jolie distinguishes between InputPorts and OutputPorts (cf. Figure 1a). InputPorts expose a public contract to clients while OutputPorts define access points used in behaviours (cf. Section 2.3) to invoke other microservices.

LEMMA provides the Endpoint concept (cf. Figure 1b) to model locations and technologies of access points, as part of a microservice API. To cope with *technology heterogeneity* in MSA [28], LEMMA treats technology information as a dedicated concern in microservice modelling. Indeed, it provides two modelling languages to (i) organise technology information in dedicated *technology models*; and (ii) assign this information to service models within dedicated *mapping models*. In the context of access points, technology models cluster Protocols and DataFormats (cf. Figure 1b) and make them available to mapping models for determining the technical endpoint characteristics.

Both Jolie and LEMMA support the specification of inbound access points: Jolie InputPorts and LEMMA Endpoints include the definition of the technological choices that define the location and the data formats of access points. However, Jolie and LEMMA differ in how they describe outbound access points:

(i) Jolie uses OutputPorts to specify, in behaviours, the interaction with the access points of other microservices. (ii) LEMMA uses the RefMicroservice concept to specify dependencies among microservices—LEMMA leaves to model processors how to interpret RefMicroservices, e.g., defining deployment precedence.

2.3 Behaviours

Behaviours specify the internal business logic of a microservice, including when the microservice accepts requests from clients and when it invokes other microservices. Jolie allows developers to use Java, JavaScript or Jolie Behaviours to express the behaviour of microservices. Jolie Behaviours are a fragment of the Jolie Language (herein, Jolie Behavioural Language), where microservice behaviours are first-class citizens that, starting from the basic service invocation, one can compose into complex behaviours via high-level *workflow operators* such as Sequence, Parallel, and Guarded Replication. The choice of these operators comes from process calculi and the study of core languages for service-oriented computing [20,27]. In this sense, the Jolie Behavioural Language is a full-fledged specification language for microservices behaviour and, borrowing LEMMA's conceptual organisation, the Jolie Interpreter as its default technology.

LEMMA does not support (yet) complete specifications of microservice behaviours. However, one can use LEMMA's malleable technology modelling language in this direction, defining a suite of technology aspects for declaring general behaviours (e.g., that a microservice is guarded by a circuit breaker) and programming new code generators to produce microservice skeletons.

3 Cross-Fertilisation and Conclusion

The conceptual similarities between Jolie and LEMMA regarding APIs, Access Points and Behaviours identified in this work open the door to cross-fertilisation.

Behaviours in LEMMA As discussed in Section 2.3, LEMMA does not support complete and general specifications of microservice behaviours. We propose to extend LEMMA with hosting of languages for programming behaviours like the Jolie Behavioural Language. In general, one can envision a suite of such guest languages that users can select from or extend. The snippet below illustrates a typical instance of this scenario where a programmer extends a microservice specification with a behaviour for `operation1`. To this end, the programmer imports a behaviour modelling language and a suitable technology for it, in this case, the Jolie Behavioural Language and the Jolie Interpreter.

```
import microservices from "example.services" as ExampleServices
import behaviour_language from "jolie.behaviour_language" as jolie
import technology from "jolie.technology" as jolie_interpreter

@behaviour_language(jolie)
@technology(jolie_interpreter)
ExampleServices::org.example.Microservice {
  operation1() { /* programmed using the given behavioural language" */ }
}
```

This requires a conceptual and technological infrastructure for language integration in some regards similar to quotation [22,7]: APIs modelled in LEMMA need to be rendered available to the guest language and aspects of behaviour interaction and composition need to be made available to LEMMA. This observation suggests that this integration infrastructure could be founded over the core concepts and behaviour operators for service-oriented programming of process calculi that already constitute the foundation of the Jolie Behavioural Language.

DDD patterns in Jolie As mentioned in Section 2.1, we can augment LEMMA's Types with DDD semantics, i.e., constraints imposed by the domain on data structures. Equipping Jolie with such a feature can increase its expressiveness in useful ways, which we discuss briefly below. Comment annotations can capture DDD patterns in Jolie. For example, we can express the Entity pattern (cf. Section 2.1) via the annotation `@entity` below, which associates the property `identity` of the pattern with two sub-nodes of the Person type (SSN and country):

```
/// @entity { identity = [ SSN, country ] }
type Person { SSN: string, country: string( length(3) ), name: string }
```

An immediate result is using DDD patterns to improve documentation, by attaching plain-text explanations of the intended usage of types—in unison with the additional constraints expressed by refinements (cf. `length(3)` above). More advanced integrations can elevate DDD patterns at the level of types, opening the door to runtime and static utilities. For instance, we can have operations "governed" by the semantics of patterns, e.g., to verify entity equality through a unique `assertEquals` operation that checks equality of the components defined in the identity annotation of the entity's type. Similarly, patterns can indicate static constraints on types, e.g., there cannot be two Persons, identified by SSN and country, whose names differ. Pattern-aware execution engines can enforce static constraints at runtime, e.g., keeping track of the (privacy-preserving) "signature" of each identified entity and its correlated immutable values.

Reference Metamodel Jolie and LEMMA are in remarkable conceptual proximity despite their distant origins—namely Programming Languages and MDE. This close match in their conceptual foundations hints at the existence of a reference metamodel for MSAs to be uncovered. This reference metamodel should identify the main concepts of MSA including their basic properties and relationships to each other. Furthermore, it should emerge from the analysis of various existing, yet fragmented bodies of MSA knowledge ranging from pattern collections, over best practices and reference solutions for certain challenges in MSA, to more formal approaches like metamodels for programming and modelling languages. Recent efforts in the area of software deployment automation [34] reveal the potential of reference metamodels as they (i) reify and organise knowledge about a specific subject area; (ii) enable the comparison and reasoning about alternative approaches to the same issue; and (iii) allow identification of migration paths and cost estimation for technology choices. We believe that a reference metamodel for MSAs would be valuable to organise efforts and unify the great number of ad-hoc solutions for recurring challenges and the heterogeneity of MSAs.

References

1. Jolie website (2020), `https://www.jolie-lang.org`
2. 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. Information and Software Technology **62**, 42–66 (2015), elsevier
3. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., et al.: Business process execution language for web services (2003)
4. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: An experience report. In: Celesti, A., Leitner, P. (eds.) Advances in Service-Oriented and Cloud Computing. pp. 201–215. Springer, Cham (2016)
5. Berners-Lee, T., Fielding, R.T., Masinter, L.: Uniform Resource Identifier (URI): Generic syntax. RFC 3986, RFC Editor (2005)
6. Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A.: Modeling language variability with reusable language components. p. 65–75. SPLC '18, Association for Computing Machinery, New York, NY, USA (2018)
7. Cheney, J., Lindley, S., Radanne, G., Wadler, P.: Effective quotation: relating approaches to language-integrated query. In: PEPM. pp. 15–26. ACM (2014)
8. Cotton, I.W., Greatorex Jr, F.S.: Data structures and techniques for remote computer graphics. In: Proceedings of the December 9-11, 1968, fall joint computer conference, part I. pp. 533–544 (1968)
9. Daigneau, R.: Service Design Patterns. Addison-Wesley, first edn. (2012)
10. Dalla Preda, M., Maurizio, G., Claudio, G., Jacopo, M., Fabrizio, M.: Interface-based service composition with aggregation. In: Paoli, F.D., Pimentel, E., Zavattaro, G. (eds.) Service-Oriented and Cloud Computing - First European Conference, ESOCC 2012, Bertinoro, Italy, September 19-21, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7592, pp. 48–63. Springer (2012)
11. Deantoni, J.: Modeling the behavioral semantics of heterogeneous languages and their coordination. In: 2016 Architecture-Centric Virtual Integration (ACVI). pp. 12–18 (2016)
12. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B. (eds.) Present and Ulterior Software Engineering, pp. 195–216. Springer (2017)
13. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. LDTA '12, Association for Computing Machinery, New York, NY, USA (2012)
14. Erl, T.: Service-oriented architecture: concepts, technology, and design. Pearson Education India (2005)
15. Evans, E.: Domain-Driven Design. Addison-Wesley, first edn. (2004)
16. Fernando, A., Giallorenzo, S., Guidi, C., Jayasoma, S., Maschio, B., Mauro, J., Montesi, F., Montesi, M., Peressotti, M., Wallnöfer, M.D., Warusawithana, L.: Ballerina and Jolie: Connecting two frontiers of microservice programming. In: Microservices 2020 (2020)
17. Frankel, D.S.: Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons (2003)
18. Freeman, T., Pfenning, F.: Refinement types for ml. In: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation. pp. 268–277 (1991)

19. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. Fundam. Informaticae **95**(1), 73–102 (2009)
20. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: : A calculus for service oriented computing. In: ICSOC. Lecture Notes in Computer Science, vol. 4294, pp. 327–338. Springer (2006)
21. Kapferer, S., Zimmermann, O.: Domain-driven service design. In: Dustdar, S. (ed.) Service-Oriented Computing. pp. 189–208. Springer (2020)
22. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. Commun. ACM **3**(4), 184–195 (1960)
23. Milner, R.: The tower of informatic models. From semantics to Computer Science (2009)
24. Montesi, F.: JOLIE: a Service-oriented Programming Language. Master's thesis, University of Bologna, Department of Computer Science (2010), `http://amslaurea.cib.unibo.it/1226/`
25. Montesi, F.: Process-aware web programming with Jolie. Sci. Comput. Program. **130**, 69–96 (2016)
26. Montesi, F., Carbone, M.: Programming services with correlation sets. In: Kappel, G., Maamar, Z., Nezhad, H.R.M. (eds.) Service-Oriented Computing - 9th International Conference, ICSOC 2011, Paphos, Cyprus, December 5-8, 2011 Proceedings. Lecture Notes in Computer Science, vol. 7084, pp. 125–141. Springer (2011)
27. Montesi, F., Guidi, C., Zavattaro, G.: Service-Oriented Programming with Jolie, pp. 81–107. Springer, New York, NY (2014)
28. Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly, first edn. (2015)
29. Oram, A.: Ballerina: A Language for Network-Distributed Applications. O'Reilly, first edn. (2019)
30. Rademacher, F.: A non-intrusive approach to extend microservice modeling languages with architecture pattern support. In: Microservices 2020 (2020)
31. Rademacher, F., Sachweh, S., Zündorf, A.: Aspect-oriented modeling of technology heterogeneity in Microservice Architecture. In: 2019 IEEE International Conference on Software Architecture (ICSA). pp. 21–30. IEEE (2019)
32. Rademacher, F., Sorgalla, J., Wizenty, P., Sachweh, S., Zündorf, A.: Graphical and textual model-driven microservice development. In: Microservices: Science and Engineering. pp. 147–179. Springer (2020)
33. 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 Information Systems **12**(8-9), 1034–1057 (2018), taylor & Francis
34. Wurster, M., Breitenbücher, U., Falkenthal, M., Krieger, C., Leymann, F., Saatkamp, K., Soldani, J.: The essential deployment metamodel: a systematic review of deployment automation technologies. SICS Software-Intensive Cyber-Physical Systems **35**(1), 63–75 (Aug 2020)
35. Zimmermann, O.: Microservices tenets. Computer Science-Research and Development **32**(3), 301–310 (2017)