A Toolchain for Checking Domain- and Model-Driven Properties of Jolie Microservices

Saverio Giallorenzo[1,2], Fabrizio Montesi[3], Marco Peressotti[3], Florian Rademacher[4], Sabine Sachweh[5], and Philip Wizenty[5](✉)

[1] Università di Bologna, Bologna, Italy
[2] INRIA, Paris, France
[3] University of Southern Denmark, Sonderborg, Denmark
{fmontesi,peressotti}@imada.sdu.dk
[4] codecentric AG, Solingen, Germany
rademacher@se-rwth.de
[5] University of Applied Science and Arts Dortmund, Dortmund, Germany
{sabine.sachweh,philip.wizenty}@fh-dortmund.de

Abstract. We present Jolie Checker Toolchain (JCT), a plugin-based toolchain for model-code consistency and compliance analysis aimed at helping developers ensure that evolving implementations of Jolie microservices remain aligned with their intended architectural design—specified with domain- and model-driven engineering approaches like LEMMA and MDSL. One of JCT's strengths lies in providing a uniform surface for plugin development, based on the analysis of abstract syntax trees, that leverages existing code generation tools, framing checks as correspondence relations between the expected and actual programs. We present two JCT plugins to respectively check the consistency of domain-driven design annotations on Jolie APIs and verify Jolie code conformity to a given LEMMA data model. We illustrate both plugins via a use case drawn from the Lakeside Mutual architecture.

1 Introduction

Background: microservices and their model-driven development Microservices are small, independent, and reusable distributed components composed via message passing [9,21]. Thanks to their granularity, they can make systems more flexible and scalable [10]. To be effective, the design of microservice architectures needs to be carefully carried out, with challenges including designing appropriate APIs and defining (micro)services around business capabilities [21,28].

To facilitate the design and implementation of microservice architectures, researchers have investigated several tools based on high-level linguistic abstractions. Some of these tools apply Model-Driven Engineering (MDE) [6]: they provide modelling languages for the specification of microservice domain models, following the principles of Domain-Driven Design (DDD) [11]. Examples of MDE tools for microservices include LEMMA [26], MDSL [32], MicroBuilder [29], and

W. Gaaloul et al. (Eds.): ICSOC 2024, LNCS 15405, pp. 161–175, 2025.
https://doi.org/10.1007/978-981-96-0808-9_13

JHipster[1]. Other tools are general programming languages oriented towards the development of services, like Jolie [20] and Ballerina [22]. Jolie, in particular, offers both a technology-agnostic API language and a language for implementing the behaviour of microservices based on a process-algebraic approach [19,20].

Recently, Giallorenzo et al. [16] observed that the high-level constructs to represent services and APIs used in some metamodels for MDE (like LEMMA and MDSL) share strong similarities with linguistic abstractions found in Jolie. This observation paved the way for the development of code generators that, given the model of a microservice architecture, automatically produce skeletons of Jolie implementations with model-compliant APIs [13,15,16,32]. The vision behind these code generators is to support developers in modelling microservice architectures by applying MDE, and then seamlessly switching to a service-oriented programming language to define concrete implementations by refining the (correct by construction) skeletons generated from the source models [15].

Problem statement: keeping service implementations consistent with architecture modelling Software evolves. Once developers translate models into code skeletons (ideally following the above-mentioned correct-by-construction approach), they have to refine the code to reach executable implementations. This refinement, the regular maintenance of the software, and interventions for its performance improvement can induce a drift between the models and their implementation.

Essentially, all these activities share a model-code compliance challenge. After any edit of the artefacts involved (models and code), the developers need to perform the work of comparing the code with its models, to make sure that the former still complies with the properties of the latter. This task suffers of at least two major disadvantages. First, developers lose momentum during the development process as they need to integrate the task of checking the consistency/correspondence of the properties. Second, manual checks do not guarantee the reliability of the process. This phenomenon is further exacerbated by large-scale codebases, where breaking consistency can be as easy as removing a field from the type of a data-transfer object. This calls for automated tool support.

This Work. We present Jolie Checker Toolchain (JCT), a plugin-based toolchain that automates consistency and conformance-violations checks between models and Jolie service implementations. We base our development on an innovative approach to code analysers for compliance. Since there exist already model-to-code generators, we want to avoid redeveloping part of the logic that maps models to service code to obtain a code analyser for consistency/conformance. Contrarily, we leverage the existing generators, avoiding logic duplication and the possible misalignments due to the parallel development of these tools. The code analysis in JCT builds on top of existing code generators, where a model-to-code tool generates code skeletons from a model and the plugin uses the skeletons as specifications for the service code under analysis. Technically, JCT plugins base their comparison on the abstract syntax trees (ASTs)

[1] https://www.jhipster.tech/jdl.

of the model-generated code and the one under scrutiny and implement their checks as correspondence relations between the two pieces of code. To enable the general reuse model-to-code generators, we offer a uniform surface for checkers, which come as plugins.

In this way, JCT can host a plethora of checkers, each verifying the properties of a given model w.r.t.@ the given Jolie code.

We represent the general structure of JCT in Fig. 1. In JCT, developers can compose the different checker plugins in successive stages, where each stage receives the AST of the Jolie program under verification, performs its checks, and outputs its results to the user. The reason behind this structure is, besides accommodating the development of plugins by developers, the integration of JCT within existing Continuous Integration/Deployment (CI/CD) DevOps pipelines.

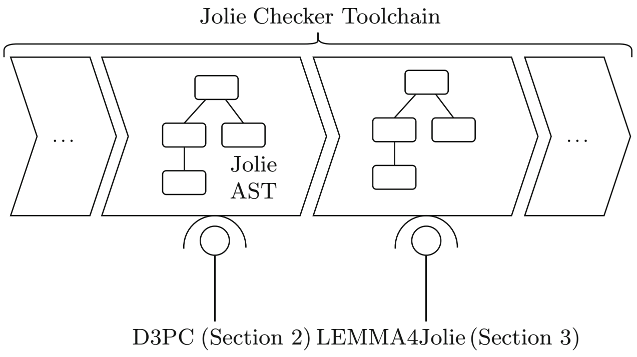

Fig. 1. Schema of the Jolie Checker Toolchain (JCT).

As shown in Fig. 1, we present two illustrative plugins for JCT. The first, introduced in Sect. 2, is a checker for consistency of Domain-Driven Design patterns defined as annotations of Jolie code. The second, described in Sect. 3, leverages the state-of-the-art LEMMA2Jolie code generator (which generates Jolie service code from LEMMA domain models [13]) to implement the LEMMA4Jolie plugin for checking the compliance between LEMMA models and their Jolie implementation. Since we build both plugins from existing work, in Sect. 2 and Sect. 3 we briefly summarise the related background work.

In Sect. 4, we discuss how developers can use JCT and our plugins to check DDD and model-code properties of Jolie programs. This discussion also provides an initial validation of our approach by presenting implementations of a scenario based on a reference MSA from the literature: Lakeside Mutual [32]. Finally, we compare our work with the related one in Sect. 5, and we draw closing remarks and future steps in Sect. 6. The tool and use case presented in this work are available at[2] together with additional examples.

2 DDD Concepts and Patterns in Jolie

2.1 Background: Jolie IDL

Jolie provides an Interface Description Language (IDL) for specifying the operations offered and consumed by each microservice and the types of the semi-structured data accepted and returned by those operations. Jolie IDL is designed

[2] https://github.com/pwizenty/JCT.

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

Fig. 2. Jolie IDL, simplified syntax (types and interfaces).

to be technology-agnostic: its types model Data Transfer Objects (DTOs) built on native types generally available in most architectures [7].

Figure 2 contains the grammar for Jolie IDL taken from [20] and updated to the latest major release at the time of writing (Jolie 1.11). 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. Jolie also supports **oneWay** operations, where the sender delivers its message to the receiver, without waiting for the latter to process it (fire-and-forget); we omit this class of operations for conciseness. 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) like **int**egers, **string**s, etc. Jolie **type** definitions (TD) have a tree-shaped structure to characterise semi-structured data typical of web technologies such as XML, JSON. At their root, we find a basic type (B)—which may include a refinement (R) to express constraints that further restrict the possible inhabitants of the type [12]. The possible branches (or fields) 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.

We can decorate interfaces, types, and their members (operations and fields) with syntactic metadata (MD) using line (*/// [...]*) and multi-line (*(/** [...] */)* documentation comments. These elements are not part of the API but are available to the Jolie interpreter and exposed to plugins by JCT.

2.2 D3PC: Realising DDD Patterns and Their Contracts

Recently, Giallorenzo et al. [16] established a connection between the metamodels of LEMMA and Jolie, paving the way for the principled mapping of DDD concepts and patterns to Jolie developed in consecutive work [14,15]. To elicit and document instances of DDD concepts in Jolie APIs, Giallorenzo at al. [15] introduced *DDD annotations* i.e., syntactic metadata comprised by expressions of the form *@Concept* and *@Concept(params)* for parametric concepts. In *op. cit.*, DDD annotations document the intent of using DDD patterns leaving the onus of correctly applying these patterns to programmers. Our Domain-driven Design Pattern Checker (D3PC) JCT plugin fills this gap. In the remainder, we summarise the patterns found in *op. cit.* and present the pattern-induced verifications performed by D3PC (implementation and a use case are in Sect. 4).

Aggregate and Part. In DDD, aggregates prescribe object graphs requiring their parts to maintain a consistent state [11]. Thus, aggregates are always loaded and stored in a consistent state and within a single transaction. Aggregates and their parts are realised as Jolie types annotated with *@aggregate* and their branches annotated with *@part*. Figure 3 shows an example of an aggregate representing a parking space booking. A DDD aggregate must consist of at least one part and at least one part must be an entity or a value object, which we discuss below. D3PC verifies that types representing aggregates follow these requirements.

```
/// @ctx(BookingManagement)

/** @aggregate
    @entity */
type ParkingSpaceBooking {
    ///@identifier
    bookingID: long
    /// @part
    timeSlot: TimeSlot
    priceInEuro: double
}
/** @valueObject */
type TimeSlot { . . . }                                Jolie
```

Fig. 3. An example of DDD annotations [15, Sect. 2.4].

Entity and Identifier. Instances of DDD entities are distinguishable by a domain-specific identity [11], e.g., a unique ID. In Jolie, DDD entities are realised as types annotated with *@entity* and their identifier as a branch annotated with *@identifier*. D3PC verifies that types representing entities follow these requirements. Continuing our example, we extend our *PSB* aggregate with an identifier (a booking ID) and annotate them accordingly.

Value Object and Domain Event. In DDD, value objects consist of data and logic that do not depend on a domain-specific identity, differently from entities. Therefore, value objects serve as data transfer objects (DTOs) to exchange data between microservices and, in asynchronous scenarios, domain events. These concepts are realised in Jolie by annotating types with *@valueObject* and *@event* respectively. D3PC verifies that types representing value objects and events follow these requirements. The type *TimeSlot*, shown in Fig. 3, is marked as a value object meaning that it can serve as a DTO and cross boundaries between microservices (see bounded contexts).

Bounded Context. In DDD for microservice architectures, the concept of bounded context makes the boundaries for domain concepts explicit. Contexts are realised as modules which are constituted by Jolie files marked with the annotation *@ctx(ContextName)* parametric in the context name. To comply with this concept, Jolie codebases must avoid imports of types and interfaces across modules marked as distinct bounded contexts if they realise DDD concepts other than value object and domain event. Likewise, operations of interfaces meant to be exposed outside a context (even if they are not themselves marked with any

```
context UserManagement {

  structure User ⟨entity⟩ {
    int id ⟨identifier⟩,
    string firstName,
    string lastName,
    Username username
  }
                                        LEMMA
```

```
  structure Username⟨valueObject⟩ {
    immutable string username
  }

  enum Status {
    ENABLED, DISABLED, INACTIVE
  }
}
                                        LEMMA
```

Fig. 4. Illustrative LEMMA domain model: the UserManagement bounded context.

DDD annotation) must not expose return types that realise DDD concepts other than value object and domain event. D3PC verifies all that the elements of a module representing a context are used accordingly.

3 LEMMA Domain Models and Jolie

The second plugin we present builds on work on LEMMA and Jolie [13] for developing MSA-based software systems. In the following, we introduce in Sect. 3.1 LEMMA's domain data modelling language for the DDD-supported design of MSAs by Domain Experts and Service Developers [21]. In Sect. 3.2, we elaborate on using LEMMA's domain data models to create Jolie source code artefacts by applying methods and techniques from Model-driven Engineering (MDE). Then, in Sect. 3.3, we introduce the methodology followed to implement the correspondence relation that underpins the comparison between the LEMMA domain data model and the Jolie service under verification.

3.1 Background: LEMMA Domain Data Modeling Language

The LEMMA ecosystem [25] for Model-Driven Engineering (MDE) offers a range of modeling languages that capture different concerns in Microservices Architecture (MSA) engineering from the perspective of architecture stakeholders. Developers can integrate these MSA models via an import mechanism—supporting referencing between elements of diverse models—to enhance the information content of captured viewpoints in a microservice architecture, also facilitating reuse.

Domain Data Modelling Language (DDML). The Domain Data Modelling Language (DDML) is a language that enables domain experts and microservice developers to construct models addressing concerns from the Domain Viewpoint. Domain experts and developers can use DDML to create domain models that capture relevant concepts from the application domain. These concepts can be enhanced with patterns from DDD [11]—a popular methodology in microservice development. DDML also implements LEMMA's type system, making domain concepts usable for typing parameters of modelled microservice operations. In

$$
\begin{aligned}
&\llbracket \textbf{context}\ id\ \{\overline{CT}\}\rrbracket && = /**\ @ctx(id)\ /*\ \overline{\llbracket CT\rrbracket}\\
&\llparenthesis \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\ \overline{OPS}\}\rrparenthesis && = [/**\ \overline{@STRF}\ \ */]\ \textbf{interface}\ id\_interface\ \{\overline{\llparenthesis OPS\rrparenthesis_{id}}\}\\
&\llbracket \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\ \overline{OPS}\}\rrbracket && = \textbf{type}\ \lceil\!\lceil \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\}\rceil\!\rceil\\
& && \quad \overline{\llbracket OPS\rrbracket}_{id}\ \llparenthesis \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{OPS}\}\rrparenthesis_{id}\\
&\llbracket \textbf{procedure}\ id\ [\langle\overline{OPSF}\rangle]\ (\overline{FLD})\rrbracket_{id_s} && = \textbf{type}\ id\_type:\ \textbf{void}\ \{self?:\ id_s\ \overline{\lceil\!\lceil FLD\rceil\!\rceil}\}\ \ldots\\
&\lceil\!\lceil \textbf{structure}\ id\ [\langle\overline{STRF}\rangle]\ \{\overline{FLD}\}\rceil\!\rceil && = [/**\ \overline{@STRF}\ \ */]\ id:\ \textbf{void}\ \{\overline{\lceil\!\lceil FLD\rceil\!\rceil}\}\\
&\lceil\!\lceil S\ id\ [\langle\overline{FLDF}\rangle]\rceil\!\rceil && = [/**\ \overline{@FLDF}\ \ */]\ id:\ \lceil\!\lceil S\rceil\!\rceil\\
&\ldots\\
&\lceil\!\lceil \textbf{unspecified}\rceil\!\rceil && = \textbf{undefined}
\end{aligned}
$$

Fig. 5. Extract of the Jolie encoding for LEMMA's domain modeling concepts [16].

this way, typing relationships help to identify the portion of the application domain on which a microservice operates and for which it is responsible.

We report an example in Fig. 4 of DDML model with integrated concepts from DDD of a UserManagement context relating to the bounded context [11].

The UserManagement bounded context contains two data structures that capture domain information. The User data structure clusters information data about the user of the software system, e.g., the primitive type attributes firstName and lastName. The User data structure contains the complex attribute UserName, which is specified as a type to make the domain concept explicit.

The domain model contains DDD concepts, e.g., **entity** assigned to User relates to the eponymous DDD concept, whose **identifier** attribute is id.

3.2 Background: From Domain Models to Jolie APIs

Our LEMMA4Jolie plugin uses in one of its steps LEMMA2Jolie [13,15], a *Model-to-Text Transformation* that generates Jolie source code from LEMMA domain models. Figure 5 contains an exceprt of this transformation.

```
/// @ctx(User)
/// @entity
type User {
    ///@identifier
    id: int
    firstName: string
    lastName: string
    username: Username
}
                                    Jolie
```

```
/// @valueObject
type Username {
    username: string
}
type Status {
     literal : string(enum([
         "ENABLED", "DISABLED",
         "INACTIVE"
     ])) }
                                    Jolie
```

Fig. 6. Example Jolie code generated from the model in Fig. 4 by LEMMA2Jolie.

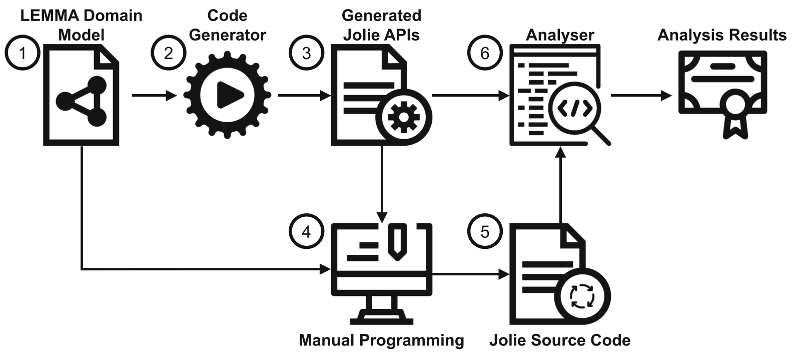

Fig. 7. Schema of the LEMMA4Jolie compliance routine.

The encoding specifies, e.g., that a LEMMA `context` is transformed into the Jolie annotations of *@ctx(...)* or that the complex type of data structure becomes a correspondingly-structured Jolie **type**. Figure 6 shows the Jolie source code generated by the transformation of the LEMMA domain model from Fig. 4.

Briefly, in Fig. 6, we find the Jolie types User and Username from the related LEMMA data structures, including the DDD concepts applied on them as Jolie comments, starting with a @-symbol to indicate DDD-specific annotations.

3.3 LEMMA4Jolie: Compliance of Jolie APIs with Domain Models

We draw in Fig. 7 the steps that characterise our LEMMA4Jolie JCT plugin, which includes code generation, manual programming, and the overall development process of the software systems.

Step ① consists of modelling the software systems domain concepts using LEMMA's domain data modelling language. The domain model, constructed by domain experts and service developers, is part of the software systems specification and, therefore, serves as the Architectual Design [30] of the system.

In step ②, we focus on the software systems' initial development. This step uses a code generator (LEMMA2Jolie) to generate the skeleton implementation of the system as Jolie source code, referenced in ③. Note that the approach does not necessarily requires the developer to start from the generated code, which they can also write from scratch (as shown in ④). Hence, the code generated from ② is only mandatory for the LEMMA4Jolie plugin as the reference specification to compare against the provided Jolie implementation.

Step ④ indicates the possible manual modifications of the Jolie code which we are going to check. This step ranges on all the ways developers can manually intervene on the Jolie code, including from-scratch implementations of the model to modifications of the model-generated code—e.g., to realise architectural designs from other models or improve performance, fix bugs, and integrate new features. All these modifications result in step ⑤, which references the Jolie source code that we want to check for correspondence with the LEMMA model.

Step ⑥ implements the correspondence relation at the heart of the compliance checks between the LEMMA2Jolie-generated Jolie specification and the Jolie source code. Concisely, the relation checks that each Jolie type and interface found in the LEMMA-generated source has a corresponding type and interface in the code under verification, including the related annotations for DDD patterns. Any structural deviation between the former and the latter results in a violation reported to the developer—the report indicates the line (if any) where the code presents a violation and its explanation (e.g., a missing expected annotation).

4 Implementation and Use Case

To illustrate the usage of JCT and the D3PC and LEMMA4Jolie plugins and provide a preliminary validation of their design, we apply them to an illustrative use case based on the textbook MSA Lakeside Mutual [32]. Specifically, we modelled in LEMMA the use case to elicit DDD Patters, we generated the Jolie APIs via LEMMA2Jolie, introducing to the latter illustrative modifications to exemplify the usage of the plugins. JCT is a terminal tool designed for both interactive and automatic reporting within a CI/CD infrastructure. Within a CI/CD, JCT shall stop the process when it detects a violation, while the interaction with the users mirrors that of a compiler, where warnings and errors carry the reason behind the reported violation to help the users identify the offending lines (if any). We exemplify both compliance/conformance reporting and the fixing of illustrative violations with the Jolie code in Fig. 8, stored within a `example.ol` file. Using D3PC, we obtain the following output.

```
///@ctx(User)
/** @entity
    @aggregate */
type Customer {
 customerRef: int
 firstName: string
 lastName: string
 ///@identifier
 username: Username
}
...                                  Jolie
```

Fig. 8. `example.ol`

```
+-------------------------------------------------------------------+
| DDD      | Line | Description                          | Status    |
+-------------------------------------------------------------------+
| Username | 16   | Value Object has no entity annotation | CORRECT   |
| Customer | 7    | Entity has an identifier             | CORRECT   |
| Customer | 7    | Aggregate is an entity               | CORRECT   |
| Customer | 7    | Aggregate misses a part              | VIOLATION |
+-------------------------------------------------------------------+
```

From the report, the code complies with the property of *@entity*, i.e., Customer has an *@identifier* subnode (username). Similarly, *@aggregate* is correctly attributed to Customer—i.e., an *@entity* can also be an *@aggregate*—but we have a `VIOLATION` because we need to associate an aggregate with one or more *@part*s. Since Username (omitted for brevity) is a complex type aggregated by Customer, we mark username *@part* to fix the violation. Running JCT with D3PC again confirms we fixed the problem.

```
type Customer {
 [...]
 /** @identifier
     @part */
 username: Username
}                                   Jolie
```

```
+-----------------------------------------------------------------------+
| DDD      | Line | Description                               | Status   |
+-----------------------------------------------------------------------+
| Username | 17   | Value Object has no entity annotation | CORRECT  |
| Customer | 7    | Entity has an identifier              | CORRECT  |
| Customer | 7    | Aggregate is an entity                | CORRECT  |
| Customer | 7    | Aggregate misses a part               | CORRECT  |
+-----------------------------------------------------------------------+
```

The LEMMA4Jolie plugin needs a source Jolie file generated by LEMMA2Jolie from the LEMMA data model intended for the considered Jolie file. Specifically, the LEMMA model for our example is the following on the left (a slight extension of the one from Fig. 3), of which we report the Jolie code generated on the right (lightly reformatted for presentation).

```
context User {

 structure Customer ⟨entity,aggregate⟩ {
  int customerRef ⟨identifier⟩,
  string firstName,
  string lastName,
  Username username ⟨part⟩
 }

 ...
}                                   LEMMA
```

```
///@ctx(User)
/** @entity
     @aggregate */
type Customer {
 ///@identifier
 customerRef: int
 firstName: string
 lastName: string
 ///@part
 username: Username
}                                   Jolie
```

Using LEMMA4Jolie, we obtain the following output.

```
+---------------------------------------------------------------------------------+
| Name        | Line | Description                         | Status    | Type       |
+---------------------------------------------------------------------------------+
| Customer    | 7    | Matching annotations                | CORRECT   | ANNOTATION |
| customerRef | 9    | Type present                        | CORRECT   | TYPE       |
| customerRef | 9    | Annotation variation, Identifier    | VIOLATION | ANNOTATION |
| firstName   | 10   | Type present                        | CORRECT   | TYPE       |
| lastName    | 12   | Type present                        | CORRECT   | TYPE       |
| username    | 13   | Type present                        | CORRECT   | TYPE       |
| username    | 13   | Annotation variation, Identifier    | VIOLATION | ANNOTATION |
| username    | 13   | Matching annotation                 | CORRECT   | ANNOTATION |
+---------------------------------------------------------------------------------+
```

By comparing the codes of `example.ol` (updated) and the related LEMMA model, the two programs differ by the assignment of the *@identifier*. Indeed, in `example.ol`, we assign the *@identifier* annotation to the username leaf. This assignment was fine for the D3PC checker, which just verified that the Customer *@entity* had an *@identifier* subcomponent. Contrarily, that association violates the LEMMA model, which specifies that customerRef is the identifier of the Customer entity. LEMMA4Jolie reports the above error at two locations. From the top of the output, we first find that customerRef is missing the *@identifier* annotation, which is incorrectly attributed to the username—reported as an unexpected annotation by the tool as the second violation.

To make it valid for the LEMMA model, we need to fix the annotations of the customerRef and username nodes as shown in the Jolie file reported above.

5 Related Work

Looking at related work from the perspective of model-driven engineering, we see our proposal close to architecture-conformance analysis [18,23], which aims to check whether a software's architecture is consistent with its intended organization of structural elements (such as packages, subsystems, and layers) to reduce software erosion. Works the closest to ours, on microservices, include measures to assess architecture conformance to microservice patterns [31]

Architecture conformance checking is an approach to address erosion in software development. Caraccio et al. introduced a unified approach for conformance checking [5]. The approach is based on the domain-specific language Dicto to specify architecture rules to support the software developer in formalising functional and non-functional aspects of the software, e.g., that the architecture design of the software systems conforms to its implementation. Probo, a tool coordination framework, uses the specified rules to check the conformance of the software system. Our work differs from the approach described in [5] in various factors. The main difference is that our approach does not require additional artefacts in the software development process, such as the architecture rules defined via Dicto. JCT plugins (e.g., D3PC and LEMMA4Jolie) work with artefacts that already exist in the software development process. Both approaches enable the integration of additional tools to analyse the architecture conformance. However, the approach described by Caracciolo et al. [5] addresses architecture conformance generally, whereas our approach is tailored for microservices.

Buckley et al. presented JITTAC [4], a tool that leverages real-time reflection modelling to inform the software developers about the architectural consequences of their programming actions and, therefore, try to raise awareness towards promoting consistency between the design and implementation of the software system. JITTAC relies on an architectural model that consists of the different components of the software systems and their relationship with each other. The components of the architecture model are mapped to the various source code artefacts to monitor their evolution. The difference between JCT and JITTAC lies in the latter considering the extensibility, scope, and integration into the development process. Both tools aim to improve the architecture compliance of the software systems. However, JITTAC is directly integrated into the Eclipse IDE and, therefore, forces the IDE usage with a fixed number of analyses. In contrast, JCT does not have those constraints for developing the software system due to the loose integration via terminal or into the CI/CD pipeline. Additionally, JCT focuses specifically on the needs of MSA developers. JITTAC supports only a central model specifying the architecture, which is a broader approach and lacks detailed MSA-specific concepts.

Another tool, InMap [27], focuses on automated interactive Code-to-Architecture Mappings recommendations. The tool uses reflection modelling to recommend a mapping between source code artefacts to their corresponding model specifications. The mappings are then used to ensure the conformance between the software systems architecture and implementation. Diferenlty from JCT, InMap does not analyse the compliance between design and implementation but rather specifies a possible mapping between the two.

6 Discussion and Conclusion

We presented JCT, a plugin-based toolchain for checking the respect of MDE and DDD properties of Jolie programs. The main idea behind JCT is leveraging the AST of Jolie programs to abstract away syntactic details and concentrate on the shape of programs. We conjecture that this abstraction step is particularly suitable for efficiently integrating different model-driven engineering code-generation technologies within a single toolchain. Indeed, besides providing facilities to obtain ASTs designed for the analysis of Jolie programs, JCT supports a plugin development methodology where constraints induced by models manifest as correspondence checks between Jolie code generated from models and the code developed by programmers. We present two illustrative JCT plugins namely, D3PC and LEMMA4Jolie, which both work on Jolie APIs. D3PC performs a consistency check among DDD annotations of Jolie code while LEMMA4Jolie verifies the conformance of a Jolie program w.r.t. a given LEMMA data model.

We conducted a preliminary validation of our toolchain with a use case consisting of the textbook MSA Lakeside Mutual [32] drawing positive results that support the design of JCT. However, the preliminary nature of this use case limits our conclusions and calls for more systematic investigations. To address the scope and limitations of our use case, we are planning to pursue two main directions: 1) retrace the steps in Fig. 4 with a larger sample of use cases and reference MSAs and 2) conduct qualitative analyses with Jolie programmers not involved in the development of JCT. We leave these as future work since the size of such studies evades the scope of this work.

We cast our work within the larger effort to provide Jolie programmers with tools to help them check properties and test their applications [13,17] in CI/CD pipelines. Immediate future steps to evolve JCT regard the implementation of new plugins for tools that already support from-model Jolie code generation, like MDSL [32], but also integrate checkers that analyse code looking for “smells” of microservices, e.g., on security [24,30].

While, in this paper, we provide a concrete implementation of the model-code consistency toolchain methodology targetting the Jolie language, we argue that the approach we propose is generally applicable to any language whose ecosystem of tools includes model-to-code generators [1]. Indeed, we conjecture that one can rely on the approach we introduced with JCT not only on paradigmatic programming languages for microservices like Jolie but also with libraries used to inject service-oriented concerns into programming languages based on other programming paradigms, e.g., with Java and the Spring framework[3]. Hence, since LEMMA can generate Java/Spring microservice skeletons, one could build a tool similar to JCT for Java/Spring microservices. A concrete step in this direction is investigating the usage of the Java Checker Framework [8] for implementing a LEMMA4Jolie-like plugin for Java/Spring-based microservices that leverages a LEMMA-to-Java/Spring code generator to compare Java/Spring programs against their expected Java-based specifications. With more support from LEMMA generators, one can envision a generic plugin for performing the same ASTs comparison performend by LEMMA4Jolie but for arbitrary languages: instead of relying on deep integration with a generator for a specific language (as in the case of LEMMA4Jolie), such JCT plugin would work in tandem with LEMMA and use generic markers left by its generators to identify the 'generated gap'. Widening our scope, we conjecture that our approach is not limited to one language per architecture and one could use a multi-target model framework, like LEMMA, to check in the same pipeline a mixed codebase of microservices, e.g., written in Jolie and Java/Spring—this extension would require adding annotations to indicate which microservices are written in which language. One could further broaden the coverage of the models and target languages. For example, one could integrate the tool by Bucchiarone et al. [3] to target also container configuration files (viz. Docker files) or the one by Belguidoum et al. [2] to cover IoT deployments.

Acknowledgements. Partially supported by the research project FREEDA (CUP: I53D23003550006) funded by the framework PRIN 2022 (MUR, Italy), the French ANR project SmartCloud ANR-23-CE25-0012, and Villum Fonden (grant no. 29518). Co-funded by the European Union (ERC, CHORDS, 101124225). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

1. Balasubramanian, K., Gokhale, A.S., Karsai, G., Sztipanovits, J., Neema, S.: Developing applications using model-driven design environments. Computer **39**(2), 33–40 (2006). https://doi.org/10.1109/MC.2006.54
2. Belguidoum, M., Gourari, A., Sehili, I.: MDMSD4IoT a model driven microservice development for IoT systems. In: Fournier-Viger, P., Yousef, A.H., and Bellatreche, L. (eds.) MEDI 2022. LNCS, vol. 13761, pp. 176–189. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-21595-7_13

[3] https://spring.io/microservices.

3. Bucchiarone, A., Soysal, K., Guidi, C.: A model-driven approach towards automatic migration to microservices. In: Bruel, J., Mazzara, M., and Meyer, B. (eds.) DEVOPS 2019. LNCS, vol. 12055, pp. 15–36. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-030-39306-9_2
4. Buckley, J., Mooney, S., Rosik, J., Ali, N.: JITTAC: a just-in-time tool for architectural consistency. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 1291–1294 (2013)
5. Caracciolo, A., Lungu, M.F., Nierstrasz, O.: A unified approach to architecture conformance checking. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture, pp. 41–50 (2015)
6. Combemale, B., France, R.B., Jézéquel, J.-M., Rumpe, B., Steel, J., Vojtisek, D.: Engineering Modeling Languages: Turning Domain Knowledge into Tools. CRC Press (2017)
7. Daigneau, R.: Service Design Patterns. Addison-Wesley (2012)
8. Dietl, W., Dietzel, S., Ernst, M.D., Muslu, K., Schiller, T.W.: Building and using pluggable type-checkers. In: Taylor, R.N., Gall, H.C., Medvidovic, N. (eds.) Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, 21–28 May 2011, pp. 681–690. ACM (2011). https://doi.org/10.1145/1985793.1985889
9. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B. (eds.) Present and Ulterior Software Engineering, pp. 195–216. Springer (2017)
10. Dragoni, N., Lanese, I., Larsen, S.T., Mazzara, M., Mustafin, R., Safina, L.: Microservices: how to make your application scale. In: Petrenko, A.K., Voronkov, A. (eds.) Perspectives of System Informatics, pp. 95–104. Springer, Cham (2018)
11. Evans, E.: Domain-Driven Design. Addison-Wesley (2004)
12. Freeman, T., Pfenning, F.: Refinement types for ML. In: Proceedings of the 1991 Conference on Programming Language Design and Implementation, pp. 268–277 (1991)
13. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F.: LEMMA2Jolie: a tool to generate microservice APIs from domain models. Sci. Comput. Program. **228**, 102956 (2023). https://doi.org/10.1016/J.SCICO.2023.102956
14. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F.: Model-driven code generation for microservices: service models. In: Dorai, G., et al. (eds.) Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022). Open Access Series in Informatics (OASIcs), 6:1–6:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl (2023). https://doi.org/10.4230/OASIcs.Microservices.2020-2022.6
15. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F.: Model-driven generation of microservice interfaces: from LEMMA domain models to Jolie APIs. In: Beek, M.H. ter, Sirjani, M. (eds.) COORDINATION 2022, DisCoTec 2022. LNCS, vol. 13271, pp. 223–240. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-08143-9_13
16. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S.: Jolie and LEMMA: model-driven engineering and programming languages meet on microservices. In: Coordination Models and Languages, pp. 276–284. Springer (2021)
17. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Unwerawattana, N.: JoT: a Jolie framework for testing microservices. In: Jongmans, S., Lopes, A. (eds.) COORDINATION 2023, DisCoTec 2023. LNCS, vol. 13908, pp. 172–191. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-35361-1_10

18. Knodel, J., Popescu, D.: A comparison of static architecture compliance checking approaches. In: 2007 Working IEEE/IFIP Conference on Software Architecture (WICSA 2007), p. 12 (2007)
19. Montesi, F., Carbone, M.: Programming services with correlation sets. In: Kappel, G., Maamar, Z., Nezhad, H.R.M. (eds.) ICSOC 2011. LNCS, vol. 7084, pp. 125–141. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25535-9_9
20. Montesi, F., Guidi, C., Zavattaro, G.: Service-oriented programming with Jolie. In: Bouguettaya, A., Sheng, Q.Z., Daniel, F. (eds.) Web Services Foundations, pp. 81–107. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-7518-7_4
21. Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly (2015)
22. Oram, A.: Ballerina: A Language for Network-Distributed Applications. O'Reilly (2019)
23. Passos, L., Terra, R., Valente, M.T., Diniz, R., Mendonča, N.: Static architecture-conformance checking: An illustrative overview. IEEE Softw. **27**(5), 82–89 (2009)
24. Ponce, F., Soldani, J., Astudillo, H., Brogi, A.: Smells and refactorings for microservices security: a multivocal literature review. J. Syst. Softw. **192**, 111393 (2022). https://doi.org/10.1016/J.JSS.2022.111393
25. Rademacher, F.: A Language Ecosystem for Modeling Microservice Architecture. PhD thesis, Universität Kassel (2022)
26. 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, Cham (2020)
27. Sinkala, Z.T., Herold, S.: InMap: automated interactive code-to-architecture mapping recommendations. In: 2021 IEEE 18th International Conference on Software Architecture (ICSA), pp. 173–183 (2021)
28. Soldani, J., Tamburri, D.A., Heuvel, W.-J.V.D.: The pains and gains of microservices: a systematic grey literature review. J. Syst. Softw. **146**, 215–232 (2018)
29. 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. Enterpr. Inf. Syst. **12**(8–9), 1034–1057 (2018)
30. Wizenty, P., et al.: Towards resolving security smells in microservices, model-driven. In: Fill, H., Mayo, F.J.D., Sinderen, M. van, Maciaszek, L.A. (eds.) Proceedings of the 18th International Conference on Software Technologies, ICSOFT 2023, 10–12 July 2023, pp. 15–26. SCITEPRESS (2023)
31. Zdun, U., Navarro, E., Leymann, F.: Ensuring and assessing architecture conformance to microservice decomposition patterns. In: Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, 13–16 November 2017, Proceedings, pp. 411–429 (2017)
32. Zimmermann, O., Stocker, M., Lübke, D., Zdun, U., Pautasso, C.: Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges. Addison-Wesley (2023)