# A Model-Driven Approach Towards Automatic Migration to Microservices

Antonio Bucchiarone[1(✉)], Kemal Soysal[2], and Claudio Guidi[3]

[1] Fondazione Bruno Kessler, Trento, Italy
bucchiarone@fbk.eu
[2] LS IT-Solutions GmbH, Berlin, Germany
kemal.Soysal@ls-it-solutions.de
[3] italianaSoftware s.r.l., Imola, Italy
cguidi@italianasoftware.com

**Abstract.** Microservices have received and are still receiving an increasing attention, both from academia and the industrial world. To guarantee scalability and availability while developing modern software systems, microservices allow developers to realize complex systems as a set of small services that operate independently and that are easy to maintain and evolve. Migration from monolithic applications to microservices-based application is a challenging task that very often it is done manually by the developers taking into account the main business functionalities of the input application and without a supporting tool. In this paper, we present a model-driven approach for the automatic migration to microservices. The approach is implemented by means of JetBrains MPS, a text-based metamodelling framework, and validated using a first migration example from a Java-based application to Jolie - a programming language for defining microservices.

## 1 Introduction

The life cycle of an application is bound to changes of domain and technical requirements. Non functional requirements as scalability and availability may lead to a rewrite of the application as is for a new architecture or programming language. DevOps [1] and Microservices-based Applications (MSA) [2,3] appear to be an indivisible pair for organizations aiming at delivering applications and services at high velocity. The philosophy may be introduced in the company with adequate training, but only if certain technological, organizational and cultural prerequisites are present [4–6]. If not, the prerequisites should be developed to guarantee adequate instruments to model and verify software systems and support developers all along the development process in order to deploy correct software.

Microservices allow developers to break up monolithic applications (MA) in a set of small and independent services where each of them represents a single business capability and can be delivered and updated autonomously without any impact on other services and on their releases. In common practice, it is


J.-M. Bruel et al. (Eds.): DEVOPS 2019, LNCS 12055, pp. 15–36, 2020.
https://doi.org/10.1007/978-3-030-39306-9_2

also expected that a single service can be developed and managed by a single team [7,8]. Microservices [8] recently demonstrated to be an effective architectural paradigm to cope with scalability in a number of domains [9], however, the paradigm still misses a conceptual model able to support engineers starting from the early phases of development. Several companies are evaluating pros and cons of a migrating to microservices [6].

Model-driven software development [10,11] supports expressing domain requirements regarding contained data, function points, workflows, configurations, requirement tracking, test cases, etc. by appropriate domain specific languages [12]. In this respect, this work discusses the provision of an model-driven approach for the automatic migration of monolithic applications to microservices. In particular, domain-specific languages (DSLs) [13] allow the definition and deployment of microservices, while model transformations are exploited to automatize the migration and the containerization phases.

The implementation of the migration framework is realised by means of JetBrains MPS (briefly, MPS)[1]. MPS is a meta-programming framework that can be exploited as modelling languages workbench, it is text-based, and provides projectional editors [14]. The choice of MPS is due to the inherent characteristics of MSAs, which are by nature collections of small services that must interacts to satisfy the overall business goal. In this respect, graphical languages do not scale with the complexity of the MSAs. Moreover, MPS smoothly supports languages embedding, such that our definition of microservice mining, specification, and deployment phases for the migration that are easily implemented.

As a validation of our approach, we have migrated a simple Java application in the corresponding Jolie[2] [15] microservices deployed inside the Docker[3] container.

### 1.1 Structure of the Paper

The remainder of the paper is organized as follows: Sect. 2 presents the basics about microservices, the metamodelling framework used in the proposed approach and the Jolie language for defining and deploying microservices. Starting from this preliminary information, Sect. 3 surveys related works, Sect. 4 presents the migration framework proposed, and Sect. 5 shows its implementation. We conclude the paper with final remarks in Sect. 6.

## 2 Background

### 2.1 Microservices

Microservices [8] is an architectural style originating from Service-Oriented Architectures (SOAs) [16]. The main idea is to structure systems by composing small independent building blocks communicating exclusively via message

[1] http://www.jetbrains.com/mps/.
[2] http://www.jolie-lang.org/.
[3] https://www.docker.com.

passing. These components are called *microservices*. The characteristic differentiating the new style from monolithic architectures and classic Service-Oriented is the emphasis on *scalability*, *independence*, and *semantic cohesiveness* of each unit constituting the system.

Each microservice it is expected to implement a single *business capability*, bringing benefits in terms of service maintainability and extendability. Since each microservice represents a single business capability, which is delivered and updated independently, discovering bugs or adding a minor improvements do not have any impact on other services and on their releases.

Microservices have seen their popularity blossoming with an explosion of concrete applications seen in real-life software [17]. Several companies are involved in a major refactoring of their backend systems in order to improve scalability [9].

Such a notable success gave rise to academic and commercial interest, and ad-hoc programming languages arose to address the new architectural style [18]. In principle, any general-purpose language could be used to program microservices. However, some of them are more oriented towards scalable applications and concurrency [19]. The Jolie programming language (see Sect. 2.4), crystalizes the basic mechanisms of service oriented computing within a unique linguistic domain in order to simplify the design and the development phases of microservices. As another advantage, Jolie has already a community of users and developers [20] and it has been validated in production environments. Finally, Jolie is available on Docker and is possible to develop and run a microservice inside a container[4].

### 2.2 Model-Driven Engineering and Domain Specific Languages

Model-Driven Engineering (MDE) [12] is a software engineering methodology that proposes to shift the focus of the development from coding to modelling. The goal is to reduce the complexity of software development by raising the level of abstraction, analyzing application properties earlier, and introducing automation in the process. In fact, models are expected to allow domain experts to reason about a certain solution by means of concerns closer to their area of expertise than to implementation details. Moreover, automated mechanisms, i.e. model transformations, can manipulate those models to evaluate attributes of the application and/or to generate implementation code.

Domain Specific Languages (DSLs) have been introduced with the aim of meeting the needs of particular software applications, industry or business challenges that would be less effectively addressed by using mainstream general-purpose languages. In fact, DSLs are languages introduced for expressing problems by using terms closer to a particular domain of application [21].

The definition of DSLs can be challenging due to a number of reasons, notably the nature of the specific domain, how the concepts should be interconnected to ease the modelling activity without sacrificing the quality of the produced

[4] https://jolielang.gitbook.io/docs/containerization/docker.

models, what kind of concrete syntaxes the users desire to exploit, and so forth. To support the development of DSLs, it is common practice to use a language workbench, that is a toolkit supporting the definition of various aspects of the DSL under development (syntax, semantics, validation constraints, generators) as well as user interaction features (code completion, syntax coloring, find usages, refactorings, etc.).

### 2.3 JetBrains MPS: A Text-Based Metamodelling Framework

MPS by JetBrains[5] is a text-based meta-programming system that enables language oriented programming [22]. MPS is open source and is used to implement interesting languages with different notations [14]. In particular, based on MPS *BaseLanguage* it is possible to define new custom languages through extension and composition of *concepts* [23]. A new language is composed by different *aspects* making its specification modular and therefore easy to maintain [13,24]. Notably, the *Structure Definition* aspect is used to define the Abstract Syntax Tree (AST) of a language as a collection of *concept*s. Each concept is composed of properties, children, and relationships, and can possibly extend other concepts. The *Editor Definition* aspect deals with the definition of the concrete syntax for a DSL: it specifies both the notation (i.e., tabular, diagram, tree, etc.) and the interaction behavior of the editor. The *Generators Definition* aspect is used to define the denotational semantics for the language concepts. In particular, two kinds of transformation are supported: (1) AST to text (model-to-text), and (2) AST to AST (model-to-model). Other aspects like the *Type System Definition*, the *Constraints Definition*, etc. are provided. For the sake of brevity, we refer the reader to [25] and [26].

### 2.4 Jolie Language for Microservices

Jolie [15] is a programming language which offers a native linguistic tool for defining microservices following a structured service oriented paradigm. In Jolie some basic concepts of service oriented computing have a direct representation within the primitives of the language. In particular:

- it provides an integrated syntax for defining API interfaces and types;
- it provides specific communication primitives for dealing with communication both synchronous and asynchronous;
- it allows for defining the service behaviour in a workflow manner thus allowing for an easy definition of orchestrators and coordinators of services.

Thanks to these features, we considered Jolie as a good candidate for demonstrating how our approach works because it allows us to directly map the microservice meta-model into a unique linguistic technology instead of exploiting a mix of technologies. Such a characteristic permits us to avoid specific technicalities related to the chosen technology selection and focusing just on the core

[5] https://www.jetbrains.com/mps/.

concepts of microservices. Starting from a jolie representation, the microservice generation can be easily extended to other technologies. A detailed discussion about Jolie is out of the scope of this paper, the reader may consult the technical documentation of Jolie for a deeper investigation [15]. In order to show how the basic primitives of Jolie work, in the following we present a simple example where a calculator is implemented in Jolie.

**A Calculator in Jolie.** The design of a Jolie service always starts from the design of its interface which permits to define the *operations* exposed by the service and its related message types.

**Listing 1: Microservice Interface in Jolie**

```
type CalculatorRequest: void {
    .operand[1,*]: double
}
type DivisionRequest: void {
    .dividend: double
    .divisor: double
}
interfaces CalculatorInterface {
RequestResponse:
    sum( CalculatorRequest )( double ),
    sub( CalculatorRequest )( double ),
    mul( CalculatorRequest )( double ),
    div( DivisionRequest )( double )
          throws DivisionByZeroError
}
```

In the example above we defined an interface for a calculator service which exposes four operations: *sum*, *sub*, *mul* and *div*. Their definitions are scoped within the language keyword *interfaces* followed by the name of the interface *CalculatorInterface*. Note that each operation comes with a request message and a response message, thus we are defining synchronous operations which receive a request message and will reply with a response one. In Jolie request-response operations are defined below the keyword *RequestResponse* within the interface definition. In the example three operations (*sum*, *sub* and *mul*) have the same signature whereas operation *div* has a different one. In particular, the first operations receive a message with type *CalculatorRequest* and reply with a native type *double*. On the contrary, the last operation receive a message with type *DivisionRequest* and reply with a double. Moreover, operation *div* could also raise a fault called *DivisionByZeroError*. The type *CalculatorRequest* defines an array of double operands which will be elaborated by the related operation, whereas the type *DivisionRequest* define a couple of field, one for the dividend and the other for the divisor. It is worth noting that in Jolie all the messages and the internal variables are structured as trees where each node is potentially a vector of elements. Moreover, in Jolie it is possible to express the cardinality for each node of tree structure as it happens for the node *operand* where it must contain at least one element (minimum cardinality is 1) and it can have as many

elements as preferred (maximum cardinality is *). Such a structure recalls those of XML trees and JSON trees, indeed a Jolie value can be easily converted in one of them.

Once a Jolie interface has been defined, it is possible to implement its operations within a service. In the following we report a possible implementation of the *CalculatorInterface*:

**Listing 2: Microservice implementation in Jolie**

```
include "CalcultorInterface.iol"

execution{ concurrent }

inputPort Calculator {
Location: "socket://localhost:8000"
Protocol: sodep
Interfaces: CalculatorInterface
}

main {
    [ sum( request )( response ) {
        response = request.operand[ 0 ]
        for( i = 0, i < #request.operand, i++ ) {
           response = response + request.operand[ i ]
        }
    }]

    [ sub( request )( response ) {
        response = request.operand[ 0 ]
        for( i = 1, i < #request.operand, i++ ) {
           response = response - request.operand[ i ]
        }
    }]

    [ mul( request )( response ) {
        response = request.operand[ 0 ]
        for( i = 0, i < #request.operand, i++ ) {
           response = response * request.operand[ i ]
        }
    }]

    [ div( request )( response ) {
        if ( request.divisor == 0 ) {
            throw( DivisionByZeroError )
        } else {
          response = request.dividend/request.divisor
        }
    }]
}
```

The definition of a service in Jolie is mainly divided in two parts: a declarative part where all the interfaces and ports are defined, a behavioral part, represented by the scope *main*, where the implementation of the operations is provided. Note that in the definition of the service above we use the primitive *include* for automatically import the code written in file *CalculatorInterface.iol* where we suppose we saved the definition of interface *CalculatorInterface*. In this case the Jolie engine reads the content of the file and put it instead of the include

declaration. The primitive *execution* defines the execution modality of the service which can assume three possible values:

- *concurrent*: the engine will serve all the received request concurrently;
- *sequential*: the engine will serve all the received request sequentially;
- *single*: the engine will execute the behaviour once then the service will stop.

Finally, the definition of the *inputPort* permits to define the listener where the messages will be received by the service. The *inputPort* is also in charge to dispatch the message to the right operation. The *inputPort* requires three elements in order to be correctly defined:

- Location: it defines where the service is listening for messages. Briefly, a location defines the medium (*socket*, the IP (in the example is *localhost*) and the port (in the example is 8000)).
- Protocol: it defines the protocol used for performing the communication. It could be HTTP, HTTPS, SOAP, etc. In the example we use *sodep* that is binary protocol released with the Jolie engine which does not require any particular header like it happens for other protocols.
- Interfaces: it lists all the interfaces joined with that port, thus it defines all the operations enabled to receive messages on that listener.

Note that more than one input port could be defined in the same Jolie service.

Finally, let us analyze the implementation of the operations. Note that each of them is defined to store the received message in a variable called *request* and take the response message from a variable called *response*. The response is automatically sent to the invoker once the body of the request response is successfully finished. This means that inside the body of each operations, the developer must fill the variable response with a proper message which matches with the related type defined in the interface. Indeed, if we analyze the code of the first three operations, we see that the service ranges over the array of operands calculating the related operation (+, − or *). in the case of operation *div*, the service checks if the divisor is equal to zero and, if it is the case, it raises a fault *DivisionByZeroError* which is automatically sent to the invoker instead of a usual response.

The calculator service can be run by simply typing the following command on a shell:

```
jolie calculator.ol
```

where *calculator.ol* is the name of the file which contains the definition of service depicted above.

Before continuing, here we also show how to create and run a Jolie client which uses the operation of the service calculator.

**Listing 3: Jolie Client implementation**

```
include "CalcultorInterface.iol"
include "console.iol"

execution{ single }

outputPort Calculator {
Location: "socket://localhost:8000"
Protocol: sodep
Interfaces: CalculatorInterface
}

main {
    with( request ) {
        .operand[ 0 ] = 1;
        .operand[ 1 ] = 2;
        .operand[ 2 ] = 3;
        .operand[ 3 ] = 4
    }
    sum@Calculator( request )( result )
    println@Console( result )()
}
```

This Jolie client calls the calculator service on its operation *sum* passing an array of four elements and receiving as a reply their sum (*10*). The result is then printed out on the shell exploiting the operation *println@Console( result )()* which comes with the built-in Jolie service imported at the beginning (*include "console.iol"*). Note that the invocation of the calculator is performed with the line

```
sum@Calculator( request )( result )
```

where we specify the name of the operation (*sum*) and the output port to be used (*Calculator*). The output port *Calculator* is defined before the scope main and it identifies the endpoint to which sending the message. It is not a case indeed, that the elements *Location*, *Protocol* and *Interfaces* of the outputPort correspond to those of the inputPort of the calculator service. The client can be easily run in a shell typing the following command:

```
jolie client.ol
```

where *client.ol* is the name of the file which contains the definition of the client.

**The Calculator Example in Java.** In the following we report how the Jolie example of the calculator described in the previous section can be implemented as a Java class.

**Listing 1: Java implementation of the Calculator**

```
import java.util.ArrayList;

public class JavaCalculator {

 public Double sum( ArrayList<Double> operand ) throws Exception {
    if ( operand.size() > 0 ) {
     throw new Exception("At␣least␣one␣operand␣must␣be␣specified");
    }
    Double response = operand.get( 0 );
    for( int i = 1; i < operand.size(); i++ ) {
        response = response + operand.get( i );
    }
    return response;
}

 public Double sub( ArrayList<Double> operand ) throws Exception {
    if ( operand.size() > 0 ) {
     throw new Exception("At␣least␣one␣operand␣must␣be␣specified");
    }
    Double response = operand.get( 0 );
    for( int i = 1; i < operand.size(); i++ ) {
        response = response - operand.get( i );
    }

        return response;
}

 public Double mul( ArrayList<Double> operand ) throws Exception {
    if ( operand.size() > 0 ) {
      throw new Exception("At␣least␣one␣operand␣must␣be␣specified");
    }
    Double response = operand.get( 0 );
    for( int i = 1; i < operand.size(); i++ ) {
        response = response * operand.get( i );
    }
    return response;
}
 public Double div( Double dividend, Double divisor )
                                  throws Exception {
    Double response = new Double(0);
    if ( divisor == 0 ) {
        throw new Exception("Division␣by␣Zero␣Error");
    }
    response = dividend / divisor;
    return response;
}
}
```

Note that in the case of Java we need to check at the beginning of methods *sum*, *sub* and *mul* that the received array contains at least one element. In Jolie the type checking is automatically performed by the engine.

**Deploying a Jolie Service in a Docker Container.** Deploying a jolie microservice within a Docker container is very simple. The first thing to do is preparing the image of the container starting from the public and available image *jolielang/jolie* which provides a core layer where both Java and Jolie are installed. In the following we show the *Dockerfile* which allows for the creation of the docker image of the service *calculator* described in the previous section.

**Listing 4: Docker file for a microservice**

```
FROM jolielang/jolie
EXPOSE 8000
COPY calculator.ol
CMD jolie calculator.ol
```

Such a Dockerfile can be used as an input for the command *docker build* for actually creating the image of the container. Note that the file *calculator.ol* must be located in the same directory from which the command *docker build* is executed. The file *calculator.ol* indeed, will be directly copied within the image. Once the image is created, a container which derives from it can be easily run following the standard procedure of Docker.

## 3 Related Work

Since the 2014, as shown by Balalaie et al. [27], microservices steadily grown as a concept, and plenty of businesses decided to migrate their monolithic and service-oriented architectures to microservices ones. Taibi et al. [6] conducted an empirical investigation and interviewed experienced practitioners in order to identify the common motivations that led to the migration of monoliths to microservices, as well as the issues they ran into. According to the interviewees, the main reasons for migrating from a monolithic architecture to a microservices one were both maintainability and scalability. Unsurprisingly, the main issue related to migration was the monetary expenditure that such operation entails. Finally, from such interviews, the authors outlined three different migration processes adopted by practitioners.

In another study, Knoche and Hasselbring [28] report that discussions with practitioners highlighted how industry looks at microservices as a promising architecture to solve maintainability issues, even in those cases where scalability is not a critical priority. First, this work shows that incremental approaches that gradually decompose a monolith in separated microservices are the most adopted, even though cases of full-scale code rewriting exist as well. Then, based on their industrial experience, the authors provide a decomposition process to achieve an incremental migration. Besides, authors argue that when dealing with critical migrations, it makes sense to first migrate clients applications, while implementing new functionalities in the existing monolith, and then incrementally migrate all the services to the microservices architecture. On the contrary, when dealing with less critical instances, authors acknowledge that these efforts are not justified, and suggest to directly implement all the new services as microservices.

Di Francesco et al. [5] conducted another empirical study, similar to the one conducted by Taibi et al. [6] but with two main differences. First, in this study the authors put greater focus on the details of each migration phase; second, they investigated not only migrations of monolithic architecture to microservices

architectures, but also of service-oriented architectures to microservice architectures. Again, authors highlight that there is no unified strategy into approaching the migration, as some businesses choose to proceed by means of increments, whereas others tackle down the problem as a single big project. They also report a surprising result, where more than half of the participants reported that the existing data is not migrated together with the architecture, arguing that this does not align well with the two microservices typical principles: hiding internal implementations details, and managing data in a decentralized fashion.

In a 2015 manuscript, Levcovitz et al. [29] proposed a technique to identify, within monoliths, service candidates for migrating to microservices based on mapping the dependencies between databases, business functions, and facades. The authors evaluated their technique on a real case study (a 750 KLOC monolith programmed in C) in the banking domain, and show that they identified successfully candidate subsystems. To the best of our knowledge, apart from our work, this is the only alternative publication that discusses migration techniques applied to a specific banking case study. Moreover, it is useful to notice that while their approach aims to automatize the identification based on the legacy monolithic deployment, our case study was primarily business-driven. Therefore, our approach had to be necessarily manual and iterative.

Again, Balalaie et al. [27,30] reported their experience of performing an incremental migration of a mobile back-end as a service (MBaaS) to microservices, coupling with DevOps methodologies. Citing the on-demand capability as the main driver of migration, they caution *a posteriori* about two important lessons learned. In first instance, the authors warn that the service contracts are critical and that changing a lot of services that only interact with each other could expose to a number of errors, as small errors in the contracts can break down substantial part of the architecture. Second, they caution against considering microservices a silver bullet, as it can be beneficial to bring scalability to services, but it can introduce higher complexity as well.

Following the previous works, the authors collected and reported some empirical migration patterns derived from medium to large-scale industrial projects, aiming to help others to perform a smooth migration [31]. They evaluate such patterns through qualitative empirical research, and cite as future work the development of a pattern language that would allow to automatically compose the patterns.

In a recent work, Furda et al. [32] agree on defining the migration to microservices a promising way to modernize monolithic architectures and to enable full-scale utilization of cloud computing. At the same time, they identify three major challenges in migrating monolithic architectures to microservices ones, namely: multitenancy, statefulness, and data consistency.

Finally, Bucchiarone et al. [33] report an experience from of a real-world case study in the banking domain, of how scalability is positively affected by re-implementing a monolithic architecture into microservices. Even if it presents a real and complex application migration, the approach proposed was not supported by an automatic migration tool but was only business-driven and

outside-in, i.e., the system has been designed and implemented one business functionality at a time.

Analyzing all the lessons learned by the previous works we can conclude saying that despite the fact that there is an extreme and increasingly emerging need to migrate applications from monolithic to microservices, rare are the approaches that try to make this process automatic and tool supported. Most of the migration approaches proposed are guided by the developers experience and are not supported by a specific tool or language. They consider case by case whether a functionality should result in a new service or not. If the business functionality seemed isolated and big enough, or it was shared among numerous other business functionalities, then it resulted in a new service.

At the same time, to the best of our knowledge, there is no Model-Driven Engineering approach addressing the support of migration in the sense addressed in this paper. The migration approach introduced in the next section shows our solution in this direction with the aim towards the realization of a general framework for automatic migration to microservices.

## 4 Model-Driven Migration Approach

In this Section we introduce the migration process, showing how a MA can be converted into a MSA and deployed in a Docker container. In this respect, our contribution is visualized in Fig. 1, which depicts the different artifacts realized and their relations. Technically, the solution is composed by two fundamental components: (a) the `Microservices Miner` and (b) the `Microservices Generator`, by two Domain Specific Languages (DSLs) (i.e., for the `Microservice` specification and for their `Deployment`), by a set of `generators` used to support the overall migration from MA (developed in Java) to MSA (developed using Jolie) and the corresponding deployment in a `Docker Container`. The following sections provide detailed descriptions of each of the previous artifacts.

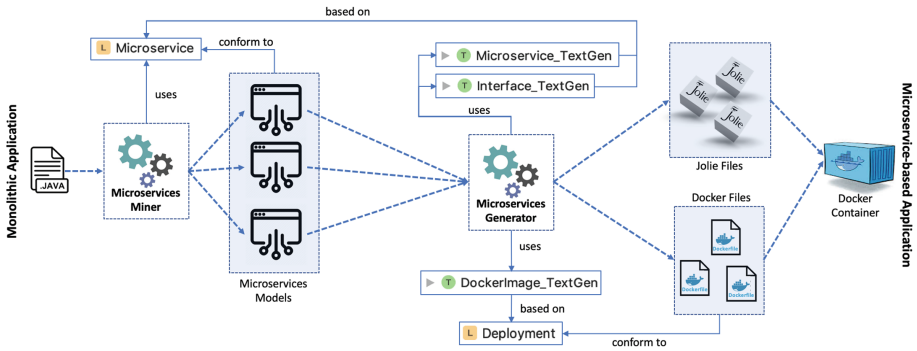


**Fig. 1.** The model-driven migration approach.

### 4.1 The `Microservice` Language

The `Microservice` language relies on how microservices can be defined in Jolie (see Sect. 2.4) and allows the developers to design concrete microservices. An excerpt on the concepts defined to realize this DSL are depicted in Fig. 2. In particular at this level of abstraction the developer can specify a microservice including its `interface`, `inputPort`, `outputPort` and the respective `behavior`. Moreover it comprises a property called `directive` that is used to set the execution modality. For example setting it to `concurrent` will allow the service to process all the incoming requests concurrently. The `behaviour` children of the `Microservice` concept is used to define the implementation of the functionalities offered by a microservice.

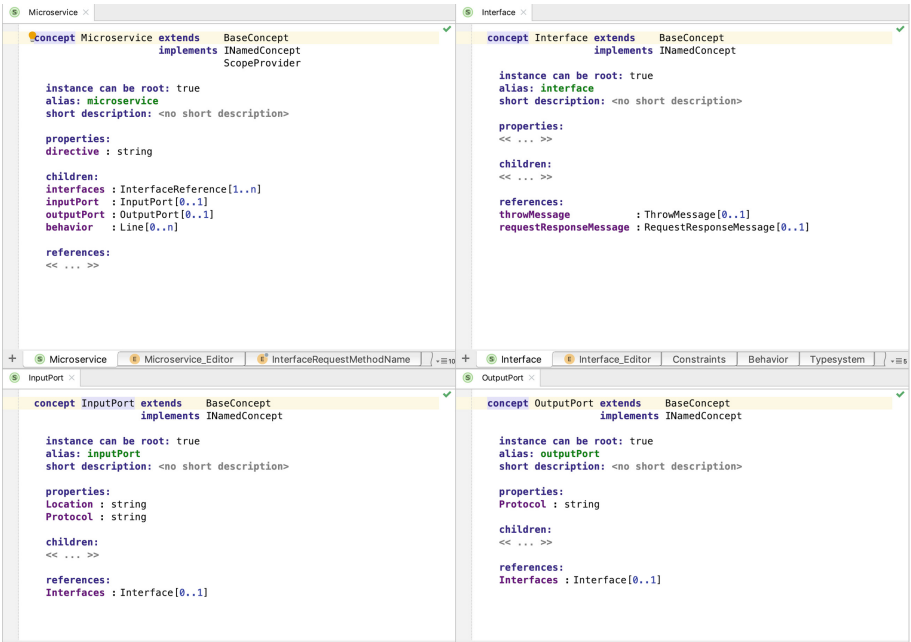


**Fig. 2.** Concepts of the microservice language.

The `Interface` concept is used to specify the type of each exchanged message (i.e., `requestResponseMessage`) in a microservice. To enable a communication between microservices, we need to specify the input and the output endpoints. In Jolie this is done by using primitives input ports and output ports. In our DSL we have introduced two concepts. `InputPort` is used for defining a listener endpoint whereas the `OutputPort` concept is used for sending messages to an inputPort.

Figure 3 shows how the `calculator` microservice can be specified using the `Microservice` language. The `calculator` microservice provides an `inputPort` which is listening on port 8999 where the `CalculatorInterface` is defined.

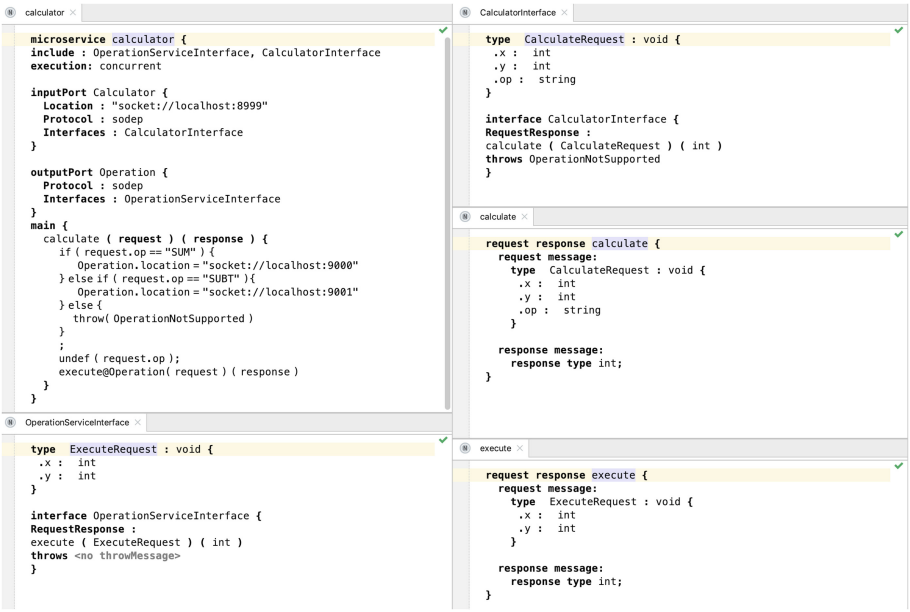


**Fig. 3.** Calculator microservice model.

In this interface the request also contains the subnode `.op:string`, which permits to specify the operation type (i.e., `SUM` or `SUBT`). Moreover, it includes also a fault sent as a response (i.e., `OperationNotSupported`).

The interface called `OperationServiceInterface` provides a `RequestResponse` operation called `execute`. A RequestResponse operation is an operation which receives a request message and replies with a response message. In this case, the request message type is defined by `ExecuteRequest`, which contains two subnodes: x and y. Both of them are integers. On the other hand, the response is just an integer.

The `outputPort` of the `calculator` microservice requires the same parameters of the `inputPort` (i.e., `Location`, `Protocol`, `Interfaces`) but in our example the `Location` is omitted because it is dynamically bound at runtime depending on the value of request node `op`. Indeed, we bind the port Operation to a different location depending on if we call the service SUM or the service SUBT in the microservice behavior (i.e., `main` part of the `calculator` microservice).

### 4.2 The `Deployment` Language

In this Section we show how a microservice modelled using our approach can be deployed inside a Docker container. Basically, the only thing to do is to create a `Dockerfile` which allows for creating a `Docker image` that can be used for generating containers. For this purpose we have defined a specific language called `Deployment` (see Fig. 4) devoted to the specification of the microservices `Dockerfiles`. At the same time is important to know that there is a `Docker image` which provides a container where Jolie is installed. Such an image can be found on `dockerhub`[6] and is used as base layer for deploying jolie services modelled or automatically generated by our approach.

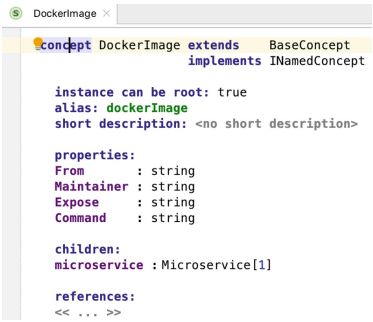


**Fig. 4.** DockerImage concept of the deployment language.

To create a docker image of a microservice (as the one in Fig. 3), it is necessary to specify down a `Dockerfile` exploiting the `DockerImage` concept of the `Deployment` language. To do this we have created an MPS editor (as shown in Fig. 5 left side) that helps developers to specify `Dockerfile` models as depicted in the right side of Fig. 5.

The `FROM` child of the `DockerImage` concept us used to load the image *jolielang/jolie*, while the `MAINTAINER` is used to specify the name and the email address of the file maintainer. `EXPOSE` is used to expose the port 8000 to be used by external invokers of the microservice. This means that the jolie microservice always listens on this port within the container. `COPY` is used to copy the file `calculatore.ol` within the image renaming it into `main.ol`. Finally `CMD` specifies the command to be executed by Docker when a container will be start from the image described by this Dockerfile.

### 4.3 Microservices Miner

To analyze the monolithic application written in Java and to retrieve from it the set of needed microservices, we have implemented the `Microservices Miner` component. Its main task is to search in the abstract syntax tree of the imported Java

[6] https://hub.docker.com/r/jolielang/jolie.

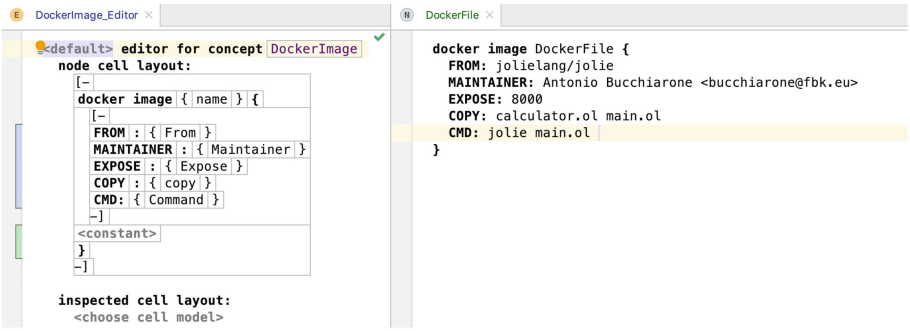


**Fig. 5.** `Dockerfile` editor and model.

code for patterns of interest and suggest to the developer the set of microservices for the migration. This is done thanks the realization of two subcomponents, the *orchestrator* and a set of *searchers*. The orchestrator is a generic implemented action in the miner language and can be invoked on the MPS *logical view* or on the specific imported Java code *editor*.

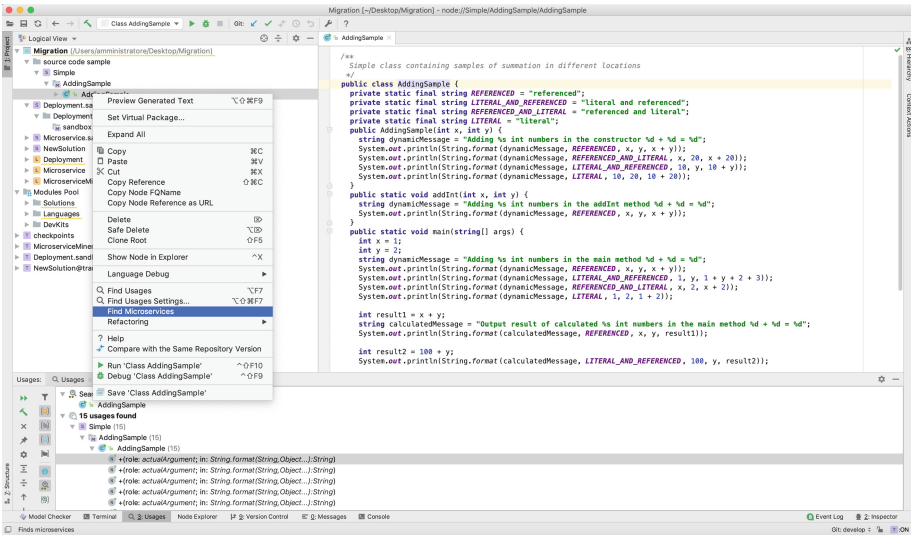

**Fig. 6.** `Microservice` finder in action.

The MPS logical view shows a tree presenting the project structure. When the implemented finder is selected (the `Find microservices` action in Fig. 6), MPS, using the *searchers* available in the project executes the searchers. The result of this search is usually visualized in the *Usage View*, as depicted at

the bottom of Fig. 6, and contains the occurrences of the *pattern* specified in the specific searcher. Each searcher is realized by implementing the interface or the abstract implementation and finds nodes in the AST that comply to the semantic understanding of the searcher. As a simple example, Fig. 7 presents the `PlusExpressionFinder` definition and its usage in the more general `MicroServiceSearcher`. When it is invoked in the project *logical view* it searches for all the occurrences of the *plus expression* in the Java source code and returns all the occurrences retrieved in the *Usage View*. This outcome is exploited by the developer to identify the set of microservices that must be specified using the `Microservice` language introduced in Sect. 4.1 and that can be directly deployed in a Docker container using the `Microservice Generator` illustrated in the next Section.

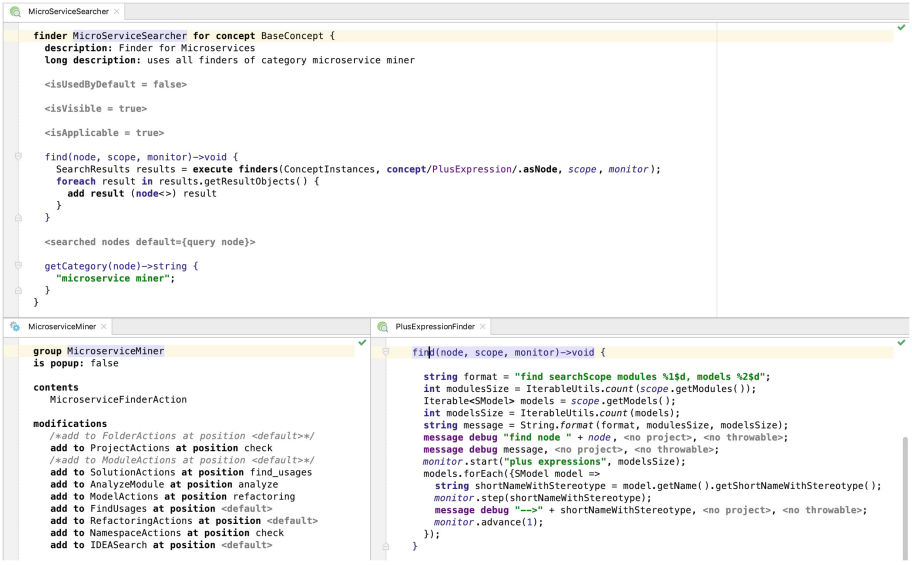


**Fig. 7.** `Microservice` miner.

### 4.4 Microservices Generator

To generate all the needed files to deploy a microservice in a Docker container we have used one of the transformation feature provided by MPS, it is the AST to text (model-to-text) transformation. As we have already introduced in Sect. 2, to run a microservice we need to provide one file for the service specification (with extension .ol), one file for each interface the microservice uses (with extension .iol), and the Dockerfile used to deploy the microservice in a Docker container. To do this we have implemented three generators (depicted in Fig. 1) called:

`Microservice_TextGen`, `Interface_TextGen`, and `DockerImage_TextGen`. As illustrated in Figs. 8 and 9, generators specifications are given by means of template mechanisms. Templates are written by using the output language (i.e., Jolie and Dockerfile in our case), and are parametric with respect to the elements retrievable from the input model through Macros, denoted by the $ symbol. In our case, the `Microservice` and the `Interface` generators are used to generate the corresponding Jolie .ol and .iol files. The `Dockerfile` generator instead is used to generate the corresponding Dockerfile document needed to deploy the microservice in a Docker container.

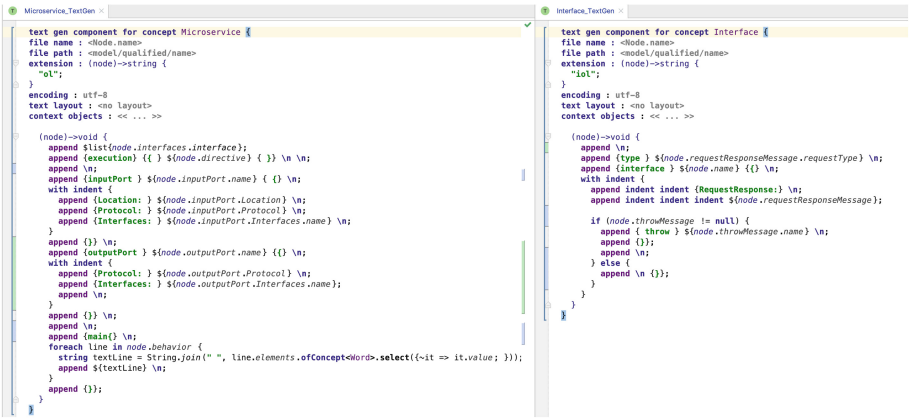


**Fig. 8.** `Microservice` and `Interface` generators.

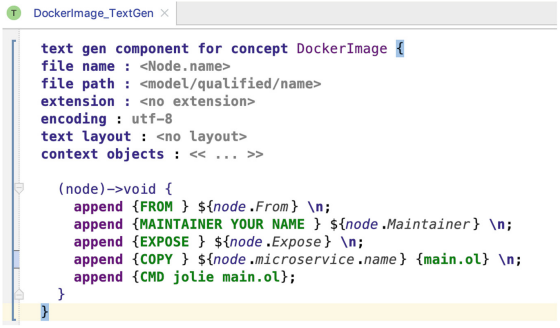


**Fig. 9.** `Dockerfile` generator.


**Fig. 10.** Prototype execution steps.

## 5 Prototype Implementation

As a first iteration towards realizing the framework proposed in this paper, we developed the model-driven approach described in Sect. 4[7]. The implementation includes all the phases depicted in Fig. 10 and has been experimented using the motivating scenario presented in Sect. 2.4.

Using our solution, a developer can start the migration process importing the Java source code of the monolithic application in phase (1). This is done using the native MPS action `Get Models Content from Source` that can be invoked by the main menu. In this way the Java code is parsed into MPS' base language and imported in the editor as input Java models. Phase (2) is used to interrogate the imported Java models for patterns of used packages, classes, methods and members to identify microservice candidates. In phase (3), with the `Microservice` domain specific language in the hand the developer can create the different models of the identified microservices. In phase (4), using the provided generators described in Sect. 4.4, the microservices models with their respective interfaces are transformed in the target Jolie files. In the end, in phase (5), for each Jolie microservice a `Dockerfile` is created and used to deploy the overall application in a Docker container.

## 6 Conclusion

In this paper we presented the experiences matured in the development of a Model-Driven approach for the migration of monolithic applications to microservice applications. The proposed solution is based on the definition of two domain specific languages, one for the microservices specification and one for their deployment in a Docker container, and on a set of generators that make the migration approach automatic and with less manual intervention by developers. The framework have been implemented by means of the MPS text-based language workbench and evaluated with an initial with the aim to demonstrate the feasibility of the approach, and calls for future research. To make is scalable and usable in real contexts we are interested to test it using different industrial case studies to further investigate the soundness of the proposed methodology and eventually to extend the specification of the provided DSLs in MPS to make it more general.

## References

1. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional, Reading (2015)
2. Fowler, M., Lewis, J.: Microservices, ThoughtWorks (2014)

[7] A prototype implementation of the approach and the related artefacts are available at the GitHub repository: https://github.com/antbucc/Migration.git.

3. Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S.: Microservices: the journey so far and challenges ahead. IEEE Softw. **35**(3), 24–35 (2018)
4. Mazzara, M., Naumchev, A., Safina, L., Sillitti, A., Urysov, K.: Teaching DevOps in corporate environments: an experience report, CoRR, vol. abs/1807.01632 (2018)
5. Francesco, P.D., Lago, P., Malavolta, I.: Migrating towards microservice architectures: an industrial survey. In: 2018 IEEE International Conference on Software Architecture (ICSA), pp. 29–2909, April 2018
6. Taibi, D., Lenarduzzi, V., Pahl, C.: Processes, motivations, and issues for migrating to microservices architectures: an empirical investigation. IEEE Cloud Comput. **4**, 22–32 (2017)
7. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**, 1053–1058 (1972)
8. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B., et al. (eds.) Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
9. 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.) PSI 2017. LNCS, vol. 10742, pp. 95–104. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-74313-4_8
10. France, R., Rumpe, B.: Model-based development. Softw. Syst. Model. **7**(1), 1–2 (2008)
11. Atkinson, C., Kühne, T.: Model-driven development: a metamodeling foundation. IEEE Softw. **20**, 36–41 (2003)
12. Schmidt, D.C.: Guest editor's introduction: model-driven engineering. Computer **39**, 25–31 (2006)
13. Voelter, M., et al.: DSL Engineering - Designing, Implementing and Using Domain-Specific Languages (2013). dslbook.org
14. Voelter, M., Lisson, S.: Supporting diverse notations in MPS' projectional editor. In: Proceedings of the 2nd International Workshop on the Globalization of Modeling Languages Co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems, GEMOC@Models 2014, pp. 7–16 (2014)
15. The Jolie language website. http://www.jolie-lang.org/
16. MacKenzie, M.C., Laskey, K., McCabe, F., Brown, P.F., Metz, R., Hamilton, B.A.: Reference model for service oriented architecture 1.0, vol. 12. OASIS Standard (2006)
17. Newman, S.: Building Microservices. O'Reilly Media Inc, Sebastopol (2015)
18. Montesi, F., Guidi, C., Zavattaro, G.: Service-Oriented Programming with Jolie. In: Bouguettaya, A., Sheng, Q., Daniel, F. (eds.) Web Services Foundations, pp. 81–107. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-7518-7_4
19. Guidi, C., Lanese, I., Mazzara, M., Montesi, F.: Microservices: a language-based approach. Present and Ulterior Software Engineering, pp. 217–225. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_13
20. Bandura, A., Kurilenko, N., Mazzara, M., Rivera, V., Safina, L., Tchitchigin, A.: Jolie community on the rise. In: SOCA, pp. 40–43. IEEE Computer Society (2016)
21. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. SIGPLAN Not. **35**(6), 26–36 (2000)
22. Ward, M.: Language oriented programming. Softw. Concepts Tools **15**, 147–161 (1994)

23. Voelter, M.: Language and IDE modularization and composition with MPS. In: Generative and Transformational Techniques in Software Engineering IV, GTTSE 2011, pp. 383–430. International Summer School (2011)
24. Voelter, M., Pech, V.: Language modularity with the MPS language workbench. In: 34th International Conference on Software Engineering, ICSE 2012, pp. 1449–1450 (2012)
25. Campagne, F.: The MPS Language Workbench, vol. 1, 1st edn. CreateSpace Independent Publishing Platform, Hamburg (2014)
26. Campagne, F.: The MPS Language Workbench Volume II: The Meta Programming System, vol. 2, 1st edn. CreateSpace Independent Publishing Platform, Hamburg (2016)
27. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables devops: migration to a cloud-native architecture. IEEE Softw. **33**, 42–52 (2016)
28. Knoche, H., Hasselbring, W.: Using microservices for legacy software modernization. IEEE Softw. **35**, 44–49 (2018)
29. Levcovitz, A., Terra, R., Valente, M.T.: Towards a technique for extracting microservices from monolithic enterprise systems. In: III Workshop de Visualização, Evolução e Manutenção de Software (VEM), pp. 97–104 (2015)
30. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Migrating to cloud-native architectures using microservices: an experience report. In: Celesti, A., Leitner, P. (eds.) ESOCC Workshops 2015. CCIS, vol. 567, pp. 201–215. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33313-7_15
31. Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D.A., Lynn, T.: Microservices migration patterns. Softw. Pract. Exp. **48**, 2019–2042 (2018)
32. Furda, A., Fidge, C., Zimmermann, O., Kelly, W., Barros, A.: Migrating enterprise legacy source code to microservices: on multitenancy, statefulness, and data consistency. IEEE Softw. **35**, 63–72 (2018)
33. Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S.T., Mazzara, M.: From monolithic to microservices: an experience report from the banking domain. IEEE Softw. **35**(3), 50–55 (2018)