JoT: A Jolie Framework for Testing Microservices

Saverio Giallorenzo[1], Fabrizio Montesi[2],
Marco Peressotti[2], Florian Rademacher[3,4], and Narongrit Unwerawattana[2]

[1] Università di Bologna, Italy and INRIA, France saverio.giallorenzo@gmail.com
[2] University of Southern Denmark, Denmark
{fmontesi,peressotti}@imada.sdu.dk, nau@sdu.dk
[3] Software Engineering, RWTH Aachen University, Germany
[4] IDiAL Institute, University of Applied Sciences and Arts Dortmund, Germany
rademacher@se-rwth.de

Abstract We present JoT, a testing framework for Microservice Architectures (MSAs) based on technology agnosticism, a core principle of microservices. The main advantage of JoT is that it reduces the amount of work for a) testing for MSAs whose services use different technology stacks, b) writing tests that involve multiple services, and c) reusing tests of the same MSA under different deployment configurations or after changing some of its components (e.g., when, for performance, one reimplements a service with a different technology). In JoT, tests are orchestrators that can both consume or offer operations from/to the MSA under test. The language for writing JoT tests is Jolie, which provides constructs that support technology agnosticism and the definition of terse test behaviours. We present the methodology we envision for testing MSAs with JoT and we validate it by implementing non-trivial test scenarios taken from a reference MSA from the literature (Lakeside Mutual).

Keywords: Microservice Architectures, Testing Frameworks, Service-Oriented Programming

1 Introduction

The paradigm of microservices is one the modern gold standards for developing distributed applications. In this setting, a distributed application emerges as the composition of multiple services (the "microservices"). Each microservice implements a set of business capabitilies, and is independently executable and deployable. Microservices interact with each other via message-passing APIs [4].

Two important factors in the diffusion of microservices are the scalability and flexibility that they support. Scaling is efficient because one can focus scaling actions precisely on those components impacted by traffic fluctuations. Flexibility is given by the usage of *technology-agnostic* APIs, which allows for using different implementation technologies for different microservices without renouncing integration.

However, the good traits of microservices do not come for free. Here, we focus on one of the most prominent elements impacted by the microservices style: testing sets

of microservices, or *Microservice Architectures (MSAs)*. Indeed, for unit testing, one can rely on existing frameworks tailored for and idiomatic to the general-purpose implementation technology used to develop a single microservice (e.g., Java, JavaScript, Rust, C). However, when tests cover more microservices, it can become cumbersome to specify the coordination and invocation of services developed with different technologies using a framework designed for testing the "internals" of a service.

To make a concrete example, imagine using JUnit [8] (in Java) to specify the connections to and the coordination and consumption of multiple operations of several microservices. This would not only entail the specification of (possibly complex) coordination logic in Java, but it would also mean adding, on top of the latter, the logic that encodes the data structures that microservices exchange, how connections are established and handled (including errors)—in terms of transport and application layers, etc. Besides their complexity, tests written in this way are difficult to be reused in other tests or under different deployment settings (imagine repurposing a test that uses HTTP endpoints to verb-based binary protocols).

Motivated by these observations we present JoT (Jolie Testing), a testing framework for MSAs based on technology agnosticism. Responding to its motivating points, JoT reduces the amount of work for a) testing for MSAs whose services use different technology stacks, b) writing tests that involve multiple services, and c) reusing tests of the same MSA under different deployment configurations or after changing some of its components (e.g., when, for performance, one reimplements a service with a different technology). In JoT, tests are orchestrators that can both consume or offer operations from/to the MSA under test. The language for writing JoT tests is Jolie [17], which provides constructs that support technology agnosticism [16] and the definition of terse test behaviours. One of the most relevant features introduced by JoT is the provision of Jolie annotations that users can use to structure and specify the sequence of actions that the tool needs to follow to run each test (e.g., test setup, cases, and clean up).

In Section 2, we discuss the methodology we envision for testing MSAs with JoT, following an example where we use JoT annotations to build a test case. In Section 3, we provide initial validation to JoT's approach by presenting implementations of non-trivial test scenarios taken from a reference MSA from the literature [23] (Lakeside Mutual). We draw conclusions, compare to related work, and discuss future steps in Section 4.

2 Methodology and Structure of Tests

To illustrate the structure of tests and the architecture of the testing framework, we start by describing the methodology we envision for building tests in JoT, i.e., the steps users should follow to define a test using the framework.

2.1 Building a Test in JoT

Following general testing practice, the first step for building a test in JoT is defining the subject under test. Our subject is an architecture of services (one or more)

that can interact with each other. Considering that the subject under test are the services of an architecture, in the remainder, we use interchangeably the terms "subject under test" and "architecture under test" and use the term "service under test" to indicate a service that is part of an architecture under test (which includes the degenerate case of an architecture made of one service). For example, in the first case in Section 3, the architecture under test is made of two services—CustomerCore and CustomerManagement–that manage the users of an online platform.

Once we defined the subject of the test, we need to identify the cases we want to test, i.e., the functionality whose implementation we want to verify. This can range from a single invocation, e.g., calling one operation of one service, to complex behaviours that compose several operations of different services. For example, by having as the subject under test the CustomerCore-CustomerManagement architecture, we can check that users are coherently created, fetched, and modified by the two services. For instance, we can interact with CustomerCore to create a user, then we update the data related to that user via the operations provided by CustomerManagement, and then verify that the update was successful, by fetching and checking the user's information from CustomerCore.

Once we defined the subject under test and the functionality we want to test, we can proceed with the actual implementation of the JoT test and its cases.

Since a JoT test is a service itself (and an orchestrator, in particular) the information we need to provide to a JoT test coincides with the three main elements that define services in general [11]. The first two are the Application Programming Interfaces, *interfaces* for short, and the *access points* which, combined, define the public contract of the services (under test). The third element is the private, internal *behaviour* of the service, which implements the logic of each test case.

Interfaces The interface of a service specifies what operations it offers to clients. There exist many guidelines and technologies for the description of interfaces [4]. However, we can abstract an interface as the set of labelled operations that a service promises to support. The description of the set of operations can also carry the messaging pattern (e.g., one-way, request-reply calls) of each operation and the structure of the data exchanged through each of them.

For example, one can provide them in the form of an informal list of resources that one can call, e.g., as URL addresses, and describe the shape of the in-/outbound data similarly. Alternatives include the usage of formal languages for the specification of service interfaces, such as WSDL, and the description of interfaces using metamodels [11,20] which support the generation of the same service interface under different formats (formal and informal).

Thanks to the flexibility of Jolie interfaces, JoT adopts a permissive attitude, where the minimal amount of information users need to provide regarding interfaces is: a) the list of operation labels that the test is going to use and b) the messaging pattern that characterises each operation.

For example, a minimal Jolie interface to test the "createCustomer" operation of the CustomerCore is

```
interface CustomerCoreInterface {
 requestResponse: createCustomer
}
```

In the code, we specify that the operation `createCustomer` has a request-response behaviour (from the user side, this means invoking the service on the operation and waiting for the server to answer with some response) and that the operation belongs to the `CustomerCoreInterface` interface (the latter's name is immaterial for the service under test, and it is just a reference to the interface's content within the test itself).

Interestingly, JoT provides support for specifying test invariants on the exchanged data already at the level of interfaces. Indeed, users can specify the structure of the data they expect to see in tests via Jolie types. Jolie types have a tree-shaped form, made of two components: the root of the tree, associated with a basic type (e.g., integer, string, etc.), and a set of nodes that defines the internal fields of the data structure—each node is an array with specified minimal and maximal cardinality.

For example, we can enrich `CustomerCoreInterface` with types, to both specify the kind of data we promise to provide within the test (cases)—in the request part of the `createCustomer` operation—and the shape of the data we expect the service under test to send back as the response.

For example, in the code below, we show one such interface where the request to the `createCustomer` operation needs to carry the name and surname of the user (as strings), while the operation responds with the identification number of the user (as an integer).

```
type CustomerRequest { name: string, surname: string }
type CustomerResponse { id: int }
interface CustomerCoreInterface {
  requestResponse:
   createCustomer(CustomerRequest)(CustomerResponse)
}
```

Access Points The access point completes the public contract of a service's interface by defining where and how to contact the service, i.e., defining the stack of technologies that clients can use to interact with the service.

Specifically, the technology stack determines the media and protocols used to support the communication between a service and its clients and the format of the data that these exchange. For instance, one can decide to use SOAP and TCP/IP as a technology stack for communication and use XML to format the data.

By relying on Jolie ports, JoT makes it easy to adapt a test to the access-point specifications of a given service incarnation. For example, this allows users to write

a test case that they initially want to run at the development stage, e.g., using a message broker [6] and some binary format, and then change the ports settings to test the service in production, e.g., switching the port to use TCP/IP, HTTP, and the JSON format—other examples include SOAP-based web services [17] and REST ones [16].

As we discuss below, JoT provides direct support to this level of flexibility via configuration parameters that the user can pass to the test, so that one can run the same test on different deployment settings programmatically. As an example, following the simple case made above, we can define the port to contact the `CustomerCore` service in a JoT test in the following way:

```
outputPort CustomerCore {
  location: parameters.customerCore.location
  protocol: parameters.customerCore.protocol
  interfaces: CustomerCoreInterface
}
```

Above, we define an `outputPort` called `CustomerCore`, which represents an external service that we can invoke. Through the port definition, we declare that we expect that the `CustomerCore` service implements the `CustomerCoreInterface`. Notice that the `location` and `protocol` of the port are (elements of the variable) `parameters`. We used this definition of the port to illustrate how the user can change the medium technology and endpoint definition (`location`) and the communication protocol and data format (`protocol`) by passing this information as parameters of the test instantiation.

Test Logic The last element of the test is the definition of the actions that the test needs to enact to implement its logic.

Here, Jolie provides different ways to define the logic of the service, e.g., by allowing developers to use Java or JavaScript. We deem using these languages a viable route, e.g., if one needs to use libraries that would be difficult to expose otherwise or wants to re-use some test logic written in those languages. Notwithstanding this possibility, we envision users to mainly write JoT tests using the Jolie behaviour language. Indeed, Jolie provides a concise-yet-expressive language for behaviour specification that makes it easy to assemble even complex coordination logic, like speculative parallelism [3] and partial joins [7], which one can use to reproduce edge cases of highly-concurrent systems.

Ports make it possible to keep the logic of Jolie programs, and JoT tests, loosely coupled w.r.t. the deployment technology. For instance, let us look at a simple behaviour snippet for our example

```
createCustomer@CustomerCore({ name = "John", surname = "Doe" })(resp)
if(resp.id <= 0){
 throw (TestFailed, "Users need to have positive id numbers")
}
```

Above, we define an elementary test for the `createCustomer` operation, where we send a legit request (according to the interface we defined) and check that

the response has the expected shape (verified by the Jolie type checker, given the interface definition of `createCustomer`) and that the identifier is positive. In case the test fails, we `throw` a fault, which interrupts the execution of the tests and reports to the user the failing case. Later in Section 3 we use the assertion library provided by JoT, which helps users in verifying the compliance of the results against the expected values even in the case of complex data structures (multi-level nested trees).

2.2 Writing a Complete Test

Before detailing the architecture of JoT, we illustrate the remaining important items that make up a JoT test. For this purpose, we show a working JoT test example by assembling the interface, port, and behaviour shown above with the remaining elements that characterise a JoT test—for brevity, we elide most of the constructs discussed above to focus on the new parts.

```
type CustomerRequest ...
interface CustomerCoreInterface { ... }

interface TestInterface {
 requestResponse:
  ///@Test
  testCreateCustomer()() throws TestFailed(string)
}

service Main(parameters: undefined){
 outputPort CustomerCore {
   location: parameters.customerCore.location
   ...
 }

 inputPort Input {
  ...
  interfaces: TestInterface
 }

 main {
  testCreateCustomer()() {
   createCustomer@CustomerCore({ name = "John", surname = "Doe" })(resp)
   ... }
} }
```

The salient additional parts in the example are four, described below following their top-to-bottom order of appearance in the code.

First, we have an interface, called `TestInterface`, which defines the sequence of operations the JoT framework shall run from the current test. This is done—similarly to other testing frameworks, e.g., JUnit—using comment annotations of the form `///@Annotation`. JoT currently supports five kinds of annotations: `///@BeforeAll`, `///@BeforeEach`, `///@AfterEach`, `///@AfterAll`, and `///@Test`.

Respectively, these indicate operations in the body of the test that we invoke once before all test cases, before calling each test case, after we called each test case, and once after we invoked all test cases. The last annotation is to indicate test-case operations. JoT does not impose order among the operations in a given annotation category.

Second, we have a `service` (conventionally called `Main`), which is the Jolie program unit that the JoT framework instantiates to run the tests. When performing the instantiation, the framework passes the configuration parameters for the test defined by the user, which the `service` holds in the `parameters` variable (here, we leave its type `undefined`). In the example, we use the `parameters` variable to carry the information to contact the `CustomerCore` in the related `outputPort`.

Third, we have an `inputPort` (complementary inbound access points to `outputPort`s) that allows the JoT framework orchestrator to govern the operations offered by the test (`service`). Indeed, the `inputPort` publishes the `TestInterface` defined earlier.

Fourth, there is the `main` execution block, which encloses the behaviour of the test cases and the surrounding operations (before-all/each and after-all/each) of the test. In the body of the `main`, we find the test `testCreateCustomer`, which, at invocation, runs the test-case behaviour we previously commented on.

2.3 Executing JoT Tests

By design, JoT does not manage the deployment of the architecture under test. This is to let developers decide the best way to run the architecture. For example, the developer of our exemplary test could execute the service locally (using private network addresses) and later on re-use the same test logic to check the behaviour of the service in production (using public addresses). JoT achieves this flexibility via file-based configurations. Concretely, JoT configurations are JSON files that contain test parameters, such as a tested service's address or protocol. Listing 1 shows an example of a JoT configuration file. It configures the execution of the JoT test whose excerpts were shown in previous listings and which is stored in a Jolie program called "TestCustomerCore.ol" (".ol" is the extension for Jolie programs).

Listing 1. Example JoT configuration file.

```
{ "testsPath": ".",
  "params": {
    "TestCustomerCore.ol": [{
      "name": "Main",
      "params": {}
    }] } }
```

The `testsPath` element specifies the file path of the test source, relative to the configuration file. The `params` element is where users link tests to parameters. For this purpose, each member of the element is a key-value pair consisting of (i) the name of the file that contains the code of the test; (ii) an array of configuration objects. Namely, the element `name` is the name of the Jolie `service` that wraps the test code (e.g., `Main`) while the `params` node contains the parameters for the test.

To execute a test with file-based configuration, the user can save the JSON data in a "params.json" file and then launch the test with the command `jot params.json`.

When testing architectures, our suggestion is to pair JoT with widely adopted microservice deployment technologies, like Kubernetes and Docker-compose, to further automate the running of test batteries. This is the practice we follow, e.g., in Section 3, where the services of the architecture under test are containers, deployed through a single Docker Compose file.

JoT's source code is available on GitHub[5], and a publicly downloadable video illustrates JoT's architecture and usage[6].

3 Validation

We now show a preliminary validation of JoT by writing a pair of tests (and related test cases) drawn from the Lakeside Mutual [23] architecture[7].

Briefly, Lakeside Mutual is a fictitious insurance company that provides its employees and customers with a software platform to, e.g., manage personal data and insurance policies. In total, Lakeside Mutual—in the continuation, we use the term to indicate the insurance company's software platform—consists of five backend microservices, four frontend components enabling users to operate on the data maintained by the backend microservices, and two infrastructure components for service discovery and technical administration.

3.1 Tested Interaction Scenarios

We implement two testing scenarios.

Scenario 1 involves the interaction of two microservices, namely `CustomerCore` and `CustomerManagement`. The `CustomerCore` microservice provides basic capabilities to manage a customer's data. The `CustomerManagement` microservice acts as a façade for `CustomerCore` and is responsible for providing clients with a stable interface, thereby facilitating the evolution of `CustomerCore`. The testing logic for Scenario 1 covers the update of an existing insurance customer triggered by a client. Figure 1 shows the specification of the scenario as a UML sequence diagram [18].

In Figure 1, the `Client` initiates the scenario by retrieving an existing customer with a given `id`, using the `CustomerManagement` operation `getCustomer`. `CustomerManagement` forwards the request to `CustomerCore` and returns the response of the latter to the `Client`. Next, the `Client` updates the received data (e.g., it can change the address of the queried customer) and calls `updateCustomer` with the updated data on `CustomerManagement`. Again, `CustomerManagement` forwards this call to `CustomerCore` to perform the actual update of the database.

[5] `https://github.com/jolie/jot`

[6] `https://drive.google.com/file/d/1VimUbh6stPQoyB_EeLJLllwLs5Vj82wX/view?usp=sharing`

[7] As retrieved at version `https://github.com/Microservice-API-Patterns/LakesideMutual/commit/aaebc590832c9ffc064fa3a22eae20db17ab31d9`

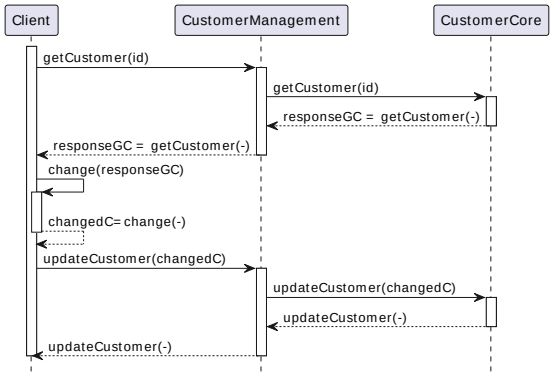

Figure 1. Specification of tested interaction Scenario 1 as a UML sequence diagram.

Scenario 2 includes, on top of the services seen in Scenario 1, another microservice, i.e., `CustomerSelfService`. In this scenario, `CustomerSelfService` provides customers with the functionality to register themselves in the system. Scenario 2 focuses on this registration process and the correct execution of the `getCustomers` operation to find the newly registered customer. Thus, differently from Scenario 1, Scenario 2 covers a dedicated business process rather than an activity that is part of several processes. Indeed, Scenario 2 is more complex than the first one and illustrates JoT's capability to perform testing of interactions comprising more than the microservices directly accessed by the test, i.e., the test entails the correct interaction between `CustomerCore` and `CustomerManagement`. Figure 2 shows the specification of Scenario 2 as a UML sequence diagram.

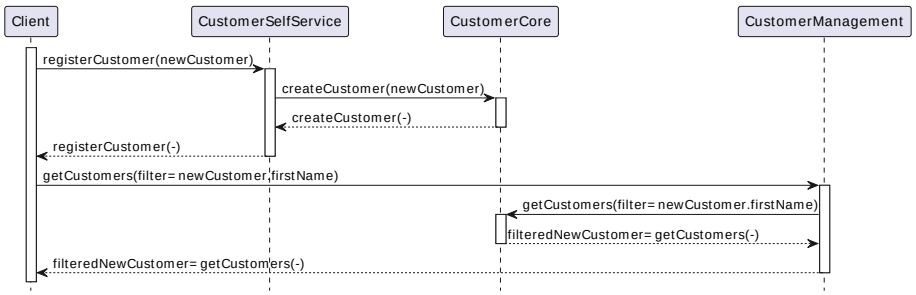

Figure 2. Specification of tested interaction Scenario 2 as a UML sequence diagram.

In Figure 2, a `Client` registers a new customer by calling the `registerCustomer` operation of the `CustomerSelfService` with the new customer's data. `CustomerSelfService` partially acts as a façade to `CustomerCore`, to which it forwards the request for customer registration as a call to `createCustomer`. After the completion of `registerCustomer`, the `Client` continues by executing the `getCustomers` operation of `CustomerManagement`. This operation allows fetching customers via filters, e.g., via their names. This call is also forwarded to `CustomerCore`, which queries its database and performs the actual fetching.

3.2 JoT Test of Scenario 1

We move to implement Scenario 1 (cf. Figure 1) using JoT. In the scenario, the JoT tests correspond to the `Client` components (cf. Section 3.1). We start by introducing the Jolie interfaces and access points for the test, and then we describe its logic.

Interfaces Following the scenario specification (cf. Figure 1), the test program must invoke the `getCustomer` and `updateCustomer` operations on `CustomerManagement` to test its correct behaviour, which entails interacting with `CustomerCore`. Here, we let the test directly interact with `CustomerCore`, in the "setup" phase, to create the customer (via the `createCustomer` operation) that we want to get and update in the test case. Listing 2 shows the Jolie interfaces of the test for Scenario 1.

Listing 2. Interfaces of the test for Scenario 1.

```
1  type CustomerProfileUpdateRequest { firstName:string, lastName:string, ... }
2
3  type CustomerResponse {
4   customerId? :string, firstName? :string, lastName? :string, ...
5  }
6
7  interface CustomerInformationHolder_CustomerCore {
8   RequestResponse:
9    createCustomer(CustomerProfileUpdateRequest)(CustomerResponse)
10 }
11
12 type GetCustomerRequest {
13  ids:string, fields?:string
14 }
15
16 type UpdateCustomerRequest {
17  customerId:CustomerId
18  requestDto:CustomerProfileUpdateRequest
19 }
20
21 interface CustomerInformationHolder_CustomerManagement {
22  RequestResponse:
23   getCustomer(GetCustomerRequest)(CustomerResponse),
24   updateCustomer(UpdateCustomerRequest)(CustomerResponse)
25 }
```

The `CustomerInformationHolder_CustomerCore` interface[8] in Lines 7–10 specifies the signature of the `CustomerCore` microservice's `createCustomer` operation used to setup the test database. The operation is a synchronous request-response operation (cf. Figure 1), and expects an instance of the `CustomerProfileUpdateRequest` type (cf. Line 1) as input and returns a `CustomerResponse`

[8] Note that the prefix `CustomerInformationHolder` refers to the microservice API pattern Information Holder Resource conceived by the developers of Lakeside Mutual, and enabling the provisioning of domain data with integrity and quality preservation [23].

(cf. Lines 3–5) as output, whereby the most of the fields of the `CustomerResponse` type correspond to those of `CustomerProfileUpdateRequest` with optional cardinality (`?`)—Jolie also provides the `*` cardinality that means a 0-to-unbound number of elements of that type. An exception is the `customerId` field by which the `CustomerCore` microservice informs invokers of `createCustomer` about the unique identifier of a newly created customer.

The `CustomerInformationHolder_CustomerManagement` interface in Lines 21–25 specifies the `getCustomer` and `updateCustomer` operations used in the test. The operation `getCustomer` expects an instance of the `GetCustomerRequest` type (cf. Lines 12–14) to determine the identifiers of the customers to be retrieved and optionally a list of relevant fields. The operation then returns matching data in a `CustomerResponse` instance. Operation `updateCustomer` requires an instance of the type `UpdateCustomerRequest` (cf. Lines 16–19) with the customer identifier to be updated by the passed `CustomerProfileUpdateRequest` instance. As for `getCustomer`, `updateCustomer` then returns its results in the form of `CustomerResponses`.

Access Points As mentioned, the JoT test for Scenario 1 has two output ports, `CustomerCore` and `CustomerManagement`. Listing 3 shows the expected bindings.

Listing 3. Access points of the test for Scenario 1.

```
1  outputPort customerCore {
2   location: parameters.customerCore.location
3   ...
4   interfaces: CustomerInformationHolder_CustomerCore
5  }
6
7  outputPort customerManagement {
8   location: parameters.customerManagement.location
9   ...
10  interfaces: CustomerInformationHolder_CustomerManagement
11 }
```

We specify at Lines 1–5 the output port for the `CustomerCore` microservice while at Lines 7–11 we report the output port for the `CustomerManagement` microservice. Notice that the actual binding of the ports (location, protocol) is parametric (passed through the `parameters` variable of the test).

Test Logic Listing 4 shows the testing logic of Scenario 1. Notice that we import the same interface `CustomerInformationHolder` from different files (i.e., `customer-core.interfaces` and `customer-management.interfaces`) and we alias them (with the `as` keyword) resp. `CustomerInformationHolder_CustomerCore` and `CustomerInformationHolder_CustomerManagement`, so that we obtain a similar result as the code in Listing 2.

Listing 4. Logic of the test for Scenario 1.

```
1  // cf. Listing 2
2  from customer-core.interfaces import CustomerInformationHolder
```

```
3   as CustomerInformationHolder_CustomerCore
4  from customer-management.interfaces import CustomerInformationHolder
5   as CustomerInformationHolder_CustomerManagement
6
7  interface TestInterface {
8   RequestResponse:
9    /// @BeforeEach
10   setup(void)(void),
11   /// @Test
12   testScenario1(void)(void)
13 }
14
15 service Main {
16  outputPort customerCore { cf. Listing 3 }
17  outputPort customerManagement { cf. Listing 3 }
18  inputPort Input { ... }
19
20  main {
21   /* Setup Test */
22   [ setup()() {
23    request << { firstName = "Jane", lastName = "Doe", ... }
24    createCustomer@customerCore(request)(actual)
25    global.user_id = actual.customerId
26   } ]
27
28   /* Test Scenario 1 */
29   [ testScenario1()() {
30    // Step 1
31    getCustomer@customerManagement({ ids = global.user_id })(resp)
32    equals@assertions
       ({ actual << resp.customerId, expected << global.user_id })()
33
34    // Step 2
35    undef(resp.customerId)
36    resp.firstName = "John2"
37    updateCustomer@customerManagement
       ({ customerId = global.user_id, requestDto << resp })(resp2)
38    equals@assertions({ actual = resp2.firstName, expected = "John2" })()
39
40    // Step 3
41    getCustomers@customerManagement({ ids = global.user_id })(resp3)
42    equals@assertions({ actual = #resp3.customers, expected = 1 })()
43   } ] } }
```

Briefly, Lines 2–5 import the types and interfaces for the `CustomerCore` and `CustomerManagement` microservices (cf. Listing 2).

Next, in Lines 7–13 we specify the `TestInterface` of the test. This has two operations with JoT-specific annotation. We use `@BeforeEach` to invoke the

`setup` operation before each test (here, just one). Then, we annotate with `@Test testScenario1`, which will execute after all `@BeforeEach` (here, one) operations.

Starting from Line 15 we find the implementation of the test, as a Jolie `service`. There, we find the output ports to access `CustomerCore` and `CustomerManagement` microservices (cf. Lines 16 and 17), the input port `Input` that offers the test operations found in the `TestInterface` to the JoT framework orchestrator. The `main` block encloses the implementation of the logic of the test.

Specifically, we find at Lines 22–26 the behaviour of the `setup` operation, which creates a `request` value with test data based on the structure of the `CustomerProfileUpdateRequest` type (cf. Listing 2) and it uses the latter in the invocation of `createCustomer` of `CustomerCore`. Since `setup` is run before all tests (as per its annotation), the `@Test`s can assume that the microservice's database has the test entry. The resulting identifier of the created customer is then stored in a `global` field called `user_id`, accessible by all test cases.

Lines 29–43 comprise the actual logic for the test operation of Scenario 1, i.e., `testScenario1`. First, the operation retrieves the test customer previously created by the `setup` operation. However, this call addresses the `CustomerManagement` rather than the `CustomerCore` microservice and thus verifies whether `CustomerManagement` actually behaves as a façade for `CustomerCore` as anticipated by Lakeside Mutual's architecture design (cf. Figure 1). In the second step, the test operation changes the name of the test customer from "Jane" to "John2" and issues a request to the `updateCustomer` operation of the `CustomerManagement` microservice. The response of the latter operation is then checked to report the new name of the customer as expected by `updateCustomer` after a successful update of customer data. In its final step, `testScenario1` verifies that the update is persistent by issuing a `getCustomers` request to the `CustomerManagement` microservice.

3.3 JoT Test of Scenario 2

We describe the JoT test of Scenario 2 (Figure 2) following the same structure of Section 3.2: interfaces, access points, logic.

Interfaces Listing 5 shows the type definitions and operations of the interfaces of the `CustomerSelfService` and `CustomerManagement` for Scenario 2 (cf. Figure 2).

Listing 5. Interfaces of the Jolie test program for Scenario 2.

```
1  type CustomerRegistrationRequest { firstName:string, lastName:string, ... }
2
3  interface CustomerInformationHolder_CustomerSelfService {
4   RequestResponse:
5    registerCustomer(CustomerRegistrationRequest)(CustomerResponse)
6  }
7
8  type GetCustomersRequest {
9   filter?:string, fields?:string, limit?:int, offset?:int
10  }
11
```

```
12 type PaginatedCustomerResponse {
13  filter?:string, limit?:int, offset?:int, size?:int
14  customers*:CustomerResponse // cf. Lines 3-5 in Listing 2
15 }
16
17 interface CustomerInformationHolder_CustomerManagement {
18  RequestResponse:
19   getCustomers(GetCustomersRequest)(PaginatedCustomerResponse)
20 }
```

The `CustomerInformationHolder_CustomerSelfService` interface of `CustomerSelfService` specifies the `registerCustomer` operation for the registration of new insurance customers with the Lakeside Mutual platform. It requires an instance of the `CustomerRegistrationRequest` type (cf. Line 1) as input and returns an instance of the `CustomerResponse` type (cf. Lines 3–5 in Listing 2).

The `CustomerInformationHolder_CustomerManagement` interface of `CustomerManagement` gathers the `getCustomers` operation (cf. Lines 17–20 in Listing 5), which lets users fetch customers based on the `GetCustomersRequest` type (cf. Line 10). An instance of the type determines the `filter` string and `fields` for customer matching. In case one of the fields of the record associated with a registered customer includes the `filter` string, the record will be part of the set of customers returned by `getCustomers`. The size of the set can be controlled by the `limit` and `offset` fields of `GetCustomersRequest`—the former prescribes the number of records in the set and the latter indicates by which offset customer matching shall start. With this mechanism, `getCustomers` supports paginated requests of customer records as modelled by the operation's return type `PaginatedCustomerResponse` (cf. Lines 12–15). An instance of the type informs the caller about the employed `filter` string, the prescribed `limit` and `offset`, as well as the `size` of the resulting record set. The set itself is comprised by the list of `CustomerResponses` in the `customers` field.

Access Points In Scenario 2, the `Client` performs direct interactions with `customerSelfService` and `customerManagement` and the test has the related ports. Since it introduces no salient elements, we omit to show the access point code for brevity.

Test Logic Listing 6 shows the test logic for Scenario 2.The imports we have at the beginning are similar to the ones included for Scenario 1, i.e., we alias `CustomerInformationHolder` for either the `CustomerManagement` and the `CustomerSelfService` resp. as `CustomerInformationHolder_CustomerManagement` and `CustomerInformationHolder_CustomerSelfService`, so that we obtain a similar result as the code in Listing 5. In the code, we use both the Jolie value-assignment operator `=` and the deep-copy operator `«`. The first just copies the topmost element of the expression on its right. The second copies the whole structure referred by the expression on the right.

Listing 6. Logic of the Jolie test program for Scenario 2.

```
1 // cf. Listing 5
2 from customer-management.interfaces import CustomerInformationHolder
```

```
 as CustomerInformationHolder_CustomerManagement
from customer-self-service.interfaces import CustomerInformationHolder
 as CustomerInformationHolder_CustomerSelfService

interface TestInterface {
 RequestResponse:
  /// @Test
  testScenario2(void)(void)
}

service Main {
 outputPort customerManagement { ... }
 outputPort customerSelfService { ... }
 inputPort Input { ... }

 main {
  [ testScenario2()() {
   // Step 1
   customer << { firstName = "Homer2", lastName = "Simpson", ... }
   registerCustomer@customerSelfService(customer)(resp1 )
   equals@assertions({ actual = resp1.firstName, expected = "Homer2" })()

   // Step 2
   getCustomers@customerManagement({ filter = "Homer2" })(resp2)
   equals@assertions({ actual = #resp2.customers, expected = 1 })()
   equals@assertions
     ({ actual = resp2.customers.firstName, expected = "Homer2" })()
  } ] }
}
```

Similar to Listing 2, we: (i) import the types and interfaces of the microservices involved in the scenario; (ii) define the `TestInterface`; (iii) specify the involved microservices' output ports; and (iv) define the test logic.

Focusing on the latter, Step 1 creates a test customer by invoking the `registerCustomer` operation of the `customerSelfService` microservice. At Step 2 we use `getCustomers` of `customerManagement` to fetch (and filter) the created customer, checking that there exists exactly one customer with the given name.

4 Related Work, Discussion, and Conclusion

We presented JoT, a testing framework for MSAs based on technology agnosticism. JoT tests are orchestrators that can consume or offer operations from/to the MSA under test. Since JoT adopts Jolie as the language for writing tests, it provides constructs supporting technology agnosticism and the definition of terse test behaviours. These elements facilitate the testing of MSAs with microservices based on heterogeneous technology stacks and the reuse of tests under different deployment configurations. Recent surveys and interviews with practitioners [21,22] substantiate this need, pointing out that developers urge for microservice-specific testing solutions.

We reference [21,22] for a comprehensive survey of the field, while, here, we compare with the closest proposals to ours. Gremlin [12] is a framework for MSAs that focuses on testing failure-handling by manipulating inter-service messages at the network layer. Quenum and Aknine [19] conceive an approach for the generation of executable test cases from requirements specifications, thereby focusing on acceptance tests for validating a software system's conformance with stakeholder expectations.

Hillah et al. [13] present an approach to automated functional testing based on formal specifications (of services, relations, etc.). Jayawardana et al. [15] propose a framework to produce test skeletons from business process models.

All mentioned related works concentrate on different aspects of MSA testing than JoT. In particular, they do not focus on the specification of advanced MSA tests tailored to technology agnosticism and expressed using a terse syntax, like the one provided by JoT thanks to the usage of the Jolie language. We plan to study the possible interplay between the mentioned work with JoT, e.g., for semi-automatic test generation geared towards specific traits of the architecture under test.

To improve the reliability of JoT we intend to conduct more comprehensive validation of our tool. One such validation entails more varied and complex scenarios, including synchronous and asynchronous interactions, design and architecture patterns, like Sagas for distributed transactions and Circuit Breaker for increased reliability.

In particular, looking at the design and architecture patterns, we foresee the language for test behaviours (inherited from Jolie) would play a fundamental role in helping users express complex testing logic spanning different services. Also this aspect deserves dedicated work, i.e., how the JoT behaviour increases the productivity of testers w.r.t. existing solutions. Both empirical studies with practitioners and applying relevant software quality metrics, comparing with both existing tools for general testing (e.g., JUnit) specific to microservices (e.g., zerocode[9], Microdot[10], [11] and MounteBank[12]).

Other future endeavours regard studying the integration of JoT with MSA modelling languages like LEMMA [20] and MDSL [23], and with choreographic testing approaches [9,10,2,1]. Such an integration would allow the generation of test behaviours and coordinators in contexts where a single orchestrator is not sufficient, e.g., in decentralized, cross-organizational deployments. Furthermore, Jolie types and interfaces provide natural support for property-based testing [5], where generators randomly run tests on valid data and operations to assert relevant invariants. In this context, one could use session types [14] to specify behavioural invariants that shall hold in the system and test these in a property-based manner.

Acknowledgements This work was partially supported by the Independent Research Fund Denmark, grant no. 0135-00219, Villum Fonden, grant no. 29518, and Innovation Fund Denmark, grant no. 9142-00001B.

[9] `https://github.com/authorjapps/zerocode`.
[10] `https://github.com/gigya/microdot`.
[11] `https://pact.io/`.
[12] `https://www.mbtest.org/`.

References

1. Coto, A., Guanciale, R., Tuosto, E.: On testing message-passing components. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12476, pp. 22–38. Springer (2020)
2. Coto, A., Guanciale, R., Tuosto, E.: An abstract framework for choreographic testing. J. Log. Algebraic Methods Program. **123**, 100712 (2021)
3. Dalla Preda, M., Gabbrielli, M., Lanese, I., Mauro, J., Zavattaro, G.: Graceful interruption of request-response service interactions. In: Service-Oriented Computing: 9th International Conference, ICSOC 2011, Paphos, Cyprus, December 5-8, 2011 Proceedings 9. pp. 590–600. Springer (2011)
4. Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216. Springer (2017). https://doi.org/10.1007/978-3-319-67425-4_12
5. Fink, G., Bishop, M.: Property-based testing: a new approach to testing for assurance. ACM SIGSOFT Software Engineering Notes **22**(4), 74–80 (1997)
6. Gabbrielli, M., Giallorenzo, S., Lanese, I., Zingaro, S.P.: A language-based approach for interoperability of iot platforms. In: 51st Hawaii International Conference on System Sciences, HICSS 2018, Hilton Waikoloa Village, Hawaii, USA, January 3-6, 2018. pp. 1–10. ScholarSpace / AIS Electronic Library (AISeL) (2018)
7. Gabbrielli, M., Giallorenzo, S., Montesi, F.: Service-oriented architectures: From design to production exploiting workflow patterns. In: Distributed Computing and Artificial Intelligence, 11th International Conference, DCAI 2014, Salamanca, Spain, June 4-6, 2014. pp. 131–139. Springer (2014). https://doi.org/10.1007/978-3-319-07593-8_17
8. Gamma, E., Beck, K.: Junit (2006)
9. Giallorenzo, S., Lanese, I., Russo, D.: Chip: A choreographic integration process. In: On the Move to Meaningful Internet Systems. OTM 2018 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, October 22-26, 2018, Proceedings, Part II. pp. 22–40. Springer (2018)
10. Giallorenzo, S., Montesi, F., Peressotti, M.: Choreographies as objects. CoRR **abs/2005.09520** (2020), https://arxiv.org/abs/2005.09520
11. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S.: Jolie and LEMMA: model-driven engineering and programming languages meet on microservices. In: Damiani, F., Dardha, O. (eds.) Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12717, pp. 276–284. Springer (2021). https://doi.org/10.1007/978-3-030-78142-2_17
12. Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M.K., Sekar, V.: Gremlin: Systematic resilience testing of microservices. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). pp. 57–66. IEEE (2016)
13. Hillah, L.M., Maesano, A.P., De Rosa, F., Kordon, F., Wuillemin, P.H., Fontanelli, R., Bona, S.D., Guerri, D., Maesano, L.: Automation and intelligent scheduling of distributed system functional testing: Model-based functional testing in practice. International journal on software tools for technology transfer **19**, 281–308 (2017)

14. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P.M., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., et al.: Foundations of session types and behavioural contracts. ACM Computing Surveys (CSUR) **49**(1), 1–36 (2016)
15. Jayawardana, Y., Fernando, R., Jayawardena, G., Weerasooriya, D., Perera, I.: A full stack microservices framework with business modelling. In: 2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer). pp. 78–85. IEEE (2018)
16. Montesi, F.: Process-aware web programming with Jolie. Sci. Comput. Program. **130**, 69–96 (2016)
17. 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 (2014). https://doi.org/10.1007/978-1-4614-7518-7_4, https://doi.org/10.1007/978-1-4614-7518-7_4
18. OMG: OMG Unified Modeling Language (OMG UML) version 2.5.1. Standard formal/17-12-05, Object Management Group (2017)
19. Quenum, J.G., Aknine, S.: Towards executable specifications for microservices. In: 2018 IEEE International Conference on Services Computing (SCC). pp. 41–48. IEEE (2018)
20. Rademacher, F.: A language ecosystem for modeling microservice architecture. Ph.D. thesis, University of Kassel, Germany (2022), https://kobra.uni-kassel.de/handle/123456789/14176
21. Waseem, M., Liang, P., Márquez, G., Di Salle, A.: Testing microservices architecture-based applications: A systematic mapping study. In: 2020 27th Asia-Pacific Software Engineering Conference (APSEC). pp. 119–128. IEEE (2020)
22. Waseem, M., Liang, P., Shahin, M., Di Salle, A., Márquez, G.: Design, monitoring, and testing of microservices systems: The practitioners' perspective. Journal of Systems and Software **182**, 111061 (2021)
23. Zimmermann, O., Stocker, M., Lübke, D., Zdun, U., Pautasso, C.: Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges. Addison-Wesley (2023)