JoT: A Jolie Framework for Testing Microservices

Saverio Giallorenzo[a], Fabrizio Montesi[b], Marco Peressotti[b], Florian Rademacher[c], Narongrit Unwerawattana[b]

[a] *Università di Bologna, Italy and INRIA, France*
[b] *University of Southern Denmark, Denmark*
[c] *RWTH Aachen University, Germany*

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. 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.

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

1. Motivation and significance

The microservice architectural style has become one of the state-of-the-art paradigms for building distributed systems. One of its main traits is consolidating the functionalities found in a distributed system in distinct, independent software units, called microservices. This consolidation action usually follows principles like coherence [4] and context-boundedness [6]; the choice of which of these principles to follow integrates concerns like scalability

Email addresses: `saverio.giallorenzo@gmail.com` (Saverio Giallorenzo), `fmontesi@imada.sdu.dk` (Fabrizio Montesi), `peressotti@imada.sdu.dk` (Marco Peressotti), `rademacher@se-rwth.de` (Florian Rademacher), `nau@sdu.dk` (Narongrit Unwerawattana)

Nr.	Code metadata description	Please fill in this column
C1	Current code version	0.0.27
C2	Permanent link to code/repository used for this code version	`https://github.com/jolie/jot`
C3	Permanent link to Reproducible Capsule	`ghcr.io/jolie/jot`
C4	Legal Code License	GNU Lesser General Public License, version 2.1
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	Jolie
C7	Compilation requirements, operating environments and dependencies	Java SDK 11, NPM 10
C8	If available, link to developer documentation/manual	`https://github.com/jolie/jot/blob/main/README.md`
C9	Support email for questions	`nau@sdu.dk`

Table 1: Code metadata (mandatory)

(so that one can scale as few microservices as possible) and code reusability (so that one can reuse the same microservice in different architectures). Moreover, microservices embrace technology-agnosticism, i.e., they give programmers the freedom to use the most suitable technologies to implement the functionalities of any given microservice.

The essential feature to support scalability, reusability, and technology-agnosticism in microservices is the usage of Application Programming Interfaces (APIs), which primarily fix the set of operations offered by a microservice, followed by the interaction patterns and technologies used to interact with them [8].

Unfortunately, a negative aspect of technology agnosticism is that it makes testing the correct interaction between the microservices in an architecture difficult to both specify and perform. Within the same microservice, one can rely on time-tested techniques and technology-specific frameworks (like JUnit[1] for Java and Cucumber [25] for Ruby) for testing its implementation (unit, integration, and end-to-end, as long as the scope of the testing routine remains within the boundaries of the same microservice).

However, when testing the interaction among different microservices, the framework cannot rely on technology-specific assumptions and only work

[1] `https://www.junit.org`.

with the APIs a microservice provides. Using languages intended for internal development—like Java, Rust, C, etc.—makes the definition of non-trivial testing scenarios complex. Even taken in isolation, consuming multiple operations of different microservices entails the definition of dedicated routines for establishing connections (and handling their state/errors) and marshalling/unmashalling the data. Moreover, tests written in this way are difficult to be reused under different deployment settings—imagine repurposing a test that uses HTTP endpoints to verb-based binary protocols.

JoT [12] is a microservice testing framework designed to support technology agnosticism. In JoT, tests are orchestrators that can both consume or offer operations from/to an architecture under test. The language for writing JoT tests is Jolie [18], which provides constructs that support technology agnosticism [17] and the definition of terse test behaviours.

While designed to support unit, integration, and end-to-end testing, we see the last two levels of testing as the distinctive ones for JoT, since these are the levels where a test necessarily needs to interact with (different) microservices through their APIs. The need for frameworks like JoT is both timely and pragmatic, for example, recent surveys and interviews with practitioners [24, 23] reported how end-to-end testing is one of the most used testing strategies, but that developers urge for microservice-specific testing solutions.

Besides JoT, other proposals tackle the area of testing microservices. Gremlin [14] is a framework focused on testing failure-handling by manipulating inter-service messages at the network layer. Quenum and Aknine [20] 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. [15] present an approach to automated functional testing based on formal specifications (of services, relations, etc.). Jayawardana et al. [16] propose a framework to produce test skeletons from business process models. All mentioned works concentrate on different aspects of testing for microservices, yet none focuses (like JoT does) on the specification of tests tailored to technology agnosticism. Furthermore, JoT offers a terse syntax that supports the definition of complex scenarios thanks to the usage of the Jolie language.

In Section 2, we present the architecture of JoT — its components, their relationship, and the logic for running tests — and the functionalities offered by the tool. Then, we exemplify the definition of JoT tests in Section 3 and describe the impact of the software and its planned evolution in Section 4.

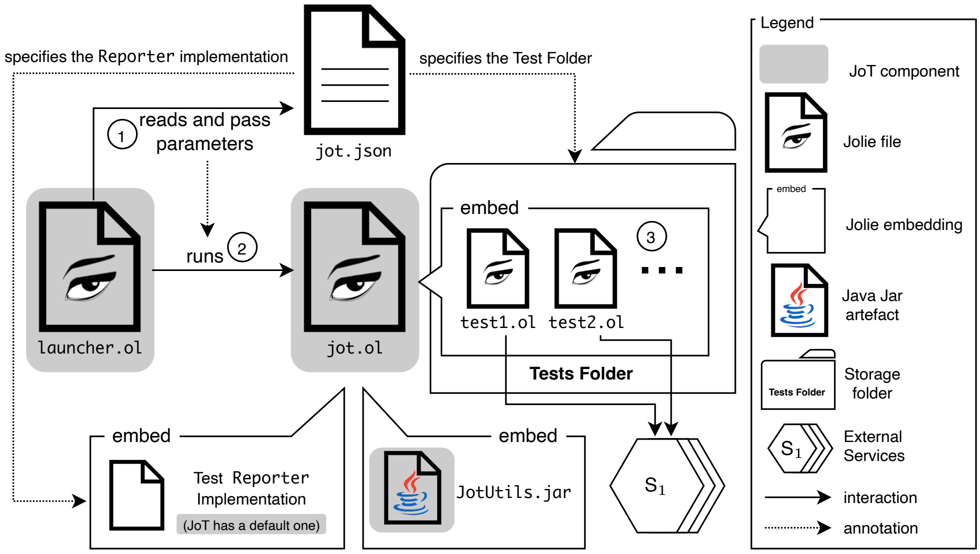

Figure 1: Overview of the JoT components interactions. The shaded elements belong to the default JoT software package; the other parts correspond to a specific set of tests.

2. Software description

We structure the description of JoT by first introducing, in Section 2.1, its components and presenting their relationships within the JoT software architecture and the components that characterise tests. Then, we overview the orchestration logic for running the tests (Section 2.1.1). In Section 2.2, we present the functionalities offered by JoT and comment on the template of a JoT test.

2.1. Software architecture

We represent the architecture of JoT and the interaction among its components and the tests in Figure 1.

Let us start from the components that make up the architecture of JoT, which are shaded in Figure 1: `launcher.ol`, `jot.ol`, and `JoTUtils.jar`. In the figure, we use the Jolie and the Java logos to indicate that the component is implemented using that language.

The first component, `launcher.ol`, launches the JoT routine, and it is the only one exposed to users (see Section 2.2 for a description of the supported commands). To start the test framework, `launcher.ol` (1) reads the parameters of the tests found in a companion file `jot.json` (see Section 2.2 for a summary of the structure of the file). Mainly, the file indicates i) what implementation of the `Reporter` service JoT shall use to output the notifications of the execution and results of the tests (as the shaded text on the bottom left side of Figure 1 hints, JoT comes with a default `Reporter` that forwards the notifications to the standard output); ii) the location of the tests (the **Tests Folder** shown in Figure 1); and iii) the parameters needed for the execution of each test (omitted in the figure), e.g., the parameters to access external services.

The `jot.ol` file encloses the orchestration logic over the tests, and it is run by `launcher.ol`, which passes to it the parameters read from the `jot.json` file (2). The `jot.ol` orchestrator uses Jolie's `embedding` feature to integrate the functionalities of both the other components that make up its architecture and the tests defined by the users. In general, Jolie's `embedding` primitive allows users to run a Jolie program inside the execution context of another one, called the embedder. The embedder and embedded services can communicate through ordinary communication ports (e.g., TCP/IP connections), but since they share the same runtime environment, they can also rely on the use of efficient in-memory channels. Drawing an analogy, we can see embedding as a mechanism for having a service use other services as libraries. For example, we can have a Console service that provides operations that write on the standard output. Then, a service can embed Console to privately access its operations for output—which is what happens when we include in a Jolie program the Console service of the language's standard library. Since the embedded services share the same runtime environment of the embedder, they cannot outlive the latter's termination. In the case of JoT, `jot.ol` uses the `embedding` primitive to internally execute i) the `Reporter`, ii) the `JoTUtils.jar` services, used to extract information on the tests (annotations in particular), and iii) the tests found in the **Tests Folder**.

As shown in Figure 1, the tests (3) can communicate with external services, e.g., microservices, databases, etc. (the hexagons at the bottom of the figure). These services are not controlled by JoT and the user has (if needed) to manage their execution and provide the parameters for the tests to access the former in the `jot.json` file.

2.1.1. Testing orchestration logic

We complete our description of the relationships among the components of the JoT architecture by briefly presenting the logic of the testing orchestration, also reported in algorithmic form in Algorithm 1 for a clearer overview.

The orchestration logic for running the tests, as shown in Algorithm 1, requires three inputs: the `tests_path`, which defines the location of the test files (the **Tests Folder** in Figure 1), the `tests_param` structure, which contains the parameters needed by the tests to run, and the `reporter`, which is the instance of the `Reporter` (cf. Figure 1) that handles the output of the notifications.

For a clearer exposition, we proceed by commenting on the steps of the testing orchestration logic by referring to the lines of Algorithm 1.

The first action we perform (Line 1 of Algorithm 1) is collecting all the test files, found under the `tests_path`. For each file, we execute the following routine.

Each test contains a set of operations that can be associated with JoT annotations[2] which JoT collects (Line 2) (using the functionalities offered by `JoTUtils.jar`). Then, JoT embeds the test file and puts it in execution, passing to it its related parameters found in the `jot.json` file. After having notified the `Reporter` of the starting of the tests on the selected file (Line 4), the JoT orchestrator first invokes all the operations of the test marked as *beforeAll* (Line 5). These are operations that shall be executed before all the tests and are usually intended for general setup (e.g., connecting to a database to prepare the pool of connections used by the tests). Then, for each `test`, we first execute all the operations marked *beforeEach* (Line 7) — JoT does not impose an ordering among the annotated operations —, followed by the execution of the `test` and the notification of the result to the `reporter` (Line 8). After the execution of the `test`, all the operations marked *afterEach* run (Line 9). Once all the tests have been run, all operations marked *afterAll* execute (Line 11, e.g., to close the connections opened at Line 5) and the `reporter` is notified of the closure of the test procedure for that `test_file` (Line 12). Then, the procedure either passes to the next file or, if no other file is available, it closes the testing session.

2.2. Software functionalities

The main functionality of JoT, provided to the users by the `launcher.ol` service (cf. Section 2.1), is the execution of the tests, which follows the logic described in Section 2.1.1. This functionality is invoked by running the command `jolie launcher.ol jot.json`, where `jot.json` is the (path to the) file containing the parameters to execute the tests.

Listing 1 shows an example of a JoT configuration file while Listing 2 illustrates the `interface`-level annotations of the operations of a test.

In Listing 1, we configure the execution of the JoT test `MyTest`, stored in a Jolie program within the file `MyTest.ol`. In Listing 1, `testsPath` specifies the file path of the test source relative to the configuration file, while under

[2]This approach is similar to other test frameworks, like JUnit, where programmers associate Java methods in test files with JUnit annotations to inform the JUnit test engine on their execution.

Algorithm 1 Pseudocode of the JoT testing orchestration logic.

```
Input
    tests_path     location of the test files
    tests_param    set of test-operation parameters
    reporter       reporter's instance

for test_file in collect( files from tests_path ) do
    test_operations ← collect( test operations from test_file )
    embed( test_file, tests_param[ test_file ] )
    notifyStart( reporter, test_file )
    invokeAllMarked( test_operations, "beforeAll" )
    for test in selectMarked( test_operations, "test" ) do
        invokeAllMarked( test_operations, "beforeEach" )
        notifyResult( reporter, invoke( test ), test )
        invokeAllMarked( test_operations, "afterEach" )
    end for
    invokeAllMarked( test_operations, "afterAll" )
    notifyEnd( reporter, test_file )
end for
```

`params` we find the parameters related to the various tests — in Listing 1, we pass the parameter `"db_address"` to the service test `main` within the `MyTest.ol` file.

Listing 1: Example JoT configuration file.

```
{
 "testsPath": ".",
  "params": {
    "MyTest.ol": [{
      "name": "main",
      "params": {
        "db_address": "..."
      }
}] } }
```

Listing 2: Example JoT test file (annotations).

```
interface TestInterface {
  RequestResponse:
  /** @BeforeAll */ setupTest()()
  /** @BeforeEach */ setupCase()()
  /// @Test
  testCase()() throws TestFailed(string)
  /** @AfterEach */ cleanupCase()()
  /** @AfterAll */ cleanupTest()() }
// binding definitions
// implementations of operations
```

We now look at Listing 2, which shows an example of the annotations found in a JoT test. These annotations, associated with the element found on their right, are either in the form `/** @annotation */` or `///@annotation` (the notation is equivalent, except the former is in multiline form while the latter is single-line). In Listing 2, we find all the annotations mentioned when presenting the logic of orchestration of tests (cf. Section 2.1.1); in particular, we note that the signature for `/// @Test`s specifies that the operation (in Listing 2, `testCase`) can fail, throwing an error labelled `TestFailed`, which can carry a string explaining the reason of the failure. The comments at the bottom of Listing 2 refer to the definition of the bindings of the test (so that, e.g., it can communicate with external services, like the database we mentioned when describing Listing 1) and the behaviour of the test, where the user specifies the actions executed at the invocation of the different operations. We omit to describe the latter here and exemplify them in Section 3.

3. Examples

To demonstrate the main functionalities of JoT, we report two illustrative and concise examples. The first example is a prototypical scenario where JoT is used to test an external (run independently of JoT) service. The second example shows how one can leverage Jolie service orientation for running and managing the service under test within JoT. Both examples are available in the repository of JoT[3]. We refer the reader to [12], which is the paper that introduces the methodology behind JoT and details how developers can write tests with the framework, including more involved test scenarios taken from a reference microservice architecture.

The examples in this section assume an environment where both Jolie[4] and JoT[5] are available. A reference environment is available as a Docker Image at `ghcr.io/jolie/jot:latest`.

3.1. Testing an external service

The first example illustrates how to use JoT to test a service running independently of JoT. The service under test is the `Greeter` service from the Jolie documentation and can be thought of as the "hello world" example for Jolie. The service offers a single operation, which takes a name and returns a greeting message obtained from combining `"Hello, "` with the provided name e.g., `"Hello, Alice"` is the response for an request carrying the value `"Alice"`. The API of this service is reported in Listing 3.

Listing 3: Interface of Greeter service, `greeter.ol`.

```
type GreetRequest { name:string }
type GreetResponse { greeting:string }

interface GreeterAPI {
    RequestResponse: greet(GreetRequest)(GreetResponse)
}
```

[3] `https://github.com/jolie/jot/tree/main/examples`

[4] Installation instructions are available at `https://www.jolie-lang.org/`

[5] Installation instructions are available at `https://github.com/jolie/jot/`

To test the service, we consider two simple test cases. In the first case (`test1`) we invoke operation `greet` of service `Greeter` with a specific parameter and compare the actual response with one expected for that parameter. The comparison is carried out using the service `Assertions` provided by the Jolie standard library.

```
test1()() {
    // Invoke operation greet at service Greeter and store the reply in
    // the variable response.
    greet@Greeter({ name = "Alice" })(response)
    // invoke operation equals at service Assertions to compare the
    // response received with the expected one.
    equals@Assertions({
        actual = response
        expected = { greeting = "Hello, Alice" }
    })()
}
```

In the second case (`test2`) we invoke `greet` with a specific parameter twice and compare the two responses. The two invocations are executed concurrently using Jolie's parallel composition operator '`|`'.

```
test2()() {
  { greet@Greeter({ name = "Bob" })(response1)
  | greet@Greeter({ name = "Bob" })(response2) }
  equals@assertions({
      actual = response1
      expected = response2
  })()
}
```

Neither test case requires initialisation or finalisation operations so the interface that the test service offers to JoT is simply as follows.

```
interface TestInterface {
  RequestResponse:
      /// @Test
      test1(void)(void) throws AssertionError(string),
      /// @Test
      test2(void)(void) throws AssertionError(string)
  }
```

The entire test is specified as the service `TestGreeter` shown in the listing below.

Listing 4: `TestGreeter.json`.

```
service TestGreeter(params) {
    // Access point for consuming service Greeter
    outputPort Greeter {
```

```
        location: params.location
        protocol: params.protocol
        interfaces: GreeterAPI
    }
    // Access point offered to JoT
    inputPort JoT {
        location: "local"
        interfaces: TestInterface
    }
    // Service behaviour
    main {
        [ test1()(){ /* ... */ } ]
        [ test2()(){ /* ... */ } ]
    } }
```

This definition consists of three main elements:

1. An *output port* (`Greeter`) that specifies the location, protocol, and interface of the service under test. The first two are initialised using parameters of the service (`params.location`, `params.location`, respectively) and the third is set to the interface `GreeterAPI` from the beginning of this example.

2. An *input port* (`JoT`) that specifies how JoT interacts with this test service. The location is set to `"local"` since JoT embeds test services and interacts with them via in-memory communication.

3. A service behaviour (`main`) that implements the test cases as operations (`test1` and `test2` given above).

By parameterising the test service in the location and protocol to use for accessing the service to be tested, we can reuse the same test with different deployments. The values used by JoT to initialise these parameters are specified in the test configuration file (by convention, `jot.json`). For instance, the listing below shows how to configure JoT to run this test against an instance of `Greeter` reachable at a specific URI (`"socket://localhost:9000"`) via JSON-RPC.

Listing 5: `jot.json`.

```
{  "params": {
    "TestGreeter.ol": [{
        "name": "TestGreeter",
        "params": {
          "location": "socket://localhost:9000",
          "protocol": "jsonrpc"
        } } ] } }
```

The complete source code for this example (greeter service, test service, and configurations) is available in Appendix A.1 and in the repository of JoT at `https://github.com/jolie/jot/tree/main/examples/greeter`.

Below we show the results of running this test using JoT natively, using the Docker images for Jolie and JoT, and using NPM. All bash commands shown for these three cases are run from the following directory — which is based on the default directory structure for Jolie projects with JoT tests created by the Jolie Package Manager (JPM).

```
.
├── test
│   └── TestGreeter.ol
├── greeter.ol
└── jot.json
```

(JPM generates a NPM-compatible configuration file `package.json`, which we omit since it is not essential for this example.)

Below we show the result of running this test with JoT after launching `Greeter` on the same machine.

```
$ jolie greeter.ol &
  ...
$ jot jot.json
  TestGreeter.ol -> TestGreeter
    ✓ pass test2
    ✓ pass test1
  passes 2 (106ms) failures 0
```

Below, we show the same result using the Docker images for Jolie and JoT.

```
$ docker pull jolielang/jolie && docker pull ghcr.io/jolie/jot:latest
  ...
$ docker run -d --rm --network host --mount=type=bind,source="$(pwd)",target=/app jolielang/jolie jolie /app/greeter.ol
  ...
$ docker run --rm --network host --mount=type=bind,source="$(pwd)",target=/app ghcr.io/jolie/jot:latest
  TestGreeter.ol -> TestGreeter
    ✓ pass test2
    ✓ pass test1
  passes 2 (155ms) failures 0
```

Below we show the result of running this test with NPM after launching `Greeter` on the same machine.

```
$ jolie greeter.ol &
  ...
$ npm run test

```

```
> jot-greeter@1.0.0 test
> jot jot.json

 TestGreeter.ol -> TestGreeter
   ✓ pass test2
   ✓ pass test1
 passes 2 (150ms) failures 0
```

3.2. Testing an embedded service

The second example illustrates how to use JoT to test a service running within JoT and managed from the test itself through Jolie's embedding mechanism — used to run a service in a sandbox but within the same runtime environment of the embedder service. In particular, embedding ties the lifecycles of the test and embedded/tested services (the second is disposed of when the first terminates), and it allows tests and services to use in-memory channels rather than the network stack.

For this example we target a service from the Jolie standard library, `StringUtils`[6], which collects several common utility operations for handling strings. For conciseness, we consider the fragment of its API shown below.

Listing 6: Interface of StringUtils.

```
interface StringUtils {
    RequestResponse:
        length(string)(int),
        toLowerCase(string)(string)
}
```

To test this service, we consider a few simple test cases each invoking an operation with specific parameters and comparing the actual response with the one expected for that parameter. The snippet below contains a case for operation `length`.

```
testLength()() {
  length@stringUtils("12345678")(result)
  equals@assertions({
      actual = result
      expected = 8
  })()
}
/* ... */
```

[6]https://docs.jolie-lang.org/v1.11.x/language-tools-and-standard-library/standard-library-api/string_utils.html.

The test cases are collected in the following interface which is offered to JoT by the test service.

```
interface TestInterface {
RequestResponse:
  ///@Test
  testLength(void)(void) throws AssertionError(string),
  ///@Test
  testToLowerCase(void)(void) throws AssertionError(string)
  /* ... */
}
```

The rest of the test service is defined similarly to `TestGreeter` above save for the fact that in this example the target service is not assumed to be running independently and is instead executed by the test within the same process using the Jolie embedding mechanism. Instead of defining an output port, the service below embeds `StringUtils`.

Listing 7: `TestStringUtils.ol`.

```
service TestStringUtils() {
    embed StringUtils
    inputPort JoT { /* ... */}
    main {
        [ testLength()(){ /* ... */ } ]
        [ testToLowerCase()(){ /* ... */ } ]
    } }
```

The complete source code for this example (test service and JoT configuration) is available in Appendix A.2 and in the repository of JoT at `https://github.com/jolie/jot/tree/main/examples/stringUtils`.

Differently from the previous case, the service under test is started and terminated by the testing service using Jolie embedding mechanism. Below we show the results of running this test using JoT directly and the Docker image for JoT. In both cases, commands are launched from the following directory (we omit the `package.json`).

```
.
├── test
│   └── TestStringUtils.ol
└── jot.json
```

Below we show the result of running this test with JoT.

```
jot jot.json
  TestStringUtils.ol -> TestStringUtils
    ✓ pass testToLowerCase
    ✓ pass testLength
  passes 2 (33ms) failures 0
```

Below, we show the same result using the Docker image for JoT.

```
$ docker pull ghcr.io/jolie/jot:latest
  ...
$ docker run --rm --network="host" --mount=type=bind,source=$(pwd),target=/app
     ghcr.io/jolie/jot:latest
  TestStringUtils.ol -> TestStringUtils
    ✓ pass testToLowerCase
    ✓ pass testLength
  passes 2 (28ms) failures 0
```

4. Conclusion: Impact and Future Plans

We present JoT, a testing framework for microservice architectures designed around technology-agnosticism [12]. The only technology dependency of JoT is the Jolie programming language. Jolie's constructs for the abstraction of interface-related service technologies allow JoT to harmonise the interaction among microservices that leverage heterogeneous protocols and data formats. JoT provides service developers with a set of annotations that streamline microservice testing following best practices and popular testing frameworks like JUnit. For example, the `@Test` annotation identifies Jolie functions that implement test logic and the `@BeforeAll` annotation supports initialization of test environments, e.g., the generation and insertion of test data into databases. Based on these annotations, service developers can rely on a unified and declarative approach to the implementation of tests, while JoT takes care of instrumenting the Jolie interpreter to execute test functions consistently and in a replicable way.

JoT's technology agnosticism provides a foundation for stable microservice tests. More precisely, once written, JoT-based tests are invariant to re-implementations of microservices in other programming languages or frameworks — provided that microservices' interfaces remain stable and Jolie supports the employed communication protocols and data formats.

The versatility of Jolie also allows for JoT to be used in a variety of microservice testing approaches such as unit, integration, and end-to-end testing [23]. For instance, unit testing becomes feasible by using JoT within the microservice under test. For integration and end-to-end testing, a JoT-based test acts as an orchestrator that implements and runs test routines spanning several microservices [12].

We expect JoT to stimulate research concerning the provisioning of scalable as well as versatile microservice testing. In particular, we deem JoT tests scalable thanks to their microservices nature and efficient thanks to the support for reuse. In addition, we consider JoT a viable starting point for investigating and evaluating new ideas in the area of microservice testing — possibly by combining technology agnosticism with further microservice principles like ad-hoc scalability, which is helpful, e.g., in systematic resilience testing [14].

Concerning how JoT can impact practitioners' processes, we note that the framework is already adopted by Jolie developers[7] and envision its usage in existing Jolie projects [7, 11, 13]—which, in turn, can provide us with context-specific testing needs to make JoT further evolve. Furthermore, since Jolie microservices can easily be wrapped into virtualized containers[8], JoT-based tests implicitly align with established continuous integration approaches like the reconciliation of local development with production environments [5].

Looking at the future of JoT, we envision investigating the usage of JoT in the context of (semi-)automatic test generation, e.g., by identifying microservice interfaces using static analysis and subsequently applying suitable algorithms for test case generation [1]. Moreover, to improve JoT's reliability and developer experience, we plan its comprehensive validation using more complex scenarios (e.g., extending the coverage of realistic architectures, as done by Giallorenzo et al. [12]). These scenarios should involve synchronous and asynchronous microservice interactions as well as design and architecture patterns that are popular in microservice architectures, e.g., Sagas or Circuit Breakers [19, 22]. Additionally, we intend to conduct empirical evaluations of JoT with practitioners and in comparison to related tools like JUnit, Zerocode[9], Pact[10], and mountebank[11]. We also expect it to be fruitful to study the integration of JoT with microservice architecture modelling languages like LEMMA [21] and MDSL [26], and with choreographic testing approaches [9, 2, 3, 10]. Another goal is to eventually apply JoT for testing new releases of Jolie and its standard libraries, thereby exploiting advantages from compiler bootstrapping like introducing and enforcing consistency between language development and testing.

[7] `https://npm-stat.com/charts.html?package=%40jolie%2Fjot`.

[8] `https://docs.jolie-lang.org/v1.11.x/language-tools-and-standard-library/containerization`.

[9] `https://github.com/authorjapps/zerocode`

[10] `https://www.pact.io`

[11] `https://www.mbtest.org/`

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.

References

[1] Andrea Arcuri. 'RESTful API Automated Test Case Generation'. In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2017, pp. 9–20. DOI: 10.1109/QRS.2017.11.

[2] Alex Coto, Roberto Guanciale and Emilio Tuosto. 'An abstract framework for choreographic testing'. In: *J. Log. Algebraic Methods Program.* 123 (2021), p. 100712. DOI: 10.1016/J.JLAMP.2021.100712. URL: https://doi.org/10.1016/j.jlamp.2021.100712.

[3] Alex Coto, Roberto Guanciale and Emilio Tuosto. 'On Testing Message-Passing Components'. In: *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*. Ed. by Tiziana Margaria and Bernhard Steffen. Vol. 12476. Lecture Notes in Computer Science. Springer, 2020, pp. 22–38. DOI: 10.1007/978-3-030-61362-4_2.

[4] Nicola Dragoni et al. 'Microservices: Yesterday, Today, and Tomorrow'. In: *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

[5] Omar Elazhary et al. 'Uncovering the Benefits and Challenges of Continuous Integration Practices'. In: *IEEE Transactions on Software Engineering* 48.7 (2022), pp. 2570–2583. DOI: 10.1109/TSE.2021.3064953.

[6] Eric Evans. *Domain-Driven Design*. Addison-Wesley, 2004.

[7] Maurizio Gabbrielli et al. '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*. ScholarSpace / AIS Electronic Library (AISeL), 2018, pp. 1–10.

[8] Saverio Giallorenzo. 'Service-Oriented Programming Paradigm'. In: *Programming Languages: Principles and Paradigms*. Cham: Springer International Publishing, 2023, pp. 473–518. ISBN: 978-3-031-34144-1. DOI: 10.1007/978-3-031-34144-1_15.

[9] Saverio Giallorenzo, Ivan Lanese and Daniel Russo. '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*. Ed. by Hervé Panetto et al. Vol. 11230. Lecture Notes in Computer Science. Springer, 2018, pp. 22–40. DOI: 10.1007/978-3-030-02671-4_2.

[10] Saverio Giallorenzo, Fabrizio Montesi and Marco Peressotti. 'Choral: Object-Oriented Choreographic Programming'. In: *ACM Transactions on Programming Languages and Systems* (Nov. 2023). ISSN: 0164-0925. DOI: 10.1145/3632398.

[11] Saverio Giallorenzo et al. 'Ephemeral data handling in microservices with Tquery'. In: *PeerJ Comput. Sci.* 8 (2022), e1037. DOI: 10.7717/peerj-cs.1037.

[12] Saverio Giallorenzo et al. 'JoT: A Jolie Framework for Testing Microservices'. In: *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings*. Ed. by Sung-Shik Jongmans and Antónia Lopes. Vol. 13908. Lecture Notes in Computer Science. Springer, 2023, pp. 172–191. DOI: 10.1007/978-3-031-35361-1_10.

[13] Saverio Giallorenzo et al. 'LEMMA2Jolie: A tool to generate microservice APIs from domain models'. In: *Sci. Comput. Program.* 228 (2023), p. 102956. DOI: 10.1016/j.scico.2023.102956.

[14] Victor Heorhiadi et al. 'Gremlin: Systematic Resilience Testing of Microservices'. In: *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 57–66. DOI: 10.1109/ICDCS.2016.11. URL: https://doi.org/10.1109/ICDCS.2016.11.

[15] Lom-Messan Hillah et al. 'Automation and intelligent scheduling of distributed system functional testing - Model-based functional testing in practice'. In: *Int. J. Softw. Tools Technol. Transf.* 19.3 (2017), pp. 281–308. DOI: 10.1007/S10009-016-0440-3.

[16] Yasith Jayawardana et al. 'A Full Stack Microservices Framework with Business Modelling'. In: *2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer)*. 2018, pp. 78–85. DOI: 10.1109/ICTER.2018.8615473.

[17] Fabrizio Montesi. 'Process-aware web programming with Jolie'. In: *Sci. Comput. Program.* 130 (2016), pp. 69–96. DOI: 10.1016/J.SCICO.2016.05.002.

[18] Fabrizio Montesi, Claudio Guidi and Gianluigi Zavattaro. 'Service-Oriented Programming with Jolie'. In: *Web Services Foundations*. Ed. by Athman Bouguettaya, Quan Z. Sheng and Florian Daniel. Springer, 2014, pp. 81–107. DOI: 10.1007/978-1-4614-7518-7_4.

[19] Fabrizio Montesi and Janine Weber. 'From the decorator pattern to circuit breakers in microservices'. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*. Ed. by Hisham M. Haddad, Roger L. Wainwright and Richard Chbeir. ACM, 2018, pp. 1733–1735. DOI: 10.1145/3167132.3167427.

[20] José Ghislain Quenum and Samir Aknine. 'Towards Executable Specifications for Microservices'. In: *2018 IEEE International Conference on Services Computing, SCC 2018, San Francisco, CA, USA, July 2-7, 2018*. IEEE, 2018, pp. 41–48. DOI: 10.1109/SCC.2018.00013.

[21] Florian Rademacher. 'A language ecosystem for modeling microservice architecture'. PhD thesis. University of Kassel, Germany, 2022. URL: https://kobra.uni-kassel.de/handle/123456789/14176.

[22] Chris Richardson. *Microservices Patterns*. First. Manning Publications, 2019.

[23] Muhammad Waseem et al. 'Design, monitoring, and testing of microservices systems: The practitioners' perspective'. In: *J. Syst. Softw.* 182 (2021), p. 111061. DOI: 10.1016/J.JSS.2021.111061.

[24] Muhammad Waseem et al. 'Testing Microservices Architecture-Based Applications: A Systematic Mapping Study'. In: *27th Asia-Pacific Software Engineering Conference, APSEC 2020, Singapore, December 1-4, 2020*. IEEE, 2020, pp. 119–128. DOI: 10.1109/APSEC51365.2020.00020.

[25] Matt Wynne, Aslak Hellesoy and Steve Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.

[26] Olaf Zimmermann et al. *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley, 2023.

Appendix A. Examples, complete sources

Appendix A.1. Testing Greeter

In this section we report the complete source code for the example discussed in Sect. 3.1. These files are available from the repository of JoT at https://github.com/jolie/jot/tree/main/examples/greeter.

```
.
├── test
│   └── TestGreeter.ol
├── Greeter.ol
├── jot.json
└── package.json
```

Listing 8: Greeter.ol.

```
type GreetRequest { name:string }
type GreetResponse { greeting:string }

interface GreeterAPI {
    RequestResponse: greet(GreetRequest)(GreetResponse)
}

service Greeter {
    execution: concurrent

    inputPort GreeterInput {
        location: "socket://localhost:9000"
        protocol: jsonrpc
        interfaces: GreeterAPI
    }

    main {
        greet(request)(response) {
            response.greeting = "Hello, " + request.name
        }
    }
}
```

Listing 9: TestGreeter.ol.

```
from assertions import Assertions
from ..greeter import GreeterAPI

interface TestInterface {
RequestResponse:
    /// @Test
    test1(void)(void) throws AssertionError(string),
    /// @Test
```

```
    test2(void)(void) throws AssertionError(string)
}

// Test parameters: Greeter access point
type TestParams {
    location: string
    protocol: string
}

// Test service
service TestGreeter(params:TestParams) {

    execution: sequential

    // Access point of Greeter
    outputPort Greeter {
        location: params.location
        protocol: params.protocol
        interfaces: GreeterAPI
    }

    // A local instance of Jolie's Assertions service
    embed Assertions as assertions

    // Access point for JoT
    inputPort JoT {
        location: "local"
        interfaces: TestInterface
    }

    main {
        // Test case, a specific input
        [ test1()(){
            greet@Greeter({ name = "Alice" })(response)
            equals@assertions({
                actual = response
                expected = { greeting = "Hello, Alice" }
            })()
        } ]
        // Test case, equal result on equal input
        [ test2()(){
            greet@Greeter({ name = "Bob" })(response1) |
            greet@Greeter({ name = "Bob" })(response2)
            equals@assertions({
                actual = response1
                expected = response2
            })()
        } ]
    } }
```

Listing 10: `jot.json`.

```
{  "params": {
    "TestGreeter.ol": [{
        "name": "TestGreeter",
        "params": {
          "location": "socket://localhost:9000",
          "protocol": "jsonrpc"
        } } ] } }
```

Listing 11: `package.json`.

```
{
  "name": "jot-greeter",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "directories": {
    "test": "test"
  },
  "scripts": {
    "test": "jot jot.json"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@jolie/jot": "^0.0.25"
  }
}
```

Appendix A.2. Testing StringUtils

In this section we report the complete source code for the example discussed in Sect. 3.2. These files are available from the repository of JoT at `https://github.com/jolie/jot/tree/main/examples/stringUtils`.

```
.
├── test
│   └── TestStringUtils.ol
├── jot.json
└── package.json
```

Listing 12: `TestStringUtils.ol`.

```
from assertions import Assertions
from string_utils import StringUtils

interface TestInterface {
```

```
RequestResponse:
  ///@Test
  testLength(void)(void) throws AssertionError(string),
  ///@Test
  testToLowerCase(void)(void) throws AssertionError(string)
}

service TestStringUtils() {
    execution: sequential

    embed StringUtils as stringUtils
    embed Assertions as assertions

    inputPort JoT {
        location: "local"
        interfaces: TestInterface
    }

    main {
  [ testLength()() {
   length@stringUtils("12345678")(result)
   equals@assertions({
    actual = result
    expected = 8
   })()
  } ]
        [ testToLowerCase()() {
   toLowerCase@stringUtils("AbC dEf_GhI")(result)
   equals@assertions({
    actual = result
    expected = "abc def_ghi"
   })()
  } ]
    } }
```

Listing 13: jot.json.

```
{ "params": {
  "testStringUtils.ol": [{
      "name": "TestStringUtils"
  } ] } }
```

Listing 14: package.json.

```
{
  "name": "jot-greeter",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
```

```
  "directories": {
    "test": "test"
  },
  "scripts": {
    "test": "jot jot.json"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@jolie/jot": "^0.0.25"
  }
}
```