INSTITUT FÜR INFORMATIK
Ludwig-Maximilians-Universität München

Verification of Micro Services based on Pact API contracts

Robin Mattis

Bachelor Thesis

Supervisor	Prof. Dr. Dirk Beyer
Mentor	Dr. Thomas Lemberger
Submission Date	April 19, 2024

Statement of Originality

English:

Declaration of Authorship

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments.

Deutsch:

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, April 19, 2024 Robin Mattis

Abstract

Due to the growing use of microservices, it is becoming increasingly important to test these systems and ensure their correctness. However, current testing techniques such as Unit Tests, Integration Tests or Property-based Testing are not sufficient to ensure the correctness of microservices. In this thesis, we describe an approach to automatically create harnesses for formal verification for microservices implemented with the Java Spring framework, using Specifications from Pact.io, a contract-based testing framework. We contribute the tool PACT-VERIFIER and show that it is possible to automatically create harnesses for formal verification that can be verified by SV-COMP verifiers in a negligible runtime. Thus, we show that automatic formal verification in a microservice architecture is possible and can be done in a reasonable time frame.

Contents

List of Figures

Listings

List of Tables

1 Introduction

Motivation. In recent years, we have seen an increased use of microservices in the industry. The microservice architecture allows the development of complex software systems by developing isolated services that communicate over a network and can be deployed and scaled independently [50]. Thanks to these advantages, microservices are being used more and more by large companies, as well as start-ups. For example Twitter [3], Netflix [45] and the german platform gutefrage.net [52] are using microservices. Uber [10] already had more than 1000 microservices running in 2016 and thus heavily relies on the correct execution, implementation and communication of these services. Achieving correctness of the implementation and communication of these services is a challenging task. Caitie McCaffrey [46] states in her article that unit tests, integration tests and property tests, that are currently frequently used, create some confidence, but are not sufficient to ensure the correctness of a distributed system. Pact.io [14] also introduces a contract testing framework to ensure the correct communication of two microservices with integration tests. However, these tests are still example-based and cannot guarantee the absence of bugs. In our approach, we therefore focus on verifying the correctness of a microservice by using formal methods based on the defined Pact.io Specification for a service.

Overview. In our approach, we focus on the automatic verification of microservice endpoints implemented in Java using the Spring framework that have a corresponding Pact.io Specification defined. Therefore, we contribute the CLI tool PACT-VERIFIER [1] that generates verification harnesses based on the implemented endpoints and the specification. Figure 1.1 illustrates the basic workflow of this tool. To run the tool, the user provides the compiled Jar-File of the project to test and the Pact.io Specification containing interaction definitions for endpoints implemented in the project. The tool then generates verification harnesses for each defined interaction. The generated harnesses are then used as input for an SV-COMP verifier (e.g. JBMC [34], GDart [48]) that get's executed directly by PACT-VERIFIER to run the verification process. The output of the verifier is then gathered, and the user is informed about the verification verdict for each interaction.

Example. Listing 1 shows the Spring RestController class `TemperatureController`. The class contains the method `getAverageTemperatureByYear` that provides a GET endpoint returning the average temperature for a given year. Listing 2 defines the corresponding Pact.io Specification for this endpoint. The specification

[1]https://gitlab.com/sosy-lab/software/verification-for-pact-io

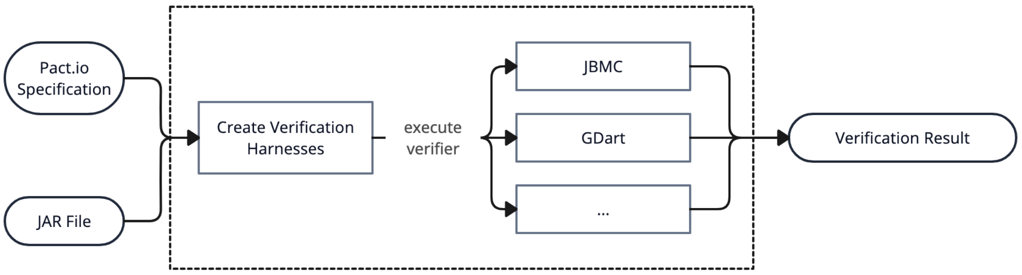

Figure 1.1: Basic workflow of the approach

```java
@RestController
public class TemperatureController {

    @GetMapping("temperature/{year}")
    public YearTemperature getAverageTemperatureByYear(
            @PathVariable("year") int year) {
        return new YearTemperature(year, 20);
    }

}
```

Listing 1: Example Spring RestController class with an endpoint returning the average temperature for a year

```json
{
  "description": "average temperature by year",
  "request": {"method": "GET", "path": "/temperature/2023" },
  "response": {
    ...
    "body": {"year": 2023,"temperature": 14},
    "matchingRules": {
      "body": {
        "$.year": {
          "matchers": [{"match": "equality"}],
          "combine": "AND"
        },
        "$.temperature": {
          "matchers": [{"match": "type", "min": -273, "max": 100}],
          "combine": "AND"
        }
      }
    }
  },
  ...
}
```

Listing 2: Corresponding Pact.io Specification for the temperature endpoint

```
1  public class TemperatureControllerVerificationContainer {
2
3      private static TemperatureController createConstructor() {
4          TemperatureRepository temperatureRepository = new TemperatureRepository();
5          TemperatureController temperatureController = new TemperatureController(temperatureRepository);
6          return temperatureController;
7      }
8
9      static void getAverageTemperatureByYear() {
10         // Verification Method for pact interaction with description "get average temperature by year"
11         TemperatureController temperatureController = createConstructor();
12
13         int year = Verifier.nondetInt();
14
15         YearTemperature result = temperatureController.getAverageTemperatureByYear(year);
16
17         assert (result.getTemperature() >= −273 && result.getTemperature() <= 100);
18     }
19 }
```

Listing 3: Generated Verification Harness for the temperature endpoint

contains an example request and response body, as well as matching rules for the response body. As defined by this specification, the response body has to contain the fields `year` and `temperature` (line 6). The matching rules then specify that the field `year` has to be equal to 2023 (line 9–12) and allows values from -273 to 100 for the field `temperature` (line 13–16), which in fact is the range of temperatures in Celsius.

Using these files as input, our approach is to create verification harnesses based on the Pact.io Specification and the underlying implementation. Listing 3 shows the generated class `TemperatureControllerVerificationContainer` containing the verification harness for the endpoint `getAverageTemperatureByYear` in the class `TemperatureController` based on the Pact.io Specification. The created class contains the constructor initialization (line 3–7, line 11) and calling the endpoint method (line 13–15) with the necessary parameters. The assertion (line 17) then checks if the temperature is within the specified range.

Within our approach, we then use the generated verification harnesses as input for SV-COMP verifiers and check if the verifier can find a property that does not hold.

Evaluation. To evaluate the approach and the contributed tool PACT-VERIFIER, we use the following methods:

RQ 1. Create Verification Harness from Specification: This research question consists of two parts:

1. Can we infer the correct Java classes used in the method of an endpoint in the request / response body from an example-based Pact.io Specification?
2. Can we generate a verification harness for the method of an endpoint that SV-COMP verifiers can verify?

Evaluation Plan: We show, that the arguments of a Pact.io Specification match with the arguments of the method in Java so that the method can be called correctly and the generated code compiles.

RQ 2. Real World Example: Can we find a real world example where we can determine an error of the application?
Evaluation Plan: Using the real-world microservice *court-case-service*, developed and maintained by the British Ministry of Justice, we show that the tool can create verification harnesses, and we can find an error in the application.

RQ 3. Negligible runtime: Is the overhead (in CPU time) of the verification harness generation negligible?
Evaluation Plan: We show that the overhead of the transformation is negligible by measuring the execution time of the CLI until the generation of the verification harnesses.

2 Related Work

Code Generation. The most basic type of validating the communication between consumer and provider is to generate the code for both parties based on a specification file (for instance by using Pact.io or OpenAPI). This way, using a strongly typed programming language, requests and responses can only be sent or received if they match the format defined in the specification. There are many tools providing code-generation for different languages and also for different specification languages. *api-codegen-ts* [12] is an example for a code-generation tool that generates a TypeScript representation of the specification as well as TypeScript functions to interact with the API operations. *gRPC-Java* [6] is a library that generates client and server code in Java based on a protocol buffer file. Several similar tools for gRPC [7] exist in various programming languages like Dart, C++, C and more.

Automatic Testing. Table 2.1 shows an (incomplete) list of testing tools for OpenAPI to automatically generate and run test-cases for a REST-API server by using Fuzzing or other similar techniques. While Step CI and Tcases only exist as open source projects, Schemathesis, EvoMaster and RESTest are also described in the literature. RESTest [44] is a tool that automatically generates test cases using fuzzing, constraint-based testing and others. It uses a black-box testing approach and due to its implementation as framework, it is able to generate code for multiple programming languages.
Other than RESTest, EvoMaster [21] uses a white-box testing approach. The goal of EvoMaster is to generate test-cases with the ability to find false HTTP return codes and to maximize the code-coverage of the generated tests.
Schemathesis [38] automatically generates test cases and uses different fuzzing techniques to find unexpected responses, server errors and crashes. Based on the API schema, Schemathesis generates structure- and semantics-aware fuzzers which can be used in unit tests or directly when running the CLI. The tool is implemented in Python and can be used for any service offering HTTP-endpoints with an OpenAPI or GraphQL schema. Schemathesis supports white-box as well as black-box testing. Even though these approaches are capable of finding many bugs in real world software-systems, it is not a formal verification and therefore cannot guarantee the absence of bugs.

Formal Verification. First strides have already been made in the field of formal methods to automatically verify both software systems and microservices. "Design-time to Run-time Verification of Microservices Based Applications" [28] introduces a tool to automatically create a formal specification from a service definition. The

Table 2.1: Incomplete list of tools for automated API testing or test generation for OpenAPI [13]

Tool	Description
Schemathesis	Automate your API Testing: catch crashes, validate specs, and save time
EvoMaster	The first open-source AI-driven tool for automatically generating system-level test cases (also known as fuzzing) for web/enterprise applications. Currently targeting whitebox and blackbox testing of Web APIs, like REST, GraphQL and RPC (e.g., gRPC and Thrift).
RESTest	RESTest: Automated Black-Box Testing of RESTful Web APIs
Step CI	Automated API Testing and Quality Assurance
Tcases for OpenAPI	A model-based test case generator

tool is able to generate these specifications for services that are built on top of *CONDUCTOR* [11], a Netflix Inc. microservice framework. The tool generates a formal specification in form of Time Basic Petri nets [37] and implements the verification with a Java program built on top of *MAHARAJA* [29].
Matteo Camili, Andrea Janes and Barbara Russo [30] describe the approach of using real-world samples to verify a software system. By capturing samples of the running production system, a formal specification of the real user's behavior is created. This specification then is used to verify the software by using probabilistic model checking.
Amazon Web Services (AWS) also published a paper [33] about verifying that the initial boot code in their data centers is memory safe. They implemented an automatic solution using CBMC [32], the same tool that is called inside JBMC [34], thus the same tool we want to use in this work.

Pact.io Tools. Currently, no such tool exists for Pact.io that provides code-generation, automatic test generation or formal verification.

3 Background

3.1 Micro-Services

There is no single definition of what exactly a microservice is. In the article "Microservices - a definition of this new architectural term" [42] Martin Fowler and James Lewis describe the architectural style of microservices as individually running and deployable services that work together to form a larger application. The services communicate over the network, can be implemented in various programming languages and may integrate different data storing technologies or other dependencies. Most definitions in the literature [36, 42, 57] have in common, that microservices are communicating over the network, mostly by using Representational State Transfer (REST) over HTTP. This is also the most important part of microservice architecture that we'll focus on in this thesis.

3.2 Spring

Spring [18] is a modular Java framework that offers a wide range of features and modules, such as Dependency Injection, Data Access Support and Web Applications. Spring Boot [19], one of many modules in the Spring framework, is a popular choice for building microservices and RESTful Web-Services. According to the 2023 Developer Ecosystem Survey by JetBrains [9], Spring is (by far) the most popular Java web framework and is also used for the development of microservices.

In a Spring RESTful Web Service [20], classes that handle HTTP requests are annotated with `@RestController` and are thus automatically recognised by Spring. Every method that exposes an endpoint is annotated with `@GetMapping`, `@PostMapping`, `@PutMapping` or `@DeleteMapping`, depending on the HTTP method the endpoint should handle. This way, Spring offers a convention over configuration approach for the development of Restful Services and Microservices.

Listing 1 shows an example for such a Spring endpoint definition. The `@GetMapping` annotation tells Spring that the `getAverageTemperatureByYear` method should be called when a GET request is made to the endpoint `/temperature/{year}` where `{year}` is a placeholder for a specific year.

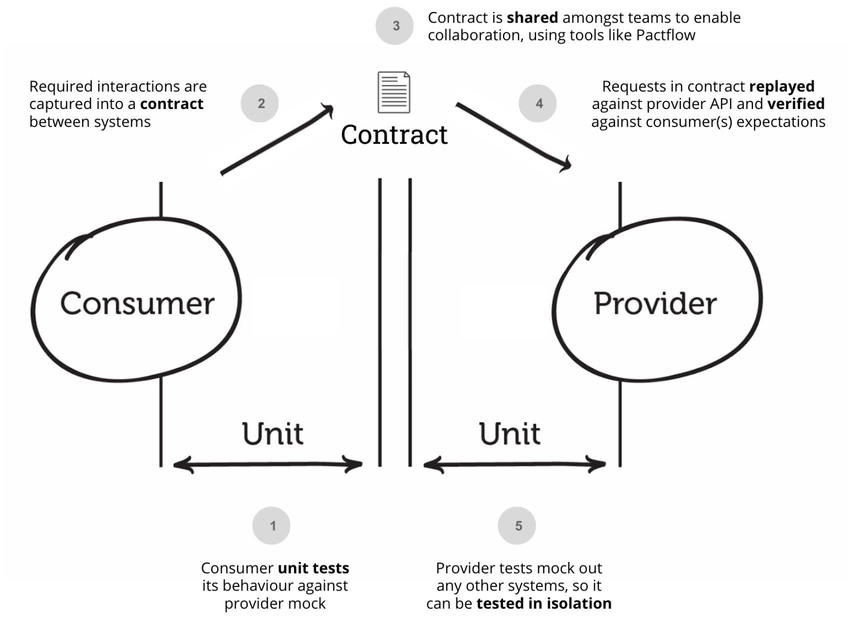
3
Contract is **shared** amongst teams to enable collaboration, using tools like Pactflow
Required interactions are captured into a **contract** between systems
2
Contract
4
Requests in contract **replayed** against provider API and **verified** against consumer(s) expectations
Consumer
Provider
Unit
Unit
1
5
Consumer **unit tests** its behaviour against provider mock
Provider tests mock out any other systems, so it can be **tested in isolation**

Figure 3.1: How Pact.io works [14]

3.3 Contract-based Testing and Pact.io

The microservice architecture, however, displays a big problem when it comes to verifying the correct communication between two services. One approach to tackle this problem is using contract-based testing [39]. The design by contract involves two parties: consumer or requester and the provider. The consumer is the party that initiates the communication and the provider is the party that responds to the consumer's request. Both parties share a contract that defines the behaviour of the client and the server for different requests and responses. Basically, the contract defines that when the client sends a specific request, the server will answer with a specific response. This is reached by using assertions that, for example define, that a specific integer in the request body always has a value greater than zero. This in return leads to the provider answering with a successful response.

One tool that provides a framework for contract-based testing is Pact.io [14]. Pact.io is a tool to establish contract testing for the communication of an application. It tests that sent and received messages conform to the documented behaviour in the contract while using mocks to test the application in isolation.

Figure 3.1 shows the overall idea of contract-based testing with Pact.io. The consumer side provides a contract with the requirements between two systems and the provider side verifies that the contract is fulfilled. When running unit-tests for the consumer, the provider is mocked with the methods returning the expected responses as defined in the Pact.io Specification. The provider on the other side uses the contract to verify its implementation based on the specification file. This also happens in isolation by mocking other systems, like a database or other services that the provider depends on.

Pact.io Specification. The contract between consumer and provider is defined in the Pact.io Specification. The specification is defined in JSON and contains basic information about consumer and provider as well as some metadata. Most importantly, the specification contains a list of interactions between the two parties which define the request and the expected response. One important concept of the specification is the different matching techniques [15] the tool offers. The rules can be defined for body, header, path, query and metadata fields.

The body matching rules are stored in a key/value map where the key is the path to the field in the json body and the value is the "MatchingRules" object. The "MatchingRules" object contains a list of matchers as well as the `combine` field that defines whether the rules should be combined with a logical *AND* or *OR*.

Listing 4 shows an example of a Pact.io Specification. The specification is in JSON format and contains the attributes `consumer`, `provider`, `metadata` and `interactions`. While `provider` and `consumer` provide general information about the consumer or provider-side of the contract, the `metadata` field contains information about the Pact.io Specification version and the tool used to create the specification. The `interactions` field contains a list of interactions to be tested and that exist between provider and consumer. In this example, the only interaction in the specification is a GET request to the path `/product/10?example=test` (line 25–27) and is sent with an Authorization header (line 16–24) that matches a specific regular expression. The expected response is a JSON object with the fields `id`, `name`, `type` and `imageUrls` (line 30–39). The matching rules for the response body are defined in the `matchingRules` field (line 41–60) and one can already see the matching rules `Type` (line 49) to match the type of the field and `MinType` (line 45) to define a minimum number for a number field. A full list of matching rules can be found in Table 4.1.

3.4 Formal Verification

As described in Sect. 3.3, Pact.io uses unit-tests to verify the correct communication between two services. Another approach to evaluate the correctness of a system is by using formal verification. The following section will give a brief overview of the theory behind formal verification and will present specific tools for the JVM environment that can be used to verify Java programs.

3.4.1 Theory

Per Bjesse defines formal (software) verification [27] as "the use of mathematical techniques to ensure that a design conforms to some precisely expressed notion of functional correctness." The formal verifier would then check if there is any input to the program that would violate the specified properties of the program. Formal verification can prove that the program always holds the specified properties for all possible inputs. One approach in formal methods is Model Checking [31]. The basic idea behind model checking is to define a model K as finite-state-transition,

```
{
  "consumer": {"name": "FrontendApplication" },
  "provider": {"name": "ProductService"},
  "metadata": {
    "pact−jvm": {"version": "4.3.10" },
    "pactSpecification": {"version": "4.0" }
  },
  "interactions": [
    {
      "comments": {"text": []},
      "description": "get product with ID 10",
      "key": "a501d946",
      "pending": false,
      "providerStates": [{"name": "product with ID 10exists"}],
      "request": {
        "headers": {"Authorization": ["Bearer 1917−09−30T13:53"]},
        "matchingRules": {
          "header": {
            "Authorization": {
              "combine": "AND",
              "matchers": [{"match": "regex", "regex": "Bearer Token (shortened)"}]
            }
          }
        },
        "method": "GET",
        "path": "/product/10",
        "query": {"example": ["test"] }
      },
      "response": {
        "body": {
          "content": {
            "id": "10",
            "name": "28 Degrees",
            "type": "CREDIT_CARD",
            "imageUrls": [{"url": "url"}]
          },
          "contentType": "application/json; charset=utf−8",
          "encoded": false
        },
        "headers": {"Content−Type": ["application/json; charset=utf−8" ]},
        "matchingRules": {
          "body": {
            "$.id": {
              "combine": "AND",
              "matchers": [{"match": "type", "min": 1}]
            },
            "$.name": {
              "combine": "AND",
              "matchers": [{"match": "type"}]
            },
            "$.type": {
              "combine": "AND",
              "matchers": [{"match": "type"}]
            },
            "$.imageUrls[∗].∗": {
              "combine": "AND",
              "matchers": [{"match": "type"}]
            }
          }
        },
        "generators": {
          "body": {
            "$.id": {"type": "RandomInt"}
          }
        },
        "status": 200
      },
      "type": "Synchronous/HTTP"
    }
  ]
}
```

Listing 4: Example Pact.io Specification

describing the software and a specification ϕ containing the correct properties of the software. An algorithm, called model checker, then decides whether

$$K \models \phi$$

and the structure K satisfies the specification formula ϕ. When the property does not hold, the model checker returns a counterexample that shows the violation of the property.

3.4.2 Tools

A small selection of tools that are available for formal verification with Java are JBMC and CoVeriTeam:

JBMC. JBMC [34] is a verification tool using Bounded Model Checking (BMC) [26] to verify Java Bytecode. The tool is an extension to the C Bounded Model Checker (CBMC) [32], a tool to verify C programs. It accepts `.class` and `.jar` files as input.

CoVeriTeam. Other than JBMC, CoVeriTeam [25] is not a verification tool itself. It rather provides "Verification as a Service" and allows the remote-execution of verification tasks. The program to verify and additional configuration options, e.g. the verifier to use or the computation resources, are sent to the CoVeriTeam server with an HTTP-Request. The server executes the verification and returns a `.zip` archive containing the results of the verification. This allows a developer to use a variety of verification tools with just one integration and without having to install the tools on their local machine.

3.4.3 SV-COMP

The Software Verification Competition (SV-COMP) [22] was established in 2012 to compare the performance of different verification tools for the C programming language. Since 2019, SV-COMP [23] also considers verification tools for Java. For the verification of Java programs, SV-COMP defines convention methods in a Java class named `org.sosy_lab.sv_benchmarks.Verifier`. The class contains methods to create non-deterministic values for the types `Boolean, Byte, Char, Short, Int, Long, Float, Double` and `String`. The methods have the name `nondet`*<Type>* and return a non-deterministic value of the specified type. The class also contains the method `assume` that can be used to assume a specific condition.
The Java participants in the latest SV-COMP competition [24] of 2023 were Coastal [55], GDart [48], Java Ranger [40, 54], JayHorn [41, 53], JBMC [34, 35], JDart [43, 47], MLB and SPF [49, 51]. The results of 2023 and more general information about SV-COMP can be found in "Competition on Software Verification and Witness Validation. SV-COMP 2023" [24].

4 Contribution

We contribute the CLI tool PACT-VERIFIER to automatically create verification harnesses for Spring endpoint methods based on a Pact.io Specification and the provider's source code to verify the method's actual implementation using SV-COMP verifiers.

4.1 Workflow and Approach

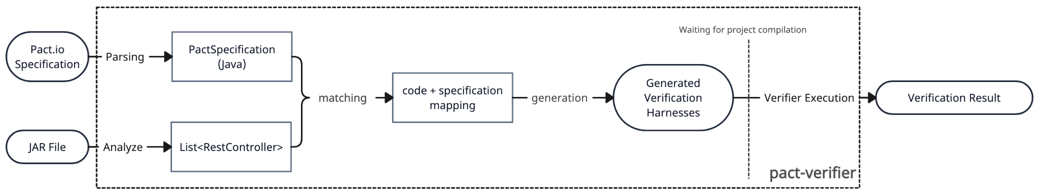

Figure 4.1: Detailed workflow of our approach

Figure 4.1 shows a detailed workflow of PACT-VERIFIER. The CLI tool takes a Pact.io Specification and a provider JAR file as input, then finds endpoints that are present in both the Pact.io Specification and the provider implementation. For every matching endpoint, PACT-VERIFIER generates a verification harness and writes it to the file system. After that, the user running the tool needs to compile the project again if a verifier is used that requires Java ByteCode as input. PACT-VERIFIER then executes a verifier on the generated verification harnesses and returns the verification results. For static configurations, PACT-VERIFIER expects a JSON configuration file in the same directory as the tool is executed. Listing 5 shows an example of such a configuration.

4.2 Parsing Pact.io Specification

The first concrete step of PACT-VERIFIER is to parse the Pact.io Specification in JSON format. We're able to parse Pact.io Specifications up to version 4 and provide backwards compatibility for older versions of the Pact.io Specification. Even though version 4 is generally supported, currently only matching rules for version 3 and below can be parsed, which causes an error when trying to parse a specification that uses matching rules from version 4. However, it would be no great effort to add

```
{
  "paths": {
    "testSourceCode": "src/test/java",
    "mainCompiledCode": "build/classes/java/main",
    "testCompiledCode": "build/classes/java/test"
  },
  "mockOverrides": {
    "com.example.BusinessLogic": "com.example.MockBusinessLogic"
  },
  "coveriTeam": {
    "serverUrl": "https://coveriteam-service.sosy-lab.org/execute",
    "verifierActorFile": "jbmc.yml"
  }
}
```

Listing 5: Example configuration for the tool

these matching rules without a specific implementation to allow such specifications to be parsed.

The tool parses the specification json, like the example in Listing 4, into a Java object that contains all fields of the specification that are required in later steps of the tool: `consumer` (line 2), `provider` (line 3) and `interactions` (line 8–70). The `metadata` field (line 4–7) of the specification is ignored, even though the information to be used to check that the version of the specification is supported. For now, we just assume that a decoding error will prevent the use of an unsupported specification version.

For every interaction, only the `request` (line 15–28), `response` (line 29–67) and `description` (line 11) fields are parsed. Besides from the `providerStates` field (line 14), that could be useful to provide mocks for the expected provider state, the other fields do not provide any helpful information for the tool.

Since the focus lies on the verification of the body, the `headers` (line 16,40) and `query` (line 27) fields of requests and responses are ignored. The `generators` (line 61–65) field is also ignored since the matching rules are used to create non-deterministic values.

4.3 Analyzing Provider JAR File

To gain information about the actual implementation of the endpoints in the provider, PACT-VERIFIER then analyzes the JAR file of the provider. This has the advantage that the tool can parse files written in different Java versions and can support other JVM languages like Kotlin or Scala. On the other side, the project to be analyzed must first be compiled so the JAR file becomes available. It is also worth mentioning that even though PACT-VERIFIER currently only supports the analysis of the project's JAR file, it would be straightforward to implement the analysis of the `.class` files of the project.

When analyzing the JAR file, all classes that are annotated with `@RestController` are considered as controller classes that provide one or multiple endpoints. Then,

all methods annotated with one of the annotations `GetMapping`, `PostMapping`, `PutMapping` or `DeleteMapping` are filtered. This already provides information about the HTTP method of this endpoint. From the annotation, we can also extract the path of the endpoint, if the field is required and if the method parameter is used as path variable, request parameter or request body. Based on this, all classes that are used as method parameters or return values in the endpoint method get recursively analyzed.
The current implementation when analyzing the JAR file does not support endpoints that use a `java.util.Map` or any other Map data structure. Those endpoints are currently ignored and not considered for the verification process. Though, it would be possible to extend the tool to support these data structures as well.

4.4 Identify and match Endpoints

After the steps in Sect. 4.2 and Sect. 4.3, we have separate information about the endpoints exposed from the provider's implementation and the endpoints defined in the Pact.io Specification. As a next step, PACT-VERIFIER tries to match these endpoints and to summarize all information at one point.
The Spring endpoint path can contain variables, as seen with `@GetMapping("temperature/{year}")` in Listing 1 (line 4). On the other hand, the Pact.io Specification is example-based and thus does not contain any generic variable-definition in the path. To find matches between both paths, we use a regular expression that matches any variables in the Spring path and replaces them with a regular expression that matches any value. Furthermore, we also check that the HTTP method of the implementation matches with the one defined in the specification.
After that, the tool recursively summarizes information from the Pact.io Specification for every endpoint method parameter and for the response object. For the request parameters, we need to distinguish between path variables, request parameters and request body and also whether the parameter is required or not. For path variables, all variables are extracted from the path to allow matching these with the matching rules defined in the Pact.io Specification. For request body parameters, we recursively iterate through all fields of the method's return type and gather information about the implementation class and the information from the Pact.io Specification. Specifically, it checks whether the type of the Java class matches the type in the Pact.io Specification. We also derive the matching rules from the Pact.io Specification and summarize this with the information from the class in the Java implementation.

4.5 Generate Verification Harnesses

For every Spring controller class that contains an endpoint that is defined in the Pact.io Specification, PACT-VERIFIER generates a "VerificationContainer" class with a method, which contains the verification harness, for every interaction. The verification container class is always written to the file system under the test directory path defined in the configuration file. In the example configuration introduced in Listing 5, the files would be written to `src/test/java` (line 3). Every "VerificationContainer" contains the method `createConstructor()` that creates a new instance of the controller class and recursively initiates the constructors for every argument in the constructor. If the argument is a primitive type, the value is initialized as a non-deterministic value.

For every interaction that matches with an endpoint in the controller class, the tool generates a method that contains the verification harness for this specific interaction. The method in the verification container has the exact same name as the method in the controller class. If there are multiple interactions that match with the same endpoint, the method is named with an incremental number at the end after the second interaction. A more advanced naming scheme could be enforced, like using the description of the interaction. This, however, would require a more complex implementation and would not provide any additional value for now.

The verification harness always starts with the initialization of the controller class which is done by simply calling the `createConstructor()` method. After that, all method parameters used in the method to verify are initialized. Since the classes used as method parameters can be rather complex, the tool recursively initializes all fields of the classes and thus supports complex classes, lists as well as primitive types. Currently, Map's as well as Set's are not supported, but could be added without much effort.

The next generated code is the line where the method to verify is actually called and the result is stored in a variable. The name of the variable is always `result` and is used in the following assertions.

After the method call, PACT-VERIFIER generates assertions based on the Pact.io matching rules. To allow recursive assertions, the tool then also generates variables for every field of the result object if the result is not a primitive type. Basically, every assertion is of the form `assert(object.getValue() <matches value>)` where `object` is some variable, `getValue()` is the getter method for a value and `<matches value>` defines some condition.

Table 4.1 contains a list of all supported matching rules and the generated assertions. In order to perform the verification with non-deterministic-values and not like Pact.io with example-based values, we do not explicitly support the matching rule *Equality* and instead use the *Type* matcher to check if the result is of the expected type. Also, the *Type* matcher is only generated for non-primitive types, since this is already checked at runtime as explained in Sect. 4.4 which allows detecting type mismatches even at runtime. For *Timestamp*, *Time* and *Date* matching rules, currently only the *Type* matcher is used as these matching rules do not require any additional assertions.

An example of a generated verification harness is shown in Listing 3. Listing 1 and Listing 4 were used as input for PACT-VERIFIER to generate the verification harness. In line 13, the only method parameter (`year`) is initialized with a non-deterministic Integer. If another matching rule - like *MinType* with a minimum of zero - were used, the tool would instead generate `Verifier.nondetInt(year -> year > 0)`. In line 17, one can see the generated assertion for the matcher *MinMaxType* for the getter method `getTemperature()` of the result object.

Table 4.1: Matching Rules and the corresponding assertions generated from PACT-VERIFIER

Matching-Rule	Example	Generated Assertion
Equality	{"match": "equality" }	See Type-Matcher Test
Type	{"match": "type" }	`assert(var instanceof <Type>)`
Integer	{"match": "integer" }	`assert(var instanceof Integer)`
Decimal	{"match": "decimal" }	`assert(var instanceof Decimal)`
Number	{"match": "number" }	`assert(var instanceof Number)`
Null	{"match": "null" }	`assert(var == null)`
Boolean	{"match": "boolean" }	`assert(var instanceof Boolean)`
RegEx	{ "match": "regex", "regex": "\\d+" }	`assert(var.matches("regex"))`
MinType	{ "match": "type", "min": 2 }	`assert(var >= 2)`
MaxType	{ "match": "type", "max": 10 }	`assert(var <= 10)`
MinMaxType	{ "match": "type", "max": 10, "min": 2 }	`assert(var >= 2 && var <= 10)`
Include	{ "match": "include", "value": "substr" }	`assert(var.contains("substr"))`
Timestamp	{ "match": "datetime", "format": "yyyy–MM–dd HH:ss:mm" }	See Type-Matcher
Time	{ "match": "time", "format": "HH:ss:mm" }	See Type-Matcher
Date	{ "match": "date", "format": "yyyy–MM–dd" }	See Type-Matcher

Besides from the verification harnesses for every Pact.io interaction, the tool also generates a `Verifier.java` file that provides the methods to generate non-deterministic values. The `Verifier.java` file contains all conventional methods provided by SV-COMP as mentioned in Sect. 3.4.3. In addition, we introduced the methods `nondetListOf` and `nondetEnum` to be able to generate non-deterministic lists and enums. `nondetListOf` takes a Supplier function for a generic type `T` and returns a list of type `List<T>`. The `nondetEnum` method takes a class of an enum type and returns a non-deterministic value of this enum type. For each non-deterministic method from the SV-COMP conventions, we provide an addi-

```
jbmc au/com/dius/pactworkshop/provider/
    TemperatureControllerVerificationContainer.
    getAverageTemperatureByYear --cp build/classes/java/test:build/classes
    /java/main
```

Listing 6: Example of how JBMC gets executed by PACT-VERIFIER

```
CHECK( init(com.example.TestControllerVerificationContainer.testMethod()), LTL(G
    assert) )
```

Listing 7: Example assertion property-file for CoVeriTeam

tional method that can make assumptions or restrictions on the values of the non-deterministic value. The method signature for a type `T` is `nondetT(Function<T, Boolean> assumption)` where `T` is one of `Boolean, Byte, Char, Short, Int, Long, Float, Double` or `String`. The `assumption` function is a lambda function that takes the non-deterministic value and returns whether the value is valid or not. This allows easily making assumptions on the non-deterministic values for request parameters based on the matching rules in the generated verification harness.

4.6 Formal Verifier Execution

As the last step, PACT-VERIFIER then executes a verifier on the generated verification harnesses. The verifier is executed for every interaction from the Pact.io Specification that matches with an endpoint in the provider implementation and thus for every generated verification method inside a "VerificationContainer"class. For all verifiers that use Java ByteCode as input, the user first needs to compile the generated test classes before the verifier can be run. When running the tool, the user is asked in this case to compile the project again and needs to confirm the successful compilation. Currently, PACT-VERIFIER supports the verifier JBMC and CoveriTeam as verification service:

JBMC. JBMC is the default verifier used by PACT-VERIFIER. We assume that the executing user has the *jbmc* binary installed locally on their system and that the executable is available in the system's PATH. For every method test, JBMC is called with the path to the generated verification method and the class paths for the main and test classes. These class paths are defined in the configuration file as Listing 5 shows. In this case, `build/classes/java/main` and `build/classes/java/test` are used as class paths and are passed as arguments to JBMC. No further arguments are provided to JBMC. Listing 6 shows an example of an executed process by the tool using JBMC as verifier.

CoVeriTeam. Another option PACT-VERIFIER provides to execute the verification is by using CoVeriTeam as verification backend and thus allows the use of any verifiers supported by CoVeriTeam. When running the tool with CoVeriTeam as verification backend, the default verifier executed is JBMC. Though, the executing the user is able

```
 1 {
 2   "cvt_program": "verifier-Java.cvt",
 3   "working_directory": "",
 4   "zipped_directories": [
 5     "main",
 6     "test"
 7   ],
 8   "coveriteam_inputs": {
 9     "verifier_path": "verifier.yml",
10     "program_path": [
11       "main",
12       "test"
13     ],
14     "specification_path": "assert_java.prp"
15   }
16 }
```

Listing 8: Example arguments in a CoVeriTeam request

to define another actor file to use another verifier running the verification process on CoVeriTeam server. Listing 5 provides an example of how a custom actor file for CoVeriTeam can be defined. The configuration example also shows that a custom url for the CoVeriTeam server is used. Both configuration options are optional and the tool uses the default CoVeriTeam server and the default JBMC actor file if no custom configuration is provided. Along with the actor definition file, we also send the verifier definition file `verifier.cvt` containing the definition, inputs and the execution configuration for Java verifiers. We use the default `verifier.cvt`, also used in the CoVeriTeam examples, that defines a verifier from the external actor-definition and executes the verification with the program and specification as input. Every request also contains the compiled main and test classes of the program to verify, uploaded as zipped directories. Since PACT-VERIFIER can create multiple methods that need to be verified, we send a request for every method to be verified. Every file mentioned so far is sent along with every request and is not dependent on the method to verify. The `assert_java.prp` property file is generated for every method to verify and contains the path to the verification method. An example of the `assert_java.prp` file is shown in Listing 7. Besides these files, the request also contains a JSON string defining all arguments and inputs used in the request as shown in Listing 8. Specifically, it defines the used cvt program (line 2), the working-directory (line 3), the zipped directories (line 4–7) and the inputs for the verifier (line 8–15). The inputs for the verifier contain the path to the actor definition file (line 9), the path to the specification property file (line 14) and a list of program paths (line 10–13).

Even though PACT-VERIFIER theoretically supports any verifier supported by CoVeriTeam, manual effort may be required to make a specific verifier work. The main problem here is that most executables of a verifier's default configuration in CoVeriTeam contain a compile step using the default Java compiler `javac`. Since most Microservices projects are built with Maven or Gradle and depend on other external libraries, this step could fail.

4.7 Technical Details

Used Libraries. To implement the tool, we used the following libraries: PACT-VERIFIER uses Javassist to analyze the bytecode of the provider JAR file. To parse the Pact.io Specification, we use Jackson. Verification Harnesses are then generated using JavaPoet. The CLI command was implemented using Picocli. Also, the tool uses Spring as dependency to be able to analyze code that uses imports from the Spring framework.

Lazy Evaluation of Class Signatures. Since class definitions in Microservices can be rather complex, we need to recursively evaluate all fields of a class to get all information about the class and its fields. Since cyclic dependencies between classes are possible, we need to evaluate the class signatures lazily to avoid StackOverflow or OutOfMemory errors. In the current implementation, this is solved by just storing the signature of classes used in an endpoint. The class is then only evaluated when the information is needed and not when the class is first encountered. The evaluated classes are then stored in a Map with its signature as key and the class as value and are thus only evaluated once, whilst a good performance is still guaranteed.

CoVeriTeam JBMC Integration. Even though CoVeriTeam provides predefined actor files to use JBMC as verifier, we needed to customize this implementation to allow the use of JBMC as verifier. The default jbmc actor file calls an executable that contains a lot of transformations and handling of features for SV-COMP. The biggest problem in the default executable was the compile step trying to compile the project using the default java compiler `javac`. Since most Microservices projects are built with Maven [1] or Gradle [4] and depend on other external libraries, the default compile step failed. Since CoVeriTeam does not support Maven or Gradle, we created a custom toolinfo module for jbmc that only executes the jbmc-binary and doesn't try to compile the project. The specific toolinfo module for JBMC used by PACT-VERIFIER in the default configuration can be found online under `https://gitlab.com/-/snippets/3643052/raw/main/jbmc-binary.py`.

5 Evaluation

5.1 Experimental Setup

To answer the research questions, we conducted a series of experiments to evaluate the correctness and performance of the tool. The benchmarks were executed in a virtual machine running Ubuntu 22.04 on a Windows 10 host with a 4-core Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GHz and 16GB of memory. The virtual machine was configured with four cores and a maximum of 8GB of memory. The verification harnesses were generated with Ubuntu 22.04 as well as on macOS Ventura version 13.6. In addition, we used the following tools:

- **BenchExec**: Version 3.12 [56]
- **Java**: Version 21.0.2 LTS (GraalVM)
- **JBMC**: Version 5.72.1-22-g1ab48b9654 [1]
- **CoVeriTeam**: We used the currently running version of the CoVeriTeam service as of 2024-03-26.

5.2 Verification Harness Generation

This section evaluates the correctness of the generated verification harnesses and whether the generated code can be verified by SV-COMP verifiers. Listing 1 introduced an example endpoint that returns the average temperature for a given year. Listing 2 shows the corresponding Pact.io Specification for this endpoint. From these inputs, our approach generates the output shown in Listing 3. After running the verification process with the tool using JBMC as verifier, the result shows that all properties are satisfied and the tool returns a proof as verification verdict. If one changes the implementation of the endpoint to return a value outside the expected range, JBMC detects that this property is violated. The verification verdict of the tool for this endpoint then is an `ERROR`. PACT-VERIFIER can also successfully verify the generated code by using CoVeriTeam as verification backend.

Thus, we could show that the tool is able to generate a verification harness for a simple endpoint and that the generated code can be verified by SV-COMP verifiers. This also includes that PACT-VERIFIER can infer the correct Java classes and methods of an endpoint from the Pact.io Specification and generate the corresponding verification harness.

[1]https://gitlab.com/sosy-lab/sv-comp/archives-2023/-/blob/main/2023/jbmc.zip

```
private static CourtCaseController createConstructor() {
  CourtCaseService courtCaseService = null; // TODO: Replace with a mock or
      define the mock in the verification config.
  GroupedOffenderMatchRepository groupedOffenderMatchRepository = null; //
      TODO: Replace with a mock or define the mock in the verification config.
  WebClientFactory webClientFactory = new WebClientFactory();
  ClientDetails clientDetails = new ClientDetails();
  WebClient client = null; // TODO: Replace with a mock or define the mock in the
      verification config.
  String oauthClient = Verifier.nondetString();
  Boolean disableAuthentication = Verifier.nondetBoolean();
  RestClientHelper restClientHelper = new RestClientHelper(client,oauthClient,
      disableAuthentication);
  OffenderRestClientFactory offenderRestClientFactory = new
      OffenderRestClientFactory(webClientFactory,clientDetails,restClientHelper);
  CourtCaseRepository courtCaseRepository = null; // TODO: Replace with a mock
       or define the mock in the verification config.
  HearingRepository hearingRepository = null; // TODO: Replace with a mock or
      define the mock in the verification config.
  OffenderMatchService offenderMatchService = new OffenderMatchService(
      groupedOffenderMatchRepository,offenderRestClientFactory,
      courtCaseRepository,hearingRepository);
  DefendantRepository defendantRepository = null; // TODO: Replace with a mock
       or define the mock in the verification config.
  OffenderRepository offenderRepository = null; // TODO: Replace with a mock or
      define the mock in the verification config.
  OffenderRepositoryFacade offenderRepositoryFacade = new
      OffenderRepositoryFacade(offenderRepository);
  OffenderUpdateService offenderUpdateService = new OffenderUpdateService(
      defendantRepository,offenderRepository,offenderRepositoryFacade);
  CaseCommentsRepository caseCommentsRepository = null; // TODO: Replace
      with a mock or define the mock in the verification config.
  TelemetryClient telemetryClient = new TelemetryClient();
  TelemetryService telemetryService = new TelemetryService(telemetryClient,
      clientDetails);
  CaseCommentsService caseCommentsService = new CaseCommentsService(
      courtCaseRepository,caseCommentsRepository,telemetryService);
  AuthenticationHelper authenticationHelper = new AuthenticationHelper();
  HearingNotesRepository hearingNotesRepository = null; // TODO: Replace with
      a mock or define the mock in the verification config.
  CaseProgressService caseProgressService = new CaseProgressService(
      hearingRepository,hearingNotesRepository);
  HearingNotesService hearingNotesService = new HearingNotesService(
      hearingRepository,hearingNotesRepository,telemetryService);
  boolean enableCacheableCaseList = Verifier.nondetBoolean();
  CourtCaseController courtCaseController = new CourtCaseController(
      courtCaseService,offenderMatchService,offenderUpdateService,
      caseCommentsService,authenticationHelper,caseProgressService,
      hearingNotesService,enableCacheableCaseList);
  return courtCaseController;
}
```

Listing 9: Constructor generation method for the CourtCaseController

```java
static void createOrUpdateHearingByHearingId2() {
  // Verification Method for pact interaction with description "a request to put a minimal court case"

  CourtCaseController courtCaseController = createConstructor();

  String hearingId = Verifier.nondetString();
  ExtendedHearingRequestResponse putHearingRequest = new ExtendedHearingRequestResponse(
    Verifier.nondetString() /* caseNo */,
    ...
    Verifier.nondetListOf(() -> new HearingDay(
      Verifier.nondetString() /* courtCode */,
      Verifier.nondetString() /* courtRoom */,
      LocalDateTime.now() /* sessionStartTime */,
      Verifier.nondetString() /* listNo */
    )) /* hearingDays */,
    Verifier.nondetListOf(() -> new Defendant(
      Verifier.nondetString() /* defendantId */,
      ...
      LocalDate.now() /* dateOfBirth */,
      ...
      Verifier.nondetEnum(DefendantType.class) /* type */,
      ...
      new PhoneNumber(
        Verifier.nondetString() /* home */,
        Verifier.nondetString() /* mobile */,
        Verifier.nondetString() /* work */
      ) /* phoneNumber */,
      new Offender(
        Verifier.nondetString() /* pnc */
      ) /* offender */,
      ...
    )) /* defendants */,
    ...
  );
  Mono<ExtendedHearingRequestResponse> result = courtCaseController.createOrUpdateHearingByHearingId(hearingId, putHearingRequest);

  // No assertions to be made
}
```

Listing 10: Example of a generated request of the court-case-service

```
static void getCourtCase() {
  // Verification Method for pact interaction with description "a request for a case
      by case number"

  CourtCaseController courtCaseController = createConstructor();

  String courtCode = Verifier.nondetString();
  String caseNo = Verifier.nondetString();
  String listNo = Verifier.nondetString();
  CourtCaseResponse result = courtCaseController.getCourtCase(courtCode,
      caseNo, listNo);

  assert(result.getCaseId() instanceof String);
  ...
  assert(result.getSessionStartTime() instanceof LocalDateTime);
  assert(result.getSession() instanceof CourtSession);
  assert(result.getPreviouslyKnownTerminationDate() instanceof LocalDate);
  assert(result.getSuspendedSentenceOrder() instanceof Boolean);
  assert(result.getBreach() instanceof Boolean);
  assert(result.getPreSentenceActivity() instanceof Boolean);

  List<OffenceResponse> offences = result.getOffences();
  offences.forEach(offenceResponse -> {
    assert(offenceResponse.getOffenceTitle() instanceof String);
    assert(offenceResponse.getOffenceSummary() instanceof String);
    assert(offenceResponse.getAct() instanceof String);
    assert(offenceResponse.getListNo() instanceof Integer);
    assert(offenceResponse.getOffenceCode() instanceof String);
    Verdict verdict = offenceResponse.getVerdict();
    VerdictType verdictType = verdict.getVerdictType();
    assert(verdictType.getDescription() instanceof String);
    assert(verdict.getDate() instanceof LocalDate);
  } );
  ...
  List<CaseProgressHearing> hearings = result.getHearings();
  hearings.forEach(caseProgressHearing -> {
    assert(caseProgressHearing.getHearingId() instanceof String);
    ...
    assert(caseProgressHearing.getHearingDateTime() instanceof LocalDateTime);
    List<HearingNoteResponse> notes = caseProgressHearing.getNotes();
    notes.forEach(hearingNoteResponse -> {
      assert(hearingNoteResponse.getNoteId() instanceof Long);
      assert(hearingNoteResponse.getHearingId() instanceof String);
      assert(hearingNoteResponse.getNote() instanceof String);
      assert(hearingNoteResponse.getCreated() instanceof LocalDateTime);
      assert(hearingNoteResponse.getAuthor() instanceof String);
      assert(hearingNoteResponse.getCreatedByUuid() instanceof String);
    } );
    ...
  } );
  ...
  List<CaseDocumentResponse> files = result.getFiles();
  files.forEach(caseDocumentResponse -> {
    assert(caseDocumentResponse.getId() instanceof String);
    assert(caseDocumentResponse.getDatetime() instanceof LocalDateTime);

    CaseDocumentResponse.FileResponse file = caseDocumentResponse.getFile();
  });
  ...
}
```

Listing 11: Example of generated assertions of an endpoint of the court-case-service

5.3 Real World Example

To evaluate whether the tool is able to generate verification harnesses and run the verification process for a real-world example, we used the *court-case-service* as a real-world example. The *court-case-service* is a microservice developed by the British Ministry of Justice and provides endpoints to access court cases. The service is built on top of Spring Boot and uses Java and Kotlin as programming languages. For the *court-case-service* as provider, two different consumers exist, the *court-case-matcher* and the *prepare-a-case*. We'll solely focus on the *court-case-matcher* as consumer for this evaluation since the Pact.io Specification from the *prepare-a-case* consumer does not provide any matching rules and was still written in the specification version 2.

The service provides 50 endpoints, has a total of 31.306 lines of Java code and 4.530 lines of Kotlin code. The *court-case-matcher* Pact.io Specification defines four different interactons. The most used matching rule in the interaction is the *Type* rule with 65 occurrences, followed by the *Date* rule with 7 occurrences and the *Integer* and *Timestamp* rule with 2 occurrences each. The exact Pact.io Specification used can be found online in the Pact Broker of the Ministry of Justice. Unfortunately, the specification does not contain any other interesting matching rules that would have been of interest for the evaluation and that would have allowed more meaningful verification.

The interactions defined in the specification are all based on endpoints located in the `CourtCaseController` and the tool only generates the class `CourtCaseControllerVerificationContainer` containing the generated verification harnesses. While the `CourtCaseController` has a total of 408 lines of code, the generated `CourtCaseControllerVerificationContainer` has a total of 510 lines of code. The generated code contains the verification harnesses for the four interactions defined in the Pact.io Specification and the method to create the constructor of the `CourtCaseController`.

Listing 9 shows the generated method to create the constructor of the `CourtCaseController`. The method recursively initializes all dependencies of the `CourtCaseController` calls the constructor and returns the created instance. In fact, this method creates 25 different objects that are needed to create the `CourtCaseController`. In line 7,8 and 26, one can see that primitive types are initialized with a non-deterministic value. This may not be correct in every use case, as values in the configuration are often Spring properties that are stored with more or less static values in the config and not expected to be non-deterministic values. Next to the primitive types, nine of the objects are initialized with a `null` value since the type is an interface or abstract class and the tool couldn't find a concrete implementation. The user needs to create a mock for these objects and can define the mock in the verification configuration file to automatically replace the `null` value with the mock. Unfortunately, we cannot specify any mocks for this use case, as we are not familiar with the business logic and this could lead to further errors in the verification process. As a result, we are unfortunately not able to make any qualitative statements about the verification result in the *court-case-service*, as null-pointer-exception in particular are found due to missing mocks. Nevertheless, this example shows that

PACT-VERIFIER is able to generate the correct constructor initialization code for a comprehensive and deeply nested class structure.

Listing 10 shows an excerpt the created verification harness for the method `createOrUpdateHearingByHearingId` and the corresponding Pact.io interaction with a `PUT` request to a specific path of the endpoint. The complete method for the verification harness consists of 91 lines of code and has been shortened to 38 lines in this listing. The method starts by calling the `createConstructor` method and storing the created `CourtCaseController` in the variable `courtCaseController` (line 4). The method then initializes the variables `hearingId` and `putHearingRequest` (line 6–34). While `hearingId` is a simple string initialized with a non-deterministic value, the `putHearingRequest` is a more complex object. One can see that it contains lists (line 10–15, 16–32) and the creation of nested objects (line 10–15, line 16–32, line 23–27, line 28–30). The two variables are then used as parameters to call the `createOrUpdateHearingByHearingId` method of the `CourtCaseController` (line 35). The method returns a `Mono<ExtendedHearingRequestResponse>` object, which is a type of the Project Reactor library. Since the Pact.io Specification contains no matching rules and no content body for the response, no assertions are made in this method (line 37).

Listing 11 shows an excerpt of the generated verification harness for the method `getCourtCase` and the corresponding Pact.io interaction with a GET request to the path `/court/B10JQ/case/1600028913`. The complete method containing the verification harness includes 129 lines of code. As already mentioned in Listing 10, the method starts by calling the `createConstructor` method, creating the non-deterministic input parameters for the method `getCourtCase` and calling the method with these parameters (line 4–9). The assertions for the return type of the endpoint method then start in line 11. The method contains a total of 74 assertions while most assertions only check that the given type is a String. Most of these String assertions have been omitted in the example listing. Assertions for other types like `LocalDateTime` (line 13, 43, 53), `LocalDate` (line 15, 30), `Integer` (line 25) and `Long` (line 40). Line 14 also contains an assertion for a type that does not come from the standard Java library, but is part of the *court-case service*. One can also see that assertions for list types are generated successfully (line 20–31, 33–48, 50–56) and even recursively in nested lists (line 38–46). We are therefore able to show that it is also possible to generate a very complex verification harness with a large number of assertions using PACT-VERIFIER. Since the class `CaseDocumentResponse` appearing in line 50 is implemented in Kotlin, we also show that the tool is also able to analyze Kotlin classes and generate the corresponding assertions.

During the qualitative verification, we observed that JBMC found errors for all generated verification harnesses. To be more precise, there are 4 out of 716 errors for method `getCourtCase` and 4 out of 419 errors for method `getHearingByHearingId`. For the method `createOrUpdateHearingByHearingId` that is used in two interactions, JBMC found 4 errors out of 383 for each generated verification harness. As already mentioned above, these results are not really representative, as for many of the classes used, `null` was passed as a value in the constructor. It can therefore be assumed that many of the errors found by JBMC are due to null pointer exceptions. However, these results show, that PACT-VERIFIER is able to generate

Table 5.1: PACT-VERIFIER runtime results for the two example projects

Project	**Execution-Type**	**CPU-Time (s)**
Pact–Workshop–JVM–Spring	Java	2.3
Pact–Workshop–JVM–Spring	Native-Image	0.042
court–case–service	Java	2.4
court–case–service	Native-Image	0.11

verification harnesses for the court-case-service and that the generated code can be verified by JBMC. Therefore, we can assume that the tool can also be used for other real-world projects and that the generated code can be verified by SV-COMP verifiers by using CoVeriTeam as verification backend.

5.4 Negligible (CPU) Time

To evaluate the performance of PACT-VERIFIER and the duration until the verification harness is generated, we measured the CPU time for two example projects using Spring and Pact.io. We explicitly neglected the verifier runtime in this evaluation, as the verifier runtime can vary depending on the verifier used and cannot be influenced by PACT-VERIFIER. The first project is a simple example project that demonstrates the usage of Pact.io with Spring Boot. The provider provides two simple endpoints and has a total of 476 lines of Java code, while the Pact.io Specification contains 6 different interactions. The second example used in the experiment is the real-world-example *court-case-service* introduced in Sect. 5.3. The Pact.io Specification provided from the consumer *court-case-matcher* contains four different interactions.

Results. The results of the benchmarks in Table 5.1 show that PACT-VERIFIER is able to generate the verification harness for the example projects in reasonable time. The results also show that executing PACT-VERIFIER as a native–image is faster than executing PACT-VERIFIER as a Java application. This can be explained by the startup time of the JVM, that in this case takes more time than the actual execution of the CLI. When comparing the results of the two projects, the *court-case-service* takes up more CPU time than the *Pact-Workshop-JVM-Spring* project for both execution types. This can be explained by the larger size of the *court-case-service* project, which has more endpoints and thus more code to analyze. Even though the *court-case-service* has more than 75 times as many lines of code as the *Pact-Workshop-JVM-Spring* project, the CPU time of the two projects is only slightly different. This indicates that the tool is able to analyze even larger projects in a reasonable amount of time. From the results, we can conclude that PACT-VERIFIER is capable of generating verification harnesses for a microservice in a negligible amount of time.

5.5 Threats to Validity

Threats to Internal Validity. One potential threat to the internal validity of the results are possible inaccuracies in the measurements of the benchmarks. Since we used BenchExec to measure the CPU time for the application, it is unlikely that the results are inaccurate, as it provides a reliable benchmarking framework that was also used in other studies and for all instances of SV-COMP. Also, PACT-VERIFIER is that fast, that even doubling the CPU time would still represent a negligible amount of time.
Another potential internal validity threat is that the generated code might not be correct and cannot be verified. The library used to generate the code is JavaPoet [8], which provides a popular framework for Java code-generation and thus should also generate code that can be compiled. Also, we showed that JBMC can verify the generated verification harnesses, which indicates that the generated code for the provided examples is correct. Although this does not necessarily apply to all SV-COMP verifiers, we are able to show that the approach works and that a verifier can verify the automatically generated code.

Threats to External Validity. On the other side, a potential thread to external validity is the usage of external libraries that are not supported by PACT-VERIFIER. This is particularly relevant when an endpoint directly returns a type from an external library. Since we have already introduced support for the `Mono<>` class of the Project Reactor [16] library, supporting other libraries as well should be easily possible by extending the code generation to support the specific return types of the library. Though, a more solid and generic solution would be to allow a developer to inject custom code to the jar analyzing logic to basically support any library without having the tool itself to support it.
Also, the *court-case-service* used as real-world-example for a microservice might not be representative for all microservices. The *court-case-service* has a total of 31.306 lines of Java code and 4.530 lines of Kotlin code. With 22 GET, 8 POST, 8 DELETE and 12 PUT endpoints, the service also has a considerable number of endpoints. Through the correct analysis of this service with a broad variety of different endpoints, one can assume that PACT-VERIFIER is able to analyze even larger and more complex services.

6 Future Work

Even though we could show that we can automatically create verification harnesses that can be verified by SV-COMP verifiers, there is still room for improvements to cover further cases or to improve the developer experience.

Support Pact.io Specification Version 4. As mentioned in Section 4.2, the tool is currently able to parse Pact.io Specifications in version 4. However, we're currently only able to generate assertions for matching rules from version 3 or below. To support the new matching rules from version 4, we need to extend the tool to generate assertions for these new rules. The fourth version of the specification mainly introduced matching rules for arrays and would be a great extension to be able to make more assertions to lists.

Integration into Build-Tools. To offer a better developer experience, the tool could also offer an easy integration into build tools like Maven or Gradle. As stated in Section 4.6, the user currently needs to manually compile the test classes after the tool generated the verification harnesses. It's also required to manually confirm the successful compilation in order to run the verification of the generated verification harnesses. These steps could be automated by developing a plugin for Maven or Gradle that automatically compiles the generated test classes and then runs a verifier on the generated verification harnesses.

Support multiple Specifications and Frameworks. As the tool name PACT-VERIFIER suggests, the tool is currently only able to generate verification harnesses for Pact.io Specifications. However, there are other popular specification frameworks like OpenAPI or GraphQL [5] that could be supported by the tool. Also, the tool currently only supports endpoints that are implemented in a JVM language using the Spring framework. The tool could be extended to support other microservice frameworks like Vert.x [2] or Quarkus [17].
Extending the tool to support multiple specifications and frameworks would not be too complex, as only a parser for the respective specification or analyzer for the framework would have to be implemented. Since we already have a clearly defined internal data structure, the parser or analyzer would only have to map the specification or framework to this internal data structure. This would allow us to provide automatic verification for a wide range of tools and frameworks for JVM languages.

7 Conclusion

This thesis presents an approach to automatically generate verification harnesses and execute the verification based on a microservice's source code and a Pact.io Specification. This allows developers to easily use formal verification in (existing) projects alongside testing techniques. We showed that the approach is practicable by implementing a tool that generates verification harnesses for projects using Java Spring Boot and Pact.io. Our contribution is able to generate the verification harnesses in a negligible amount of time, and the generated code can be verified by SV-COMP verifiers. Therefore, the tool can already be used in practice to utilise formal verification in microservice projects. In particular, Java projects that use Pact.io as contract-based-testing framework can already benefit a lot from the contributed tool.

Disclaimer. This thesis was written with the use of ChatGPT and DeepL to improve and check wordings and grammar. For the development of the contributed tool, GitHub Copilot was used as auto-completion tool.

Bibliography

[1] Apache maven project website. [Online; accessed 23-March-2024].

[2] Eclipse vert.x website. [Online; accessed 16-April-2024].

[3] Finagle: A protocol-agnostic rpc system. [Online; accessed 05-April-2024].

[4] Gradle build tool website. [Online; accessed 23-March-2024].

[5] Graphql github. [Online; accessed 16-April-2024].

[6] grpc-java github. [Online; accessed 07-March-2024].

[7] grpc supported languages. [Online; accessed 07-March-2024].

[8] Javapoet github. [Online; accessed 28-March-2024].

[9] Jetbrains developer ecosystem - java. [Online; accessed 02-March-2024].

[10] Lessons learned from scaling uber to 2000 engineers. [Online; accessed 05-April-2024].

[11] Netflix conductor github. [Online; accessed 12-March-2024].

[12] Openapi generator github. [Online; accessed 19-January-2024].

[13] Openapi tooling. [Online; accessed 07-March-2024].

[14] Pact.io documentation. [Online; accessed 04-March-2024].

[15] Pact.io specification v4. [Online; accessed 04-March-2024].

[16] Project reactor website. [Online; accessed 28-March-2024].

[17] Quarkus website. [Online; accessed 16-April-2024].

[18] Spring. [Online; accessed 08-March-2024].

[19] Spring boot documentation. [Online; accessed 02-March-2024].

[20] Spring rest-service documentation. [Online; accessed 02-March-2024].

[21] A. Arcuri. Restful API automated test case generation with evomaster. *ACM Trans. Softw. Eng. Methodol.*, 28(1):3:1–3:37, 2019.

[22] D. Beyer. Competition on software verification - (SV-COMP). In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 504–524. Springer, 2012.

[23] D. Beyer. Automatic verification of C and java programs: SV-COMP 2019. In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 133–155. Springer, 2019.

[24] D. Beyer. Competition on software verification and witness validation: SV-COMP 2023. In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 495–522. Springer, 2023.

[25] D. Beyer, S. Kanav, and H. Wachowitz. Coveriteam service: Verification as a service. In *45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023*, pages 21–25. IEEE, 2023.

[26] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Adv. Comput.*, 58:117–148, 2003.

[27] P. Bjesse. What is formal verification? *ACM Sigda Newsletter*, 35, 12 2005.

[28] M. Camilli, C. Bellettini, and L. Capra. Design-time to run-time verification of microservices based applications - (short paper). In A. Cerone and M. Roveri, editors, *Software Engineering and Formal Methods - SEFM 2017 Collocated Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Trento, Italy, September 4-5, 2017, Revised Selected Papers*, volume 10729 of *Lecture Notes in Computer Science*, pages 168–173. Springer, 2017.

[29] M. Camilli, A. Gargantini, P. Scandurra, and C. Bellettini. Event-based runtime verification of temporal properties using time basic petri nets. In C. W. Barrett, M. D. Davies, and T. Kahsai, editors, *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, volume 10227 of *Lecture Notes in Computer Science*, pages 115–130, 2017.

[30] M. Camilli, A. Janes, and B. Russo. Automated test-based learning and verification of performance models for microservices systems. *J. Syst. Softw.*, 187:111225, 2022.

[31] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[32] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[33] B. Cook, K. Khazem, D. Kroening, S. Tasiran, M. Tautschnig, and M. R. Tuttle. Model checking boot code from AWS data centers. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 467–486. Springer, 2018.

[34] L. C. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtík. JBMC: A bounded model checking tool for verifying java bytecode. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 183–190. Springer, 2018.

[35] L. C. Cordeiro, D. Kroening, and P. Schrammel. JBMC: bounded model checking for java bytecode - (competition contribution). In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 219–223. Springer, 2019.

[36] B. Familiar. *Microservices, IoT and Azure*. Apress Berkeley, CA, 2015.

[37] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. A unified high-level petri net formalism for time-critical systems. *IEEE Trans. Software Eng.*, 17(2):160–172, 1991.

[38] Z. Hatfield-Dodds and D. Dygalo. Deriving semantics-aware fuzzers from web API schemas. *CoRR*, abs/2112.10328, 2021.

[39] R. Heckel and M. Lohmann. Towards contract-based testing of web services. In M. Pezzè, editor, *Proceedings of the International Workshop on Test and Analysis of Component Based Systems, TACoS 2004, Barcelona, Spain, March 27-28, 2004*, volume 116 of *Electronic Notes in Theoretical Computer Science*, pages 145–156. Elsevier, 2004.

[40] S. Hussein, Q. Yan, S. McCamant, V. Sharma, and M. W. Whalen. Java ranger: Supporting string and array operations in java ranger (competition contribution). In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 553–558. Springer, 2023.

[41] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. Jayhorn: A framework for verifying java programs. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 352–358. Springer, 2016.

[42] F. M. Lewis James. Microservices - a definition of this new architectural term. [Online; accessed 03-March-2024].

[43] K. S. Luckow, M. Dimjasevic, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamaric, and V. Raman. Jdart: A dynamic symbolic analysis framework. In M. Chechik and J. Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 442–459. Springer, 2016.

[44] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés. Restest: automated black-box testing of restful web apis. In C. Cadar and X. Zhang, editors, *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, pages 682–685. ACM, 2021.

[45] T. Mauro. Adopting microservices at netflix: Lessons for architectural design. [Online; accessed 05-April-2024].

[46] C. McCaffrey. The verification of a distributed system. *Commun. ACM*, 59(2):52–55, 2016.

[47] M. Mues and F. Howar. Jdart: Portfolio solving, breadth-first search and smt-lib strings (competition contribution). In J. F. Groote and K. G. Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 448–452. Springer, 2021.

[48] M. Mues and F. Howar. Gdart: An ensemble of tools for dynamic symbolic execution on the java virtual machine (competition contribution). In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*, volume 13244 of *Lecture Notes in Computer Science*, pages 435–439. Springer, 2022.

[49] Y. Noller, C. S. Pasareanu, A. Fromherz, X. D. Le, and W. Visser. Symbolic pathfinder for SV-COMP - (competition contribution). In D. Beyer, M. Huisman, F. Kordon, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pages 239–243. Springer, 2019.

[50] A. Panda, M. Sagiv, and S. Shenker. Verification in the age of microservices. In A. Fedorova, A. Warfield, I. Beschastnikh, and R. Agarwal, editors, *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, pages 30–36. ACM, 2017.

[51] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.

[52] F. Pfeiffer. A scalable and resilient microservice environment with apache mesos and apache aurora. Dublin, 2015. USENIX Association.

[53] A. Shamakhi, H. Hojjat, and P. Rümmer. Towards string support in jayhorn (competition contribution). In J. F. Groote and K. G. Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II*, volume 12652 of *Lecture Notes in Computer Science*, pages 443–447. Springer, 2021.

[54] V. Sharma, S. Hussein, M. W. Whalen, S. McCamant, and W. Visser. Java ranger: statically summarizing regions for efficient symbolic execution of java. In P. Devanbu, M. B. Cohen, and T. Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 123–134. ACM, 2020.

[55] W. Visser and J. Geldenhuys. COASTAL: combining concolic and fuzzing for java (competition contribution). In A. Biere and D. Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, volume 12079 of *Lecture Notes in Computer Science*, pages 373–377. Springer, 2020.

[56] P. Wendler and D. Beyer. sosy-lab/benchexec: Release 3.21.

[57] E. Wolff. *Microservices: Grundlagen flexibler Softwarearchitekturen*. dpunkt. verlag, 2018.