Building your Automated Test Case Generation Tool for REST APIs with RestTestGen

Mariano Ceccato

mariano.ceccato@univr.it

This work has been done in collaboration mainly with **Davide Corradini, Michele Pasqua** and **Sofia Mari**

UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

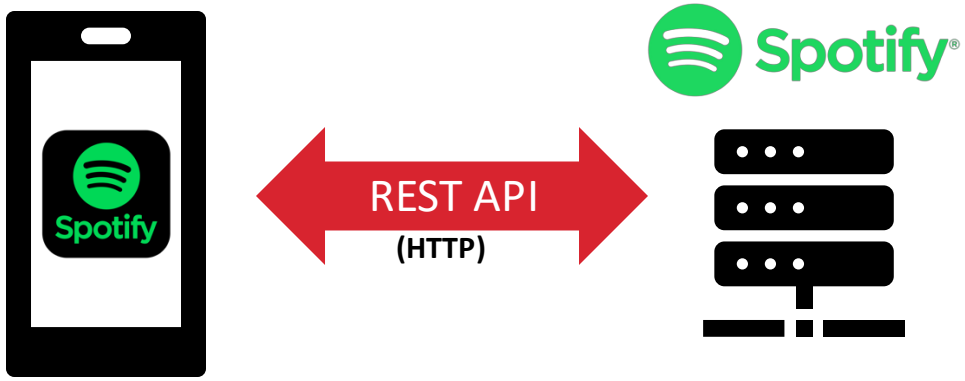
Credits

Davide Corradini

Michele Pasqua

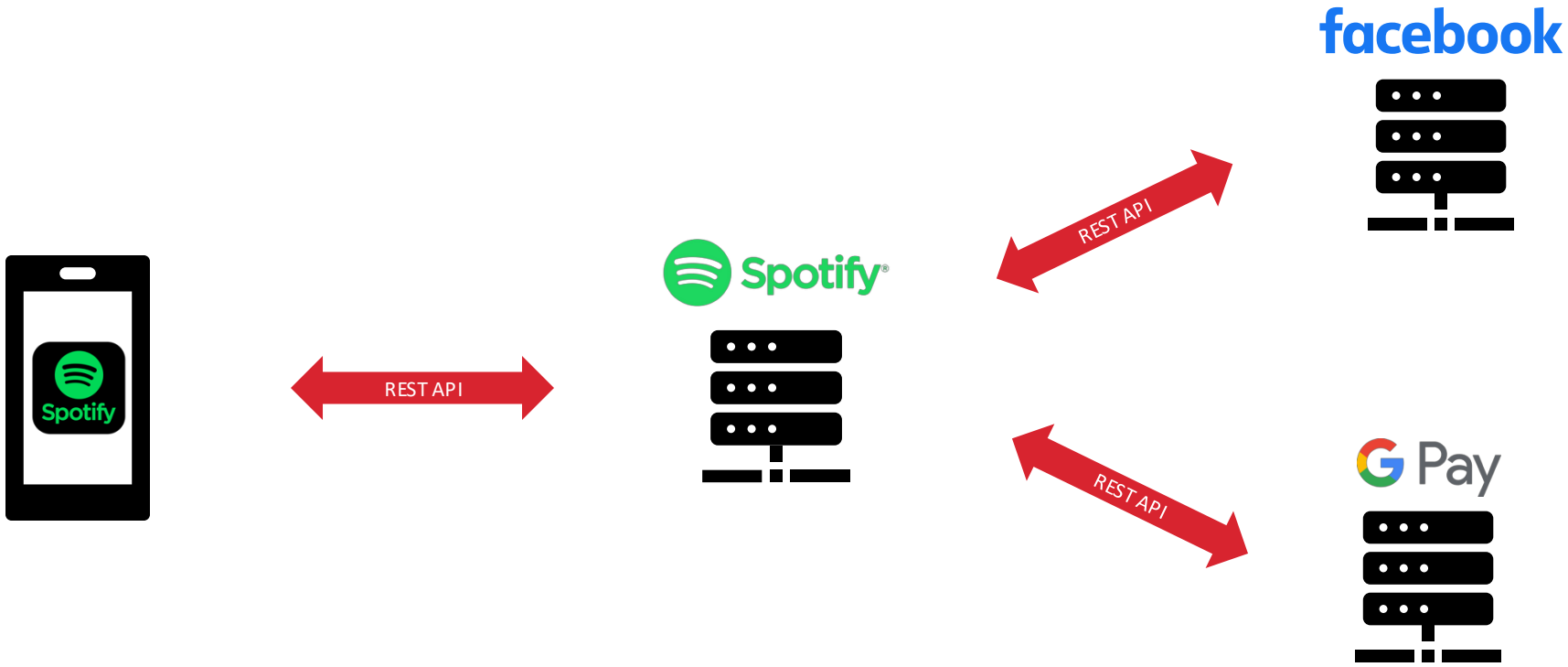
Sofia Mari

What is a Web API or REST API?



Create
Read
Update
Delete

What is a Web API or REST API?



OpenAPI Specification

GET /search Search for Item

Get Spotify catalog information about albums, artists, playlists, tracks, shows, episodes or audiobooks that match a keyword string.
Note: Audiobooks are only available for the US, UK, Ireland, New Zealand and Australia markets.

Parameters | Try it out

Name	Description
q * required string (query)	remaster%20track:Doxy%20artist:Miles%20Davis
type * required array[string] (query)	*Available values* : album, artist, playlist, track, show, episode, audiobook album artist playlist track

Code	Description	Links
200	Search response Media type application/json Controls Accept header. Example Value | Schema	*No links*

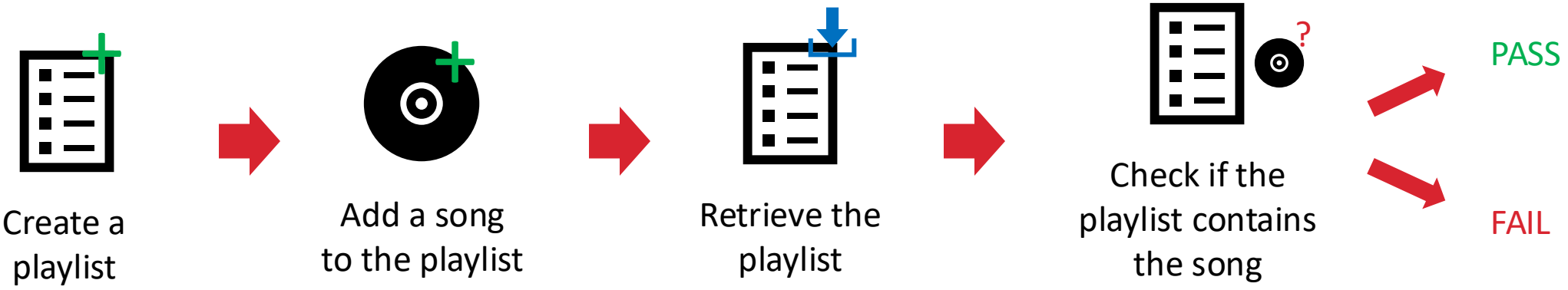

```
{
  "albums": {
    "href": "https://api.spotify.com/v1/me/shows?offset=0&limit=20\n",
    "limit": 20,
    "next": "https://api.spotify.com/v1/me/shows?offset=1&limit=1",
    "offset": 0,
    "previous": "https://api.spotify.com/v1/me/shows?offset=1&limit=
1",
    "total": 4,
    "items": [
      {
        "album_type": "compilation",
        "available_markets": [
          "CA",
          "BR",
          "IT"
        ],
        "external_urls": {
          "spotify": "string"
        },
        "href": "string",
        "id": "2up3OPMp9Tb4dAKM2erWXQ",
        "images": [
```

Problem definition

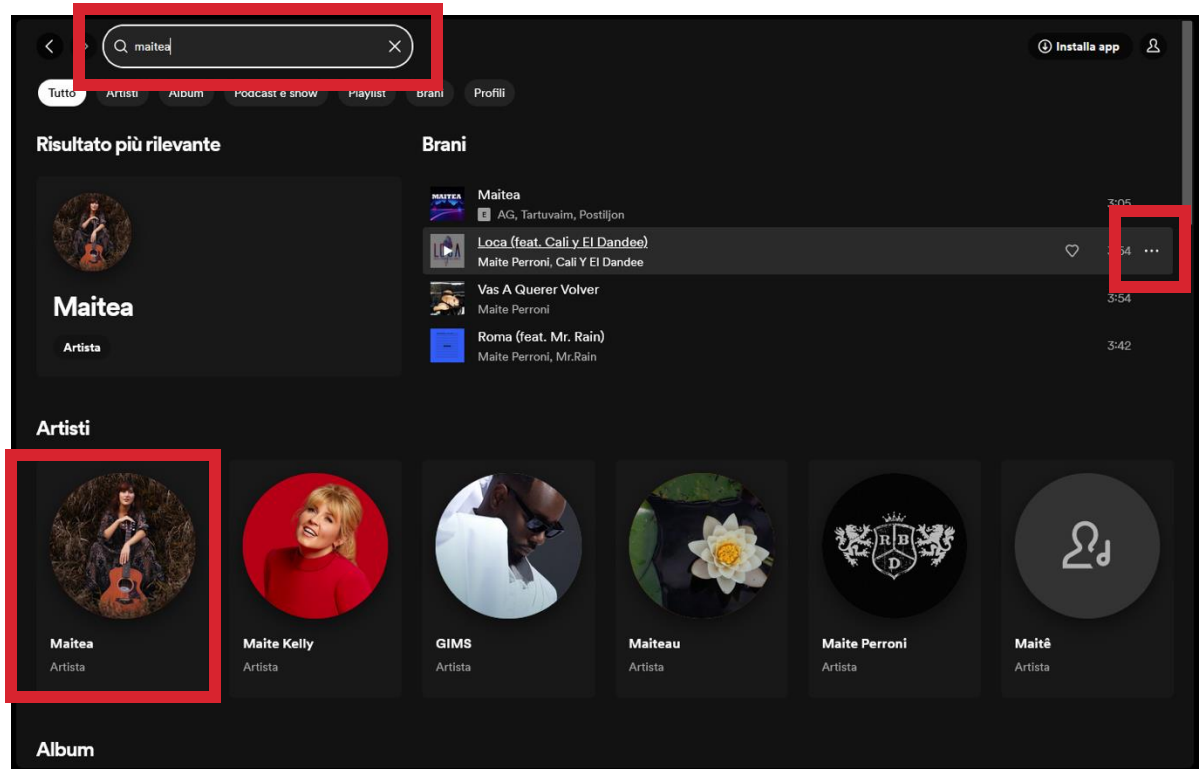
- The number of REST APIs grows larger and larger
- REST APIs contain programming defects and/or vulnerabilities
- Manual writing of test cases is limiting and costly

Automated black-box test cases generation for REST APIs

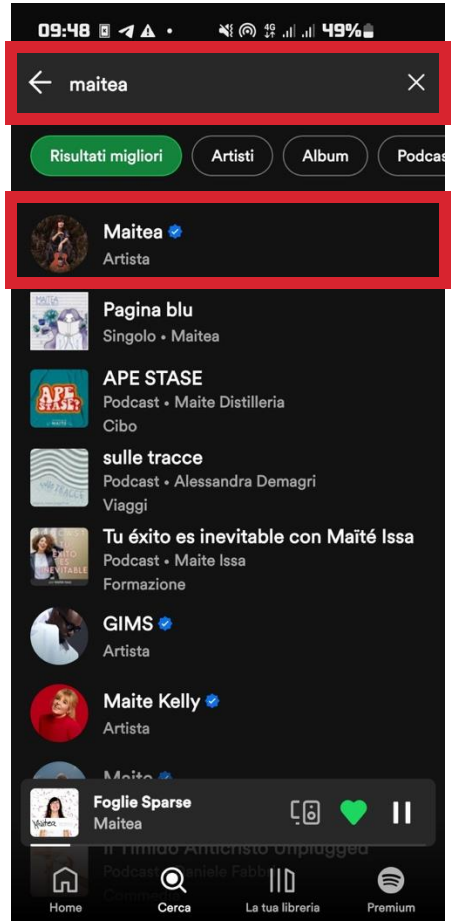
Test case


Create a playlist
Add a song to the playlist
Retrieve the playlist
Check if the playlist contains the song
PASS
FAIL

Testing a web/app ui



Spotify Web App



Spotify Android App

Challenge 1: Operations Testing Order

Albums

GET	/albums	Get Several Albums
GET	/albums/{id}	Get Album
GET	/albums/{id}/tracks	Get Album Tracks
GET	/artists/{id}/albums	Get Artist's Albums
GET	/browse/new-releases	Get New Releases
DELETE	/me/albums	Remove Users' Saved Albums
GET	/me/albums	Get User's Saved Albums
PUT	/me/albums	Save Albums for Current User
GET	/me/albums/contains	Check User's Saved Albums

Tracks

GET	/albums/{id}/tracks	Get Album Tracks
GET	/artists/{id}/top-tracks	Get Artist's Top Tracks

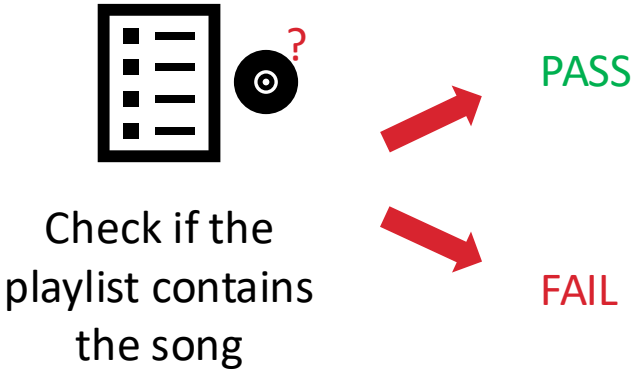
Spotify's REST API specification

Challenge 2: Test Input Values

- What are suitable <u>input values</u> for input parameters?
  - API specifications often do not provide example values
  - Validity of values might depend on the state of the API
    - E.g., resource identifiers

Challenge 3: The Oracle Problem

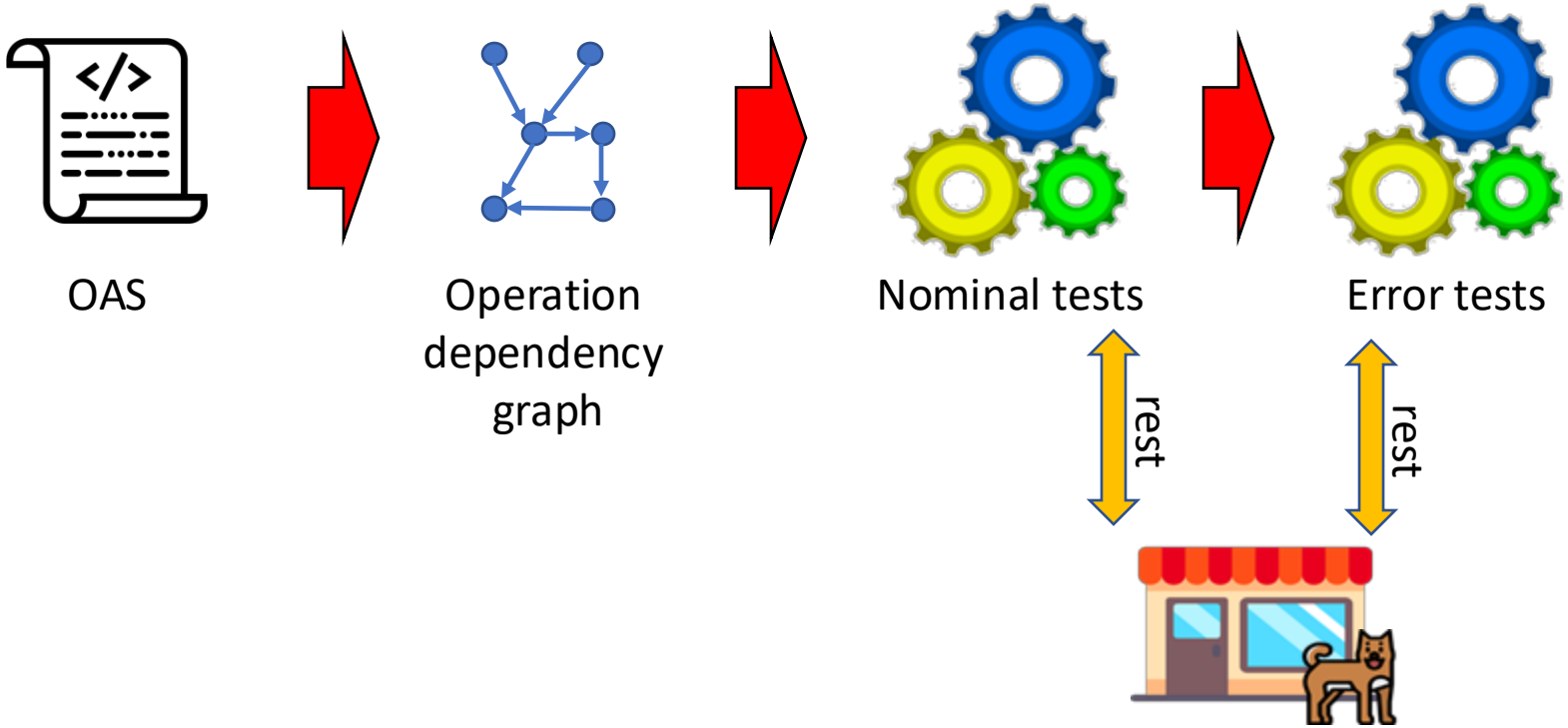
- Did the SUT behave as expected during/after the test scenario?



RestTestGen: Automated Black-Box Testing of Nominal and Error Scenarios in RESTful APIs

UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

Initial testing approach



Operation Dependency


```
/pets:
  get:
    summary: List all pets
    operationId: getPets
    tags:
      - pets
    responses:
      '200':
        description: PetIds
        content:
          application/json:
            schema:
              type: array
              items:
                type: object
                properties:
                  petId:
                    type: integer
```

output

```
/pets/{petId}:
  get:
    summary: Info for a specific pet
    operationId: getPetById
    tags:
      - pets
    parameters:
      - name: petId
        in: path
        required: true
        schema:
          type: string
```

input

Case mismatch

petID, **petid**, **petId**

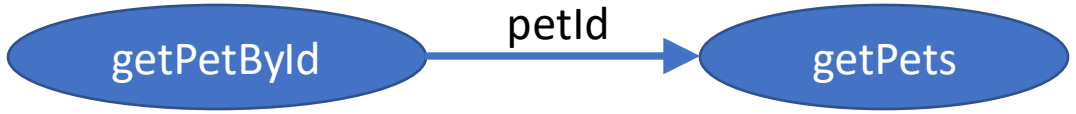
Id completion

/getPet ➠ Pet

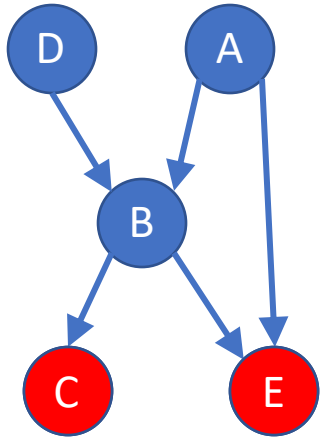
pet.id ➠ petId

Stemming

pets ➠ pet



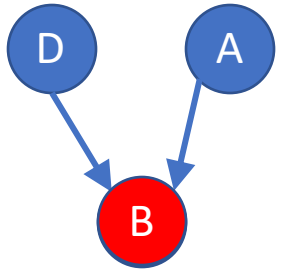
Operation Testing Order



- Leaf nodes are selected (no outgoing edges)
  - No input
  - Input is not available on operations output
- To maximize the likelihood of a successful test, resources might require to be in a certain status
- Leaf nodes are order based on the CRUD semantics

1. head
2. post
3. get
4. put/patch
5. delete

Operation Testing Order



- Tested operations are removed from the graph
- New operations become leaf nodes and can now be tested

The order in which operations are tested can not be precomputed, because it depends on what operations we succeed in testing

Input Value Generation

- Based on response dictionary
  - Map (name→values) of data observed at testing time, while testing previous operations
    - Exact name match petId ✓ petId
    - Concatenation of object + field pet.id ✓ petId
    - Name edit distance < threshold petsId ✓ petId
    - Key is a substring myPetId ✓ petId

- Based on OAS definition
  - Enum, example, default values
  - Random values (compatible with constraints)

HTTP Status Code Oracle

- 2xx means correct execution
  - 200: ok
  - 201: successful resource creation

- 4xx means error that is correctly handled
  - 400: bad request
  - 404: not found

- 5xx means error
  - 500: server crash

Schema Validation Oracle

```yaml
responses:
  '200':
    description: Expected response to a valid request
    content:
      application/json:
        schema:
          $ref: "#/components/schemas/Pet"
```

```yaml
components:
  schemas:
    Pet:
      type: object
      required:
        - id
        - name
        - tag
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
        tag:
          type: string
```

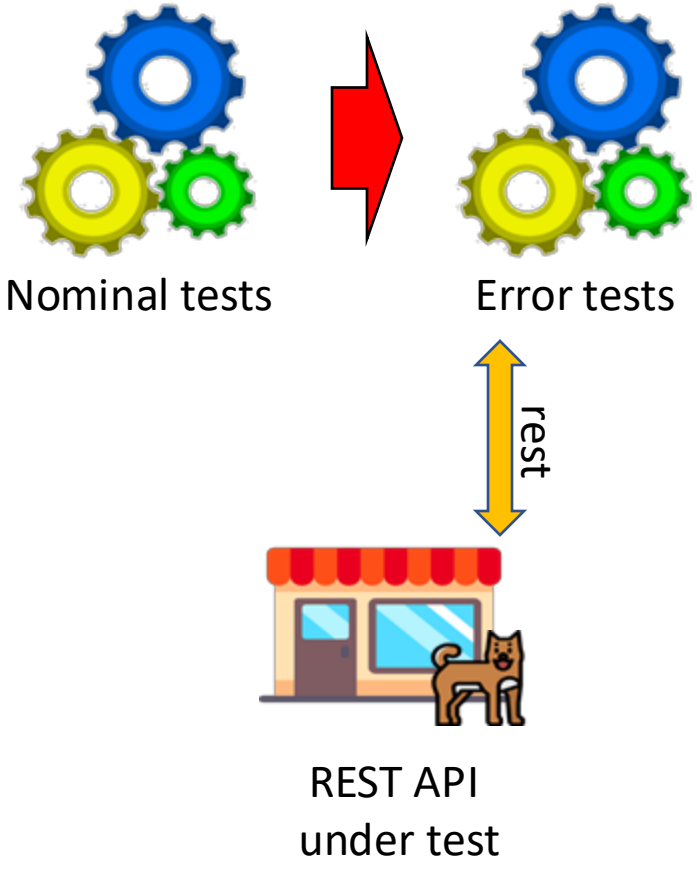
```
{
  "id": 1,
  "name": "doggy",
  "tag": "dog"
}
```

```
{
  "id": 1,
  "name": "doggy"
}
```

```
{
  "id": 1,
  "name": "doggy",
  "tag": 5
}
```

Testing of Error Cases

- Analyses how an API behaves when it is given wrong input data
- Mutation operators
  - Remove a **`required`** input field
  - Change field type
  - Change field value



HTTP Status Code Oracle

- 2xx means correct execution
  - 200: ok
  - 201: successful resource creation

✗

- 4xx means error that is correctly handled
  - 400: bad request
  - 404: not found

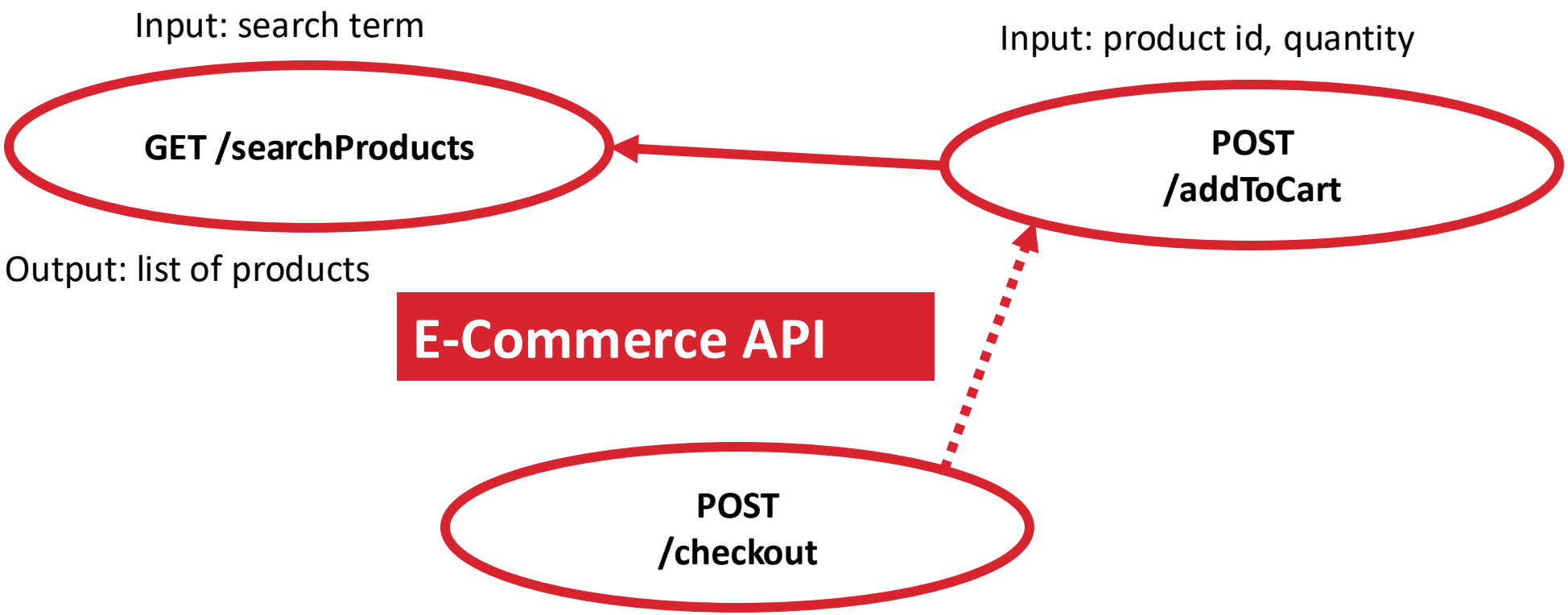
?

- 5xx means error
  - 500: server crash

✗

Deep Reinforcement Learning-Based REST API Testing

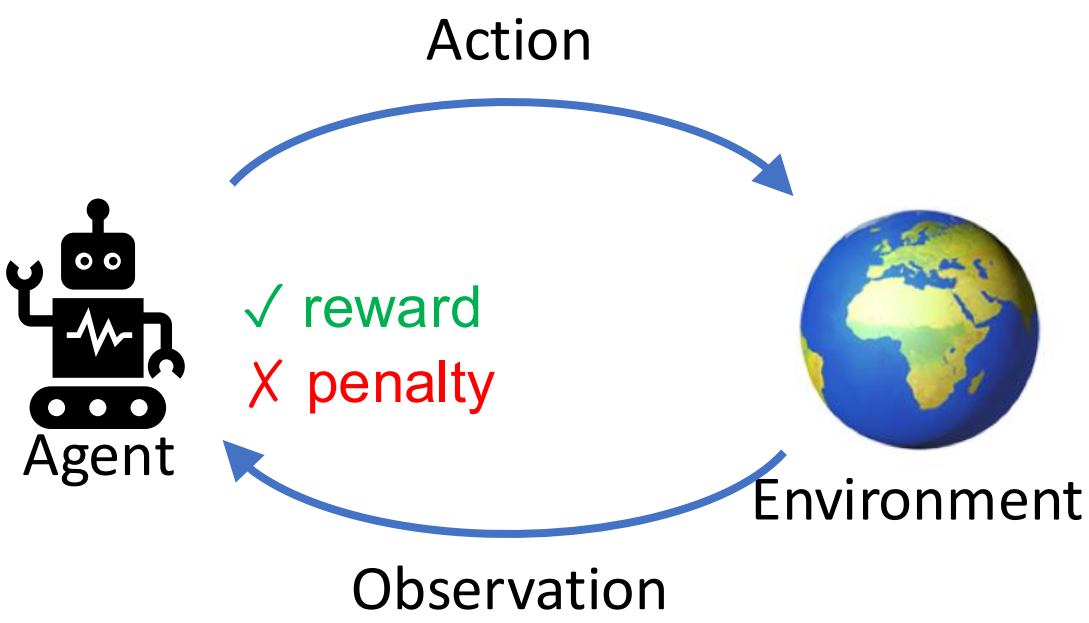
E-Commerce API



Problem: implicit dependencies are ignored!

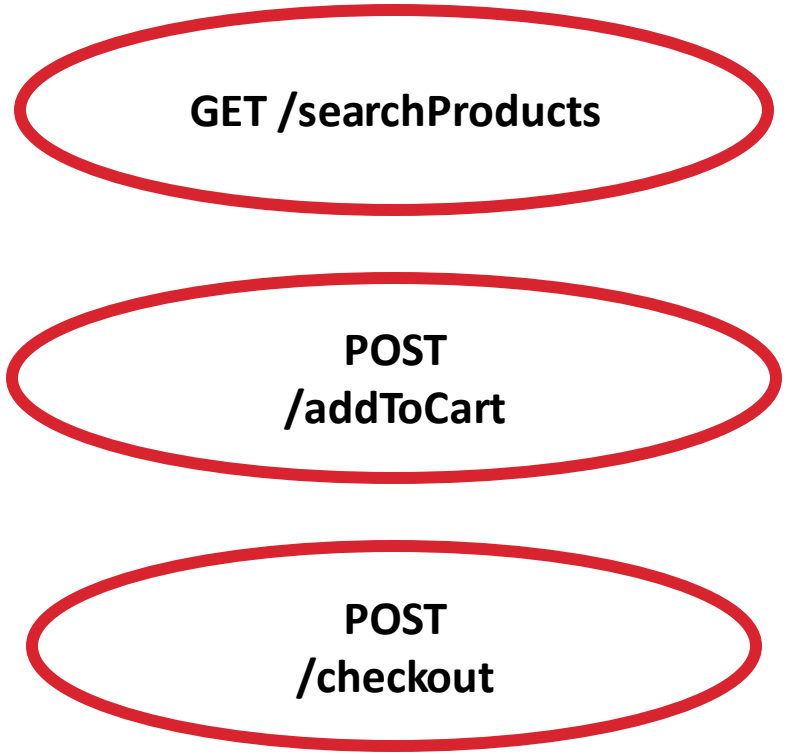
Reinforcement Learning

- **Action Space**: what we can act on
- **State Space**: what we measure of the environment
- **Reward Function**: the feedback signal



Action

- GET /searchProducts
- POST /addToCart
- POST /checkout



State

[0,0,0]

Search

Add to cart

Checkout

GET /searchProducts

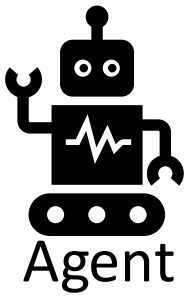
POST /addToCart

POST /checkout

State transition

$$[0,0,0] \xrightarrow{\text{Search}} [1,0,0] \xrightarrow{\text{Add to cart}} [1,1,0] \xrightarrow{\text{Checkout}} [1,1,1]$$

Reward: curiosity driven

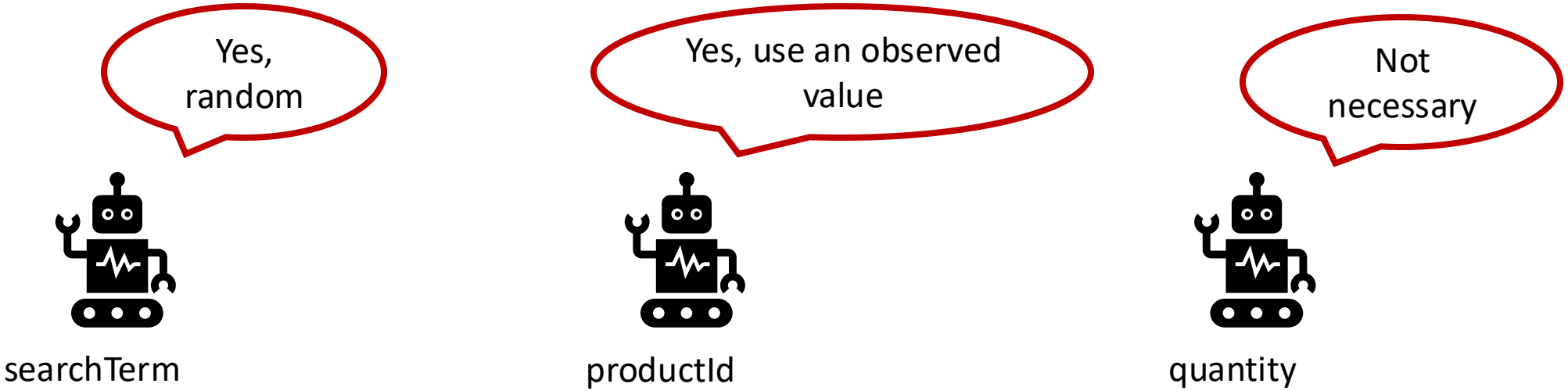


✓ reward
X penalty

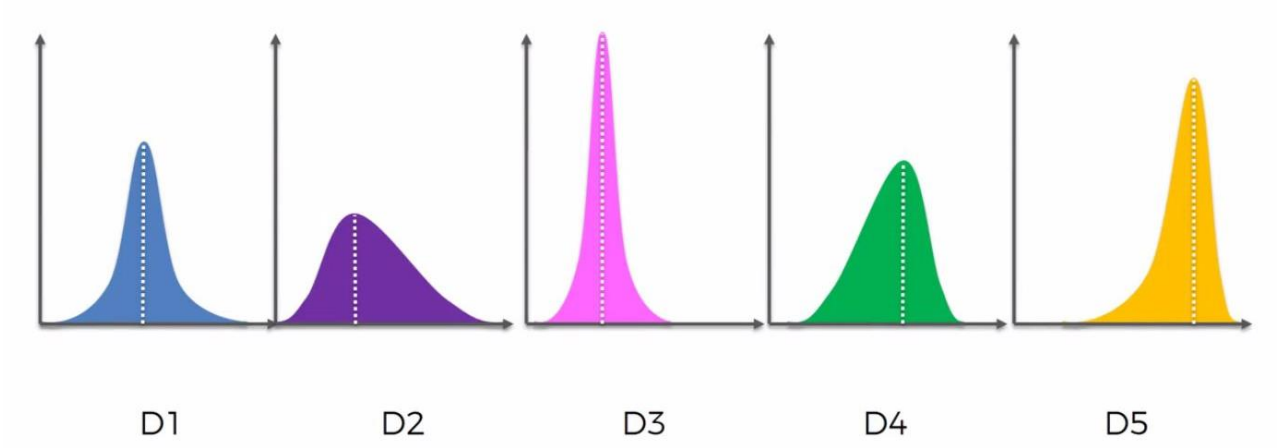
- Positive: Successfully tested a new operation (never visited so far)
- Negative: Successfully tested an operation that was already tested
- Slightly negative: Fail in testing an operation

Input Generation: Experience Driven

- Random, example values, response dictionary, etc.

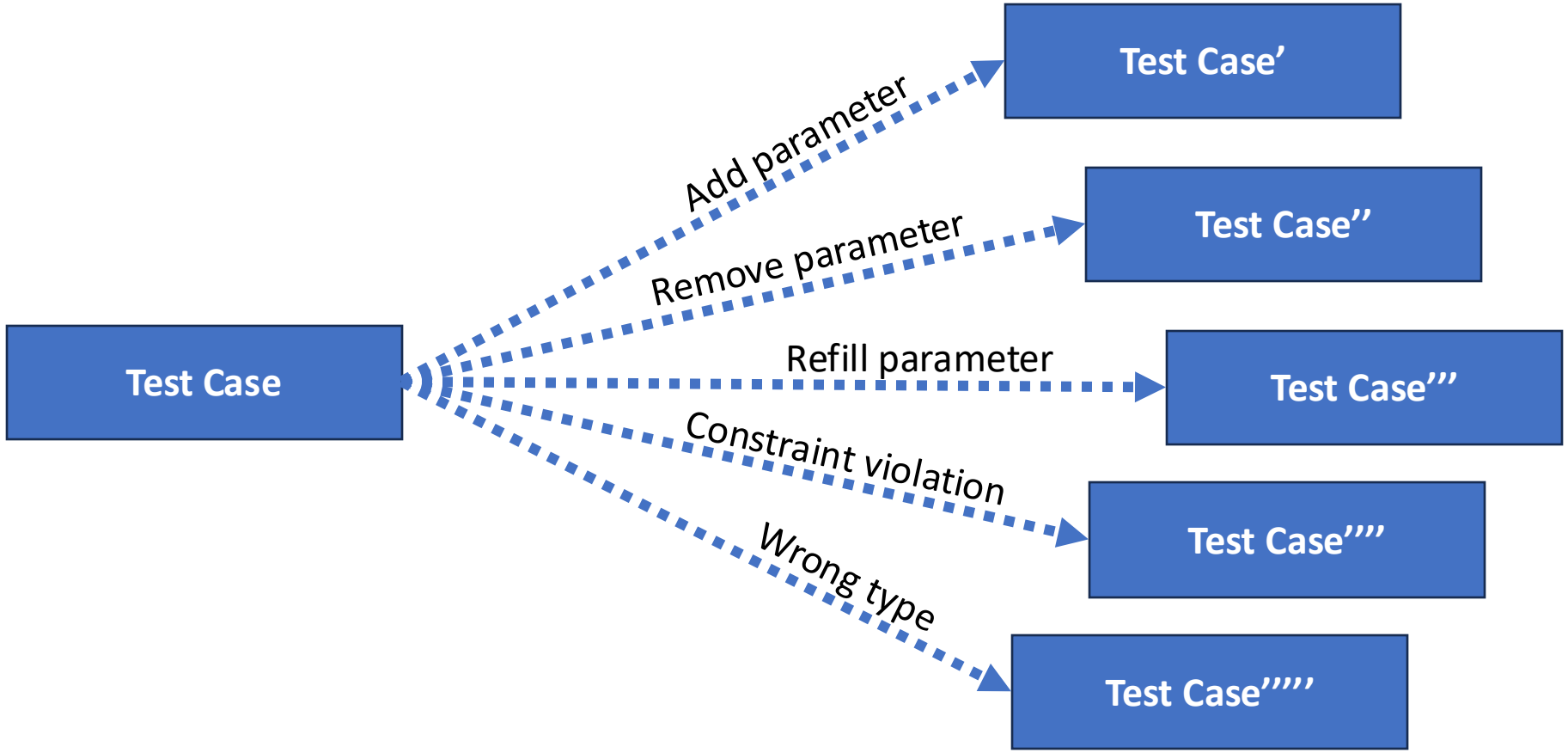


The Multi-Armed Bandit Problem

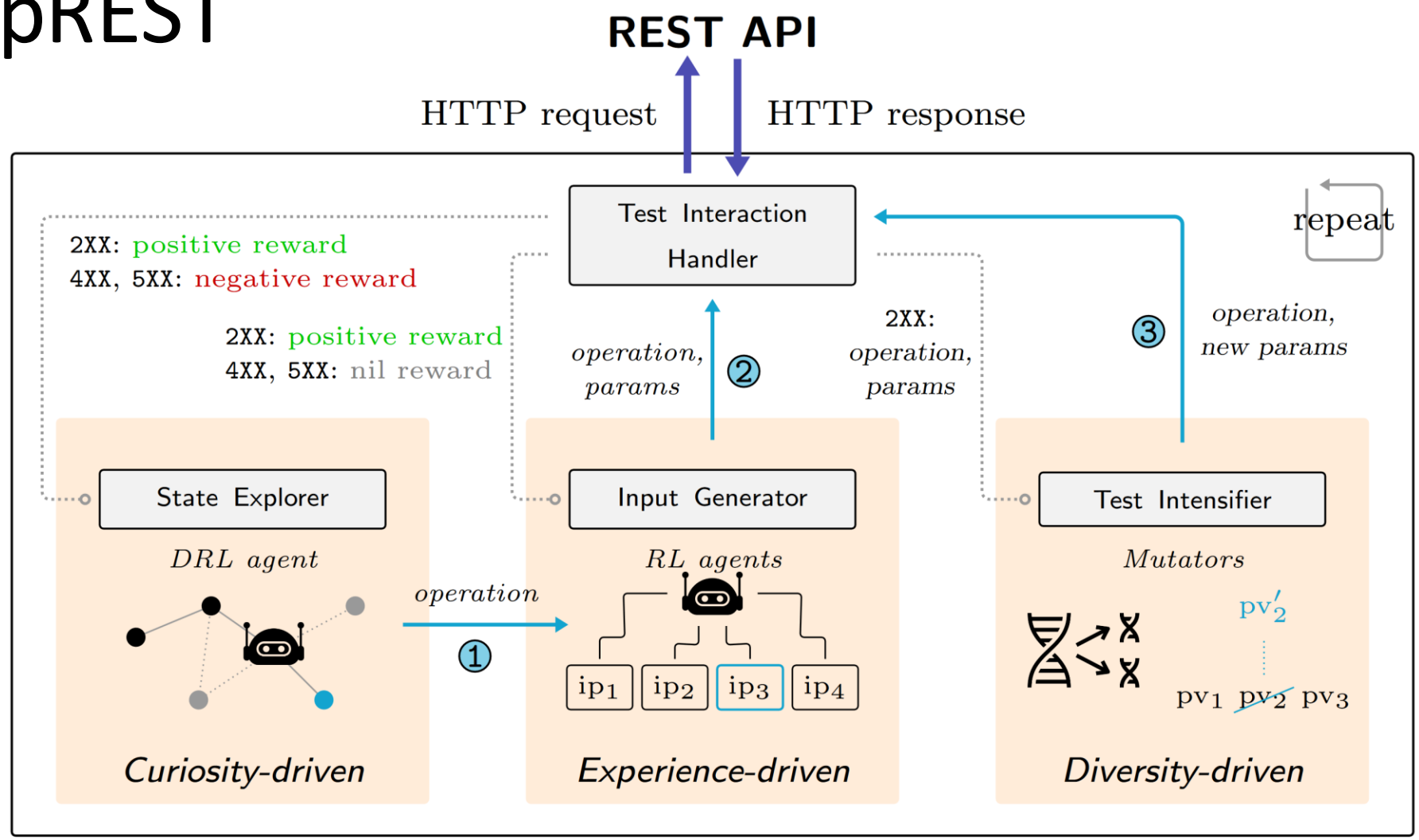


- Epsilon-Greedy algorithms

Test Intensification

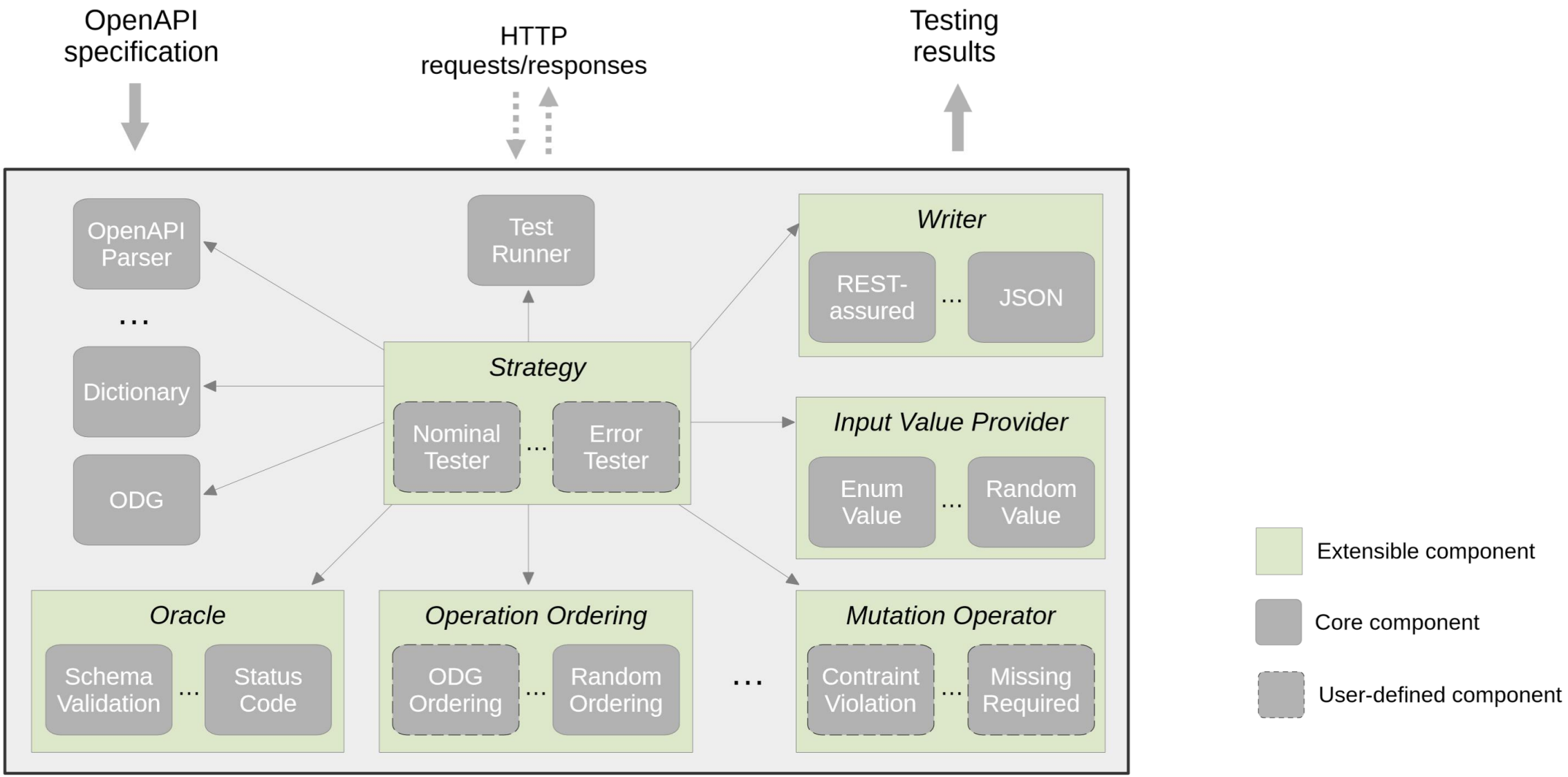


DeepREST


REST API
HTTP request
HTTP response
Test Interaction Handler
repeat
2XX: positive reward
4XX, 5XX: negative reward
2XX: positive reward
4XX, 5XX: nil reward
operation, params
2
2XX: operation, params
3
operation, new params
State Explorer
DRL agent
operation
1
Input Generator
RL agents
ip1
ip2
ip3
ip4
Test Intensifier
Mutators
pv'2
pv1
pv2
pv3
Curiosity-driven
Experience-driven
Diversity-driven

Evolution towards a reusable research tool

RestTestGen Framework



1. Core components

- A collection of ready-to-use classes that a researcher may directly integrate into a testing strategy
  - OpenAPI Parser
  - Parameters, Operations
  - Test interaction, Test sequence, Test runner, Test result
  - Operation dependency graph

2. Extensible components

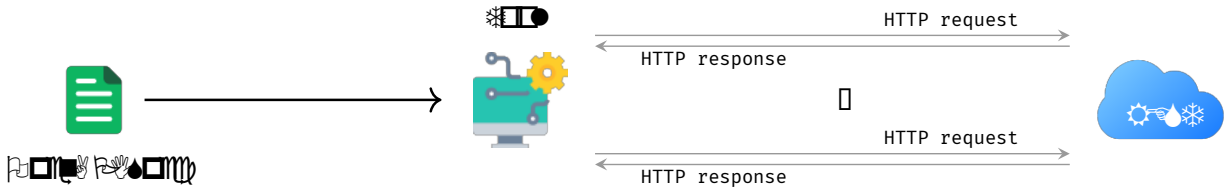
- A set of abstract and concrete classes for which researchers might want to provide a new implementation to deliver their new testing algorithm
  - Operation sorter
  - Fuzzer
  - Parameter value provider
  - Mutator
  - Oracle
  - Writer
  - Strategy

RTG documentation wiki

- git clone https://github.com/SeUniVr/RestTestGen-Wik
- cd RestTestGen-Wiki
- docker compose up –d
- open http://localhost:3000

Research on fuzzing REST APIs

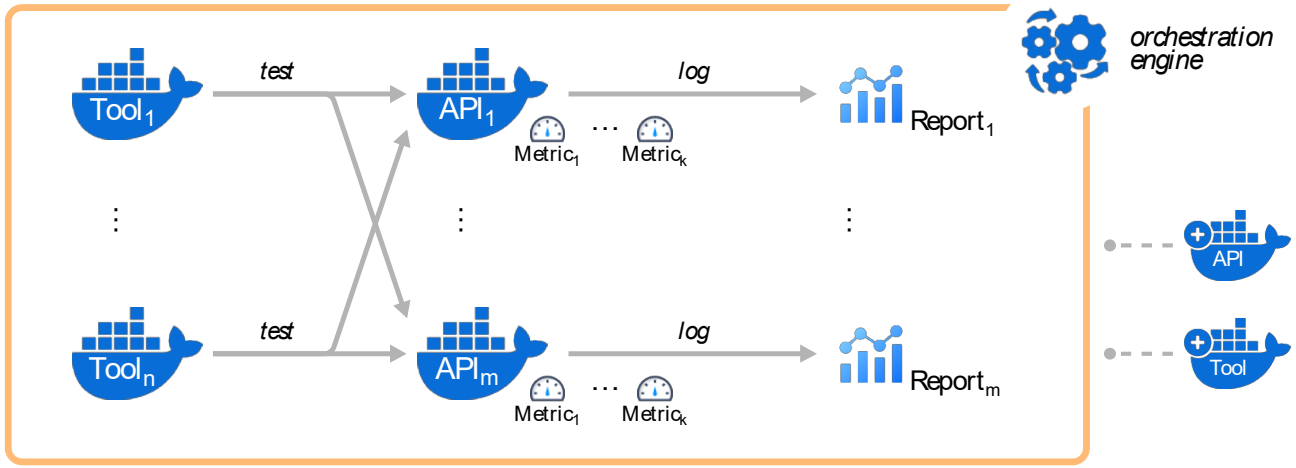
- Defining effective testing strategies
- Find working, testable case studies
- Compute testing metrics
- Compare with the baseline



RESTler	SwaggerFuzzer	Morest	DeepREST
ARAT-RL	RESTest	EvoMaster	APIFuzzer
Dredd	bBOXRT	RestCT	TnTFuzzer
RestPL	RestTestGen	QuickREST	Schemathesis

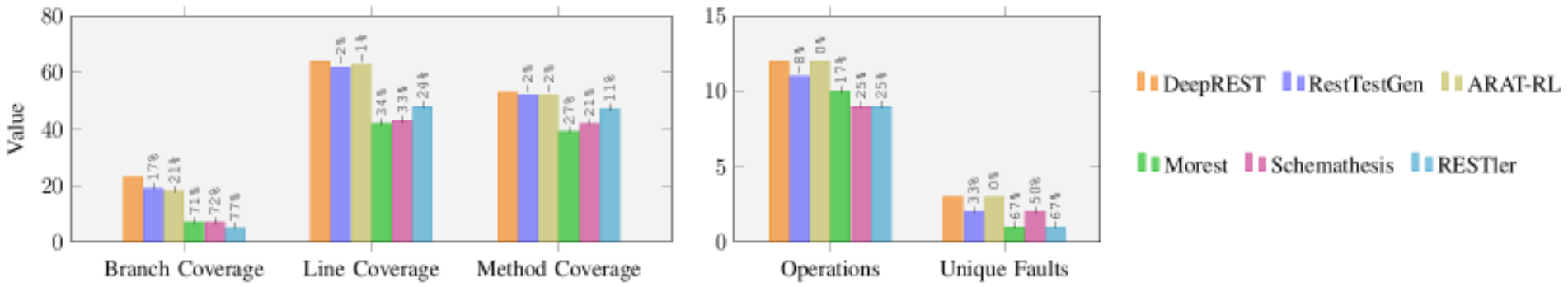
RestGym: a compassion testbed for researches

- Extensible container-based testing infrastructure
- Automated orchestration engine
- 6 Built-in test case generation tools and APIs
- 11 Built-in testing metrics
- Aggregate results and provide detailed reports

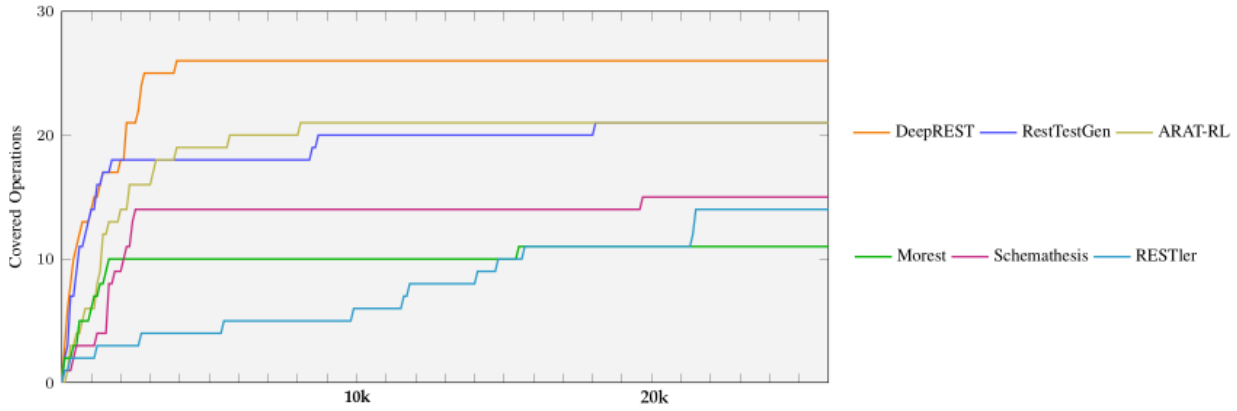


Testing reports

- Aggregate effectiveness results on all the APIs



- Efficiency trends on a single API



Tools competition

- Workshop on Search-Based and Fuzz Testing @ICSE26

- Structured empirical comparison
- Common ground
  - Hardware, OS, time budget
  - Case studies, with different features (complexity, …)
  - Metrics (e.g., coverage, defect/vulnerability revealing, …)
- Objective:
  - ~~Winner/loser~~
  - Relation between tool capabilities and case study features

Contributions

- Open-source tool implementation
  - https://github.com/SeUniVr/RestTestGen

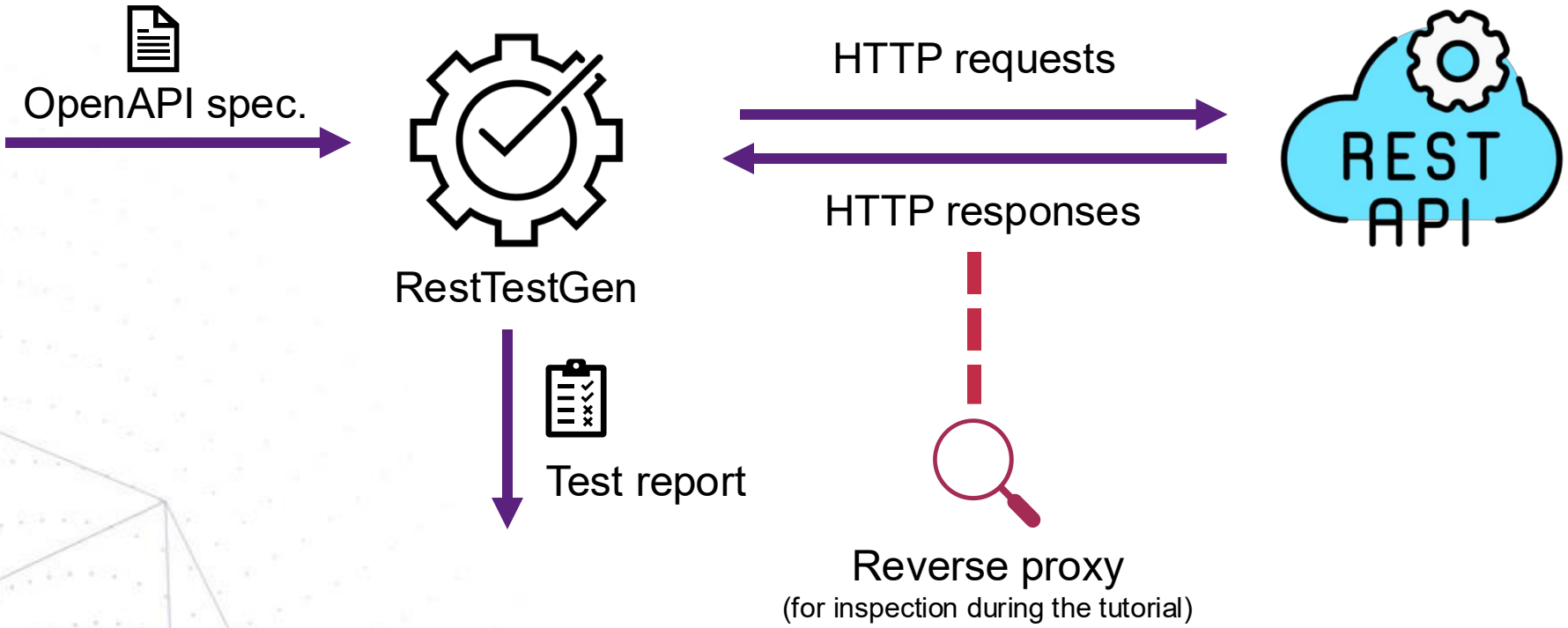
RestTestGen in practice

Davide Corradini
University of Luxembourg

Mariano Ceccato
University of Verona

UNIVERSITY OF LUXEMBOURG

Architecture



The Book Store API

Lists all books	GET	/books
Gathers data of a specific book	GET	/book/{bookUuid}
Creates a new book	POST	/book
Updates data of a book	PUT	/book
Deletes a book	DELETE	/book

Book entity

```
{
  "bookUuid": "ad850181-6399-4512-8ae6-2e9cbfb5cab0",
  "title": "Software Engineering, Vol. 1",
  "author": "Mariano Ceccato",
  "isbn": "string",
  "price": 16.5
}
```

Challenge: Generate successful requests to the API

1. The overall goal of today's tutorial is to implement a testing strategy capable of successfully test all operations exposed by the Book Store API
   - 200 status code class
2. Why successful requests are important?
   - a successful request is **understood** and **completed** by the server
   - only successful requests **can test application behavior**, trigger new resource creation, retrieval, or modification
   - in real scenarios, users provide correct data and sequences. Tools must **mirror users** by generating not just error cases, but valid use cases
   - once successful interactions are possible, **more complex test scenarios become feasible**

Basic strategy

1. Random ordering of calls to API operations
2. Random input data generation

New challenge: realistic input data

1. Problem:
   - the search space for valid input is huge
   - random generators cannot generate relevant and realistic input
   - example: it is unlikely to generate a valid ISBN for a string parameter
2. **Solution: a large langue model can generate realistic data**

New challenge: valid data (e.g., valid UUIDs)

1. Problem: Validity of part of the input data is state-dependent.
   - E.g., an UUID is only valid if a resource is available in the system with a specific UUID.
2. Observation: The API outputs valid data in some of its responses
3. **Solution: we can store this data and reuse it in subsequent requests**

New challenge: operation ordering

1. Problem: the random order undermines the effectiveness of approach
   - If ‘producer’ operation are executed at last, we miss useful data

2. **Solution: intelligent ordering of operation calls based on data-dependencies**
   - ‘producer’ operations must be executed first, ‘consumers’ later

RestTestGen

1. Open-source tool implementation

2. https://github.com/SeUniVr/RestTestGen

RestTestGen internals

Configuration file

- The file is "rtg-config.yml"
  - **strategyClassName**: strategy to run
  - **apiUnderTest**: the REST API to test.

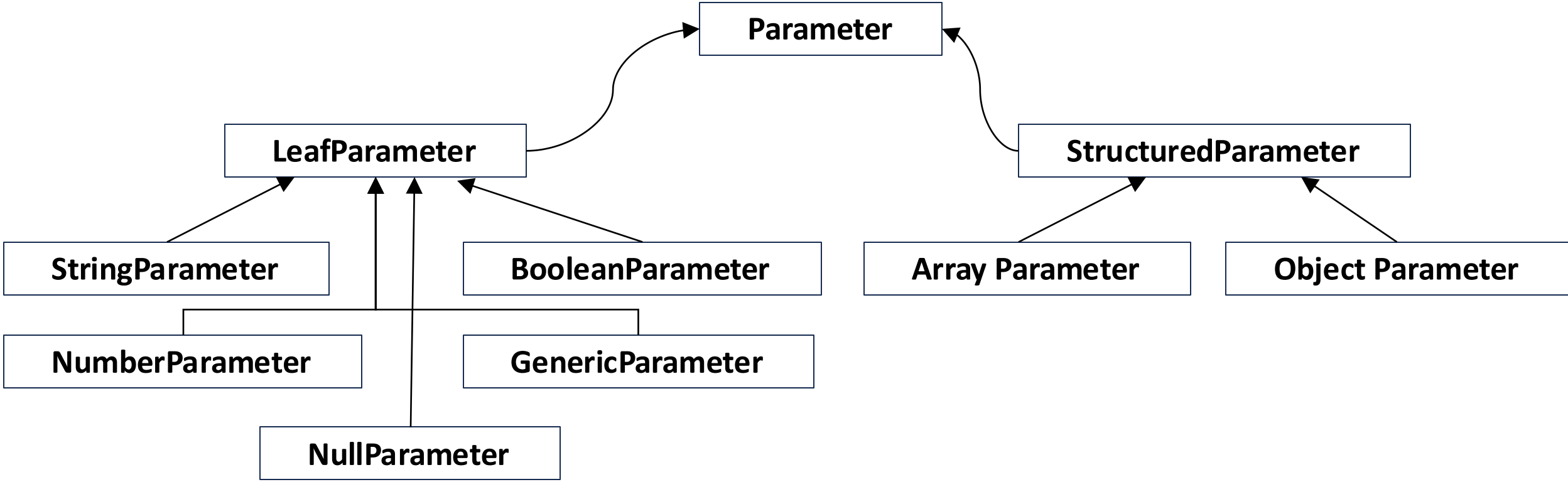
OpenAPI parser

- Very robust, able to deal with most of the common syntax mismatches and inconsistencies, often present in OAS files
- Fill internal structure to be used at testing time:
  - List of operations
  - **`Operation`**
    - Endpoint
    - HTTP method
    - Input/output schema (format)
    - **`DataTemplate`** for each input parameter (atomic or compound)
      - Name, type, domain, constraints

Operation

Operation
Method *//GET, POST, PUT, DELETE, …* Endpoint Parameters TypeOfCurdOperation *// CREATE, READ, UPDATE, DELETE* Validation Rules Request body *//StructuredParameter* Response *//StructuredParameter*
getAllRequestParameters() getHeaderParameters() getPathParameters() getCookieParameters() getOutputParameters()

Parameter types



Input/output parameter

Parameter
Name NormalizedName SchemaName Required Format Location DefaultValue EnumValue Examples Description Operation Parent
getChildren() getValue() setValueManually(Object) setValueWithProvider(ParameterValueProvider) deepClone()

Testcase

- **`TestSequence`**: an ordered list of interactions
- **`TestInteraction`**: data to send a single request to an operation by the **`TestRunner`**

TestInteraction
reference operation requestMethod *//GET, POST, PUT, DELETE, …* requestURL requestHeaders requestSentAt responseProtocol *//e.g. HTTP/1.1* responseStatusCode *//200, 404, 500, …* responseBody responseReceivedAt

TestSequence
testInteractions
isExecuted() inferVariablesFromConcreteValues()

TestRunner
- instance
run(TestSequence) tryTestInteractionExecution(TestInteraction)

ODG: Operation dependency graph

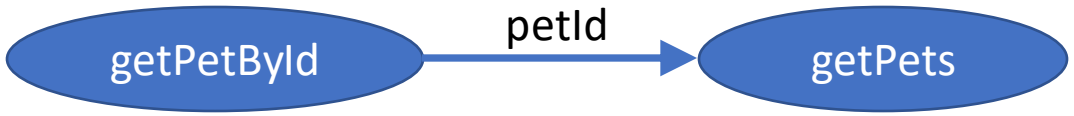
- **ODG** captures the producer-consumer relation among operations

```
/pets:
  get:
    summary: List all pets
    operationId: getPets
    tags:
      - pets
    responses:
      '200':
        description: PetIds
        content:
          application/json:
            schema:
              type: array
              items:
                type: object
                properties:
                  petId:
                    type: integer
```

output

```
/pets/{petId}:
  get:
    summary: Info for a specific pet
    operationId: getPetById
    tags:
      - pets
    parameters:
      - name: petId
        in: path
        required: true
        schema:
          type: string
```

input

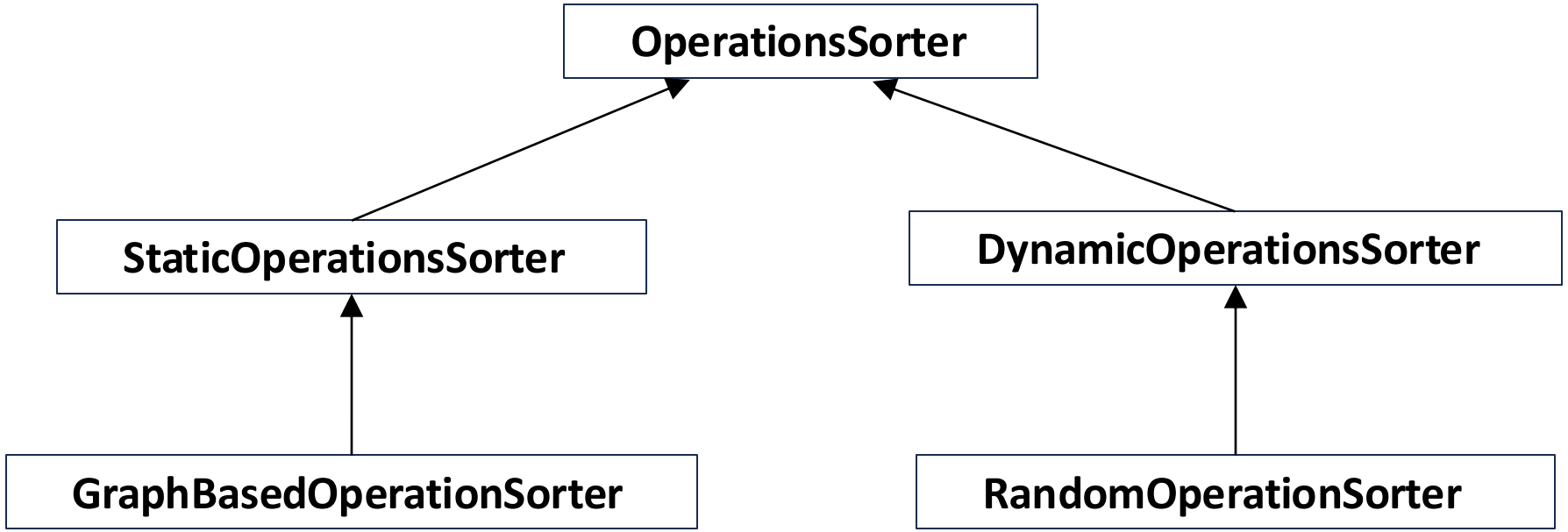


Operation sorter

- **`OperationSorter`** is responsible for deciding the order of operations in a test sequence

- **Static**: the ordering is performed before starting the execution of the **`TestSequence`**
- **Dynamic:** the order within the **`TestSequence`** is defined during the test execution as the next operation to be tested depends on the outcome of the previous ones

Operation sorter



Dictionary

- The **`Dictionary`** is used to store and retrieve values observed while testing

- Global dictionary for the whole testing session
- Possibly local dictionaries that can store the values observed in a smaller set of Test Interactions.

Input value provider

- **`ParameterValueProvider`** is responsible of providing a value for a parameter under consideration, based on its **`DataTemplate`**:

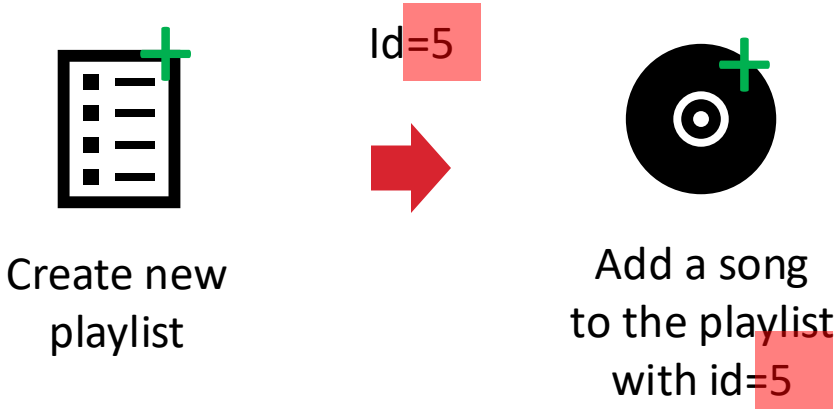
- **`ExampleValueProvider`**: returns a random value from among the examples.
- **`DefaultValueProvider`**: returns the default value of a parameter.
- **`RandomValueProvider`**: generates a random value based on the specification parameter pattern.
- **`DictionaryValueProvider`**: chooses a value from the dictionary, under the condition that a value for a parameter with the same name has already been observed in the test session.

Other input value providers

- **`NarrowRandomValueProvider`**: similar to the **`RandomValueProvider`**, but some of the values are generated in a narrower range.

- **`RequestDictionaryValueProvider`**: chooses a value for the parameter from a dictionary of values used for successful requests (i.e. 2XX status code).
- **`ResponseDictionaryValueProvider`**: chooses a value for the parameter from a dictionary of response values observed in previous interactions.
- **`LastRequestDictionaryValueProvider`**: is the same as the Request Dictionary Value Provider, but the value assigned to the parameter is the last one observed.
- **`LastResponseDictionaryValueProvider`**: is equal to the Response Dictionary Value Provider, but the value that is assigned to the parameter is the last one observed.

Other input value providers



Multi-strategy input value provider

- **`RandomSelectorInputValueProvider`**: randomly chooses a single-strategy input value provider from those available and compatible for an input parameter.
- **`EnumExamplePriorityInputValueProvider`**: prioritise enum and example values as they are more likely to be effective, selecting them with high probability and selecting the remaining single-strategy providers with lower probability.
- **`GlobalDictionaryInputValueProvider`**: priority is given to the use of a global dictionary.
- **`KeepLastIDInputValueProvider`**: the main objective of this strategy is to maintain and reuse the last observed ID value for a parameter.
- **`LocalDictionaryInputValueProvider`**: priority is given to the use of a local dictionary. A local dictionary is defined as a dictionary within which values from a sub-set of **`TestInteraction`** have been stored.

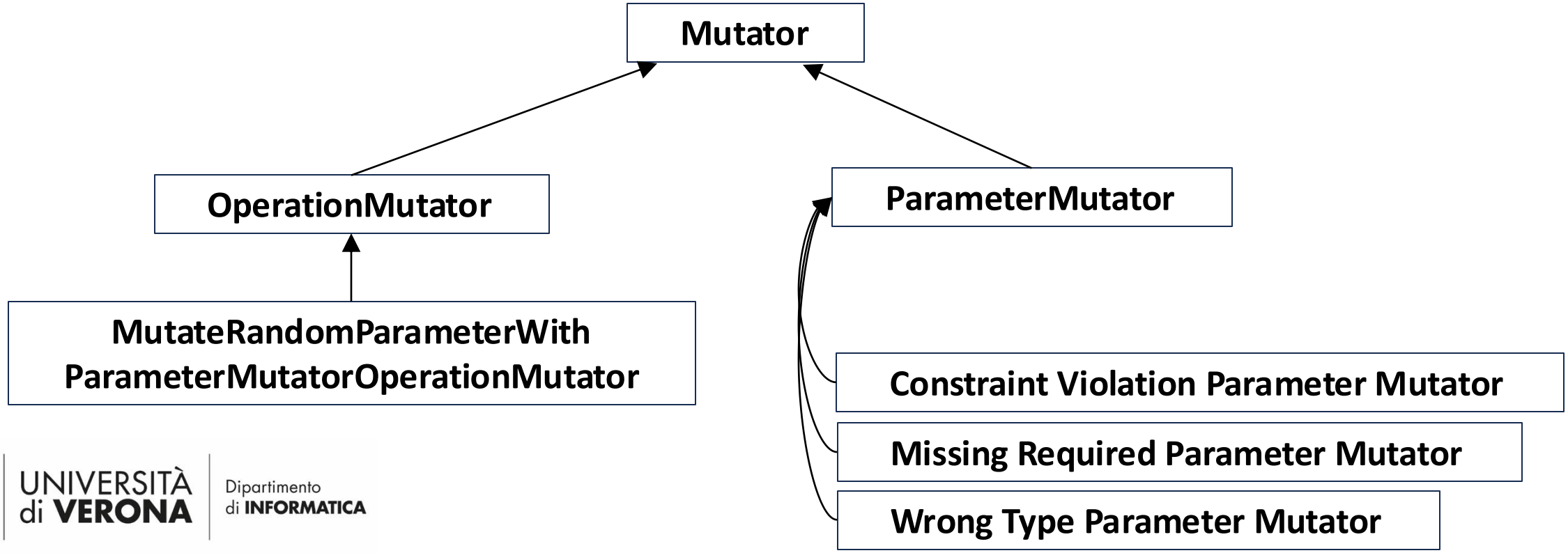
Fuzzer

- A **`Fuzzer`** generates the test sequence(s), including the operation order and their input values

- **`NominalFuzzer`**: simulate different input scenarios to test the behavior of an operation.
- **`ErrorFuzzer`**: simulate erroneous inputs to test the API's error handling
- **`MassAssignmentFuzzer`**: generating sequences to check the vulnerability of mass assignments

Mutation Operators

- A **`Mutator`** changes the value of a given parameter, e.g., to intensify testing on a given operation or to try and make it fails after it succeeded



Oracle

- An **`Oracle`** makes a decision on a **`TestSequence`**, by emitting a **`TestResult`**
  - PENDING: the test case has not yet been executed
  - PASS: the test case has passed (no defect)
  - FAIL: the test case did not pass (defect revealed)
  - ERROR: the test case has encountered an error during execution
  - UNKNOWN: the oracle is not able to make a decision

Available oracles

- **`StatusCodeOracle`**
  - PASS: 2XX status code
  - FAIL: 5XX status code
  - UNKNOWN: 4XX status code

- **`SchemaValidationOracle`**
  - PASS: data in the HTTP response matches schema
  - FAIL: otherwise

- **`MassAssignmentOracle`**
  - FAIL if the vulnerability was successfully exploited twice
  - PASS if the exploit attempt was unsuccessful

Writer

- Export test sequences to files, e.g., to be used externally

- **`ReportWriter`**: executed test sequence as a JSON file, including the HTTP request/response of each interaction and the result of the oracle.
- **`RestAssuredWriter`**: test sequence as a Java test case in Java using the RESTAssured library
- **`CoverageReportWriter`**: detailed test coverage as JSON file

Coverage metrics

Input coverage metrics

- Path coverage
- Operation coverage
- Parameter coverage
- Parameter value coverage
- Request content-type coverage

Output coverage metrics

- Status code class coverage
- Status code coverage
- Response content-type coverage

Metrics are computed as defined by Martin-Lopez et al.* with adaptations in some cases to make them operative.

* A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, “Test coverage criteria for RESTful web APIs,” in Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, 2019, pp. 15–21.

Strategy

- The **`Strategy`** is the entry point of the framework
- Represents business logic generating test cases
- Integrates the framework components, possibly after extending and/or customizing them

```
OperationsSorter sorter = new GraphBasedOperationsSorter();
while (!sorter.isEmpty()) {
    Operation operationToTest = sorter.getFirst();
    NominalFuzzer nominalFuzzer = new NominalFuzzer(operationToTest);
    List<TestSequence> nominalSequences = nominalFuzzer.generateTestSequences(
                                                        config.getNumberOfSequences());

    for (TestSequence testSequence : nominalSequences) {
        // Run test sequence
        TestRunner testRunner = TestRunner.getInstance();
        testRunner.run(testSequence);
        // Evaluate sequence with oracles
        StatusCodeOracle statusCodeOracle = new StatusCodeOracle();
        statusCodeOracle.assertTestSequence(testSequence);

        // Write report to file
        ReportWriter reportWriter = new ReportWriter(testSequence);
        reportWriter.write();
        RestAssuredWriter restAssuredWriter = new RestAssuredWriter(testSequence);
        restAssuredWriter.write();
      }
   sorter.removeFirst();
}
```

```
private TestSequence generateTestSequence() {
    editableOperation = operation.deepClone();

    resolveCombinedSchemas();
    populateArrays();
    setValueToLeaves();

    // Create a test interaction from the operation
    TestInteraction testInteraction = new TestInteraction(editableOperation);

    // Encapsulate test interaction into test sequence
    TestSequence testSequence = new TestSequence(this, testInteraction);
    String sequenceName = !editableOperation.getOperationId().isEmpty() ?
            editableOperation.getOperationId() :
            editableOperation.getMethod().toString() + "-" + editableOperation.getEndpoint();
    testSequence.setName(sequenceName);
    testSequence.appendGeneratedAtTimestampToSequenceName();

    // Create and return test sequence containing the test interaction
    return testSequence;
}
```