COTS: Connected OpenAPI Test Synthesis for RESTful Applications

Christian Bartolo Burlò[1][http://orcid.org/0000-0002-0016-086X], Adrian Francalanza[2][https://orcid.org/0000-0003-3829-7391], Alceste Scalas[3][https://orcid.org/0000-0002-1153-6164], and Emilio Tuosto[1][https://orcid.org/0000-0002-7032-3281]

[1] Gran Sasso Science Institute, L'Aquila
[2] Department of Computer Science, University of Malta, Msida, Malta
[3] DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark

Abstract. We present a novel model-driven approach for testing RESTful applications. We introduces a (*i*) domain-specific language for OpenAPI specifications and (*ii*) a tool to support our methodology. Our DSL, dubbed COpenAPI, is inspired by session types and enables the modelling of communication protocols between a REST client and server. Our tool, dubbed COTS, generates (randomised) model-based test executions and reports software defects. We evaluate the effectiveness of our approach on several open source applications. Our findings indicate that our methodology can identify nuanced defects in REST APIs and achieve comparable or superior code coverage when compared to much larger handcrafted test suites.

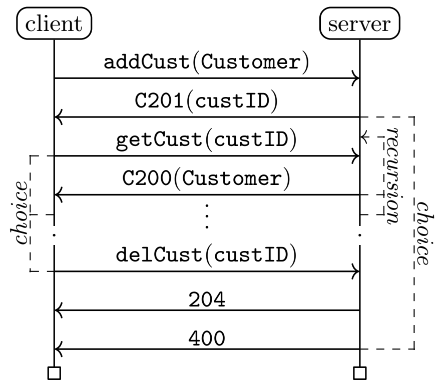
1 Introduction

Modern software is increasingly composed of possibly remote components that are independently developed and coordinated by exchanging data across a communication network. The interaction of such components may be based on classic client-server Internet protocols (such as SMTP, IMAP, and POP3), various forms of remote procedure calls (RPCs), web-based standards — such as the REpresentational State Transfer (REST) architectural design.

Ensuring the correctness and reliability of these applications is notoriously hard. Software developers typically create handcrafted test suites, which must address many (and ever-growing) application usage scenarios: It is often the case that such manually-written test suites are developed intermittently, over long periods of time, by a variety of testers. This makes the software development and testing process error-prone and susceptible to inconsistencies [23,33]. A number of efforts have emerged with the aim of simplifying the testing process of component based-software by streamlining the amount of manual work through automatic test generation. Most of these efforts focus on automatic test generation for applications exposing an API over a network, where there is a considerable interest in addressing the popular REST API style [4,5,7,13,15,21,30–33,36,42].

```
 1  paths:
 2    /customer:
 3      POST:
 4        operationId: addCust
 5        responses: '201' ...
 6    /customer/id:
 7      GET:
 8        operationId: getCust
 9        responses: '200' ...
10      DELETE:
11        operationId: deleteCust
12        responses: '204' ...
13    /card:
14      POST:
15        operationId: addCard
16        responses: '201' ...
17    /address:
18      POST:
19        operationId: addAddr
20        responses: '201' ...
```

(a) OpenAPI spec. for an online store.



(b) Message Sequence Chart outlining the invocation dependencies in Fig. 1a.

Fig. 1: Typical documentation provided for REST APIs.

Example 1. Fig. 1a shows a fragment of a REST API of the *SockShop* web application [37] written as an OpenAPI specification [28]. A new customer can be created by passing the necessary credentials as payload to the operation with ID `addCust` (lines 3-4); this operation returns a unique reference identifier for the customer, meant to be used in subsequent operations to create a payment card (`addCard`) and an address (`addAddr`) to associate with (previously created) customers. The customer information including any cards and addresses linked to it can be retrieved with `getCust` and deleted with `deleteCust`. □

The OpenAPI in Fig. 1a is a typical instance of a REST API description. OpenAPI is used pervasively for such specifications: it provides information on the available URIs, the corresponding HTTP methods and respective responses with the associated data formats (omitted from Fig. 1a). However, OpenAPI does not specify the relationships and dependencies between different API invocations. For instance, Fig. 1a does *not* express that:

- Customer information can be added (`addCard`, `addAddr`), retrieved (`getCust`), or deleted (`deleteCust`) only after the customer is created (`addCust`).
- Operations `addCard`, `addAddr` or `getCust` cannot be executed after the customer has been deleted (`deleteCust`).
- Operations `addCard`, `addAddr` and `getCust` may be interleaved without effecting the successful outcome of the other invocations.

– The variable `custId`, an ID given to the customer upon creation is to be used to perform any operation on the customer (such as retrieval or deletion).

These dependencies induce an ordering in the API invocations, depicted in the message sequence chart in Fig. 1b. Unfortunately, this information is often omitted (cf. [8,19,37]) or only informally stated (as e.g., in [14]). In development contexts, this is often sufficient as developers are typically able to deduce the intended order of invocation by examining the descriptions of requests and the data types exchanged by each operation. However, this omission becomes problematic when testing. Deviations from the dependencies identified earlier in Fig. 1b can be categorized as *logic-based faults*, that is, errors stemming from incorrect or unintended actions within the business logic of the application. Such violations are often intricate and context-specific, arising from the unique interactions and workflows within the application. These faults are not easily detectable through basic system responses or standard error codes and typically necessitate comprehensive, scenario-based testing for identification. Furthermore, these faults are intrinsically linked to the essential operations and branching mechanisms of the application, impacting its ability to perform as intended.

Previous work on REST API testing proposes various *fully-automated* test generation strategies [5, 21, 33, 41]. These *push-button* tools excel at detecting *systemic errors* that are related to the system's overall functioning rather than specific logical operations, such as server crashes, malformed requests, unauthorized access attempts, or resource not found errors. However, it is unlikely that logic-based errors are uncovered by invoking the API without an explicit predefined model of the intended interaction with the API. *E.g.,* in the *SockShop* application from Example 1, such tools are able to detect if the SUT crashes when creating a new customer, but it is unlikely that they are able to detect whether the SUT allows the retrieval of a previously deleted customer. A test to detect such errors should create a customer and then try to retrieve the same customer after having deleted it. It has been empirically shown that the quality of tests substantially improves when conducted in a *stateful* manner, i.e., by exploring states of the system under test (SUT) that are reachable via sequences of invocations [5]. A recent work concludes that: *"existing tools fail to achieve high code coverage due to limitations of the approaches they use for generating parameter values and detecting operation dependencies"* [22].

The state-of-the-art of testing REST APIs can be seen on a spectrum. At one end, manual tests, though labor-intensive, excel at uncovering complex, logic-based errors. At the opposite end are fully-automated testing tools, which, while requiring minimal effort, fall short in detecting eloaborate errors. This work seeks a middle ground, leveraging a *model-based* approach. Our goal is to automate the identification of more logic-based faults, thus bridging the gap between labor-intensive manual testing and the scope of fully-automated tools.

Contributions We present a model-based, tool-supported methodology for the automatic testing of web applications exposing REST APIs. Our approach allows the specification of API dependencies and constraints. Our contributions are:

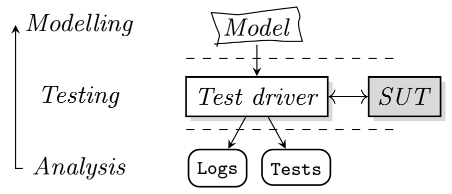


Fig. 2: Methodology overview

1. COpenAPI: a domain-specific language designed to specify dependencies between requests in an OpenAPI specification that capture the state of the interaction with the SUT and the sequencing of message exchanges;
2. COTS: an automated tool leveraging COpenAPI models to generate tests that interact with the SUT and assess the correctness of its responses.
3. Experimental results demonstrating the effectiveness of COpenAPI and COTS using case studies from various application domains. We compare our results against RESTful API testing tools and manually-written tests.

The key benefits of our approach are in terms of (*i*) a high degree of expressiveness due to the possibility of specifying data dependencies, (*ii*) effectiveness of the approach that can identify logic-based faults, and (*iii*) high level of coverage.

2 A Model-Driven, DSL-Based Methodology

2.1 Background

Based on the REST architectural style [16] and HTTP protocol, REST APIs facilitate software services on the web by exposing *resources* through HTTP URIs, which are manipulated using standard HTTP *request methods* like `GET`, `POST`, `PUT`, and `DELETE`. Responses are represented by standard HTTP codes (e.g., 200 for success, 404 for missing resources, 500 for server errors). While OpenAPI and GraphQL [18] are prominent for documenting REST APIs, our emphasis on OpenAPI stems from its broad adoption and tool support for generating client libraries across various programming languages. Though our methodology primarily targets OpenAPI, it's flexible for other web API standards like GraphQL.

Testing REST APIs introduces unique challenges due to the necessity for a novel modeling language and methodology that can accommodate the complex, state-dependent interactions and evolving nature of web services. Traditional model-based testing methods are insufficient for the dynamic and heterogeneous web API landscape.

To bridge this gap, we introduce the COpenAPI DSL (§ 3), designed to articulate the intricate sequences and state dependencies characteristic of RESTful interactions, enabling precise testing beyond the capabilities of existing models.

2.2 Methodology

Our methodology consists of the three phases depicted in Fig. 2:

(P1) *modelling phase:* construct a model that describes how a client application might interact with the SUT, and how the SUT is expected to respond to the client's inputs;

(P2) *testing phase:* automatically generate a *test driver* that tests the SUT by interacting with it according to the model, and reports whether the SUT responses violate the model;

(P3) *analysis phase:* inspect the outputs produced by test driver to identify faults in the SUT.

The phases **(P1)**, **(P2)** and **(P3)** are iterated when the model has to be refined according to the findings of the analysis or it has to be evolved according to the life cycle of the SUT. The main goal of our approach is to automate the testing phase, and let testers shift their efforts to the modelling phase, reducing the need to develop (and maintain) a large suite of handcrafted test.

The first step towards the methodology entails determining a suitable model for phase **(P1)**, allowing for the automatic derivation of tests for RESTful applications. To write such model, we design COpenAPI (cf. § 3) taking inspiration from *(binary) session types* [20].

The next step in our methodology is **(P2)**, i.e., the testing phase. In this phase, the test driver interacts with the SUT by sending and receiving messages according to the COpenAPI model. The test driver needs to perform two types of interactions with the SUT:

– the test driver must send requests to the SUT of the correct format and payload type; and
– the test driver needs to check that the responses received from the SUT do not violate the COpenAPI model. These violations may be of three forms:
  - the response code received is not valid, or
  - the payload data does not have the expected format, or
  - the response code and payload data format are valid, but the payload data causes an assertion violation in the model (e.g., by violating a constraint involving data from a previous request/response).

The last phase of the methodology Item **(P3)** is devoted to the identification of faults in the SUT, by inspecting the outputs of the test driver. The test driver produces a successful test when it manages to complete a full traversal of the COpenAPI model without finding any of the aforementioned errors. To help with the analysis, the test driver produces two outputs:

1. a log file with information about every performed test: the random seed used to generate it, the sequence of messages sent and received and their respective payloads, and the test outcome (pass/fail); and
2. an offline representation of failed tests, usable for manually reproducing faults of the SUT without re-executing the test driver, useful for, e.g., bug reporting.

3 A DSL for OpenAPI

We introduce a DSL based on an augmented form of *session types* [12, 20, 35]. Intuitively, a session type specifies the valid sequences of message exchanges (i.e., the *protocol*) that regulate the interaction between communication programs; here we focus on *binary session types*, which model how a program is expected to interact with just one other program (in our case, a client with a RESTful server).

The formal syntax of our augmented session types S is:

$$\begin{array}{rcll}
\mathsf{S} & ::= & +\{!\mathit{operationId}_i(\mathsf{gen}_i).\mathsf{S}_i\}_{i\in I} & \textit{(internal choice)} \\
& \mid & \&\{?\mathit{responseCode}_i(\mathsf{dec}_i)\langle A_i\rangle.\mathsf{S}_i\}_{i\in I} & \textit{(external choice)} \\
& \mid & \mathsf{rec}\ X.\mathsf{S} & \textit{(recursion)} \\
& \mid & X & \textit{(recursion variable)} \\
& \mid & \mathsf{end} & \textit{(termination)} \\
\mathsf{T} & ::= & \texttt{Int} \mid \texttt{String} \mid \ldots & \textit{(data types)} \\
\mathsf{G} & ::= & \texttt{Int}(g) \mid \texttt{String}(g) \mid \ldots & \textit{(data types with generators)}
\end{array}$$

where I is a finite non-empty set of indexes and for $i \in I$, gen_i is a *generator* assignment $x_{i,1} : \mathsf{G}_{i,1}, \ldots, x_{i,n_i} : \mathsf{G}_{i,n_i}$ (cf. Example 2) and dec is a type assignment $x_{i,1} : \mathsf{T}_{i,1}, \ldots, x_{i,n_i} : \mathsf{T}_{i,n_i})$ with $x_{i,1}, \ldots, x_{i,n_i}$ pairwise distinct.[4]

The communication units of our syntax are the *output* and *input* prefixes, respectively $!\mathit{operationId}$ and $?\mathit{responseCode}$ where *operationId* is the corresponding identifier referring to the specific request in the OpenAPI specification and *responseCode* is the HTTP response code that is returned from the service after the request is made (see § 5). Assertions A are boolean expressions that may refer to the variables occurring in preceding payload descriptions.

We illustrate the usage of COpenAPI with the following example which elaborates on the protocol in Fig. 1b.

Example 2. Fig. 3 shows a testing model written in COpenAPI. The model corresponds to the message-sequence diagram from Fig. 1b. The model specifies that the interaction protocol should first create a customer by invoking `addCust` together with two payload generators:

1. `getApiKey` which retrieves a key to authenticate with the API; and
2. `getCustInfo` to generate the customer information.

The server is then expected to reply with the response code `201` indicating that the customer creation was successful, while also including an identification code of type `String` which is bound with the variable `custId`. Next, the driver chooses between sending `addCard`, `addAddr`, `getCust` or `deleteCust`, including the data required in the respective payloads by either invoking the specified generators, or use previously-bound variables. For instance, in the case where

[4] We use standard data types, including standard types (e.g., `Int`, `String`, ...) and user-defined types. Generators are also user-defined, tailored to the particular request being sent.

```
1 S_shop = !addCust(apiKey: String(getApiKey), c1: Customer(getCustInfo)).
  ↪  ?C201(custId: String).
2   rec X.(
3     +{ !addCard(apiKey, custId, card: Card(getCardInfo)) .
4           ?C201(CardId: String) . X,
5         !addAddr(apiKey, custId, addr: Address(getAddrInfo)) .
6           ?C201(addressId: String) . X,
7         !getCust(apiKey, custId) .
8           ?C200(c2: Customer)<checkCustomer(c1,c2)> . X,
9         !deleteCust(apiKey, custId).?C204() . end})
```

Fig. 3: Example of a COpenAPI model using the OpenAPI specification in Fig. 1a, formalising the message-sequence diagram in Fig. 1b.

`addCard` is selected, the payload must include: *(1)* the API key stored in the variable `apiKey`, *(2)* the specific customer ID stored in the variable `custId`, and *(3)* the card information, by invoking the generator `getCardInfo`.

The model also specifies the use of a user-provided assertion `checkCustomer` (line 5) to check the response sent by the SUT for the response `getCust`. The assertion checks whether the retrieved customer information (stored in $c2$) matches with the information provided when the customer was created when `addCust` was invoked (stored in $c1$). If the assertion succeeds, the test driver loops and performs the choice (line 3) once again. Otherwise, the test run terminates and the test is marked as failed. The recursion repeats until the test driver selects `delCust`, after which the protocol would terminate successfully. □

3.1 Semantics of COpenAPI

We give a denotational semantics of COpenAPI by mapping its model to sets of finite sequences of input/output *events*. Given a output prefix π, $[\![\pi@i]\!]$ denotes the set of values in the codomain of the generator of the i-th parameter of π while for an input prefix π, $[\![\pi@i]\!]$is the set of values inhabiting the type of the i-th parameter of π. We define

$$E = \{\pi(v_1, \dots, v_n) \mid \pi \text{ is a prefix and } \forall 1 \le i \le n : \; v_i \in [\![\pi@i]\!]\}$$

Let M a COpenAPI model and ρ be a map assigning subsets of $E^\star$ to free recursion variables of M. The semantics of M in ρ, is the set $[\![M, \rho]\!]$ defined according to the equations in Fig. 4. The semantics is defined by induction on the structure of the COpenAPI models and returns a set of finite traces; the idea is that this set represents all the executions allowed by a COpenAPI specification. Basically, $[\![M, \rho]\!]$ yields the possible tests that the test driver might use. For example, the internal choice is defined as the set of all possible output prefixes in the choice with the different variations of the values that can be generated from the generators. The case for the external choice is similar, with for all $i \in I, \pi_i$ output prefix, $\boldsymbol{x}_i = x_{i,1}, \dots, x_{i,n_i}$ and $\boldsymbol{v}_i = v_{i,1}, \dots, v_{i,n_i}$

$$[\![+ \{ \pi_i(x_{i,1} : \mathsf{G}_{i,1}, \dots, x_{i,n_i} : \mathsf{G}_{i,n_i}).\mathsf{S}_i \}_{i \in I}]\!] \triangleq$$
$$\bigcup_{i \in I} \{ !\pi_i(\boldsymbol{v}_i).r_i \mid \forall 1 \le j \le n_i : v_{i,j} \in [\![\pi_i @ j]\!] \,:\, r_i \in [\![\mathsf{S}_i[{}^{\boldsymbol{v}_i}/_{\boldsymbol{x}_i}], \rho]\!] \}$$

for all $i \in I, \pi_i$ input prefix, $\boldsymbol{x}_i = x_{i,1}, \dots, x_{i,n_i}$ and $\boldsymbol{v}_i = v_{i,1}, \dots, v_{i,n_i}$

$$[\![\& \{ \pi_i(x_{i,1} : \mathsf{G}_{i,1}, \dots, x_{i,n_i} : \mathsf{G}_{i,n_i}).\mathsf{S}_i \}_{i \in I}]\!] \triangleq$$
$$\bigcup_{i \in I} \{ \pi_i(\boldsymbol{v}_i).r_i \mid \forall 1 \le j \le n_i \,:\, v_{i,j} \in [\![\pi_i @ j]\!], A_i[{}^{\boldsymbol{v}_i}/_{\boldsymbol{x}_i}] \Downarrow \mathsf{tt}, r_i \in [\![\mathsf{S}_i[{}^{\boldsymbol{v}_i}/_{\boldsymbol{x}_i}], \rho]\!] \}$$

$$[\![\mathsf{rec}\ X.\mathsf{S}, \rho]\!] \triangleq \bigcap \{ R \mid [\![\mathsf{S}, \rho[X \mapsto R] \subseteq R]\!] \} \qquad [\![X, \rho]\!] \triangleq \rho(X) \qquad [\![\mathsf{end}, \rho]\!] \triangleq \{\epsilon\}$$

Fig. 4: Semantics for the COpenAPI language.

the only exception that once the values are replaced in assertions of branches should hold. The semantics of end is standard. The map ρ keeps track of the recursive variables in S; initially we assume that M is *closed* (namely) it has no free occurrences of recursion variables. This makes the semantics of recursion standard.

3.2 Implementation

We implement the COpenAPI language as part of a tool called COTS. Implemented in the Scala programming language, takes as input the COpenAPI model and the optional preamble, and generates the Scala source code of an executable *test driver* that interacts with the SUT according to the model. The test driver, in turn, interacts with the REST API exposed by the SUT by using a Scala API which is autogenerated from the provided OpenAPI specification, using OpenAPI Generator.[5]. When the test driver runs, it invokes such Scala API methods to send HTTP requests to the SUT, and to receive and parse its responses; the model determines which requests are sent (and in what order) by the test driver, and which responses are expected. After completing the test runs, the test driver produces the following output:
(1) the level of model coverage achieved by the test runs;
(2) a log file with information about every performed test: the random seed used to generate the test, the sequence of requests/responses and their payloads, and the test outcome (pass/fail);
(3) an offline representation of failed tests as sequences of `curl`[6] commands, which can be executed from a shell to reproduce faults of the SUT without re-executing the test driver.

[5] https://openapi-generator.tech
[6] https://curl.se

Application	**Notes**
FeaturesService [14]	Used in empirical evaluation of [4,24,33,40]
RESTCountries [27]	Used in the empirical evaluation of [40]
GestaoHospital [34]	Used in the empirical evaluation of [40]
LanguageTool [29]	Used in the empirical evaluation of [24,40]
PetClinic [8]	Used in the empirical evaluation of [10]
UsersRegistry [19]	Uses non-trivial API authentication
PetStore [9]	Used in the empirical evaluation of [24]
SockShop [37]	Used in the empirical evaluation of [1,6]
Openverse [38]	Industry app, part of WordPress project

Table 1: Case studies.

4 Evaluation

In this section, we conduct a comprehensive evaluation of our methodology and its practical application in COTS. This assessment is twofold:

1. **Qualitative Analysis**: We aim to determine if the test-models defined by COpenAPI and the test drivers generated by COTS are effective in identifying logic-based faults.
2. **Quantitative Analysis**: We assess the effectiveness of our methodology in terms of code coverage, which is a standard metric for evaluating testing tools.

Given the novelty of our approach in user-written, model-based testing for REST-APIs, we establish our baseline comparison with: *i)* fully-automated REST testing approaches; and *ii)* manually crafted REST API tests.

4.1 Experiment setup

To conduct such evaluation, and obtain the results presented in § 4.2, we follow the preparatory steps illustrated in the rest of this section: we first select the artefacts, build their COpenAPI model, and we determine an adequate number of test runs per application.

Artefact selection. To conduct such an evaluation, we consider a sample of third-party applications listed in Table 1, satisfying the following criteria:

1. they must be open source, to facilitate reproducibility of our results;
2. they must have OpenAPI specifications, needed by COTS;
3. they should be non-trivial and thus representative of real-world RESTful applications;
4. ideally, they should include manually-written tests of their REST APIs, should be amenable to code coverage measurements, and should have been already used for evaluation in previous literature on testing.

All applications in Table 1 are open source, provide OpenAPI specifications, and are non-trivial; moreover, all of them (except *UsersRegistry* and *Openverse*) have been used in previous testing literature. The first 5 applications also satisfy the rest of our "ideal" criteria (item 4 above): they include handcrafted test suites for their REST APIs, and their architecture (consisting of a single Java-based executable) allow us to easily collect and analyse code coverage information using standard tools. Two other applications (*LanguageTool* and *PetStore*) also have similar architecture (allowing us to analyse code coverage), but do not include a test suite for their REST APIs.

Building the COpenAPI Models. As we are not the authors of the SUTs in Table 1, we were required to infer the usage of the SUTs as intended by their developers — in particular, what sequences of operations (request/responses) are valid, and how some requests may depend on others. As mentioned earlier, this information is typically only given informally in the application documentation, it may not be up-to-date, and is often omitted. Therefore, we often inferred this information by examining the pre-existing handcrafted tests, or by studying existing REST API clients. We also enriched our test models with application usage scenarios that, although possible, might have been overlooked in the existing handwritten tests. This way, our models may leverage SUT functionalities beyond those covered by the existing tests, and potentially reveal new faults in the SUTs.

Example 3. FeaturesService was the only application with documentation about some of its operations. For instance, the documentation says:

> *"The application should allow one to add a constraint such that when a feature requires another feature to be active, the latter feature cannot be deactivated without first deactivating the former."*

To test whether *FeaturesService* respects such a description, we formalise the COpenAPI model shown in Fig. 5 (abridged). This model specifies that the test driver should first add two features by invoking `addFeature`; their names are created by random generators and bound in variables *feat1* and *feat2*. Then, the test driver should invoke `addConstraint` specifying that *feat2* requires *feat1*. Finally, the test driver should attempt to delete *feat1* via `delFeature`, expecting the SUT to answer with a `400` "bad request" response. If the SUT was to answer with anything other than a `400`, (such as `200`, indicating that the operation was successful) it would indicate a fault in the logic of the SUT. □

Determining an adequate number of test runs. After writing the COpenAPI model of each SUT and generating its test driver using COTS, we established an adequate number of test runs for each SUT. The optimal number of test runs is the smallest number that maximises *(1)* the number of discovered faults, *(2)* the level of code coverage, and *(3)* the level of model coverage. This number depends on the complexity of the SUT and the COpenAPI model in use. To determine the optimal number, we adopted an incremental approach: gradually increase the number of test runs until the rate of bug discovery and code coverage plateau. This was determined by analysing the logs generated by COTS for each application and verifying that the model was being fully traversed.

```
S_featuresService = rec X.(
    +{ !addFeature(feat1: String(genFeatName)).?C201().
         !addFeature(feat2: String(genFeatName)).?C201().
           !addConstraint(feat1, feat2).?C201().
             !delFeature(feat1).?C400().X, ...})
```

Fig. 5: COpenAPI model for testing *FeaturesServices* (abridged), discussed in Example 3.

4.2 Results

We now present our results, in terms of discovered faults (§ 4.2) and code coverage (§ 4.2).

Discovered Faults Table 2 reports the number of faults discovered by our test models and the COTS-generated test drivers, categorised by the test oracle that detected the fault. We found 25 faults across the 9 selected applications and reported some of the most representative faults to the application maintainers; when we received a reply, in almost all cases the developers acknowledged that the faults were real bugs.[7]

Table 2 classifies the faults detected by COTS into logic-based and systematic categories. Logic-based faults, embedded in the SUT's logic, are discerned through testing specific operational sequences, while systematic faults, inherent in the SUT's overall function, can be identified by testing the requests in isolation of other requests. COTS successfully uncovered 10 logic-based and 15 systematic faults. Detecting the logic-based faults is attributed to COpenAPI models, which incorporate the SUT's domain knowledge, enabling COTS to navigate complex, interdependent request-response sequences. Such intricate sequences are unlikely to be deduced by fully-automated tools (e.g., Example 3), that, as mentioned earlier focus more on systematic faults. Furthermore, as Table 2 demonstrates, COTS effectively detects both fault types.

Some examples of logic-based faults follow:

- In the *PetClinic* model, we included an assertion to verify that a newly created resource can be subsequently retrieved from the SUT. However, COTS discovered a fault: if a resource is updated, subsequent retrieval attempts fail, indicating an issue in the SUT's retrieval operation.
- Another fault in the *PetClinic* application involved inconsistent identification numbers. The SUT issues an ID number when a resource is created, but the list of resources returned shows different IDs than those originally assigned.

[7] For *LanguageTool*, we reported 3 cases of *"500 Internal Server Error"* uncovered by our test model; such errors are considered faults in REST testing literature, but *LanguageTool* produces such errors to signal an unsupported functionality. We are discussing with the developers whether e.g. a *"400 Bad Request"* response code would be more appropriate for this purpose.

- The *FeaturesService* application did not adhere to the model shown in Example 3. It incorrectly permitted the deactivation of *feat1*, which should have been disallowed, and erroneously returned a successful HTTP status code of 204 instead of indicating failure.
- In the *SockShop* application, a significant fault was identified: the application fails to retrieve resources immediately after their creation. This fault was detected by first creating the resource and then attempting to retrieve it via the API sequence modelled via COpenAPI.

Application	***Bad status code***	***Bad response body***	***Assertion fail***	***Total***	***Logical errors***	***Systematic errors***
FeaturesService	6	0	0	6 (1, 0)	2	4
RESTCountries	0	2	0	2 (2, 0)	2	0
GestaoHospital	1	0	0	1 (1, 0)	0	1
LanguageTool	3	0	0	3 (3, 3)	0	3
PetClinic	5	0	2	7 (3, 3)	4	3
UsersRegistry	1	0	0	1 (1, 1)	0	1
PetStore	1	0	0	1 (1, 0)	0	1
SockShop	1	1	0	2 (2, 0)	1	1
Openverse	1	1	0	2 (2, 2)	1	1
Total	17	5	2	25 (16, 6)	10	15

Table 2: Faults discovered by COTS test drivers and oracles, using our COpenAPI models.

Code Coverage Table 3 reports the size of the case studies,[8] and compares COTS model-based testing coverage results against handcrafted REST API test suites (when available as part of the selected case studies), and against fully-automated REST API testing.

The table shows the sizes (in lines of code) of our COpenAPI models against the handwritten REST API test suites. The COTS model sizes include the preambles, whereas the handwritten test sizes exclude their comments. Table 3 also shows the number of test runs performed using COTS and the number of lines covered by 1. COTS; 2. the applications' handwritten REST API tests; and 3. the fully-automated tool Morest [24, 25]. We focus on Morest because a recent study [24] shows Morest to be superior to the other fully-automatic REST API testing tools in literature. Therefore, we selected Morest as representing the state-of-the-art, and included some of its case studies (*FeaturesService*, *LanguageTool* and *Petstore*) in our experiments.

[8] The *LanguageTool* application has 5 API endpoints but only 2 are testable.

Application	***Application LOC***	***#endpoints***	***COpenAPI model size (LOC)***	***Manual tests size (LOC)***	***Number of COTS test runs***	***COTS tests line coverage***	***Manual tests line coverage***	***Morest avg. line coverage [24]***
RESTCountries	2409	22	247	300	100	**1722**	896	
GestaoHospital	4427	20	138	463	50	**2857**	2532	
PetClinic	10,416	35	225	1321	100	3099	**3127**	
UsersRegistry	5452	11	68	246	50	**2035**	1906	
FeaturesService	2026	18	98	377	150	**1626**	1576	360
LanguageTool	18,053	2	75		20	**4999**		935
PetStore	3693	20	111		100	**1987**		763.20
SockShop	3392	15	99		50			
OpenVerse	7117	16	124		1			

Table 3: Size information about the case studies, and comparison between COTS-based testing results (with COpenAPI models developed by us, executed for up to 1 minute with default initial randomness seed), handwritten tests (part of the SUT source code), and Morest (average across 5 repetitions, each requiring 8 hours of execution). The coverage is measured using JaCoCo 0.8.8 (https://www.eclemma.org/jacoco) with standard configuration; JaCoCo is also used in the Morest evaluation [24].

In Table 3 we can observe that COpenAPI models are smaller than the handwritten tests provided by the SUTs — and yet, their code coverage is comparable, and higher for COTS in most cases. In the case of *RestCountries*, the COpenAPI model size is quite close to the handwritten tests size, but the COTS coverage is significantly higher. This suggests that a concise COpenAPI model can replace (part of) a larger handwritten test suite, allowing testers to "*concentrate on a (data) model and generation infrastructure instead of hand-crafting individual test*" [11].

With respect to the fully-automatic testing tool Morest, the benefits of developing a COpenAPI model are evident in the significantly higher line coverage achieved by COTS. In the three common applications we examined, COTS achieved 3 times as much coverage in *PetStore*, 4 times as much in *FeaturesService*, and 5 times as much in *LanguageTool*. Our results were obtained in a few seconds (once the COpenAPI model was developed), whereas the Morest coverage is averaged over 5 executions taking 8 hours each. This suggests that, after the initial investment of time needed to write a COpenAPI model, our model-based approach pays off in terms of coverage achieved and time saved over repeated executions.

4.3 Threats to Validity and Limitations

The time taken to develop COpenAPI models for these case studies varied from one application to another, depending on the complexity of the requests of the applications. On average, we estimate that it took us 30 hours to complete each model — including the time we used to infer the intended usage of the application in question. We believe that a developer or tester more familiar with the application domain and requirements would write equivalent (or better) test models in less time.

Internal The method for determining optimal test runs could lead to imprecisions due to local maxima. Our lack of precise knowledge about expected request/response sequences necessitated designing COpenAPI models based on application tests and source code, potentially introducing bias (Table 3). Additionally, the comparison does not account for the qualitative distinction between creating concrete test cases and developing more abstract COpenAPI models, with the latter possibly being viewed as more challenging.

External Our case studies, though diverse, may not universally represent all application types, limiting generalizability (§ 4.1). We aligned our study with REST API testing discourse, selecting artefacts common in literature to minimize bias and ensure relevance. Despite these efforts, the specific selection could limit applicability, though it was necessary for comparability with existing research.

5 Related Work

In contrast to existing methodologies, our COpenAPI DSL allows for a more nuanced and comprehensive modelling of REST API behaviour. It enables the description of complex sequences and dependencies that are essential for thoroughly testing RESTful services. This approach is a significant departure from traditional models, which typically address simpler scenarios or focus on individual API operations in isolation.

We classify studies on testing of REST APIs into two broad categories: *model-based* and *fully automatic*.

Model-based Testing of REST APIs Our work enriches model-based testing of REST APIs by introducing a sophisticated DSL, COpenAPI, for detailed modeling and testing, diverging from existing approaches by supporting complex test sequences and state dependencies within a single model (§ 3). Unlike Chakrabarti *et al.* [7] and Fertig *et al.* [15], who focus on isolated sequences or operations, our DSL enables multiple test paths and value assertions, surpassing the limitations of singular test sequence models and isolated API testing. Furthermore, while Seijas *et al.* [32] and Pinheiro *et al.* [30] contribute valuable perspectives on property-based and state machine-based modeling, they lack in addressing dynamic data generation and comprehensive evaluation. Francisco *et al.*'s [17] constraints-driven approach and Aichernig *et al.*'s [2] business rule modeling offer insights into precondition and postcondition specification, yet neither adequately tackles the RESTful API stateful interaction complexity our COpenAPI DSL is designed for. Our approach uniquely facilitates capturing intricate dependencies across REST API requests and responses, a critical aspect for thorough testing of sophisticated web services.

Fully Automatic Testing of REST APIs This category includes a variety of testing approaches [13, 31, 33, 42] and tools such as RestTestGen [36], RESTler [5], EvoMaster [4], QuickREST [21], RestCT [39] and Morest [24, 25]. Starting from an API specification like OpenAPI [28], these tools automatically generate tests primarily to check the correctness of individual API responses. They mostly rely on random data generators or generators automatically derived from the specifications, which might not fully capture the intricacies of the API's behaviour. In contrast, in our work, the sequencing of valid requests and responses is explicitly specified in the model, enabling complex interactions with the SUT. This is complemented by our use of non-random, tester-assigned data generators in the COpenAPI model, which allows for a more targeted and effective testing approach. This method provides a stark contrast to AGORA's invariant detection-based approach [3] and the approach in [26] focussing on the specification of inter-parameter dependencies. While these methodologies offer robust solutions in their respective areas, they do not offer the same level of specificity in interaction sequencing and tailored data generation as our COpenAPI model-driven approach does.

6 Conclusion

We introduced COTS, a tool implementing a model-based testing approach for RESTful applications using COpenAPI DSL and OpenAPI specifications, which generates executable test drivers for model-based testing of web services. Our evaluation against third-party open source applications and comparison with existing automated REST API testing tools and manual test suites demonstrates the efficacy of COTS. It achieved comparable or better code coverage and fault detection with smaller, more manageable models, highlighting the potential for reduced effort in test creation and maintenance. Future work includes extending COTS to support other API specifications like GraphQL and gRPC, and enhancing COpenAPI's expressiveness by incorporating additional test oracles and timing constraints.

References

1. Aderaldo, C.M., Mendonça, N.C., Pahl, C., Jamshidi, P.: Benchmark requirements for microservices architecture research. In: ECASE@ICSE. pp. 8–13. IEEE (2017)

2. Aichernig, B.K., Schumi, R.: Property-based testing of web services by deriving properties from business-rule models. Softw. Syst. Model. **18**(2), 889–911 (2019)
3. Alonso, J.C., Segura, S., Ruiz-Cortés, A.: AGORA: automated generation of test oracles for REST apis. In: ISSTA. pp. 1018–1030. ACM (2023)
4. Arcuri, A.: Restful API automated test case generation. In: QRS. pp. 9–20. IEEE (2017). https://doi.org/10.1109/QRS.2017.11
5. Atlidakis, V., Godefroid, P., Polishchuk, M.: Restler: stateful REST API fuzzing. In: ICSE. pp. 748–758. IEEE / ACM (2019)
6. Avritzer, A., Ferme, V., Janes, A., Russo, B., Schulz, H., van Hoorn, A.: A quantitative approach for the assessment of microservice architecture deployment alternatives by automated performance testing. In: ECSA. Lecture Notes in Computer Science, vol. 11048, pp. 159–174. Springer (2018)
7. Chakrabarti, S.K., Kumar, P.: Test-the-rest: An approach to testing restful web-services. In: 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns. pp. 302–308 (2009). https://doi.org/10.1109/ComputationWorld.2009.116
8. community, S.P.: Pet clinic application (2022), https://github.com/spring-petclinic/spring-petclinic-rest
9. community, S.A.: Pet store application (2023), https://github.com/swagger-api/swagger-petstore
10. Corradini, D., Zampieri, A., Pasqua, M., Ceccato, M.: Empirical comparison of black-box test case generation tools for restful apis. CoRR **abs/2108.08196** (2021)
11. Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M.: Model-based testing in practice. In: ICSE. pp. 285–294. ACM (1999)
12. Dezani-Ciancaglini, M., de'Liguoro, U.: Sessions and session types: An overview. In: WS-FM. LNCS, vol. 6194, pp. 1–28. Springer (2009)
13. Ed-Douibi, H., Izquierdo, J.L.C., Cabot, J.: Automatic generation of test cases for REST apis: A specification-based approach. In: EDOC. pp. 181–190. IEEE Computer Society (2018)
14. noz Ferrara, J.M.: Features model microservice (2016), https://github.com/JavierMF/features-service
15. Fertig, T., Braun, P.: Model-driven testing of restful apis. In: WWW (Companion Volume). pp. 1497–1502. ACM (2015)
16. Fielding, R.T., Taylor, R.N.: Architectural Styles and the Design of Network-Based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
17. Francisco, M.A., López, M., Ferreiro, H., Castro, L.M.: Turning web services descriptions into quickcheck models for automatic testing. In: Erlang Workshop. pp. 79–86. ACM (2013)
18. GraphQL: (2023), https://graphql.org
19. Henrique, R.: Users registry application (2022), https://github.com/Throyer/springboot-api-rest-example
20. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016). https://doi.org/10.1145/2873052, https://doi.org/10.1145/2873052
21. Karlsson, S., Causevic, A., Sundmark, D.: Quickrest: Property-based test generation of openapi-described restful apis. In: ICST. pp. 131–141. IEEE (2020)

22. Kim, M., Xin, Q., Sinha, S., Orso, A.: Automated test generation for rest apis: No time to rest yet. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 289–301 (2022)
23. Lenarduzzi, V., Daly, J., Martini, A., Panichella, S., Tamburri, D.A.: Toward a technical debt conceptualization for serverless computing. IEEE Softw. **38**(1), 40–47 (2021)
24. Liu, Y., Li, Y., Deng, G., Liu, Y., Wan, R., Wu, R., Ji, D., Xu, S., Bao, M.: Morest: Model-based restful API testing with execution feedback. In: ICSE. pp. 1406–1417. ACM (2022)
25. Liu, Y., Li, Y., Liu, Y., Wan, R., Wu, R., Liu, Q.: Morest: Industry practice of automatic restful API testing. In: ASE. pp. 138:1–138:5. ACM (2022)
26. Martin-Lopez, A., Segura, S., Müller, C., Ruiz-Cortés, A.: Specification and automated analysis of inter-parameter dependencies in web apis. IEEE Trans. Serv. Comput. **15**(4), 2342–2355 (2022)
27. Matos, A.: REST Countries, https://restcountries.com
28. OpenAPI: (2023), https://www.openapis.org
29. Organisation, L.T.: Language tool, https://github.com/languagetool-org/languagetool
30. Pinheiro, P., Endo, A.T., da Silva Simão, A.: Model-based testing of restful web services using uml protocol state machines (2013)
31. Segura, S., Parejo, J.A., Troya, J., Cortés, A.R.: Metamorphic testing of restful web apis. IEEE Trans. Software Eng. **44**(11), 1083–1099 (2018)
32. Seijas, P.L., Li, H., Thompson, S.J.: Towards property-based testing of restful web services. In: Erlang Workshop. pp. 77–78. ACM (2013)
33. Stallenberg, D.M., Olsthoorn, M., Panichella, A.: Improving test case generation for REST apis through hierarchical clustering. In: ASE. pp. 117–128. IEEE (2021)
34. de Vargas, V.P.: Gestao hospital system, https://github.com/ValchanOficial/GestaoHospital
35. Vasconcelos, V.T.: Fundamentals of session types. Inf. and Comp. **217**, 52–70 (2012). https://doi.org/10.1016/j.ic.2012.05.002
36. Viglianisi, E., Dallago, M., Ceccato, M.: RESTTESTGEN: automated black-box testing of restful apis. In: ICST. pp. 142–152. IEEE (2020)
37. Weaveworks: Sock shop (2021), https://github.com/microservices-demo/microservices-demo
38. Wordpress Foundation: Openverse, https://wordpress.org/openverse
39. Wu, H., Xu, L., Niu, X., Nie, C.: Combinatorial testing of restful apis. In: ICSE. pp. 426–437. ACM (2022)
40. Zhang, M., Arcuri, A.: Open problems in fuzzing RESTful APIs: A comparison of tools. CoRR **abs/2205.05325v2** (2022), https://arxiv.org/abs/2205.05325v2, version 2, published on 7 July 2022
41. Zhang, M., Marculescu, B., Arcuri, A.: Resource-based test case generation for restful web services. In: GECCO. pp. 1426–1434. ACM (2019)
42. Zhang, M., Marculescu, B., Arcuri, A.: Resource and dependency based test case generation for restful web services. Empir. Softw. Eng. **26**(4), 76 (2021)