Exploring API behaviours through generated examples

Software Quality Journal (2024) 32:729–763
https://doi.org/10.1007/s11219-024-09668-2

RESEARCH

Exploring API behaviours through generated examples

Stefan Karlsson[1,2] · John Hughes[3,4] · Robbert Jongeling[2] · Adnan Čaušević[1] · Daniel Sundmark[2]

Accepted: 17 March 2024 / Published online: 22 April 2024

Abstract
Understanding the behaviour of a system's API can be hard. Giving users access to *relevant* examples of how an API behaves has been shown to make this easier for them. In addition, such examples can be used to verify expected behaviour or identify unwanted behaviours. Methods for automatically generating examples have existed for a long time. However, state-of-the-art methods rely on either white-box information, such as source code, or on formal specifications of the system behaviour. But what if you do not have access to either? This may be the case, for example, when interacting with a third-party API. In this paper, we present an approach to automatically generate relevant examples of behaviours of an API, without requiring either source code or a formal specification of behaviour. Evaluation on an industry-grade REST API shows that our method can produce small and relevant examples that can help engineers to understand the system under exploration.

Keywords Property-based testing · Examples · Automated testing · API testing · REST

✉ Stefan Karlsson
stefan.l.karlsson@se.abb.com; stefan.l.karlsson@mdu.se

John Hughes
rjmh@chalmers.se

Robbert Jongeling
robbert.jongeling@mdu.se

Adnan Čaušević
adnan.causevic@se.abb.com

Daniel Sundmark
daniel.sundmark@mdu.se

1 ABB AB, Västerås, Sweden

2 Mälardalen University, Västerås, Sweden

3 Chalmers University of Technology, Gothenburg, Sweden

4 Quviq AB, Gothenburg, Sweden

1 Introduction

Understanding and verifying the behaviour of an API can be a difficult task. One way of alleviating some of this burden is through access to examples of the API's behaviour (McLellan et al., 1998; Novick & Ward, 2006; Nykaza et al., 2002; Robillard, 2009; Robillard & DeLine, 2011; Shull et al., 2000). Generated examples can significantly aid users in understanding the behaviour of an API (Gerdes et al., 2018), and many approaches have been proposed to automatically produce them (Barnaby et al., 2020; Buse and Weimer, 2012; Gu et al., 2019; Holmes et al., 2006; Kim et al., 2009; Mar et al., 2011; Montandon et al., 2013; Moreno et al., 2015). A common theme for these approaches is that they require access to white-box information such as existing usage examples from source code. In contrast, Gerdes et al. (2018) proposed a black-box approach. However, it required not only the implementation itself, but also a formal specification of the code, from which tests could be generated. The formal specification played a key role in selecting examples of "interesting" behaviour.

What if we do not have a specification of behaviours? Can we still generate tests to *explore the behaviour of the system*, and among them select interesting examples from which the user can gain a new understanding of the system's behaviour? Can we automate the behaviour of "playing with the system" to understand parts of what it does? Gaining such understanding of a system is vital for end-users in using an API to successfully interact with the system, testers verifying the actual behaviour of a system, and developers making changes to the system, wanting to ensure the behaviour has been altered correctly.

In our work, we consider this situation and generate examples of the system's behaviour without requiring any white-box information, such as source code, or a formal specification. As illustrated in Fig. 1, an example generating approach, such as we propose, fits in an interactive workflow where understanding of an API is refined over time. As shown in Fig. 1, a user of the approach interacts with the approach by generating examples of behaviours. Through search iterations, the approach interacts with an API to generate the examples. The user of the approach inspects the examples to better understand how the API behaves.

For example generation to be useful, we must consider the following three requirements: (i) generated examples should illustrate interesting and different behaviours, i.e. how different API operations interact (Robillard & DeLine, 2011). Moreover, to avoid overwhelming

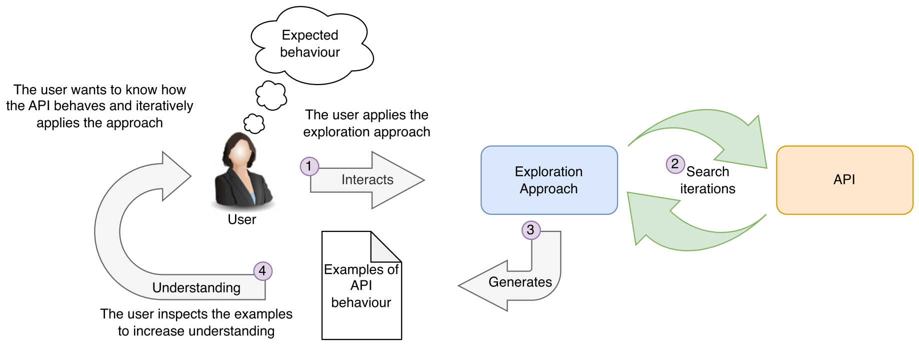
Expected behaviour
The user wants to know how the API behaves and iteratively applies the approach
The user applies the exploration approach
User
1 Interacts
Exploration Approach
2 Search iterations
API
3 Generates
Examples of API behaviour
4 Understanding
The user inspects the examples to increase understanding

Fig. 1 The proposed approach can be used in an interactive exploration process, where a user—based on the output—refines the understanding of an API

the user, (ii) the number of examples should be small and (iii) each example should be minimal (Gerdes et al., 2018; Robillard & DeLine, 2011). Minimal means that all the operations in the example should be essential to its point—no operation can be removed without breaking the example. Indeed, a generated example consisting of a random sequence of API operations would not be helpful in understanding a system's behaviour. Therefore, we focus on generating *relevant* examples.

In this work, we judge an example as relevant if it fulfils the requirements mentioned from prior work in this area (Gerdes et al., 2018; Robillard & DeLine, 2011). Concretely, we judge an example as relevant if it shows an interaction between the state of the system and a sequence of two or more operations in such a way that a distinct behaviour is demonstrated, in accordance with the aforementioned requirements. For example, a sequence of three operations that query the state of the system, create an entity, and query the state again could demonstrate a state-changing sequence in which the state of the system is enlarged—the system stores a larger number of entities. We exclude sequences of only one operation due to the fact that such a sequence cannot produce an example of interaction between the operation and the state of the system. We would have no general behaviour to relate the operation to, since we do not know how it behaves over multiple invocations or in sequence with other operations.

Potential uses of generated examples include helping users and developers understand the behaviour of an API. Users could be, e.g. end-users who want to understand its functionality better, developers integrating with APIs of 3rd-party systems or developers who make changes to the codebase that the API exposes. For the latter type of user, generated examples of the actual behaviour could then show both expected pre-existing behaviours and unexpected new behaviours, whereas an existing test suite would be limited to finding potential regressions in existing behaviours, but would not uncover new, potentially unwanted, behaviours.

Another potential use case is in systems consisting of services with APIs created by different teams, such as in a microservice architecture (Fowler, 2016), where the system is composed of many services (sometimes hundreds). For a developer implementing an API for such services, generated examples can serve as a basis for usage documentation, speeding up the documentation process. In addition, examples of how operations interact could yield documentation in a tutorial fashion, rather than a flat list of API operations. In this way, the generated examples would complement a typical API reference documentation. Example interactions would alleviate the need for the user to have a mental model of the state of the system and how the operations interact when reading the API documentation.

Our approach generates examples as sequences of operations invoking the API. The examples are generated by automatically searching for predefined commonly occurring categories of general behaviour that we expect many systems to display instances of. We call these general behaviours "meta-properties". A set of generated examples of general behaviours can indicate specific behaviours of an API. These indicated specific behaviours emerge from the generated examples. An example of such a specific behaviour is that of a query-operation—an operation that has the general behaviour of not changing the state of the system. Figure 2 shows how these concepts fit together in our proposed approach; meta-properties are used to search for examples of general API behaviours which then indicate specific API behaviours. We assess whether meta-properties hold or not, based on the observations we make of responses from the API when we execute a generated sequence of operations. In essence, a meta-property defines behaviour in terms of how responses from executing a sequence of generated operations relate. Meta-properties are central to our approach as they define the behaviours that our exploration searches for.

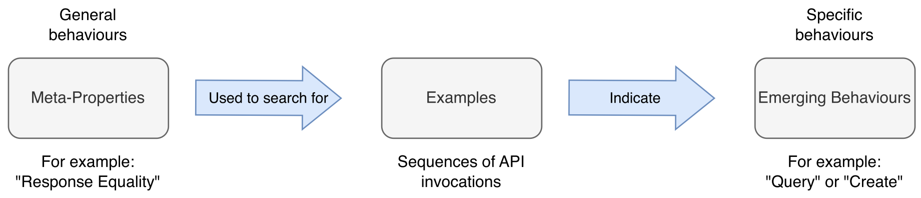

Fig. 2 Meta-properties define general behaviours. Examples are produced from the meta-properties. A set of examples, or the lack of, indicate specific emergent behaviours exposed by an API

Specifically, we make the following contributions:

- An approach to automatically generate *relevant* examples of actual system-under-test behaviours, as exposed by an API, without the need for a formal specification.
- A set of general abstract meta-properties used to categorise behaviours.

The proposed approach is enabled by the key ideas of abstracting the API of the system under exploration and using meta-properties to generate examples, as shown in Fig. 3.

We evaluate the ability of the proposed approach to generate relevant examples using the proposed meta-properties. We do so by applying the approach on GitLab, an industry-grade REST API.

The remainder of this paper is structured as follows: Section 2 describes our approach in detail. Section 3 shows an application of our approach to an illustrative example. Section 4 includes the evaluation of our approach. Section 5 discusses the approach in the context of our evaluation. Section 6 relates our work to the literature. Section 7 concludes the article.

2 Our approach for exploring behaviours

In this section, we explain the details of our approach to generate relevant examples by exploring the behaviours of an API. Before describing the details of specific parts of the process, we present a schematic overview of the components and process of the proposed approach, as shown in Fig. 4.

2.1 Process overview

To explore the behaviour of an API provided by a *System Under Exploration*① , the example generation process must be able to call the operations of the API. To be able to make

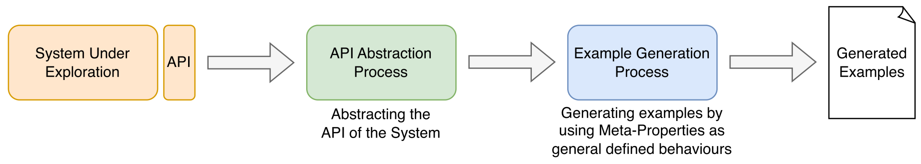
System Under Exploration
API
API Abstraction Process
Abstracting the API of the System
Example Generation Process
Generating examples by using Meta-Properties as general defined behaviours
Generated Examples

Fig. 3 An overview of the key ideas and how they fit in the process of generating examples

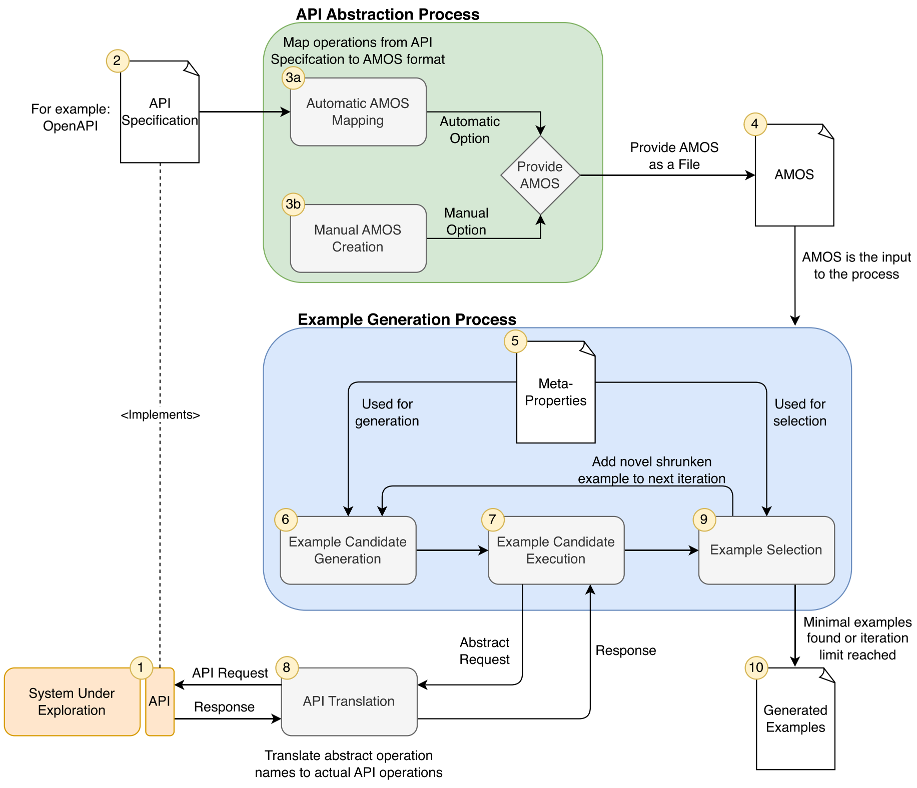

Fig. 4 Approach overview; the *API Abstraction* process produces an AMOS as input to the *Example Generation process*. The *Example Generation process* interacts with an API and produces *Generated Examples*

any calls to an API, we need to know at least the operations it provides and a schema of the required inputs for those operations. Without any of those, we would be forced to send random bits to the API, a process with a low probability of being useful when searching for behaviours. We aim to propose a general approach, potentially targeting any type of API implementation. Therefore, we represent the operations of the API in an abstract format, with no details of how the operations are to be executed. We refer to this format as the *Abstract Method and Operations Specification* (AMOS). The details of the AMOS are further explained in Sect. 2.2.

To automatically generate the AMOS describing the operations of an API, an *API Specification*(2) would be needed. Some examples of existing API specification formats are OpenAPI[1] specification (WebAPIs), a GraphQL[2] schema (WebAPIs), or an Async API[3] specification (message-driven APIs). This kind of specification could potentially be used to generate an AMOS automatically. However, constructing a component of *Automatic AMOS Mapping*(3a) is a one-time engineering effort for the specific type of

1 https://www.openapis.org/
2 https://graphql.org/
3 https://www.asyncapi.com/

API, e.g. once a mapper from OpenAPI specifications to AMOS exists, then any REST API (Fielding, 2000) described using OpenAPI can use it. Alternatively, if no such mapper exists for a type of API, then the AMOS can be created manually(3b), since it is expressed in a human-readable notation. Both of these options can be seen in Fig. 4 in the *API Abstraction* part of the process—the goal of which is to provide an AMOS. The AMOS④ is an input to our approach and only includes the API operations, not how the operations relate, i.e. no behaviour—that is what our API exploration aims to find.

Given an AMOS as input and a set of *Meta-Properties*⑤—descriptions of general API behaviours, detailed in Sect. 2.4—examples are generated by the *Example Generation Process*, further explained in Sect. 2.3. This process generates an example candidate⑥, which is then executed on the API⑦. As the *Example Generation Process* only uses abstract operations—as specified in the AMOS④—the abstract candidate operations must be translated to actual API operations. This translation process is done by an *API Translation*⑧ component. The response of an executed candidate operation sequence is selected⑨ according to the conformance of the candidate to *Meta-Properties*⑤. Example candidate operation sequences, which pass the selection criterion, are included in the *Generated Examples*⑩. In addition, found examples are made available to future iterations of the candidate generation⑥, to avoid repeatedly generating examples which are already known.

The following subsections detail the different parts of the proposed approach.

2.2 Abstracting the system with AMOS

The Abstract Method and Operations Specification (AMOS)④ provides a general format for specifying available operations for the generation of examples. Such a specification is necessary when building a method and tool useful on many different types of SUT such as web-services, user interfaces, and libraries. One of the problems we aim to solve with this work is the burden of specifying SUT behaviour by a human. Thus, it is important to consider the effort of creating the AMOS. We must not move the burden from specifying the SUT itself to specifying the AMOS for the SUT. Since the AMOS only contains operations possible to perform on the SUT, the specification is not as detailed as a behavioural specification of a SUT, hence reducing effort. Since the AMOS does not specify behaviour, the effort of mapping the operations of the SUT to the AMOS can potentially be automated, given that the SUT conforms to a common format. The domain of web services using the OpenAPI specification is an example of where the creation of an AMOS could be automated.

Figure 5 shows the AMOS for an example application used to illustrate our approach, that is introduced in Sect. 3.1. The AMOS is described in extensible data notation (edn).[4] The essential components of the AMOS are as follows:

- Method of invocation - *Which API Translation component*⑧ *should be used to perform the executions?* Line 3 in Fig. 5.
- Operations - *Which operations are available?* Lines 11–22 in Fig. 5.

[4] https://github.com/edn-format/edn

```
 1 {:amos/id   1
 2  :amos/name "Demo App"
 3  :amos/method :clojure-api ;; <= method of invocation
 4  :amos/data-specs
 5  [{:data-spec/id   1
 6    :data-spec/name "Person"
 7    :data-spec/key  :entity/person
 8    :data-spec/spec [:map
 9                     [:person/name string?]
10                     [:person/age [:and int? [:> 0]]]]}]
11  :amos/operations ;; <= a list of available operations
12  [{:operation/id                1
13    :operation/name              "Get Persons"
14    :operation/key               :get-persons}
15   {:operation/id                2
16    :operation/name              "Create Person"
17    :operation/key               :post-person
18    :operation/parameter-spec [:ref :entity/person]} ;; <= param spec
19   {:operation/id                3
20    :operation/name              "Delete Person"
21    :operation/key               :delete-person
22    :operation/parameter-spec string?}]}  ;; <= param spec
```

Fig. 5 AMOS in edn format

- Parameter specification - *What is the shape of required parameters to execute the operations?* Lines 18 and 22 in Fig. 5, where the example on Line 18 refers a specification defined on Lines 5–10.

In addition to the essential components required to execute any successful operations, the AMOS can also serve as a place for further enrichment. The more we learn of a SUT, the more information can be added, either based on a human user judging generated examples or by the tool itself. An example of such additional information is whether an operation is a query operation that leaves the state of the SUT unchanged.

To specify the input parameter schema, we use a format inspired by the *Malli*[5] schema library. The schema allows input data structures to be defined in a system-agnostic way. In addition, this schema provides enough information to allow for the automatic creation of input generators used in the example generation process. The notation does support some simple constraints on inputs (for example, on line 10), but does not support logical dependencies between operations, such as, for example "if parameter X is larger than 0 for operation Y, then parameter Z should be negative". Support for parameter dependencies could be a means for enrichment, when such constraints are realised by generated examples they could be included. However, logical dependencies are left as future work as it is not essential to the proposed approach.

2.3 Example generation process

The exploration of API behaviour is performed automatically based on a set of general API *Meta-Properties*⑤. The basic exploration process is the following: we start in the current state of the SUT; this state is then transformed by executing a sequence of API operations,

[5] https://github.com/metosin/malli

and we observe the effects of the execution. The execution of the sequence transforms the starting state of the SUT into the final state.

We observe changes to the system state in two different ways. The most direct way is to query the state before and after each operation; in this way, state-changes are directly observable. But being able to query the state of the system requires a query operation which tells us the current state of the system, and at the beginning of exploration, we have no such operation available. However, we can observe state-changes indirectly; when the *same* operation is repeated at different points in the execution sequence but returns *different* results, we can infer that the state of the system must have changed between the two.

The *Example Generation Process* generates sequences of operations, in the *Example Candidate Generation*⑥. A property-based testing (Claessen & Hughes, 2000) library is used for the generation of these sequences. Since our prototype was created using the Clojure programming language, we have used the QuickCheck-inspired library test.check.[6] The generated example candidate is executed on the SUT in the *Example Candidate Execution*⑦. The sequence generated depends on the specific *Meta-Property*⑤ used to generate the example candidate. For example, some properties might describe the general behaviour that there should be a difference in the system state after executing a sequence of API operations. Therefore, the candidate generation process needs to generate sequences where the first and last operations are the same, so that a difference in their results can be detected by comparing their responses. We obtain examples by testing the *negation* of the meta-property, *i.e.* that "there exist no sequence of operations which match this condition (defined by the meta-property)". If a counter-example is found, it is a generated operation sequence that does indeed satisfy the meta-property, and this is an example of the kind we seek.

Generated candidate examples that match a *Meta-Property*⑤ in the *Example Selection*⑨ are shrunk to a minimal example by the property-based testing library. The shrinking process re-tests smaller and smaller examples by reducing the number of operations and trying different parameters, searching for a minimal counterexample (*i.e.* a minimal example matching the metaproperty).

Early implementations of QuickCheck did not automatically shrink generated values, but newer versions do (Hughes, 2007). Different implementations of such automatic shrinking strategies exist, such as creating a *rose-tree* (a tree data structure with a variable and unbounded number of branches per node[7]) consisting of simplified values (used in the Clojure implementation of test.check) or with internal reduction of random values (used in the Python implementation of Hypothesis) (MacIver & Donaldson, 2020). This means that "modern" QuickCheck-like implementations provides, in many cases, shrinking "for free". In our approach, we automatically create generators for parameter values based on the AMOS parameter schema. Complex types, such as the "Person" example in Fig. 5, are defined in terms of scalar types. Even if a complex type refers to other complex types in the schema, the leaves of such a tree must be scalar types. The implementation consequence of our approach, regarding parameter shrinking, is that if the implementation uses a modern PBT library, the job of the approach is to translate the parameter schema into basic generators, and then shrinking of parameters—even complex ones—is provided "for free". If a modern PBT library is not available, the approach is still applicable, but the implementor of the approach must now also automatically generate the shrinking strategies. However, the automatically provided shrinking does not always fit the problem domain. In our

[6] https://github.com/clojure/test.check

[7] https://en.wikipedia.org/wiki/Rose_tree

case, the built-in shrinking usually suffices for basic parameter values, but in addition, we need to provide a method to shrink examples.

The built-in shrinking strategy for a list of operations just drops list elements from the list, but such a strategy might result in an invalid example if we drop an operation creating a value that later operations refer to. It is important that shrinking discards such “broken examples”; it is not important though that shrinking preserves the meta-property, since candidate examples that fail to match the meta-property will be discarded anyway during the shrinking search. (However, there is an *opportunity* to define meta-property-specific shrinking strategies, for example strategies which never drop the first or last operation in an example, but while this may speed up the shrinking search, it is not *necessary* to obtain a good result.) The combination of automatically creating generators based on the AMOS, providing a custom shrinking strategy for example sequences, and basing the sequence failing condition on the meta-properties, enable good shrinking results with no requirement of a formal specification of behaviour from a user of the approach.

The example generation process is repeated over several iterations, where the goal of each iteration is to generate an example not previously seen. Thus, all known shrunken examples are input to the next generation iteration. In this way, the process creates a growing list of new examples previously not seen for the given *Meta-Property*. The final set of selected examples are the output of the process, in the form of *Generated Examples*⑩.

The description of the *Example Generation Process* shows how the effort to apply the approach to a specific type of API depends on whether previous engineering work has been done on the components for the specific target type of API—the *Automatic AMOS mapping*③ and the *API Translation*⑧. The *Example Generation Process* of the approach is abstracted from the API and is thus the same, but the API-type-specific components will most likely vary in implementation effort for different types of APIs.

2.3.1 Generating operations

The *Example Candidate Generation*⑥ component generates calls to the API in several different ways, leveraging information about the abstract operations in the AMOS④. We distinguish three different types of abstract operations, based on the way the operation’s input parameters are generated. The first represents operations with randomly generated input. The second represents operations that, as their input, select a value from the parameter of a previous operation. The third represents operations that select as their input a value from a previous operation’s response.

In the exploration of the behaviour of the system, the process generates⑥ and executes⑦ a sequence of operations. Each type of abstract operation parameter generation has its own precondition. The precondition is a predicate that, given the generated execution sequence of operations so far, decides what types of operation parameters can be generated—one of the three types mentioned above. For example, when generating an operation with a parameter which is a reference to a previous operation, then the sequence so far cannot be empty.

Each operation in the generated sequence of abstract operations, selected from the AMOS, is transformed into a concrete operation for execution by the *API Translation*⑧ component. Any references (to other operations’ parameters or responses) can be resolved during this execution since, after execution of the operation referenced, we have the concrete value. It might sound as though creating a *API Translation* component is a lot of work, but, as with the *Automatic AMOS Mapping*(3a) component, mapping specifications to AMOS, it can be done once per API type, as it is a pure translation between the abstract operation and how operations are executed in the given API type. Once a translation is created for REST APIs, for example, then it can be reused on all REST APIs, since all abstract operations would be mapped to HTTP methods.

When the generated sequence of operations has been transformed and executed, then the responses are checked for the specific *Meta-Property* of interest. This corresponds to the *Example Selection*(9) part of the process, in Fig. 4.

2.3.2 Symbolic references

References are a means of enabling the proposed approach to generate operations with a dependency on other values—previous parameters or responses. The context of such references is restricted to one example, i.e. we do not store and refer to values of operations in previous example sequences—only previous values in the current sequence being generated are considered, and new example candidate sequences start with an empty list. Consider the candidate example in Fig. 6: in this example, a sequence of three operations have been generated where a “Person” is first created, all available “Person” entities are then queried, and then a deletion is performed. However, to create a “Person”, the operation parameter must provide a “name” (randomly generated as “a2h6” in this example), but to delete a “Person”, an “id” must be provided. The “id” of a created “Person” is created by the system, not the client. The only way for the client to get the “id” is to perform a query of the “Person” entities in the system. Thus, to successfully delete the created entity, the “id” of the created entity must be provided.

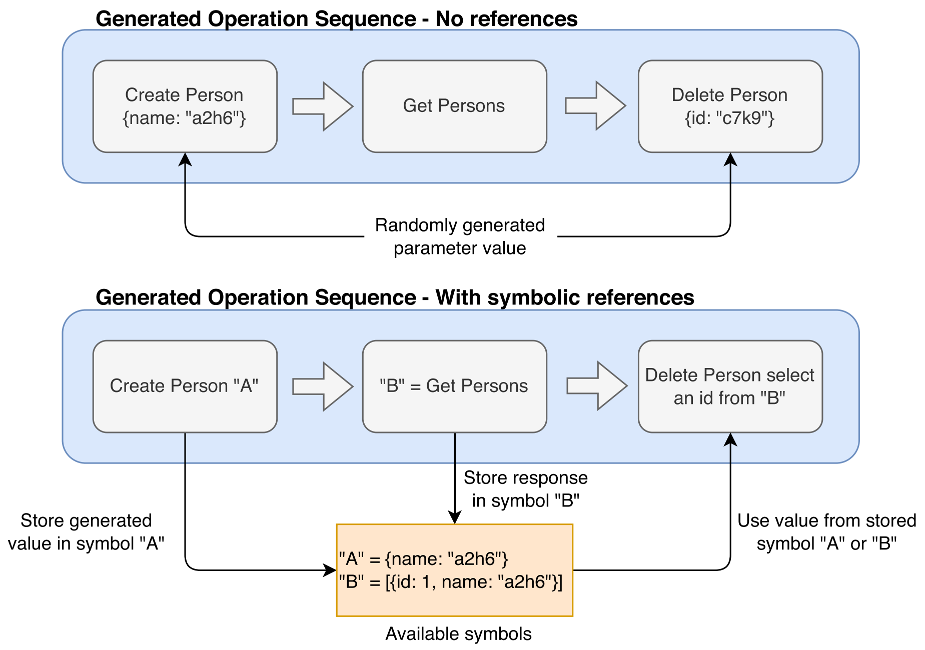
Generated Operation Sequence - No references
Create Person {name: "a2h6"}
Get Persons
Delete Person {id: "c7k9"}
Randomly generated parameter value
Generated Operation Sequence - With symbolic references
Create Person "A"
"B" = Get Persons
Delete Person select an id from "B"
Store generated value in symbol "A"
Store response in symbol "B"
Use value from stored symbol "A" or "B"
"A" = {name: "a2h6"}
"B" = [{id: 1, name: "a2h6"}]
Available symbols

Fig. 6 Two generated example sequences; one with randomly generated parameters and one which uses symbolic references for some parameters

Generating such references can be done in several ways. The simplest way is to randomly generate the input parameters, which might occasionally generate the required referred value—as in the top example sequence using no references in Fig. 6. A somewhat better way is to randomly select, as input to the current operation, any previously seen value in the candidate example—in Fig. 6 this would mean that the parameter of “Delete” would be randomly selected from “A” or ”B”. However, this could often select references that are not valid inputs for the current operation (as in the case of “A”, which does not contain a required “id” for the “Delete”). Finally, the method with the highest probability of selecting a relevant reference is to base the choice on the actual schema of responses and parameters (only “B” would match the schema for “Delete”, since an “id” is required). Schemas might not always match perfectly, rather a schema for one parameter might be a sub-schema of another parameter. For example, in the situation of Fig. 6, “Get Persons” would provide a response schema of a list of “Person” containing the fields “id” and “name”, but the schema of “Delete” only requires an “id”. In this case, we use a simple heuristic to allow for the selection of a matching value. When performing a reference look-up, we search for parameters that match the sought schema, either as an exact match or as a selection of another value. This selection is what makes it possible for the “Delete” operation to select an “id” value from the reference “B” in the example in Fig. 6. The downside of selecting with schema is that the user must provide a schema for responses (remember, we already have the schema for input in the AMOS④, since those are required for any calls to the API to be meaningful). The default reference method that we use is to generate references that match the operation schema. Since a schema for the response of an operation is optional, we fall back to random selection of previously seen values in the example sequence if no schema is available.

Note that we sometimes need the concrete value of a reference, and sometimes its symbolic representation. At execution time⑦, we need to pass the API⑧ a concrete value. However, if we execute the same example several times, then the concrete values in the system may be different each time. Thus, in order to supply the correct concrete value during re-execution, we need to retain references in their symbolic form when we report *Generated Examples*⑩. When re-executing a generated example, the symbols expressed in the example can be substituted for the actual concrete value in the specific execution.

The value of symbolic references to the approach will differ depending on the API being explored. If the responses and parameters of the API never relate, then references are not needed. However, for APIs where operations do relate, it is an essential part of the approach. As described with the example in Fig. 6, the probability of randomly finding a successful example with the correct random values is far lower than referring to prior seen and generated values in the candidate example. We confirmed the value of symbolic references with an evaluation of the effect on one of the state-based meta-properties, described in Appendix B, supporting using schema references as the default method to generate references.

2.4 Meta-properties

In this section, we define a set of *Meta-Properties*⑤ that we use to evaluate our approach by exploring an API. As described, our approach uses meta-properties as definitions of general behaviours. The proposed approach does not depend on any specific kind of meta-properties, but we need some in order to evaluate the approach. Thus, the approach is relevant to many different kinds of APIs, since new meta-properties can be expressed that are relevant to new kinds of APIs. In this paper, we propose a set of exploration meta-properties that are general in the sense that the properties do not make assumptions about the type of the SUT (web-service, user-interface, library, etc.) or any internal (white-box) information, such as source code or usage examples.

In the most general case, observations are formulated without knowledge of what responses from the SUT mean (recall that we do not have any domain knowledge at this point). Hence, observations in the most general case are defined as the difference between the SUT state before and after the execution of an operation in the exploration example candidate sequence or a difference in the response of an operation.

Given the two different possible ways of observing state changes (querying the state of the system, or analysing the response of an operation), there are two categories of meta-properties. It is worth noting that when we start the exploration, *we do not know how to query the state of the SUT!* Hence, we can not use the meta-properties defined in terms of changes to the system state until we have discovered a potential candidate query operation using meta-properties based on operation response changes. The conclusion of this reasoning is that, to be able to find system state-changing operations, we must first identify candidate query operations.

Although the exploration process is general, we need to be specific to show the applicability of the proposed approach for some type of API. Therefore, in this paper, we primarily target APIs that manipulate state, i.e. stateful APIs. Such a behaviour would include *Creating*, *Updating*, *Querying*, and *Deleting* entities stored in the system. We believe that these basic operations are general and cover many stateful APIs. We will now propose a set of general meta-properties. The proposed properties are based on intuitions about how Create, Update, Delete, and Query operations should behave. We do not claim this to be an exhaustive list, but a good starting point for evaluation of the usefulness of the proposed approach.

2.4.1 Meta-properties based on response changes

The set of general properties based on the proposed operation response changes is as follows:

- **MP-R-1: response equality** - *Is the observed response for an execution sequence of the same operation with the same parameters equal?* We would expect a query operation to return the same response if executed multiple times in a row with the same arguments.
- **MP-R-2: response inequality** - *A property of a query-like operation is that there may be a difference in the response, if the operation is executed before and after a sequence of state changing operations.* We expect this property to result in examples showing that when a create operation (for example) is called between two query operations, the responses of the query operations differ.

The main purpose of MP-R-1 and MP-R-2 is to suggest probable query operations. Recall that to formulate properties on parts of the system state, we need to know how to actually query the state.

2.4.2 Meta-properties based on state changes

The probable query operation is the operation used to observe state changes. The responses from the probable query operation, in different positions in a sequence, are used to assess state changes. The proposed set of general properties based on system state changes, given a probable query operation, is the following:

- **MP-S-1: State identity, without observed state change** - *Is there a sequence of operations that, given state S, after execution results in the same state S and* ***no observed*** *state changes took place?* An example generated by this property would only contain non-state-changing operations in the execution sequence, which might indicate query-like operations or operations that sometimes cause state changes but failed. The main difference between this property and MP-R-1 is that in MP-R-1, the sequence consists of the same operation with the same parameters, while in this property, the sequence can contain any operation with any parameters.
- **MP-S-2: State mutation** - *Is there a sequence of operations that, given state S, after execution results in a state not equal to S?* Operation sequences that match this property would include state-changing operations, which could be creations, updates, deletions, or all of them.
- **MP-S-3: State identity, with observed state change** - *Is there a sequence of operations that, given state S, after execution, results in the same state S, but we observed a change in state during execution?* An example generated by this property would contain at least one state-changing operation in the execution sequence. In addition, if we start from an empty state, the sequence would contain both a create operation, resulting in an observed state change, and a delete operation, bringing the state back to the identity state. The example could also contain update operations between the create and the delete operation, updating the state of the created entity before its deletion.

2.4.3 Meta-properties based on state change and size

The final meta-properties are based not only on state-changes, but also on the *size* of the state-change. To measure the size in a SUT-agnostic way, we use the length of the compressed (gzip) response. We compress the response to reduce the weight of the common structure and increase the weight of the novel values.

- **MP-S-4: State mutation, state increase** - *Is there a sequence of operations that, given state S, after execution, results in a state not equal to S, and the size of the state has increased?* Examples generated by this property will contain at least one operation increasing the state, i.e. a create-like operation.
- **MP-S-5: State mutation, state decrease** - *Is there a sequence of operations that, given state S, after execution results in a state not equal to S, and the size of the state has decreased?* This property will generate examples where the execution sequence contains at least one operation reducing the state, i.e. delete-like. For this property to make sense, at least when the starting state is empty, we must first generate and execute at least one operation before the rest of the execution sequence. It is not possible to successfully delete an entity if it has not been created first. All such operations are included in generated examples of this property since they are relevant to the example.

2.5 Reporting examples

The result of the example generation process is that after the given iterations, several examples are produced (unless no new novel examples were found). As we are aiming to increase understanding of the software, it is important not to overwhelm the user with too many examples. We prefer shorter examples to longer ones because we want to show an example of the meta-property with as high a signal-to-noise ratio as possible. We reason that if a shorter sequence can produce a specific behaviour, it is preferable to a longer sequence with the same property.

The reporting and presentation of the examples are important if they are to be of use to a human user. In addition, it is also beneficial for the output to be machine-readable, for any future report extensions. How humans best want to be presented with examples needs further empirical investigation and is beyond the scope of this paper. However, to present the examples in the evaluation of this paper, we have built a proof-of-concept reporter producing a report in a human-readable representation. The creation of reports suitable for a specific type of API remains a manual effort—analogous to how the *API Translation* component must be done once per type of API. For example, for the RESTful type of API, engineers might expect the report to contain URL invocations in some common format for that type, such as CURL.

Figure 7 shows three different ways of reporting the same generated example. First, the data structure that represents it (L2–19), in human-readable edn-format (the same format

```
 1 ;; Example in raw edn-format
 2 [{:operation/key
 3   {:operation.key/value  :post-person
 4    :operation.key/symbol 'a}
 5   :operation/parameter
 6   {:operation.parameter/value  {:person/name "0" :person/age 65}
 7    :operation.parameter/symbol 'b}}
 8  {:operation/key
 9   {:operation.key/value  :get-persons
10    :operation.key/symbol 'c}
11   :operation/parameter  {:operation.parameter/symbol 'd}}
12  {:operation/key
13   {:operation.key/value  :delete-person
14    :operation.key/symbol 'e}
15   :operation/parameter
16   {:operation.parameter/value            "0"
17    :operation.parameter/path             [0 :person/name]
18    :operation.parameter/symbol           'f
19    :operation.parameter/reference-symbol 'c}}]
20
21 ;; Same example in 'humanized' form
22 ["State S = response from the given query operation"
23  "execute :post-person with #:person{:name "0", :age 65}"
24  "a = response from executing :get-persons"
25  "execute :delete-person with 0, selected from [0 :person/name] in a"
26  "when the query operation is executed again, the response is
27   equal to S"]
28
29 ;; Same example as a synthesized Clojure test
30 (deftest state-identity-with-state-change
31   (let [_      (execute-fn :reset)
32         s-pre  (execute-fn :get-persons)
33         _      (execute-fn :post-person #:person{:name "0", :age 65})
34         a      (execute-fn :get-persons)
35         _      (execute-fn :delete-person
36                  (get-in a [0 :person/name]))
37         s-post (execute-fn :get-persons)]
38     (is (= s-pre s-post))))
```

Fig. 7 Reporting

as used in the AMOS). Second, the example is formatted as a list of strings, which is more accessible to a human reader—using our "humanised" reporter. Note the use of a symbolic reference (as explained in Sect. 2.3.2) in L24–25. The use of the symbol in L25 shows both the concrete value for this execution and the path in the value referenced by the symbol. As an example of how a *Generated Example* can be transformed for multiple uses, the final output is in the form of a test case capturing the behaviour of the meta-property used to produce this example. In this case, the usage of symbolic references is essential. If we relied on the concrete value, we might not be able to rerun the test case, depending on the system state. In addition, in the case where a user transforms the *Generated Examples* to test cases, it is important to remember that since we do not require a formal specification of behaviour, we can not judge test cases as correct or not. The test cases show the current behaviour of the system. A manual effort is required to assess if the test cases show faults or nominal behaviour. However, there is potential to automate some of this work. For example, a user of the approach might know that a query-like operation should never appear in a *Response Inequality* example and can automatically control that the output does not contain such examples. Thus, in the cases where the output is a large set of examples, a simple tool can check if the output contains instances of examples that should never be allowed by the system before any test cases are saved for future use.

3 An illustrative exploration

In this section, we apply the proposed approach to a small API to illustrate how the approach works, before going into our evaluation. We limit the exploration to the meta-properties based on responses (MP-R-1 and MP-R-2 from Sect. 2.4.1) and save the use of the other properties for the evaluation in Sect. 4.

3.1 The application

To illustrate our approach, we use a stateful application as an example. The example represents a simple API implementation including the basic operations for managing entities: post, get, and delete. In this illustrative example application, in order to make the implementation as clear as possible, we have made the simplification of using "name" as an identifier of entities. This is not an industry-grade solution, where the server most likely would generate IDs, as in the evaluation in Sect. 4. The application provides the following API operations:

- `post-person` - Adds the provided person to the database, indexed by their name. The initial implementation does not check whether the person to be added already exists in the database or not—it will just be overwritten if it already exists. This lack of input validation is a behaviour for which our method can produce an example and is something we will address when we explore the behaviour of this application in Sect. 3.2.
- `get-persons` - Return the persons in the database or an empty list if the database is empty.

- `delete-person` - Deletes a person with the given name. The initial implementation does not check whether the person to be deleted exists in the database or not.

For the interested reader, a walk-through of the source code of the running example is provided in Appendix A. However, understanding the code is not essential to understand how the proposed approached is applied.

3.2 The exploration

We run through an exploration by interactively applying our proposed approach, as shown in Fig. 1. Starting from the application described in Sect. 3.1, we iteratively update the application based on the generated examples and the realisations they enable.

When generating the examples in this exploration, the state of the system is reset between generations. A reset operation key is sent to the *API Translation* component, as with any other operation, and it is up to the translator to perform the reset of the SUT if possible. Our approach can be used with and without an implemented reset function. However, to keep this exploration simple, we use a reset function and discuss the differences caused by not having one in the evaluation in Sect. 4.

As we described in Sect. 2.4, the meta-properties based on system state changes require a query operation. However, at the start of the exploration, we do not yet know which operations show a query-like behaviour. Therefore, the first step in exploration is to find candidate query operations, but in that process, we might also discover other behaviours. We do so by generating examples conforming to "response equality" (MP-R-1) and "response inequality" (MP-R-2).

To put the exploration into the context of the approach overview in Fig. 4, the input to the *Example Generation Process* is the AMOS④ describing the operations of the API we explore. In this particular case, the AMOS used is shown in Fig. 5. However, its details are not important for the purpose of understanding the exploration results. In addition, the user selects the meta-properties to explore, in this case, as mentioned above, we start with MP-R-1 and MP-R-2. Figure 8 shows the result. For readability, the result is shown in our "humanised" string format. Transforming the output data-format of the generated examples into a richer presentation format, such as PDF, HTML, or similar, is just an engineering effort and not central to understanding the proposed method.

```
;; 1 Example of Response inequality
["Executing :get-persons"
 "then executing :post-person with #:person{:name "", :age 1}"
 "and then executing :get-persons again, results in different
  responses"]

;; 3 Examples of Response equality
["Executing :post-person with #:person{:name "", :age 1} twice
  results in equal responses"]

["Executing :delete-person with an empty string twice
  results in equal responses"]

["Executing :get-persons twice results in equal responses"]
```

Fig. 8 First exploration result of the example application

Looking at the result, we can observe that the `get-persons` operation conforms to the "response-inequality" meta-property (L1–4); it was the only operation doing so, and we can see the shortest shrunken example producing this behaviour. The shrinking process not only shrinks the sequence, it also tries to shrink the parameter values. This is the reason why all example names are empty strings, which is the smallest passing value. The example is very intuitive of how a query operation should behave; the response of the operation is different when an entity has been created in between two executions. This example also gives a strong indication that the `post-person` operation is a state-changing operation. Remember when looking at the examples, names of operations have no meaning to the generation method, we draw no conclusions from them—they could be any random string (but obviously, this would be much harder for humans to understand).

All three operations are found in examples of the "response equality" property. Why is that? The most intuitive example might be for the get-operation. We expect a query operation to give the same response if executed twice in a row with the same parameters. As for the delete- and post-operations, it might not be as clear why these examples are generated. Looking back at the description of the application we explore, in Sect. 3.1, recall that in the post- and delete-operations, it is not checked whether the person is actually in the database. In consequence, the delete-operation tries to remove persons whether or not they exist, with the same response to the client for the same input (which is a reasonable API behaviour), and the post-operation successfully adds the same person twice without any difference in the client result. Is this correct behaviour? We do not attempt to judge this; only the requirements of the application can state whether it is acceptable for the post-operation to add persons with the same name, potentially overwriting a person, or whether this is a bug.

Suppose that we consider the behaviour of the post-operation in this example to be a bug. We can easily fix it by adding a check before adding a person to the database: if the person exists, we return a failure instead. Let us see how this changes the exploration result.

The second exploration produces the result in Fig. 9. What can we learn from this? As intended, two executions of the post-operation with the same parameters now display a response inequality. The first call adds the person, while the second call fails. Also, note that we can no longer find any example of the post-operation for response equality.

This could have been the end of our exploration, but requirements for software constantly change. Suppose that our application should now only allow senior citizens to be

```
;; 2 Examples of Response inequality
["Executing :post-person with #:person{:name "", :age 1} twice
  results in different responses"]

["Executing :get-persons"
 "then executing :post-person with #:person{:name "", :age 1}"
 "and then executing :get-persons again, result in different
  responses"]

;; 2 Examples of Response equality
["Executing :delete-person with an empty string twice results in
  equal responses"]

["Executing :get-persons twice results in equal responses"]
```

Fig. 9 Second exploration result of the example application

```
;; 2 Examples of Response inequality
["Executing :post-person with #:person{:name "0", :age 65} twice
  results in different responses"]

["Executing :get-persons"
 "then executing :post-person with #:person{:name "0", :age 65}"
 "and then executing :get-persons again, result in different
  responses"]

;; 3 Examples of Response equality
["Executing :delete-person with an empty string twice results in
  equal responses"]

["Executing :get-persons twice results in equal responses"]

["Executing :post-person with #:person{:name "", :age 1} twice
  results in equal responses"]
```

Fig. 10 Third exploration result of the example application

added to the database and also that empty names are no longer considered valid. We add the following input validation predicate to the implementation of post-operation:

```
(and (>    (:person/age  person) 64)
     (not= (:person/name person) ""))
```

Once again, we want to explore the consequences of the actual behaviour of this change. The third and final result for this exploration example is shown in Fig. 10.

We can now observe that the post-operation has generated examples for *both* the properties. The inequality property example now shows the behaviour of two consecutive calls with valid input, and the response equality example shows two invalid calls. Note that, due to shrinking, the examples of a post-operation with valid input produce the smallest input passing the boundary condition. We are now satisfied with the behaviour of our application and have concluded the exploration.

It is worth recalling that the generation of these examples did not require any white-box information, such as source code, usage examples, or any formal specification. In summary, with this illustrative exploration, we have shown the flow of working with an example generating method, and the value of examples in conveying an understanding of how an application actually behaves and evolves through application changes.

4 Evaluation

In this section, we evaluate the ability of our approach to produce *relevant* (see definition in Sect. 1) examples without requiring a formal specification of behaviour or other white-box information. To evaluate if our approach can support interactive exploration of an API, we show that it can generate relevant examples to display API behaviours that can lead to further exploration steps of the API's behaviour. To evaluate our approach, we explore a set of API operations of the DevOps platform GitLab.[8] GitLab provides an industry-grade API that can be run locally, making it a suitable evaluation target also in several other case studies of REST API fault-finding methods (Atlidakis et al., 2019; Karlsson et al., 2020; Wu et al., 2022). Moreover, as a DevOps platform, GitLab provides APIs to manage entities such as users, groups, code issues, and code change requests, fitting the context of a stateful API.

4.1 Setup

We ran GitLab on a local installation on developer hardware.[9] An AMOS was automatically created based on a prototype mapper from OpenAPI specifications and then manually complemented where the prototype was lacking. It was created for the Group API operations, GET, POST, and DELETE. We used an *API Translation* component specific to the REST API-type of API. Each meta-property was executed with 100 example tests per iteration and 5 iterations. The research prototype implementation used is available.[10]

4.2 Exploration result

4.2.1 Finding a query candidate

To be able to use any of the meta-properties based on state-change, we first need a query-operation candidate. As in our running example, we use the response-based meta-properties, MP-R-1 and MP-R-2, to try to find a good candidate.

Figure 11 shows the first exploration result. The response-inequality property shows two operations, `:post-groups` and `:get-groups`. Since there is an inequality in the response with the same input and no operation is required between them, the result for `:post-groups` indicates that there is a condition that prevents an entity from being recreated with the same parameters. This is further supported by the fact that `:post-groups` is found to have response equality for calls with empty strings for `name` and `path`.

The other operation with examples of response inequality is `:get-groups`. This example (L8–17) shows a sequence of different operations, resulting in the response of the first and second `:get-groups` being different. A query operation should not change the state by itself. To be a candidate for a query operation, there should be at least one state-changing operation between the first and second calls to the query operation. Furthermore, a query operation should have a response equality without any state-changing operation in between, which is the case for `:get-groups` (L26).

With these first exploration results, we have examples that indicate that `:get-groups` behaves as a query operation should do (response inequality with state-changing operations in between and response equality without). The `:post-groups` operation shows examples of being a state-changing operation. It is found in an example as the middle operation between (what we think are) two query operations and hence must have been the cause of the state-change. In addition, there are examples of response changes with two subsequent calls of this operation, indicating that it must be a type of operation that can change the state.

[8] https://about.gitlab.com/
[9] MacBook Pro, 2.9 GHz Intel Core i9, 16 GB RAM.
[10] https://figshare.com/s/123e5bee7ec2ea893abb

```
 1 ;; Response inequality
 2 ["Executing :post-groups with
 3   {"name" "2U", "path" "00", "description" "64k9",
 4    "membership_lock" false, "visibility" "private",
 5    "share_with_group_lock" true}
 6   twice results in different responses"]
 7
 8 ["Executing :get-groups with {"page" 15}"
 9  "then executing :post-groups with
10   {"name" "1Ujt", "path" "10", "description" "NI9134n",
11    "membership_lock" true, "visibility" "internal"}"
12  "then executing :post-groups with
13   {"name" "Eo", "path" "", "description" "9GD"
14    "visibility" "public", "share_with_group_lock" false}"
15  "then executing :delete-groups with {"id" 0}"
16  "and then executing :get-groups again, result in different
17   responses"]
18
19 ;; Response equality
20 ["Executing :delete-groups with {"id" 0} twice results in
21   equal responses"]
22
23 ["Executing :post-groups with {"name" "", "path" ""} twice results
24   in equal responses"]
25
26 ["Executing :get-groups with {} twice results in equal responses"]
```

Fig. 11 First exploration result of GitLab with no reset

Recall that our approach aims to produce the smallest example of a behaviour. This objective is not achieved in Fig. 11. For example, is the `membership_lock` parameter really needed for `:post-groups`? The GitLab documentation[11] states that it is not. Also, the sequence in the example on lines 8–17, showing `:get-groups` to have response inequality, contains two `:post-groups` operations and one `:delete-groups`. This is clearly not the shortest possible sequence; it should suffice with a get-post-get sequence. Why is shrinking not performing as well as we would expect? The answer lies in the difference between using a soft-reset, and not.

4.2.2 Soft-reset

The process of shrinking examples works best when responses from the system are deterministic. When an example is produced, in the test generation process, the shrinking process will repeat tests with simplified input. This process continues until the minimal input producing the failing test (in our case, an example still behaving according to the property we explore, as described in Sect. 2.3) is found. When behaviour is non-deterministic, the shrinking process may get different results when trying to minimise—a test that failed may now succeed, resulting in a larger than possible shrunk example. In particular, this can happen when state changes induced by one test affect the behaviour of the next.

One way to alleviate this is to use a soft-reset function. As we propose a black-box method, we do not want the reset function to depend on internal knowledge of the application. The

[11] https://docs.gitlab.com/ee/api/groups.html#new-group

```
 1 ;; Response inequality
 2 ["Executing :get-groups with {"page" "2"} twice results in different
 3   responses"]
 4
 5 ["Executing :post-groups with {"name" "0", "path" "00"} twice results
 6   in different responses"]
 7
 8 ;; Response equality
 9 ["Executing :delete-groups with {"id" 0} twice results in
10   equal responses"]
11
12 ["Executing :post-groups with {"name" "", "path" ""} twice results
13   in equal responses"]
14
15 ["Executing :get-groups with {} twice results in equal responses"]
```

Fig. 12 Exploration result of GitLab with reset

implementation of the reset function is injected—thus, external to the core exploration process of our proposed approach—and system dependent, but it can still be implemented in a black-box manner using the external API of the system. To the internals of our approach, this is just an abstract “reset” operation passed to the translation components between the core process of our approach and the SUT. The reset operation is not part of the AMOS, rather it is sent to the translation component before the operations of an example candidate sequence. It is up to the specific translation of the reset operation to choose what to do, if anything. This is what we did in the GitLab case. By querying all the available entities for a specific type, groups in our case, we can then call delete on all groups received. Thus, we do not depend on any white-box information on how GitLab is implemented, although we do need to know how to delete groups.

Figure 12 shows the updated result when exploring with a soft-reset function.

The result is much shorter, but does it tell the same story? The examples related to `:post-groups` (L5–6, L12–13) look similar to those without the reset function, but smaller. However, there is a difference in the result for the `:get-groups` operation. It now has response-inequality with only two calls, which is not expected of a candidate query operation. After some investigation, we discovered that the deletion operation for groups is asynchronous. This means that when groups are deleted in the soft-reset, there is not enough time for them to be fully deleted before the next test starts. The result is that the state is still changing between calls to `:get-groups`, and so the response is different and the example is stored. This illustrates the problems that asynchronous operations can cause for automated API exploration. We fix this by adding a short sleep at the end of our black-box soft-reset, to allow the deletions to complete.

Figure 13 shows the result. These examples are the sharpest so far and also minimal. These three explorations show the trade-off between using a soft-reset or not. The method works without one, but tests run more slowly because the state grows larger, and the resulting examples may not be minimal.

4.2.3 State-change meta-properties

For the state-changing meta-properties, we need to provide a query operation. Based on the results of MP-R-1 and MP-R-2, discussed in the previous sections, the `:get-groups` operation is the one with examples expected of a query operation.

MP-S-1 (state identity, without observed state change), with results in Fig. 14, shows another example expected of the get and post-operation, the get-post-get sequence where the parameters

```
;; Response inequality
["Executing :get-groups with {}"
 "then executing :post-groups with {"name" "0", "path" "00"}"
 "and then executing :get-groups again, result in different
  responses"]

["Executing :post-groups with {"name" "0", "path" "00"} twice results
  in different responses"]

;; Response equality
["Executing :delete-groups with {"id" 0} twice results in equal
  responses"]

["Executing :post-groups with {"name" "", "path" ""} twice results in
  equal responses"]

["Executing :get-groups with {} twice results in equal responses"]
```

Fig. 13 Exploration result of GitLab with reset and sleep

of the post-operation are outside the previously discovered input boundary (empty name and path). The other identity sequences show examples getting and an unsuccessful delete, also resulting in no state change. In summary, the new information we obtained from the exploration is that unsuccessful post and delete operations do *not* change the state.

The state mutation (MP-S-2) property in Fig. 15 shows two executions. The first execution shows the result with our best candidate for a query operation, `:get-groups`, and the other shows the result if we provide `:post-groups` as another query candidate—which a user exploring the system might do. With `:get-groups` as a candidate, we get the expected get-post-get examples (with some non-state-changing operations in some of the sequences), but this was already indicated by MP-R-1, so we get no new information. What may be more disturbing is the result of using `:post-groups` as the query candidate. Lines 25–28 show an example where a first successful post is executed, followed by a non-successful post, and then another post with the same parameters as the first. The second call will fail since, as we have discovered previously, we cannot post the same parameters twice. As this property seeks for changes between the first and last operations with a sequence in between, this

```
;; state identity without state change
["State S = response from the given query operation"
 "execute :delete-groups with {"id" 0}"
 "when the query operation is executed again, the response is equal
  to S"]

["State S = response from the given query operation"
 "execute :post-groups with {"name" "", "path" ""}"
 "when the query operation is executed again, the response is equal
  to S"]

["State S = response from the given query operation"
 "execute :get-groups with {}"
 "when the query operation is executed again, the response is equal
  to S"]
```

Fig. 14 Exploration result of GitLab with MP-S-1

```
;; state mutation with :get-groups as query operation
["State S = response from the given query operation"
 "execute :delete-groups with {"id" 0}"
 "execute :post-groups with {"name" "0", "path" "00"}"
 "when the query op. is executed again, the response is NOT equal
  to S"]

["State S = response from the given query operation"
 "execute :get-groups with {}"
 "execute :post-groups with {"name" "0", "path" "00"}"
 "when the query op. is executed again, the response is NOT equal
  to S"]

["State S = response from the given query operation"
 "execute :post-groups with {"name" "0", "path" "00"}"
 "when the query op. is executed again, the response is NOT equal
  to S"]

;; state mutation with :post-groups as query operation
["State S = response from the given query operation"
 "execute :delete-groups with {"id" 0}"
 "when the query op. is executed again, the response is NOT equal
  to S"]

["State S = response from the given query operation"
 "execute :post-groups with {"name" "", "path" ""}"
 "when the query op. is executed again, the response is NOT equal
  to S"]

["State S = response from the given query operation"
 "execute :get-groups with {}"
 "when the query op. is executed again, the response is NOT equal
  to S"]
```

Fig. 15 Exploration result of GitLab with MP-S-2

example matches that pattern. However, this property is not doing a good job of providing relevant new examples.

Executing MP-S-3 (state identity, with observed state change) in Fig. 16 results in our first example with a response reference. To be able to successfully delete an entity and get back to the identity state, the delete operation must be provided with an existing id. In GitLab, these ids are created on the server side. As can be seen in this generated example, an entity is created with the `post-operation`, then `get-groups` is called with the response stored in the symbol `a`. Finally, `delete-groups` is called, with a value selected from the previous response and stored in the symbol `a`. This sequence has a very high signal-to-noise ratio since it shows the interaction of all three operations.

The last two properties, MP-S-4 and MP-S-5, are also based on a state mutation like MP-S-2, i.e. the state differs between the first and last operations. However, MP-S-4 and MP-S-5 add the additional check that the state should not only be different, but also differ in size.

Examples of MP-S-4 (state mutation with state size increase) are found in Fig. 17, both for `:get-groups` and `:post-groups` as given query candidates. As before, we see the get-post-get sequence, and for examples that must include a `:get-groups` (L13-17) or `:delete-groups` (L7-11) in the sequence, the same pattern appears with the addition of the required extra operation. As we can note in lines 19–20, no sequence was found

```
;; state identity with state change
["State S = response from the given query operation"
 "execute :post-groups with {"name" "0", "path" "00"}"
 "a = response from executing :get-groups"
 "execute :delete-groups with {"id" 41600}, selected from [0 :body 0]
  in a"
 "when the query operation is executed again, the response is equal
  to S"]
```

Fig. 16 Exploration result of GitLab with MP-S-3

that satisfied this property with :`post-groups` as the query operation candidate. These results indicate that MP-S-4 does a good job of verifying query-like behaviour.

MP-S-5, where the size of the state should decrease, will result in the same type of sequences as MP-S-3 (post-get-delete), but also result in the same benefit as MP-S-4; it gives no false positives, in our exploration, given a non-query operation as input.

```
;; state mutation with increase, :get-groups as query operation
["State S = response from the given query operation"
 "execute :post-groups with {"name" "0", "path" "00"}"
 "when the query op. is executed again, the response is NOT equal
  to S"]

["State S = response from the given query operation"
 "execute :delete-groups with {"id" 0}"
 "execute :post-groups with {"name" "0", "path" "00"}"
 "when the query op. is executed again, the response is NOT equal
  to S"]

["State S = response from the given query operation"
 "execute :get-groups with {}"
 "execute :post-groups with {"name" "0", "path" "00"}"
 "when the query op. is executed again, the response is NOT equal
  to S"]

;; state mutation with increase, :post-groups as query operation
"no novel example found"
```

Fig. 17 Exploration result of GitLab with MP-S-4

4.2.4 Exploration summary

The presented evaluation of the proposed meta-properties indicates that the response-based properties are strong in providing minimal, relevant examples of both query-like and create-like behaviour. In addition, these properties correctly show examples of boundaries where operation parameters result in different behaviours. The state-based meta-properties are more varied in their usefulness. To further verify the findings of the response-based properties, our evaluation strongly suggests that not only a change in the state should be checked, but also the *size* of the change.

Relevant examples of query- and create- operations can be found by our approach, using MP-R-1 and MP-R-2, and further verified with state mutation properties which consider sizes, MP-S-4 and MP-S-5. Other proposed meta-properties provide no additional relevant examples and can include false positives.

We have shown that by using general meta-properties we can find relevant short examples, and doing so without the need for a formal specification or access to source code. We achieve this by generating abstract example sequences, with parameters and references, and select and shrink sequences that conform to the sought general behaviour.

We end this section with an experience from our evaluation to further show how we interactively learnt more about the API under exploration. While experimenting on GitLab, without the soft-reset enabled, we got inconsistent results. Some properties we expected to give examples did not. It turned out that, as many industry-grade APIs do, GitLab implements a paging mechanism for its query operations. The results we got were due to the fact that when more than 20 entities existed, only the first 20 were returned, resulting in no apparent state change when an entity had been created. These experiments taught us about the paging mechanism of the API. The solution was to enrich the AMOS with the information that the get operation is a ranged operation. The *API Translation* component for a REST API would then interpret this information and perform multiple calls, returning the complete state to the example generation process. In terms of internals of the method, the get operation was still one operation with one response, only the translation component needed to be adapted.

5 Discussion

In this section, we discuss our approach in the context of our evaluation and how the findings apply to other types of API.

5.1 Supporting interactivity

We think the proposed exploration approach is suitable to be interactive, where a user of the approach gets automation support to explore an API. To be interactive, the speed at which a result is obtained is of high importance. The execution time of an API operation can range from nanoseconds to seconds, depending on if it is an in-process application or library (as the application in Sect. 3) or a network-based service (as in the evaluation in Sect. 4). Additional factors affecting the time taken include (i) the number of API operations, (ii) the complexity of the operation parameters, (iii) the implementation of the API, and (iv) randomness (as our approach uses random-based algorithms).

As an indication, the typical execution time for the generated examples in Sect. 3.2 is in the order of seconds, while in the evaluation in Sect. 4, it ranged from seconds to a few minutes. The main limiting time factor of the approach is the time of executing API operations. The generation of abstract operation sequences and their parameters is typically very fast while executing the generated sequence takes the majority of the time.

Within the scope of this paper, we prioritise an evaluation of the *relevance* of the generated examples over an evaluation of the execution time of the approach. We must first evaluate the ability of our approach to generate relevant examples that can be used in an interactive workflow to learn more about the system and continue the exploration in-depth or broad, by further exploration. Future work includes an evaluation containing a range of different APIs. To have empirical value, we argue that a rigorous evaluation of the execution time needs to contain a range of different APIs and multiple different APIs of each type (i.e. a set of RESTful APIs, a set of File APIs, a set of Collection APIs). Such an evaluation shall then consist of experiments considering evaluations of random-based algorithms (Arcuri & Briand, 2014) that require a large number of executions to be able to draw meaningful conclusions on how different configurations affect the time to generate examples.

5.2 Applicable types of APIs

We propose a general approach applicable to many APIs. The first set of proposed meta-properties are tailored to the general state-based behaviours of Create, Update, Delete, and Query. These meta-properties are applicable to any API with stateful side effects. However, if APIs with no side effects are of interest, more meta-properties based on responses could be defined—the core process of the proposed approach would not change. To make the point of generalisation more clear, we consider some examples.

The “CRUD” style of APIs exposed by RESTful APIs is a perfect example of a stateful API where operations enable us to query the state, observe stateful side-effects, and change the state by creating, updating, and deleting entities.

For another example API, consider the `ls` (list files), `touch` (create a file), and `rm` (remove a file) commands found in Unix-like operating systems. A basic AMOS file with these operations would define the operations and the parameters of the operations. An implementation of the *API Translation* component for this type of API would translate the abstract operation to a Unix-like shell invocation of the command and capture the shell process output as the response of the operation. The proposed set of meta-properties could be used to generate examples such as *State Mutation*, [`ls`, `touch "foo"`, `ls`], this sequence would result in a change of the observed result of the first and last operation. *State Identity* would include examples with `rm`, [`ls`, `touch "foo"`, `ls`, `rm "foo"`, `ls`], the first and last `ls` would have the same response, while we observed a state change with the middle `ls`. We could also consider a “Change” operation in this mix. The operation `chmod` (change file mode) will have a change effect on `ls`, but only if a parameter to include file modes in the output is included. This could generate relevant examples, where `chmod` only is included in sequences where `ls` contain certain parameters, such as `ls -l` (list in long format).

Another type of API where the proposed meta-properties would be applicable is process handling—if we can list processes, start processes, and kill processes, we have a basis for generating examples. It does not matter if these operations are provided by a Java, C#, or C API, as long as there is an observable effect. File APIs (stateful observable effects on files) or collection APIs (stateful observable effects on collections such as hash-maps or lists) are other types of APIs covered by the proposed meta-properties.

APIs where there are no observable effects among a set of operations are not a good fit for the proposed state-based meta-properties. In addition, the list of all possible behaviours is probably infinite; thus, there are types of behaviour relevant to users not included in the first set of proposed meta-properties. What set of uncovered behaviours are not possible to formulate in the proposed approach is out of the scope of this paper.

As long as there is an observable effect and some relation between the API operations, we can apply the proposed state-based meta-properties. APIs consisting of functions without side effects are not going to provide an interesting result. For example, considering an API with mathematical functions, we could apply the meta-properties based on responses, but not the meta-properties based on state. It is also debatable if the response-based meta-properties will lead to any examples considered relevant, although we would be very surprised if we would find an example of [sqrt(4), sqrt(4)] of response inequality.

As described, the proposed approach can refer to and re-use values previously used in an example sequence. For some APIs, a value might require additional transformations before being used as an input to a later operation in the example sequence. If such transformation operations are part of the API, then those would be included in the AMOS and, hence, included in the generation process when generating candidate examples, allowing the required operation to reference the transformed value. However, if such a transformation is not part of the API, then the current implementation of the approach is not applicable. Future work could enable the approach to contain a set of transformation functions, automatically applied to values as trial examples, or the AMOS could allow for custom functions the be added.

6 Related work

Much work has been done in the area of producing examples to help developers understand an API, by different means of automation (Barnaby et al., 2020; Buse & Weimer, 2012; Gerdes et al., 2018; Gu et al., 2019; Holmes et al., 2006; Kim et al., 2009; Mar et al., 2011; Mittal & Pari, 1994; Montandon et al., 2013; Moreno et al., 2015). The most common approach has been to rely on white-box information, a corpus of source code example uses of the API (Barnaby et al., 2020; Buse & Weimer, 2012; Gu et al., 2019; Holmes et al., 2006; Kim et al., 2009; Mar et al., 2011; Montandon et al., 2013; Moreno et al., 2015), which we do not.

In addition to fully automated approaches, Head et al. propose an interactive and iterative approach where the user is involved in selecting code examples from their own code (Head et al., 2018). As demonstrated, we also base our method on an interaction with the user, where knowledge gained from one set of properties are input to the next set of properties. But again, we do not require access to the code of the system.

Instead of only generating usage examples as API interactions, examples in natural language can be produced, based on different kinds of specifications (Burke & Johannisson, 2005; Lavoie et al., 1996; Swartout, 1982). In this work, we make an initial attempt to "humanize" the operation interactions of generated examples, in natural language, while still abstracting away from the details of the SUT. However, how users prefer to be presented with generated examples remains to be investigated.

A different approach, from using existing example usages of an API as a source, is introduced by Mittal et al. In this approach, examples of Lisp syntax are generated, given a Lisp syntax grammar Mittal and Pari (1994). As with our method, Mittal et al. have a strong focus on interesting examples. However, their scope is limited to Lisp, whereas we aim to be more generally applicable, abstracting the SUT.

The most similar work to ours is the method proposed by Gerdes et al. (2018). Gerdes et al. presented heuristics for choosing examples for a stateful API and showed that subjects shown those examples were better able to predict the API's behaviour than subjects shown examples selected to cover all the code. When generating test cases, code coverage is a common metric for selection, but Gerdes at al. showed that different criteria are needed to select relevant examples. As Gerdes et al. do, we also base our method on finding examples in a black-box fashion with the use of a test generation technique with shrinking (property-based testing). However, the major difference in our methods is that while Gerdes et al. require a formal specification of the behaviour of the SUT, we do not. Instead, we only require a specification of available operations and their inputs, with no behavioural information. Our main novelty is the use of a set of general meta-properties to explore and generate examples of the behaviour of the SUT.

7 Conclusions

Automatically generated examples can help developers to understand an API. In addition, exploring the behaviour of an API can reveal unwanted behaviours or validate expected behaviour. In this paper, we have proposed a novel method to generate relevant examples of API behaviours. We do so by using test generation, searching for examples of behaviours. The behaviours are based on meta-properties, abstracting the behaviour of a specific API in a general way. Our evaluation, on an industry-grade REST API, shows the applicability of the method in finding good, relevant, and small examples of behaviour. We also show how different meta-properties provide new knowledge, which can aid developers and users in understanding their systems, without the labour of producing a formal specification.

We identify some areas in which this work can be extended in future work. One such area is to define and assess additional meta-properties, to further explore a system. More studies would be needed to assess how additional general meta-properties could be used on different kinds of systems.

Another area of future work is the reporting of results. As we have mentioned, producing different kinds of reports is an engineering effort. However, to know how users best absorb the information in the report and what they want to be included, and if this is different for different kinds of examples, would require further study. For example, should examples be presented in a more visually appealing manner, or as a report including natural text, or in some domain language relevant to the domain under exploration?

Finally, how users best use, deploy, and integrate a system exploration approach in their workflow is an open question. As previously shown, users who were provided with generated examples were better able to predict the behaviour of the system, but how can we best help the users to interactively explore the system by themselves—extracting as much knowledge about the systems behaviour with as little effort as possible.

We introduced this paper with the question: *Can we generate tests to explore the behaviour of the system without access to a formal specification or source code?* We believe the answer to this is yes. We have presented an approach that does not rely on any inputs except for general behaviours and have shown that this indeed suffices to enable the automatic generation of relevant examples that allow us to explore a system's behaviour.

Appendix A. Code walkthrough

In this appendix, we provide a code walkthrough of the running example application presented in Sect. 3.1. The running example is written in Clojure (Hickey, 2020) (see Fig. 18), and pseudocode of the same example is provided (in Fig. 19), for the reader unfamiliar with Clojure.

The code in Fig. 18 starts by defining the in-memory state, using the `atom` function, named `persons-db`. The `persons-db` is initialised as an empty hash-map, with the `{}` literal.

The `post-person` function uses the `swap!` function to update the current state of the database. The second argument to `swap!` is a function that produces the new value of the database. In this case, we `assoc`iate the current state of the database with a new hash-key and value. The name field of the person is the hash-key, and the value for this hash-key is the person entity given. Note that there is no check of whether a user with a given name already exists—it will just be overwritten if so. This is a behaviour for which our method can produce an example and is something we will address when we explore the behaviour of this application in Sect. 3.

The function `get-persons` returns the values stored in the database. The current state of the database is read with the `@` literal. The database is stored as a hash-map with names as keys and the actual person entities as values. In the case of `get-persons`, we only want to return the entities, not the index keys. Therefore, we use the `vals` function to extract only the values of the hash-map. Finally, we put the values in a `vec`tor. If there is no `seq`uence of values in the database, we return an empty vector.

The last function, `delete-person`, also uses the `swap!` function to update the current state of the database. In this case, we `dissoc`iate the key in the hash-map corresponding to the given name, removing this entry from the database.

```clojure
(def persons-db (atom {}))

(defn post-person
  [db person]
  (swap! db assoc (:person/name person) person))

(defn get-persons
  [db]
  (if (seq @db)
    (vec (vals @db))
    []))

(defn delete-person
  [db name]
  (swap! db dissoc name))
```

Fig. 18 An example application with a simple API

```
persons_db = []

post_person (person p) {
  persons_db[p.name] = p
}

get_persons () {
  if (not-empty persons_db)
    return persons in persons_db
  else
    return []
}

delete_person (string name) {
  delete person_db[name]
}
```

Fig. 19 Pseudo-code of example application with a simple API

Appendix B. Performance of symbolic references

When describing our approach, we have introduced the concept of *Symbolic References* (Sect. 2.3.2). When a value is generated or received as a response from the SUT, we can store these values with a symbolic name. When further operations are generated in a trial example sequence, stored symbols can be referred to, and thus previously seen values can be reused from prior operations in the same example sequence. The first operation in an example sequence starts with random values, as there are no prior values in the sequence to refer to. If an example is successfully generated with the use of symbols, these can be used in the resulting example output, making it more succinct compared to repeating values.

Using this method is in contrast to using random values for all input values of the operation. A reasonable assumption is that references to values already in the sequence, or otherwise observed from a response in the sequence, should outperform random values. But is that true? Using symbolic references increases the cost of the implementation, as it is easier to rely on random values, rather than implementing a mechanism for reference storage and resolutions. Thus, there should be a significant performance improvement in the number of test cases needed in the search for an example to justify the extra implementation effort. In this section, we describe our evaluation of using references over random values in the generated sequences. We perform this experiment in order to make an informed decision on whether to use symbolic references or not in our main evaluation of an industry-grade API in Sect. 4.

B.1 Experiment setup

We used the application previously described in Sect. 3.1, as our controlled application. We also produced a version with the incorporated input validation from Sect. 3, which allows persons to be created only if their age is above 64 and their name is non-empty.

The setup results in four different configurations denoted A-D.

(A) Random parameter generation, no input validation.
(B) Random reference generation, no input validation.
(C) Random parameter generation, input validation.
(D) Random reference generation, input validation.

Table 1 Summary statistics of test case data

Cfg	Min	1st Qu.	Median	Mean	3rd Qu.	Max
A	2.0	811.0	1000.0	839.8	1000.0	1000.0
B	2.0	28.0	62.0	136.5	214.0	1000.0
C	40.0	1000.0	1000.0	998.6	1000.0	1000.0
D	7.0	31.0	55.5	116.5	157.2	836.0

"Random parameter generation" (A and C) means that all input values of generated operations are generated as random values. "Random reference generation" (B and D) means that input values can refer to previously stored symbols, but the symbol to use is selected randomly.

We collected data by running each configuration 1000 times with a maximum budget of 1000 test cases to find an example. In order for references to previous values to be relevant, the meta-property used should be a state-based one. For this experiment we used

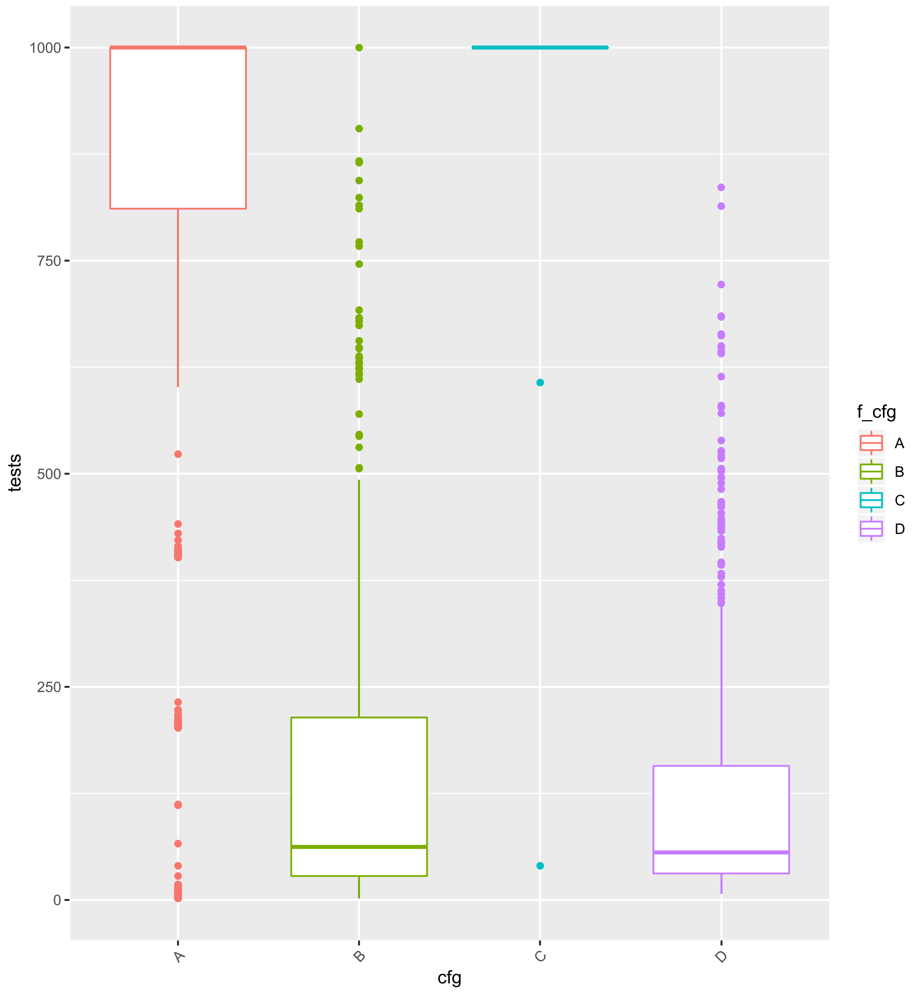

Fig. 20 Box plot of the number of test cases needed for each configuration

Table 2 Wilcoxon-Mann–Whitney test and Varga-Delaney effect size

Cfg	p-value	$\hat{A}_{12}$
A-B	$< 2e^{-16}$	0.92501
B-C	$< 2e^{-16}$	0.00117
A-C	$< 2e^{-16}$	0.36801
B-D	0.5593	0.50754
A-D	$< 2e^{-16}$	0.92925
C-D	$< 2e^{-16}$	0.99935

the meta-property MP-S-3, "State identity, with observed state change". As described in Sect. 2.4.2, successfully generated examples of this meta-property would contain both a create operation and a delete operation. Thus, the delete operation must be performed on the entity previously created in the sequence—making the use of symbolic references relevant. Data were analysed using a pairwise comparison with a *Wilcoxon-Mann–Whitney* test and *Varga-Delaney A measure* to measure the effect size. This setup was chosen considering the guidelines for evaluations of random-based algorithms (Arcuri & Briand, 2014).

B.2 Result

The statistics of the collected data are presented in Table 1. Figure 20 shows the data in a graphical representation, in the form of a box plot. When visualising the data, as in Fig. 20, we can see a strong indication that configurations B and D (both using random reference generation) performed much better than A and C (both using random parameter generation). The visualisation also indicates that there is a big difference in the configurations using random parameter generation when input was validated, A vs. C. However, it is harder to visually see if there is a significant difference in input validation being used or not with configurations using random reference generation, B vs. D.

Table 2 shows the result of the pairwise Wilcoxon-Mann–Whitney test and the effect size. The Varga-Delaney $\hat{A}$ measure is 0.5 if there is no effect of the pairs—they perform best in an equal amount of cases, 0.0 would mean that the first alternative is always better, while 1.0 that the second alternative is always better. We can see significant differences in all configurations except B-D (p-value 0.5593). In addition, there are very strong effect sizes towards B and D when compared to A and C.

The statistical analysis confirms that configurations using random reference generation are significantly better than random parameter generation, with very large effect sizes. In addition, there is also a significant difference between A and C, but not between B and D. This means that input validation has a significant effect on configurations using random parameter generation, but **not** on configurations using random reference generation.

B.3 Discussion

Although the application we used in this experiment is very simple and not industry grade, we see no reason that random input generation would perform better given a more complex application. Indeed, if random input generation cannot perform better in this simple scenario, we consider it unlikely that it would outperform references in an industry-grade scenario.

We can put this result in the context of our main evaluation of an industry-grade REST API (Sect. 4). REST API test generation is time consuming as it is performed at the system level (Zhang et al., 2022). Additionally, finding relations between operations in sequences in the generation of REST API tests is reported as one of the main challenges (Kim et al., 2022). Our results show that the number of tests required to reach a conclusion can be significantly reduced by referring to previously used and observed values in a sequence, which provides further evidence that research on how to effectively find references between operations for REST APIs is worth pursuing.

In summary, random reference generation, using stored symbolic references, requires significantly fewer tests to reach a conclusion than configurations based on parameter generation. In addition, reference generation configurations are robust to whether input validation is used or not. Our analysis gives us strong support for going forward with a random reference generation configuration if no explicit references are provided.

Acknowledgements This work is supported by ABB, the industrial postgraduate school Automation Region Research Academy (ARRAY) funded by The Knowledge Foundation (KKS).

Author contributions All authors contributed to the writing of the main manuscript and the development of the presented approach. The first draft of the manuscript was written by Stefan Karlsson; then, all other coauthors updated and commented on previous versions of the manuscript. All authors reviewed the manuscript and approved the final version.

Funding Open access funding provided by Mälardalen University. This work is funded by ABB, and The Knowledge Foundation (KKS).

Data availability The proof of concept implementation used is available at https://figshare.com/s/123e5bee7ec2ea893abb. Any further data clarifications are available from the corresponding author upon request.

Declarations

Competing interest The authors declare no competing interests.

References

Arcuri, A., & Briand, L. (2014). A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *24*, 219–250. https://doi.org/10.1002/stvr.1486

Atlidakis, V., Godefroid, P., & Polishchuk, M. (2019). RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (pp. 748–758). https://doi.org/10.1109/ICSE.2019.00083

Barnaby, C., Sen, K., Zhang, T., Glassman, E., & Chandra, S. (2020). Exempla Gratis (E.G.): Code examples for free. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* ESEC/FSE 2020 (pp. 1353–1364). https://doi.org/10.1145/3368089.3417052

Burke, D. A., & Johannisson, K. (2005). Translating formal software specifications to natural language. In *Logical Aspects of Computational Linguistics* (pp. 51–66).

Buse, R. P. L., & Weimer, W. (2012). Synthesizing API usage examples. In *2012 34th International Conference on Software Engineering (ICSE)* (pp. 782–792). https://doi.org/10.1109/ICSE.2012.6227140

Claessen, K., & Hughes, J. (2000). QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming* ICFP '00 (pp. 268–279). https://doi.org/10.1145/351240.351266

Fielding, R. (2000). *Architectural styles and the design of network-based software architectures*. Irvine, US: University of California.

Fowler, S. J. (2016). *Production-ready microservices*. O'Reilly.

Gerdes, A., Hughes, J., Smallbone, N., Hanenberg, S., Ivarsson, S., & Wang, M. (2018). Understanding formal specifications through good examples. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang* Erlang 2018 (pp. 13–24). https://doi.org/10.1145/3239332.3242763

Gu, X., Zhang, H., & Kim, S. (2019). CodeKernel: A graph kernel based approach to the selection of API usage examples. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 590–601). https://doi.org/10.1109/ASE.2019.00061

Head, A., Glassman, E. L., Hartmann, B., & Hearst, M. A. (2018). Interactive extraction of examples from existing code. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (pp. 1–12).

Hickey, R. (2020). *A history of Clojure* (vol. 4). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3386321

Holmes, R., Walker, R. J., & Murphy, G. C. (2006). Approximate structural context matching: An approach to recommend relevant examples. *32*, 952–970. https://doi.org/10.1109/TSE.2006.117

Hughes, J. (2007). QuickCheck testing for fun and profit. In M. Hanus (Ed.), *Practical aspects of declarative languages* (pp. 1–32). Berlin, Heidelberg: Springer, Berlin Heidelberg.

Karlsson, S., Čauševiś, A., & Sundmark, D. (2020). QuickREST: Property-based test generation of OpenAPI-described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)* (pp. 131–141). https://doi.org/10.1109/ICST46399.2020.00023

Kim, J., Lee, S., Hwang, S. -W., & Kim, S. (2009). Adding examples into Java documents. In *2009 IEEE/ACM International Conference on Automated Software Engineering* (pp. 540–544). https://doi.org/10.1109/ASE.2009.39

Kim, M., Xin, Q., Sinha, S., & Orso, A. (2022). 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* ISSTA 2022 (pp. 289–301). https://doi.org/10.1145/3533767.3534401

Lavoie, B., Rambow, O., & Reiter, E. (1996). The ModelExplainer. In *Proceedings of the 8th International Workshop on Natural Language Generation* (pp. 9–12).

MacIver, D. R., & Donaldson, A. F. (2020). Test-case reduction via test-case generation: Insights from the hypothesis reducer. In R. Hirschfeld, & T. Pape (Eds.), *34th European Conference on Object-Oriented Programming (ECOOP 2020)* (pp. 13:1–13:27). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Retrieved from: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2020.13, https://doi.org/10.4230/LIPIcs.ECOOP.2020.13

Mar, L. W., Wu, Y. -C., & Jiau, H. C. (2011). Recommending proper API code examples for documentation purpose. In *2011 18th Asia-Pacific Software Engineering Conference* (pp. 331–338). https://doi.org/10.1109/APSEC.2011.18

McLellan, S., Roesler, A., Tempest, J., & Spinuzzi, C. (1998). Building more usable APIs. *15*, 78–86. https://doi.org/10.1109/52.676963

Mittal, V. O., & Paris, C. (1994). Generating examples for use in tutorial explanations: Using a subsumption based classifier. In *In Proceedings of the 11th European Conference on Artificial Intelligence*.

Montandon, J. E., Borges, H., Felix, D., & Valente, M. T. (2013). Documenting APIs with examples: Lessons learned with the APIMiner platform. In *2013 20th Working Conference on Reverse Engineering (WCRE)* (pp. 401–408). https://doi.org/10.1109/WCRE.2013.6671315

Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., & Marcus, A. (2015). How can I use this method? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (vol. 1, pp. 880–890). https://doi.org/10.1109/ICSE.2015.98

Novick, D. G., & Ward, K. (2006). What users say they want in documentation. In *Proceedings of the 24th Annual ACM International Conference on Design of Communication* SIGDOC '06 (pp. 84–91). https://doi.org/10.1145/1166324.1166346

Nykaza, J., Messinger, R., Boehme, F., Norman, C. L., Mace, M., & Gordon, M. (2002). What programmers really want: Results of a needs assessment for SDK documentation. In *Proceedings of the 20th Annual*

International Conference on Computer Documentation SIGDOC '02 (pp. 133–141). https://doi.org/10.1145/584955.584976

Robillard, M. P. (2009). What makes APIs hard to learn? Answers from developers. *26*, 27–34. https://doi.org/10.1109/MS.2009.193

Robillard, M. P., & DeLine, R. (2011). A field study of API learning obstacles. *16*, 703–732. https://doi.org/10.1007/s10664-010-9150-8

Shull, F., Lanubile, F., & Basili, V. (2000). Investigating reading techniques for object-oriented framework learning. *26*, 1101–1118. https://doi.org/10.1109/32.881720

Swartout, W. R. (1982). GIST English Generator. In *In AAAI*, pp. 404–409.

Wu, H., Xu, L., Niu, X., & Nie, C. (2022). Combinatorial testing of RESTful APIs. In *44th International Conference on Software Engineering (ICSE '22)*. https://doi.org/10.1145/3510003.3510151

Zhang, M., Arcuri, A., Li, Y., Xue, K., Wang, Z., Huo, J., & Huang, W. (2022). *Fuzzing microservices in industry: Experience of applying EvoMaster at Meituan*. https://doi.org/10.48550/ARXIV.2208.03988