Specification Based Testing

Nikhila KN

Abstract

In this research, we aim to develop a method for testing RESTful APIs, with a special focus on the basic operations of Create, Read, Update, and Delete (CRUD). Our primary goal is to ensure that APIs function correctly across different situations. To achieve this, we define and test specific preconditions and postconditions — outlining what should happen before, during, and after each API call. These conditions are essential for creating a wide range of quality test cases to thoroughly test the APIs.

By carefully designing these preconditions and postconditions, we can identify logical errors that may occur when different API actions interact with one another. We started by analysing sequence diagrams and existing specifications, which led us to develop a new, formal approach for API testing. This method is designed to rigorously evaluate API operations, find out logical errors, and ensure that the APIs perform reliably in various scenarios.

1 Introduction

In the rapidly evolving landscape of modern software systems, RESTful APIs (Representational State Transfer Application Programming Interfaces) have become fundamental components that enable communication and data exchange between client applications and server-side services. These APIs form the backbone of web services, micro-services architectures, and cloud-based applications, making their reliability and functionality critical to the overall performance of the systems they support. As the complexity of these systems grows, ensuring that RESTful APIs perform their intended tasks correctly under a wide range of conditions is a crucial aspect of software quality assurance.

The correct functioning of RESTful APIs is not merely a matter of convenience but a necessity for the stability, security, and efficiency of the entire software ecosystem. Any failure in these APIs can lead to significant disruptions, ranging from minor glitches in user interfaces to major system outages that can affect millions of users. For instance, an improperly tested API could expose vulnerabilities, leading to potential security breaches, or it might fail under high load conditions, causing performance degradation. Consequently, rigorous testing of these APIs is imperative to ensure that they meet the required standards of functionality, security, and performance.

Traditional API testing methods have played a significant role in verifying the correctness of RESTful APIs. Techniques such as manual testing, automated script-based testing, and the use of sequence diagrams have been employed extensively to identify and rectify issues in API behaviour. Sequence diagrams, for instance, provide a visual representation of the interactions between different components in a system, helping testers to understand the flow of information and identify potential errors. However, while these traditional approaches have their strengths, they also have inherent limitations. Sequence diagrams are often informal and rely on visual representations that can be open to interpretation. This lack of formalisation can lead to ambiguities in understanding the expected behaviour of the system, potentially resulting in incomplete or incorrect test cases. For APIs with intricate business logic or dependencies between operations, sequence diagrams may not adequately capture all possible states or transitions. This can leave critical interactions untested and increase the risk of undetected bugs.

Moreover, newer approaches such as COTS: Connected OpenAPI Test Synthesis for RESTful Applications have been introduced to address some of the shortcomings of traditional methods [1]. COTS leverages the OpenAPI specification, which is a standard format for defining RESTful APIs, to generate test cases automatically. This approach offers the advantage of automation and can potentially increase test coverage by systematically exploring different API endpoints and parameters. However, despite its promise, COTS and similar methods still fall short in several areas. For example, they may not fully account for the dynamic and stateful nature of APIs, where the outcome of one operation can influence subsequent operations in complex ways. Furthermore, they may struggle to capture the intricate dependencies between different API calls, leading to incomplete testing and missed bugs.

Given the limitations of both traditional and newer testing approaches, there is a need for a more robust and comprehensive method that can systematically cover all possible scenarios and interactions within an API. This is where formal specification-based testing comes into play. In this research, we propose a formal specification-based approach to API testing, which aims to overcome the deficiencies of existing methods by providing a more structured and rigorous framework for test case generation.

Formal specification-based testing is grounded in the precise definition of an API's expected behaviour using formal specifications. In the context of API testing, formal specifications define the preconditions that must be met before an API operation can be executed, the operation itself, and the postconditions that describe the expected outcome after the operation is completed. By deriving test cases directly from these formal specifications, we can ensure that the tests are exhaustive and that they systematically cover all relevant scenarios, including those that might be overlooked by other methods.

One of the key advantages of formal specification-based testing is its ability to handle edge cases and complex sequences of operations. Edge cases, which are scenarios that occur at the extreme ends of the operating parameters, are often challenging to identify and test using traditional methods. However, by explicitly defining the expected behavior of the API under these conditions, formal specifications can help to uncover potential issues that might otherwise go undetected. Similarly, by considering sequences of operations, this approach can reveal logical bugs that arise when multiple API calls are made in a specific order, which might not be apparent when testing individual operations in isolation.

In addition to improving test coverage, formal specification-based testing also enhances the transparency and maintainability of the testing process. Since the test cases are derived from formal specifications, they are inherently tied to the documented requirements and expectations of the API, making it easier to trace and justify the tests that are performed. This traceability is particularly valuable in complex systems, where the relationships between different components and operations can be difficult to manage.

2 Method

The core of this approach lies in the formal specification of each CRUD operation. Each operation is specified in terms of preconditions, the API call itself, and postconditions. This formalisation ensures that all potential scenarios are considered and that the API behaves as expected in every case.

2.1 Preconditions

Preconditions are the conditions that must be true before an API operation can be executed. They typically include requirements related to user authentication, authorisation, and the existence or non-existence of the resources being manipulated. Preconditions serve as guards, ensuring that operations are only performed when appropriate.

Example: For the *createDomain* call, the preconditions might specify that the user must be authenticated and have the $ADMIN$ role, and that a domain with the specified name does not already exist.

PRECONDITION:

```
(t -> u in T AND u in U) AND
(ADMIN in u.roles) AND
(exists d in D s.t. d.id = domainId) AND
(not_exists s in SD s.t. s.name = subdomainName)
```

2.2 API call

This is the actual API call, including its inputs and the expected output. The formal specification of the API call details how the API should behave when invoked with specific parameters, assuming the preconditions are met.

Example: The *createDomain* operation might involve calling the API with a *token*, a *programID*, and a *domainname*, and it is expected to return a success code and a *domainID*.

API Call:

```
createSubdomain(t: Token, domainId: string, subdomainName: string)
==> (OK, id: string)
```

2.3 Postconditions

Postconditions define the expected state of the system after the API call has been executed. They ensure that the API call has had the intended effect and that the system remains in a consistent state.

Example: After successfully creating a domain, the postcondition might state that the new domain should be added to the set of existing domains. POSTCONDITION

```
SD' = SD union subDomain{id, subdomainName, domainId}
```

2.4 Abstract Test Case Design

Abstract test cases are designed based on the formal specifications. Each abstract test case describes a generic scenario that can be instantiated with specific values to create concrete test cases. Abstract test cases are a key component of the formal specification-based testing approach, as they ensure that all relevant scenarios are considered.

2.5 Test Case Components

An abstract test case typically consists of the following components:

- Test Case ID: A unique identifier for the test case, providing easy reference and traceability.
- Reset State: A description of the initial state of the system, including any necessary configuration or data preparation required before making the API call. The setup ensures that the system is in a valid state to perform the API call.
- Assumptions: Explicit conditions or assumptions about the system's state or environment that must hold true before the API call is performed. Assumptions set the context for the test, ensuring that the preconditions for the API call are met.
- API Call: The specific steps involved in performing the API call. This includes the parameters passed to the call, as well as any intermediate steps required.
- Assertions: Conditions that are checked after the API call is performed to determine if the test case has passed or failed. Assertions verify that the postconditions defined in the specification are met.

Example Abstract Test Case:

```
----------------------------------------------------------------------
Test case ID: A_createSubdomain_Success

reset
ASSUME(uid -> p in U) AND (ADMIN in u.roles)
(rcode1, t) := login(uid, p)
ASSERT(rcode1 = OK)

ASSUME(exists d in D s.t. d.id = domainId)
ASSUME(not_exists s in SD s.t. s.name = subdomainName)
(rcode2, subdomainId) := createSubdomain(t, domainId, subdomainName)
ASSERT(rcode2 = OK)
----------------------------------------------------------------------
```

- Assumptions:

  The user (*uid*) is authenticated and has the $ADMIN$ role.

  No existing domain in the system has the name *domname*.

- Setup:

  The system is in its initial state (reset), with no domains created.

  The user logs in with valid credentials to obtain a token (t)

- Execution:

  Call $createDomain(t, progid, domname)$ to create a new domain with the name *domname*.

  Call $getDomain(t, domainid)$ using the $domainID$ returned from the *createDomain* call.

- Assertions:

  Verify that the *createDomain* call returns OK and a valid domain ID (*domainid*).

  Verify that the *getDomain* call returns the correct domain data, with $data.id = domainid$ and $data.name = domname$.

Example Concrete Test Case:

```
--------------------------------------------------------------------
Test Case ID: C_createAndGetDomain_Success1 (A_createAndGetDomain_Success)

reset
uid := "admin"
p := "admin@123"
(rcode1, t) := login(uid, p)
ASSERT(rcode1 = OK)

progid := "prog001"
domname := "somedomain"
(rcode2, domainid) := createDomain(t, progid, domname)
ASSERT(rcode2 = OK)

(rcode3, data) := getDomain(t, domainid)
ASSERT((rcode3 = OK) AND (data.id = domainid) AND (data.name = domname))
--------------------------------------------------------------------
```

- Assumptions:

  The user is logged in with the username *admin* and password *admin@123*.

  There is no existing domain with the name *somedomain* in the system.

- Setup:

  The system is reset to its initial state.

  The user logs in with valid credentials to obtain a token (t).

- Execution:

  Call `createDomain(t, ‘‘prog001’’, ‘‘somedomain’’)` to create a new domain with the name somedomain.

  Call `getDomain(t, domainid)` using the *domain ID* returned from the `createDomain` call.

- Assertions:

  Verify that the `createDomain` call returns *OK* and a valid domain ID (*domainid*).

  Verify that the `getDomain` call returns the *domaindata*, with $data.id = domainid$ and $data.name =$ "*somedomain*".

References

[1] C. B. Burlò, A. Francalanza, A. Scalas, and E. Tuosto. Cots: Connected openapi test synthesis for restful applications. In I. Castellani and F. Tiezzi,

editors, *Coordination Models and Languages*, pages 75–92, Cham, 2024. Springer Nature Switzerland.