Model-Driven Integration Testing of Hypermedia Systems

Henry Vu, Tobias Fertig and Peter Braun

PENTASYS AG, Rüdesheimer Str. 9, 80686 Munich, Germany
University of Applied Sciences Würzburg-Schweinfurt, Würzburg, Germany
E-mail: henry.vu@pentasys.de; tobias.fertig@fhws.de;
peter.braun@fhws.de

Received 12 January 2019;
Accepted 04 June 2019

Abstract

The proper design of Representational State Transfer (REST) APIs is not trivial because developers have to deal with a flood of recommendations and best practices, especially the proper application of the hypermedia constraint requires some decent experience. Furthermore, testing RESTful APIs is a missing topic within literature. Especially hypermedia testing is not mentioned at all. Manual hypermedia testing is time-consuming and hard to maintain. Testing a hypermedia API requires many test cases that have similar structure, especially when different user roles and error cases are considered. In order to tackle this problem, we proposed a Model-Driven Testing (MDT) approach for hypermedia systems using the metamodel within our existing Model Driven Software Development (MDSD) approach. This work discusses challenges and results of hypermedia testing for RESTful APIs using MDT techniques that were discovered within our research. MDT allows white-box testing, hence covering complete program structure and behavior of the generated application. By doing this, we are able to achieve a high automated test coverage. Moreover, any

runtime behavior deviated from the metamodel reveals bugs within the generators.

Keywords: REST, Integration Testing, RESTful API, Hypermedia Testing, MDSD, MDE, MDT, Model-Driven Testing.

1 Introduction

The Web has become the de facto deployment environment for software systems and applications. Office productivity applications and corporate tools such as invoicing, purchasing and expense reporting systems have migrated to the Web [19]. Web APIs have become the vital backbones for these applications and services. While the number of Web APIs is increasing, the need for good API design has become more crucial than ever before. Good APIs can be among a company's greatest assets, as customers invest heavily in buying, writing and learning them. However, bad APIs can also be among a company's greatest liabilities as they result in never-ending streams of maintenance and support [2].

1.1 Representational State Transfer

In 2000 Fielding [8] presented an architectural design approach for building networkbased applications that resemble the Web through principled use of architectural constraints called Representational State Transfer (REST). In its simplest form, REST requires that an application server must adhere to a set of constraints, such as client-server, stateless, cache, uniform interface, layered system and hypermedia. But many APIs that claim to be RESTful often neglect the hypermedia aspect. Fielding also criticizes in his blog [7] that many existing APIs are called RESTful even though they lack hypermedia controls. This circumstance leads to a widely-held misconception of RESTful API design among the developer community, subsequently there is a lack of existing RESTful APIs that adhere to the hypermedia constraint. Since hypermedia is not present there is no awareness for testing it. An API defined as RESTful is fully matured when making use of every Fielding's constraint including hypermedia [15]. Hypermedia constraint has tree jobs according to [14]:

1) It tells the client how to construct an HTTP request, what method to use, what URL to use, what HTTP headers and/or entity-body to send. 2) It makes promises about the HTTP response, suggesting the status code, the HTTP headers, and/or the data the server is likely to send. 3) It guides the client through the application workflow given by the server, and hence complying to the stateless constraint. An API that do not generate any hyperlink response to navigate a client through its application workflow has two major problems: a) For application states do not generate any follow up hyperlinks, clients are forced to construct these hyperlinks piece by piece which would require prior knowledge about the implementation details of the server and b) since there are no hyperlinks for clients to follow, there is no application workflow, thus it is rather a static API. Besides the lack of a workflow, this described client-server architecture is tightly coupled and is likely to break due to changes on either side: If the server change the URIs, the clients will break and if any client is to be modified, the server must remain the same.

1.2 Model-Driven Software Development of RESTful APIs

We decided to tackle the challenges of RESTful API development with a MDSD approach. In 2015 we proposed *Generating Mobile Applications with RESTful Architecture*(GeMARA) [16]. Instead of a data-first approach which uses data models for creating an API, we went for an API-first approach which aims at a proper API design first which then automatically creates the underlaying database. The main idea of this project is to take RESTful API development to a higher level of abstraction by using a metamodel as input. We use our own Domain Specific Language (DSL) to describe a metamodel which is then to be translated into a RESTful API. This way, we can force consistency and achieve a higher standard of quality by encapsulating reliable and wellknown libraries, frameworks and RESTful best practices behind our DSL.

1.3 Model-Driven Testing

As this project matures, we also explored the possibility of Model-Driven Testing (MDT) [5] and realized the lack of information about quality assurance for MDT and for our domain regarding RESTful systems. In general, MDSD processes are very sensitive to the introduction of defects. Any defect in a model or a model transformation can be easily propagated to the subsequent stages, thus causing the production of faulty software [11]. However, MDT is a possible way to achieve correctness the generators. Test cases can be generated from the underlying model. Any deviating behavior of the application on a runtime environment could reveal possible bugs within the generators. Moreover, even third-party frameworks and libraries can be updated or replaced over time, these generated test cases can also be used to verify our platform code. In 2017 we proposed several approaches to deal with MDT for RESTful systems which include server-side testing and client-side testing [20]. The server-side testing is separated into two phases: static and dynamic analysis. The main goal of the static analysis is to reveal errors at the highest level of abstraction, guaranteeing that a model must be designed correctly before triggering any source code transformation. Our findings within the static analysis were presented in [21] covering up an automatic verification process for the input-model, thus ensuring its hypermedia characteristic as a Finite-State Machine (ε-NFA).

This paper is a follow-up contribution within our research regarding testing RESTful APIs using MDT techniques from [20] and [21]. Facing the need for better RESTful API design and deficits in its quality assurance, we are focusing on answering the following research questions (RQ):

RQ 1) How to generate appropriate test cases from an existing meta-model to test role-based behavior for every application state within a RESTful API?

RQ 2) How can test cases deviated from RQ 1 be verified automatically on runtime?

RQ 3) Can the proposed approach completely relieve API developers from integration testing?

We tackle this problem domain by using design science to produce artifacts. In our case, these artifacts are models and prototypes which can be used to help us to gain better grasp of the problem and to re-evaluate it.

2 Related Work

The established literature concerning REST such as [1], [14] and [22] reveal little to no information about its quality assurance. Moreover, hypermedia testing is not mentioned at all. They explain what hypermedia is good for, but do not present any approach to test it. REST API integration testing by sending HTTP-requests and verifying the received responses was mentioned in [22].

In [10], the author suggests three different "entry levels" for integration testing: 1) At HTML level using a WebDriver tool such as Selenium [13] to interact with HTML/CSS elements such as filling a form or clicking on a button 2) At HTTP level by sending HTTP requests and checking HTTP responses 3) At controller level by directly testing the methods. The main idea of this book is testing must be efficient and economic, thus recommending against a high test coverage and complex logic. However, our MDT approach enables automation of integration testing with a high test coverage with minimum manual effort, hence generating great economic return. Nevertheless, the author also neglects the hypermedia by suggesting to manually craft each HTTP request with a fix URL.

In [3], the authors present their own framework Test-the-Rest to test HTTP based web services. The test cases are written in a test specification language based on XML to give the tester a structured approach. Other than that, response validation only depends on checking media type and status code. This approach does not fully address the challenges of testing RESTful APIs.

To the best of our knowledge, there is limited information about testing RESTful systems. Especially, the hypermedia constraint is often praised as the key feature of a RESTful system but testing it is not mentioned at all. As opposed to existing literature, the aim of this work is to fill in this gap by providing guidance and methodology for testing hypermedia.

3 Challenges

Building distributed hypermedia systems using REST requires some decent experience and knowledge, particularly in the design phase.

REST has become a major paradigm, but unfortunately it means different things to different people. Some call it a standard, others call it a specification, while REST purists and its creator Fielding consider it as an architectural style. So, in order to fix these misconceptions, we have carefully analyzed Fielding's dissertation [8] to derive REST key components for our model from our experience in implementing several RESTful APIs over the last years. The focus lies in the development of a metamodel that has the ability to describe key components of REST and to form their relationships via hypermedia in a convenient way while considering all other REST constraints.

The most important component in our metamodel to describe RESTful API is the application state. An application state is a pair of a HTTP method and a resource. It represents a valid REST request to access a resource. Figure 1 shows a simplified class diagram that expresses our application state concept. Also one of the central key elements in a RESTful system is the resource. It is important to note that resource is not a storage object but is, instead, a conceptual entity. A resource represents a single object or a collection of objects. The intention of HTTP verbs should be fixed, and developers should not be free to choose wrong verbs. The four basic operations to create, retrieve, update and delete (CRUD) resources are mapped to the four HTTP verbs POST, GET, PUT and DELETE. This mapping is unambiguous regarding to the HTTP specification [9].

A transition is our formal way of modeling relationships between application states. A client can navigate from one application state to another via transitions. Technically speaking, a transition is comprised by a hyperlink, a media type and a relation type. A relation type is

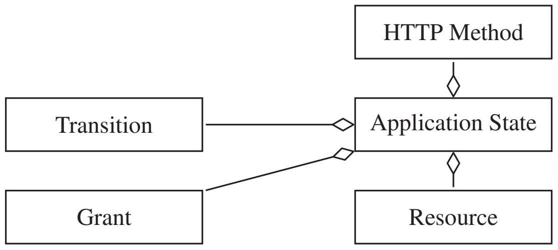

Figure 1 UML class diagram for application state.

akin to the `rel` attribute of HTML link tags. Our application state can be protected from unauthorized access by adding specific grant feature to deal with authorization e.g., role-based access control and authentication with HTTP Basic or OAuth.

Our MDSD approach has two distinct artifacts that need to be tested on the serverside: the model and the generated code from the model. These artifacts are to be tested in a bottom up manner because they are built on top of each other. Errors within the model can propagate subsequent bugs in the generated source code which leads to undesired system behavior at runtime or production. Therefore, our Model-Driven hypermedia testing on the server-side is divided into two sequential steps: static and dynamic analysis.

MDSD is a software engineering paradigm that promotes the utilization of models as primary artifacts in all software engineering activities [11], therefore, we have to ensure that there are no defects in our model in the first place. The static analysis verifies our model and a model is verified when its design represents a ε-NFA. In theory, a static analysis requires no code execution which is comparable with a code review or a Unified Modeling Language (UML) model check [12]. However, we consider using our MDSD tool to solve this problem in a sophisticated way. Since our metamodel already provides concrete information about every application state and its possible transitions, we are able to perform an automatic verification process. On the implementation level, our metamodel is comprised of Java objects. This model includes all necessary information for the generators to transform it into a RESTful API. The ε-NFA check process takes advantage of this and is able to extract all required information from the model to carry out its verification without any additional external libraries [21]. The transformation process can start if the model is correct-meaning every application state has at least one incoming and one outgoing transition. Whenever there is a missing incoming or outgoing transition, it would throw an appropriate exception to make the API designer aware of this. The goal of this ε-NFA check is to identify any error at the highest level of abstraction before triggering any source code transformation.

Dynamic analysis, as opposed to static analysis, always requires the execution of the software. The simplest form of dynamic testing is the execution of the software by a test person, thereby the tester can enter any input to operate the software. This is an unsystematic ad-hoc approach and thus inaccurate and usually not reproducible [12]. However, using MDT techniques allows us to follow a more novel approach. Once the static analysis is successfully carried out, our tool will generate a functional server. This generated server should work correctly and include all the features described by its model. But since the generators are manually implemented, we cannot guarantee this. Besides that, the platform code in our MDSD approach consists of third-party components and these components will eventually be updated or replaced over time. Therefore, we also have to check whether these changes affect the transformation process. Due to these reasons, we must carry out a dynamic analysis to test the generated code on a runtime environment. The aim of this procedure is to test the correct functionality of the transformation process, thus detecting bugs within the generators. The dynamic analysis can be used to test several aspects of a RESTful API: 1) It checks whether a ε-NFA-compliant model correctly produces a ε-NFA-compliant RESTful API 2) We also have to consider the authorization concept of the application. 3)Additionally, we can also provoke negative tests to see how the server handles error cases on runtime.

The main goal of the dynamic analysis is to automate hypermedia test case generation using our existing Model-Driven approach. The generated test cases must guarantee the correct hypermedia behavior of the actual generated RESTful API as designed in the model. Hypermedia behavior concerning role-based accessibility of application states, response validation and negative testing will also be tested. The dynamic analysis is a rather more complex task, thus it must be divided into three phases: First, model crawling phase, second, building an HTTP crawler, and third, generation of test classes. Each of this phase has its own sub-goals, these sub-goals are combined together to achieve the main goal presented above.

To simplify understanding, it is necessary to present an application example which will be the basis to demonstrate our further approaches. Say, we want to build an online shop that sells items. There are two user roles: customer and admin. A customer can view items whereas a shop

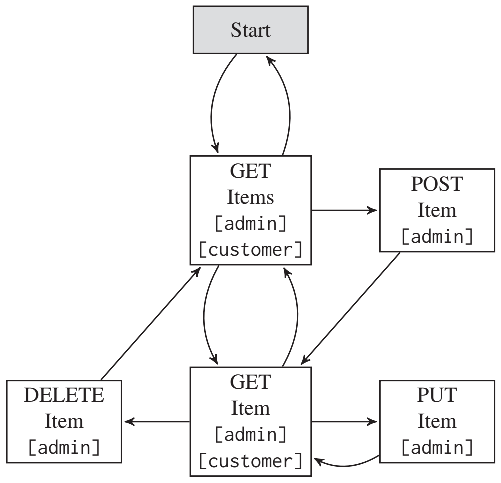

Figure 2 Workflow of the application example.

admin can also create, update or delete items. For the sake of clarity, we omitted transitions back to the dispatcher state and self-pointing transitions. The entire application workflow with all application states and possible transitions is described as an ε-NFA in the Figure 2.

3.1 ε-NFA Check

The concept of ε-NFAs revolves around a finite set of states with a particular initial state, a set of possible inputs and rules to map each state to another state, or to itself, for any of the possible inputs [23]. Speaking in REST term, a client can enter the application workflow through the initial state and only be in exact one state at any given time and it can only change its current state by navigating through a directed transition with a valid input. ε-NFA compliance means that every state within an application is accessible. The client should be able to start from start state which is the entry point of the RESTful API, and it should be able to visit every application state and go back to the start state without getting stuck in a dead-end. We argue that within a RESTful application there are no accepting states. An accepting state assumes that a client has finished a task within the application workflow after a sequence of inputs. The question arises "When is a task finished?". It is hard to say when a client finishes a task and want to leave the REST application. Taking Facebook for example: Entering the Facebook app gives us our newsfeed. This could be argued as a task done or an accepting state, but navigate to a friend's page could also be one and so is creating, modifying or deleting a post. It does not matter if the client wants to stay in the loop of the application workflow or leave it at any given state. Due to these argumentations, we can perform this simplification that our ε-NFA model of a RESTful API does not have accepting states.

Responses of application states must contain a finite set of hyperlinks through which the user or automaton can obtain choices and select actions. This means that we have to check whether responses of each application state provide a correct finite set of hyperlinks at runtime. Since each application state represents a valid REST request in our model, we have to send request to every available application state in order to get the corresponding response. A generated RESTful API is considered ε-NFA compliant when a client can navigate through the application workflow visiting every states without getting stuck in a dead-end.

3.2 Role-based Authorization

Role-based access control is a common concept in RESTful APIs. Another challenge of hypermedia testing is to check whether the generated RESTful API is delivering appropriate hyperlinks based on the client's active user role. Each role receives a different representation of the API depending on its access rights. For instance, a customer is only allowed to view a certain resource, whereby an administrator is allowed to create, modify or delete this resource.

Each role is only permitted to see its designated hyperlinks to navigate or operate through the application. In other words, the API must guide the client through the workflow based on its current role, otherwise the authorization concept would be corrupt. If we were to write test cases for this application manually, we would have to write two similar test cases to validate whether both roles (customer and admin) can view items. This would require enormous effort if implemented manually for a system with a dozen roles and hundreds of application states. Figure 2 emphasizes this problem: Each role has a different workflow on the same application, but they also share a common set of allowed actions.

3.3 Error Handling

Until now, we have looked at several challenges to check correct hypermedia-driven behavior. This falls under the category of positive testing because our model allows white-box testing, hence revealing complete program structure and behavior. Therefore, we can perform a sequence of valid inputs and check for expected outcomes. But what if the client makes invalid requests? We must also test the robustness of the system by checking its ability of handling error cases. The correct response for an invalid request within a hypermedia-driven system should contain at least a self-descriptive message [8] to tell a client what to do next.

The dynamic analysis can also be used to provoke negative tests. Instead of verifying hypermedia behavior, we can test how the server reacts to a client's non-hypermedia behavior. Since dynamic testing allows runtime execution of the to-be-tested application, a client can intentionally send false requests to see how the server handles to error cases: Whether it remains functional or breaks, or whether an appropriate response is given in the event of an unauthorized request.

Each state-to-state transition requires specific inputs in order to perform. For instance, if a client wants to enter the *POST Item* state its request header must include authorization with credentials of an admin and its request body must include a proper `Item` resource representation (e.g. JSON) as payload. The server has to validate two input types and handle several error cases:

1. Authorization header: If the client is not authenticated to the server, the server must response with a proper HTTP code such as `401 Unauthorized`, indicating that the request lacks valid authentication credentials for the target resource. Or if the client is authenticated to the server but does not have permission to access the application state, the HTTP response code has to be `403 Forbidden`, indicating that the server understood the request but refuses to authorize it due to the application logic.

2. Entity-body: If the request body is empty or not in the correct format, the server has to return a `400 Bad Request`, indicating the request could not be understood by the server due to malformed syntax. The client should not repeat the request without modifications. If the request body is in the correct format but contains invalid data e.g. price of an item should not be smaller than 0, the server has to respond with a `422 Unprocessable Entity`, meaning the server understands the content type of the request entity but was unable to process the contained instructions due to the application logic.

In addition to self-descriptive messages, we also have to ensure that the client will be redirected to its previous application state or at least to the dispatcher state.

4 Approach

The dynamic analysis is considered successful when a client is able to travel and perform all its permitted and unpermitted actions generated by the server for every user role. This again can be verified by the underlying model. Ultimately, any undesired behavior can reveal bugs within the generators. In order to carry out a dynamic analysis, it is necessary to perform two crawling processes: First model crawling and then HTTP crawling. The model crawling process is intended to derive information from the underlying model and to build appropriate test cases. Afterwards, an HTTP client will test against these test cases when the server is deployed on a runtime environment. It is also necessary to generate test data based on the model to prevent a client from navigating on an empty server. At last, a generator will combine test cases information provided by the model crawler and functionalities of the HTTP client to produce runnable test classes.

4.1 Model Crawling

The model crawling process is divided into three parts to retrieve crucial information for our test case generation: Verifying hypermedia response, deriving test paths to visit every state based on the application workflow and negative testing.

Verifying Hypermedia Responses A client expects to only see its permitted hyperlinks at any given state. If a request is valid because the client is authorized to enter an application state, response of this application state must contain the same hyperlinks as given by the model to this role. If a request goes wrong, the client must be provided with an appropriate response code or redirected to the dispatcher state.

To accomplish this task, we have to map every incoming transition of an application state with its outgoing transitions with respect to a specific user role. This is necessary, because the only URI known to the client is the one that leads to the entry state of the RESTful API. Every other hyperlink is dynamically generated by the server, and therefore clients cannot make any presumption about them. Technically speaking as shown in Listing 1, a transition in our model is comprised of a hyperlink, a media type and a relation type. The hyperlink is generated by the server. The media type is given so the client can understand the representation of a response. The relation type is akin to the rel attribute of HTML link tags and this serves as a method call for the client. In other words, the client, when assigned with a user role, can only move forwards from the dispatcher state by making requests to these relation types.

```
.transitions ()
  .fromState ("Start")
    .toState ("GetItems")
    .usingRelationType ("getItems")
  .fromState ("GetItems")
    .toState ("PostItem")
    .usingRelationType ("createItem")
```

Listing 1 Textual definition of transitions in our metamodel.

So, in order to determine whether a hypermedia response is correct we have to map every relation type to permitted following relation types with respect to the current assigned user. This way, an HTTP client is able to make a request to a relation type and expects to see the exact set of following relation types as designed in the model by checking these mappings. In order to accomplish this, we loop through every state and check if a state can be accessed by the current user role. From any given state: A user role is permitted to access a set of following states. So we can map a state's incoming transition as a key to a set of permitted outgoing transitions.

Deriving Positive Test Paths In the previous step we know what hypermedia response to expect after making a request to a particular application state through our mappings. Nevertheless, once the server is deployed on a runtime environment, the HTTP client cannot verify these responses by making direct requests the URI endpoints but it rather must start from the entry state and navigate through the application workflow to visit every state. This way, we can make sure that responses of every application state have been tested at least once. The main challenge here is to derive all possible test paths within a role-based workflow from the existing model to generate test cases for the crawler.

A naive solution approach would be letting an HTTP client embody a user role and start from the dispatcher state and randomly chooses next transition to walk through the application and it will eventually visit all application states. This approach is problematic because: First, due to its random nature, we cannot determine the time crawler requires to visit every state, it can take very long to fully test a large API. Second, if the crawler fails, we cannot reproduce the test case to find the error cause.

Another more sophisticated approach is to develop a depth-first search algorithm that derives all possible paths for a specific user role from the model. This algorithm spans a role-based specific workflow from a directed graph into a tree. The tree represents every possible path within a workflow. Each path represents a task. For instance, a task could be update an item. To manage this task, a client with admin role must sequentially visit these application states: *Start, GetItems, GetItem, PutItem*. The root element is the *Start* state, it represents the entry point of the API. A valid path ends when the client is directed back to one of the previous states in the path. For example, after changing the price of an item from a *PutItem* state, the server will redirect the client to *GetItem* state which the client has visited before. When a valid path ends, it is marked as visited and the HTTP client starts at the *Start* state again to crawl next test path.

A task is considered successfully tested when an HTTP client is able to visit all edges of the role-based specific application workflow. By doing this, any occurring error can be tracked down by looking at the path where the client fails to walk through. By design, every task within the workflow must be manageable from the entry point, so this is the best way to approve this premise. Once the client manages to walk through every path of the tree and to verify every obtaining response on its way, then we can make a definite statement about the correct hypermedia behavior of the RESTful API.

This step is to derive role-based positive test paths from the model for an HTTP client to test against later on runtime. To achieve this, we have to span the role-based representation of the application workflow into a tree with the start state as root. In our formal definition, every hypermedia system represents a ε-NFA or it can also be formally defined as an complete unweighted directed graph $G = (V, E)$ with *V* is the set of possible application states and *E* is the set of possible state-to-state transitions.

Assuming our ε-NFA is a directed graph $G = (V, E)$ with $V = \{A, B, C, D, E\}$ and $E = \{(A, B), (B, A), (B, C), (C, B), (B, D), (D, C), (C, E), (E, B)\}$ shown in Figure 3.

At the beginning we have not walked down any path yet, hence the *Visited edges* set and the *List of paths* are empty and the *Unvisited edges* set contains all available edges. A path is defined as a sequential

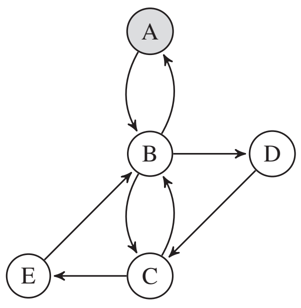

Figure 3 Graph representation of the application workflow.

list of edges and an edge is equivalent to a transition in our RESTful context.

- *Unvisited edges* = {(*A, B*), (*B, A*), (*B, C*), (*C, B*), (*B, D*), (*D, C*), (*C, E*), (*E, B*)}
- *Visited edges* = {}
- *List of paths* = {}

Next, we start to walk down from *A* until we find a leaf which is a node that we have visited before along the current path, which is *B* in this case. A path is derived once we found a leaf, then we can mark its incoming edge as visited and start to walk backwards until we find a node that has unvisited edges. In this example, node *C* has an unvisited edge. Node *E* does not have any unvisited edge. Therefore, its incoming edge (*C, E*) will also be marked as visited as shown in Figure 4.

At this point, we have the following data:

- *Unvisited edges* = {(*A, B*), (*B, A*), (*B, C*), (*C, B*), (*B, D*), (*D, C*)}
- *Visited edges* = {(*E, B*), (*C, E*)}
- *List of paths* = {{(*A, B*), (*B, C*), (*C, E*), (*E, B*)}}

The algorithm terminates when the last edge has been visited at *(B, A)*, it walks backwards and finds no further unvisited outgoing edge from every node along the path, subsequently it marks all edges left as visited.

Our final list of paths now contains all possible paths from the start node *A* and there is no unvisited edge left.

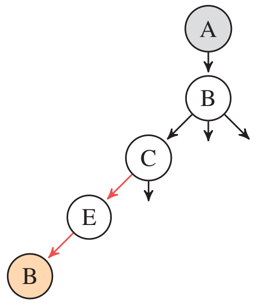

Figure 4 Deriving first path.

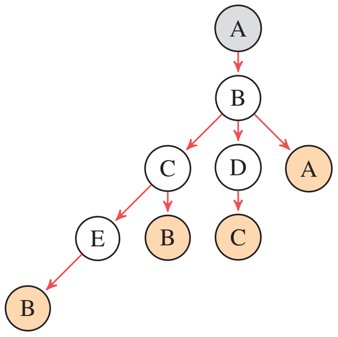

Figure 5 Deriving last path.

- *Unvisited edges* = {}
- *Visited edges* = {(*A*, *B*), (*B*, *A*), (*B*, *D*), (*D*, *C*), (*E*, *B*), (*C*, *E*), (*C*, *B*), (*B*, *C*)}
- *List of paths* = {{(A, B), (B, C), (C, E), (E, B)},
  {(A, B), (B, C), (C, B)},
  {(A, B), (B, C), (C, B)},
  {(A, B), (B, A)}}

4.2 Generation of Test Data

In order to test our generated server, we have to populate it with test data. Otherwise a client would not be able to obtain or modify any resource while navigating through the application workflow. Our MDSD approach allows automated test data generation based on our existing metamodel. This means, once written, this process will start to generate adequate test data by looking for available resources within our model. Our resource definition is straightforward as presented in Listing 2.

```
.defineResourceWithName("Item")
 .withAttribute("name"). asString()
 .withAttribute("price"). asFloat()
```

Listing 2 Textual definition of a resource in our metamodel.

4.3 Building HTTP Client

For our hypermedia testing purpose, we need to build our own hypermedia-driven HTTP client. First, the client represents a specific user role while testing, so it makes sense to save this authentication information. Then it must to be able to make request to relation types instead of sending request directly to URIs. A response link of our of our RESTful API consists of four parts: resource URI, `rel` as relation type, for media type and `method` for HTTP verb.

These elements need to be extracted and parsed, so the HTTP client can understand and make request to it. This is necessary because technically speaking an HTTP client requires merely a URI to make request to, but it is not recommended to hard code these so called "REST endpoints" into clients [6], because they can change, e.g, to resource renaming. So our hypermedia response is always provided with a fixed relation type serving as a method call to guide action and of which the client has knowledge.

4.4 Generation of Test Classes

The actual model is encapsulated behind the generation process, and therefore in order to run these test cases out of the box, we have to combine the retrieved data from the model crawling process with the functionalities of the HTTP client and embed them in generated test classes. This can be done by a generator producing test class files as shown in Figure 6. There are three crucial parts of information that must be persisted in a test class for an HTTP client to be able to begin with the test cases: 1) List of paths that represents given positive test cases,

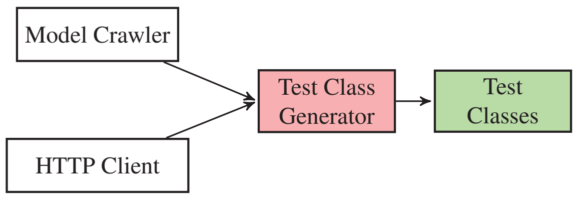

Figure 6 The model crawler and the HTTP client are required by the test class generator to create test classes.

Algorithm 1: Integration testing procedure for hypermedia in pseudo code.

```
initialize $client$ with specific user role;
initialize $mapRelToPermittedRels$ with relation type to permitted
  following relation types;
initialize $mapRelToUnpermittedRels$ with relation type to unpermitted
  following relation types;
initialize $allPaths$ as list of paths;
initialize $entryUri$ as entry point of the API;
foreach $path$ in $allPaths$ do
    $client$ sends request to $entryUri$;
    foreach $relationType$ in $path$ do
        $client$ makes request to $relationType$;
        if request is successful then
            verify response by looking up in $mapRelToPermittedRels$;
            foreach $relationType$ in $mapRelToUnpermittedRels$ do
                test all unpermitted relationTypes;
                verify response code;
            end
        end
        else
            report error;
        end
    end
end
```

2) mappings between relation type and correct outgoing relation types which allow the HTTP client to verify response after each request and 3) mappings between relation type and unpermitted relation types which allow the HTTP client to perform negative testing. This information will be generated and initialized as hard-coded objects in each test class.

For the sake of clarity, we explain the dynamic integration testing procedure with the aid of pseudo code as listed in Algorithm 1 instead of listing the generated source code. First, the HTTP client loops through each path in *allPath* given a specific user-role. Each path is comprised of a sequential list of relation types starting from the entry URI. Therefore, the client must enter the API here. For a client to make a proper request to a relation type, it must parse the hypermedia response and extract the URI, and whether it is necessary to send an entity-body (e.g. at POST or PUT). After each successful request the client starts to validate the obtained response by comparing its content with the one given in the mappings. Following that, the client starts with negative testing phase by looking for unpermitted relation types at this state in the correlate mappings and try to perform these actions. The application behavior will be validated by the test cases generated from the model, any misbehavior will be reported as error.

5 Conclusion

The main goal of this paper is to automate integration testing with the focus on hypermedia testing using MDT. In order to accomplish this goal, we have separated the server-side hypermedia testing into static and dynamic analysis.

First, we carry out a model verification process via static analysis. This procedure makes sure our model is designed as an ε-NFA, hence hypermedia compliant, before allowing any source code transformation [21]. Afterwards, we continue with a dynamic analysis. The main goal of the dynamic analysis is to automate hypermedia test class generation using our MDT approach. The generated test classes must guarantee the correct hypermedia behavior of the actual generated RESTful API as designed in the model. This approach consists of four parts: model crawling process, test data generation, building HTTP client and generation of test classes. The model crawling process extracts information from the model to build proper test cases, such as role-based test paths, response validation and negative testing. To verify these test cases, an HTTP client was built with the ability to authenticate, navigate via relation types and understand hypermedia responses of application states. Finally, we had to combine both: test case data retrieved by the model crawler and functionalities of the HTTP client to generate ready-to-run test classes via a generator. We present a small application example to demonstrate our approach. This application also includes role-based access control with two user roles: admin and customer.

To address RQ 1, we propose a model crawling process to extract information from the model to build appropriate test cases. These test cases must consider role-based access of application states, hypermedia response validation and negative testing. In order to retrieve information for role-based access of application states, we have developed an algorithm to derive all possible test paths for each user role. According to our assumption every task within a hypermedia system must be manageable from the entry state, and therefore, a path must represent a distinct task. Our algorithm is able embody an user role to navigate through the application workflow, visiting every transition and application state. As a result, it delivers a set of distinct test paths for each role. Validation of application state responses requires mapping information between relation type and role-based follow-up relation types. We have achieved these mappings by letting the model crawler loop through every application state and map all incoming transitions of an application state with permitted outgoing transitions for each user role. Our model crawler also managed to retrieve information for negative testing by mapping relation type to unpermitted relation types. Our model also allows a straightforward approach towards test data generation. This was achieved by generating test data based on meta information of the resources given by the model.

Approaching RQ 2, we first build an HTTP client to verify hypermedia test cases. This HTTP client must be able to authenticate as a pre-defined user role, make request to relation types and understand hypermedia responses. We made use of an open source Java HTTP client named OkHttp [18] and added necessary features. These objectives were accomplished with no further complication. Afterwards, we combine both information of role-based test cases retrieved by the model crawler and functionalities of the HTTP client to generate ready-to-run test classes. For this purpose, we used an open source library named JavaPoet [17] to handle source code generation. As a result, we successfully generated role-based test classes, including test paths, validation mappings, negative testing, HTTP crawler and all required imports.

In order to answer RQ 3, we must take several aspects of integration testing into consideration: By using the underlying model, we were able to generate test classes to cover all possible tasks within the example application workflow, assuring white-box hypermedia testing of the

Table 1 Overview of test paths and requests generated for different user roles

	Admin	Customer
Number of test paths	4	2
Valid requests	8	4
Invalid requests	0	3
Total requests	8	7

overall system. This would be a time consuming task if implemented manually because a) deriving all distinct role-based test paths without an algorithm would be inconceivable, b) role-based hypermedia test cases are repetitive causing a developer to write many similar test cases and c) negative testing also requires enormous amount of time when unpermitted actions have to be verified for every application state. To illustrate the effort saved by our Model-Driven approach, we take a look at the number of generated test cases. Our application example has two user roles, one resource, six application states and ten transitions. Table 1 indicates the amount of generated requests for each user role, which the dynamic analysis is able to generate for our application example.

In comparison to the existing related works which only focus on functional testing of RESTful APIs and completely neglect the hypermedia constraint, our Model-Driven approach automatically generates testing artifacts to ensure its presence.

6 Outlook

On the server-side, we managed to generate positive test cases, by making valid requests with valid data as inputs and checking whether the application response as expected. These results are quite satisfactory as our generated test cases could cover all possible tasks within a application and verify role-based responses of every application state. Nevertheless, the type of negative testing we were able to accomplish within the scope of this work was sending unauthorized requests once the HTTP client enters an application state. There are many other possibilities to generate different types of negative testing, such as applying not allowed methods, trying to access a (sub-)resource after deleting it or forcing the HTTP client to send wrong resource representations. We can extend the model crawler to extract more possible test cases to achieve larger test coverage. These features only need to be implemented once and test cases will be generated for free. We can strive to reveal more bugs or as Dijkstra states in his article [4]: testing can only reveal the presence of bugs, but not their absence.

Additionally, we should discuss more about how the application should behave in case of invalid requests. As for the scope of this work, we only expect to see an appropriate response code. We have also discussed a bit about whether the server should send a client back to its previous state. However, this approach would violate the stateless constraint. Always sending a client back to the start state would, on the other hand, reduce the usability of the application, forcing the user to repeat many unnecessary steps again, especially when the intended task is nested deep within the application workflow. Further research would be needed to clarify this matter.

RESTful APIs can be consumed by third-party clients. Clients that make proper use of hypermedia would require less manual adaption to server updates than those that do not. Once our server-side hypermedia testing process is fully automated, we will address the client-side hypermedia testing. Our motivation on the client-side testing is to find out whether a client is hypermedia-driven or not. This assumption can be confirmed if a running client can handle certain types of change within the server. We will discuss the degree to which a server can be changed without negatively impacting hypermedia clients. In our future works, we will address the practicality of automating server updates for hypermedia client testing.

References

[1] Mike Amundsen. *RESTful Web Clients – Enabling Reuse Through Hypermedia*. Sebastopol: O'Reilly Media, 2017. ISBN: 978-1-491-92190-6.

[2] Joshua Bloch. *How to design a good API and why it matters*. http://static. googleusercontent.com/media/research.google.com/en/pubs/archive/32713.pdf. Last accessed on May 23, 2018. 2014.

[3] S.K. Chakrabarti and P. Kumar. "Test-the-REST: An Approach to Testing REST-ful Web-Services". In: *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World:* Nov. 2009, pp. 302–308.

[4] Edsger W. Dijkstra. "The Humble Programmer". In: *Commun. ACM* 15.10 (Oct. 1972), pp. 859–866. ISSN: 0001-0782. DOI: 10.1145/355604.361591. URL: http://doi.acm.org/10.1145/355604.361591.

[5] Tobias Fertig and Peter Braun. "Model-driven Testing of RESTful APIs". In: *Proceedings of the 24th International Conference on World Wide Web Companion*. WWW '15 Companion. Florence, Italy: International World Wide Web Conferences Steering Committee, 2015, pp. 1497–1502. ISBN: 978-1-4503-3473-0. DOI: 10.1145/2740908.2743045. URL: http://dx.doi.org/10.1145/2740908.2743045.

[6] Roy T. Fielding. *WTF is a "REST endpoint"*. https://twitter.com/fielding/status/324448353180061696. Last accessed on May 22, 2018. 2013.

[7] R.T. Fielding. *REST APIs must be hyper-text driven*. http://roy.gbiv.com/ untangled/2008/rest-apis-must-be-hypertext-driven. Last accessed on May 23, 2018. Oct. 2008.

[8] R.T. Fielding. "REST: Architectural Styles and the Design of Network-based Software Architectures". Doctoral dissertation. University of California, Irvine, 2000.

[9] R.T. Fielding, UC Irvine, and J. Gettys. *Hypertext Transfer Protocol – HTTP/1.1*. https://tools.ietf.org/html/rfc2616. Last accessed on May 22, 2018. 1999.

[10] Nicolas Frankel. *Integration Testing from the Trenches*. Leanpub, 2015. ISBN: 978-2-955-02143-9.

[11] Carlos A. González and Jordi Cabot. "Test Data Generation for Model Transformations Combining Partition and Constraint Analysis". In: *Theory and Practice of Model Transformations*. Ed. by Davide Di Ruscio and Dániel Varró. Cham: Springer International Publishing, 2014, pp. 25–41. ISBN: 978-3-319-08789-4.

[12] Peter Liggesmeyer. *Software-Qualität – Testen, Analysieren und Verifizieren von Software*. 2. Aufl. Berlin Heidelberg: Springer Science and Business Media, 2009. ISBN: 978-3-827-42056-5.

[13] The Selenium Project. *Selenium*. https://www.seleniumhq.org/. Last accessed on May 24, 2018. 2018.

[14] L. Richardson, M. Amundsen, and S. Ruby. *RESTful Web APIs*. O'Reilly Media, 2013. ISBN: 9781449359737. URL: https://books.google.de/books?id= ZXDGAAAAQBAJ.

[15] Leonard Richardson. *The Maturity Heuristic*. https:// www.crummy.com/writing/speaking/2008-QCon/act3.html. Last accessed on May 16, 2018. 2009.

[16] V. Schreibmann and P. Braun. "Model-Driven Development of RESTful APIs". In: *Proceedings of the 11th International Conference of Web Information Systems and Technologies*. (Lisbon, Portugal). INSTICC. SciTePress, May 2015.

[17] Inc. Square. *JavaPoet*. https://github.com/square/javapoet. Last accessed on May 23, 2018. 2017.

[18] Inc. Square. *OkHttp*. http://square.github.io/okhttp/. Last accessed on May 23, 2018. 2017.

[19] Antero Taivalsaari and Tommi Mikkonen. "The Web as a Software Platform: Ten Years Later". In: *Proceedings of the 13th International Conference of Web Information Systems and Technologies*. (Porto, Portugal). INSTICC. SciTePress, May 2017.

[20] H. Vu, T. Fertig, and P. Braun. "Towards model-driven hypermedia testing for RESTful systems". In: *WEBIST 2017 – Proceedings of the 13th International Conference on Web Information Systems and Technologies*. 2017.

[21] Henry Vu, Tobias Fertig, and Peter Braun. "Verification of Hypermedia Characteristic of RESTful Finite-State Machines". In: *Companion Proceedings of the The Web Conference 2018*. WWW '18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 1881–1886. ISBN: 978-1-4503- 5640-4. DOI: 10.1145/3184558.3191656. URL: https://doi.org/10.1145/3184558.3191656.

[22] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice – Hypermedia and Systems Architecture*. Sebastopol: "O'Reilly Media, Inc.", 2010. ISBN: 978-1-449-39702-9.

[23] David Wright. *Finite State Machines*. http://www4. ncsu.edu/∼drwrigh3/docs/courses/csc216/fsm–notes.pdf. Last accessed on May 16, 2018. 2005.

Biographies

Henry Vu is a software engineer and consultant. He attended the University of Applied Sciences Würzburg-Schweinfurt Germany (FHWS) where he received his B.Eng. in Computer Science in 2015 and his M.Sc. in Information Systems in 2017. After completing his master's degree, he was employed as a research associate and lecturer at the FHWS until 2018. During this time, he has published several papers about Model-Driven Engineering of systems with REST architecture. He is currently a software consultant at PENTASYS AG Germany where he is working for big names in the automobile industry (Porsche, BMW) in innovative topics such as autonomous driving and virtual reality.

Tobias Fertig completed his bachelor's degree at the University of Applied Sciences Würzburg-Schweinfurt (FHWS). He was the best of year and also scholarship holder. He then switched to the Friedrich-Alexander-University Erlangen-Nürnberg (FAU) to do his master's degree in Computer Science. As part of his master's thesis, he dealt with the offline support of RESTful Systems. Since his graduation, Tobias has been employed as a research associate and lecturer at the FHWS. He teaches programming, software engineering and operating systems. In addition to his work as a research assistant, he is a Ph.D. student in cooperation with FAU. In his dissertation he is focusing on the automated measurement of information security awareness within companies. Before starting his dissertation, he was researching in Model-Driven software development and low-code platforms. In addition, Tobias advises companies on blockchain technology and is author of the german book "Blockchain for Developers".

Peter Braun is software architect, computer scientist, and entrepreneur. He works as Professor of Computer Science at the University of Applied Sciences Würzburg-Schweinfurt. Before, he was

Head of Technology (CTO) of match2blue in Jena, Germany. The company develops and markets complex mobile information systems. Besides match2blue, Peter has co-founded two other companies in the area of mobile solutions. Peter earned a doctorate degree (Ph.D.) in software engineering from University of Jena. He is co-author of about 40 peer-reviewed papers and author of the first text-book about mobile software agents.