Detection of REST Patterns and Antipatterns: A Heuristics-based Approach

Francis Palma[1,2], Johann Dubois[1,3], Naouel Moha[1], and Yann-Gaël Guéhéneuc[2]

[1] Département d'informatique, Université du Québec à Montréal, Canada
moha.naouel@uqam.ca
[2] Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
{francis.palma, yann-gael.gueheneuc}@polymtl.ca
[3] École supérieure d'informatique, *eXia.Cesi*, France
johann.dubois@viacesi.fr

Abstract. REST (REpresentational State Transfer), relying on *resources* as its architectural unit, is currently a popular architectural choice for building Web-based applications. It is shown that *design patterns*—good solutions to recurring design problems—improve the design quality and facilitate maintenance and evolution of software systems. *Antipatterns*, on the other hand, are poor and counter-productive solutions. Therefore, the detection of REST patterns and antipatterns is essential for improving the maintenance and evolution of RESTful systems. Until now, however, no approach has been proposed. In this paper, we propose SODA-R (Service Oriented Detection for Antipatterns in REST), a heuristics-based approach to detect patterns and antipatterns in RESTful systems. We define detection heuristics for eight REST antipatterns and five patterns, and perform their detection on a set of 12 widely-used REST APIs including BestBuy, Facebook, and DropBox. The results show that SODA-R can perform the detection of REST patterns and antipatterns with high accuracy. We also found that Twitter, DropBox, and Alchemy are not well-designed, *i.e.*, contain more antipatterns. In contrast, Facebook, BestBuy, and YouTube are well-designed, *i.e.*, contain more patterns and less antipatterns.

Keywords: REST · Antipatterns · Patterns · Design · Heuristics · Detection

1 Introduction

At present, there is a major paradigmatic shift from the traditional stand-alone software solutions towards the service-oriented paradigm to design, develop, and deploy software systems [1]. REST (REpresentational State Transfer) [7] architectural style is simpler and more efficient than the traditional SOAP-based (Simple Object Access Protocol) Web services in publishing and consuming services over the Web [16, 20]. Thus, RESTful services are gaining an increased attention. Facebook, YouTube, Twitter, and many more companies, all leverage REST.

However, the increased usage of REST for designing and developing Web-based applications confronts common software engineering challenges. In fact, likewise any software system, RESTful systems must evolve to handle new web entities and resources, *i.e.*, meet new business requirements. Even, changes in underlying technologies or protocols may force the REST APIs to change. All these changes in requirements or underlying technologies/protocols may degrade the design of REST APIs, which may cause the introduction of common poor solutions to recurring design problems—*antipatterns*—in opposition to *design patterns*, which are good solutions to the problems that software engineers face when designing and developing RESTful systems. Antipatterns and patterns may be introduced even in the early design phase of RESTful systems. Antipatterns in RESTful systems not only degrade their design but also make their maintenance and evolution difficult, whereas design patterns facilitate them [3, 5, 6].

Forgetting Hypermedia [18] is a common REST antipattern that corresponds to the absence of *hypermedia*, *i.e.*, links within resource representations. The absence of such links hinders the state transition of RESTful systems and limits the runtime communication between clients and servers. In contrast, *Entity Linking* [6]—the corresponding pattern—promotes runtime communication via links provided by the servers within resource representations. By using such hyper-links, the services and consumers can be more autonomous and loosely coupled [6]. Moreover, *cacheability* in REST helps developers implementing high-performance and scalable REST services by limiting repetitive interactions, which is often ignored by the developers due to its complexity to implement. This bad practice to ignore cacheability is commonly known as *Ignoring Caching* antipattern [18]. The corresponding pattern, *Response Caching* [6] is a good practice to avoid sending duplicate requests and responses by caching all response messages in the local client machine. For REST APIs, the automatic detection of such patterns and antipatterns is an important activity by assessing their design (1) to ease their maintenance and evolution and (2) to improve their design quality.

REST patterns and antipatterns require a concrete detection approach, to support their rigorous analysis, which is still lacking. Despite the presence of several technology-specific approaches in SCA and Web services (*e.g.*, [3, 9–11, 14]), they are not applicable for detecting patterns and antipatterns in REST. Indeed, the key differences between REST architecture and other SOA standards/styles prevents the application of these approaches because: (1) traditional service-orientation is operations-centric, whereas REST is resources-centric, (2) RESTful services are on top of JSON (or XML) over HTTP, whereas traditional Web services are on top of SOAP over HTTP or JMS (Java Message Service), (3) Web services use WSDL (Web Service Definition Language) as their formal contracts; REST has no standardised contract except the human-readable documentations, (4) traditional services are the set of self-contained and independently developed software artefacts where operations are denoted using verbs; resources in REST are denoted by nouns and are directly-accessible objects via URIs, and (5) REST clients use the standard HTTP methods to interact with resources; Web services clients implement separate client-stub to consume services.

Among many others, the differences discussed above, motivate us to propose a new approach, SODA-R (Service Oriented Detection for Antipatterns in REST) to detect patterns and antipatterns in RESTful systems. SODA-R is supported by an underlying framework, SOFA (Service Oriented Framework for Antipatterns) [9] that supports static and dynamic analyses of service-based systems.

To validate SODA-R, first, we perform a thorough analysis of REST patterns and antipatterns from the literature [2,5,6,8,12,13,18] and define their detection heuristics. A detection heuristic provides an indication for the presence of certain design issues. A heuristic, for instance "*servers should provide entity links in their responses*", suggests that REST developers need to provide *entity links* in the responses that REST clients can use. For such case, we define a detection heuristic to check if the response header or body contains any resource location or entity links. However, a heuristic is not a rule that must be enforced during the design. A heuristic should be considered as a criteria that, if not conformed, indicate a potential design problem.

Following the defined heuristics, we implement their concrete detection algorithms, apply them on REST APIs widely used, and get the list of REST services detected as patterns and antipatterns. Our detection results show the effectiveness and accuracy of SODA-R: it can detect five REST patterns and eight REST antipatterns with an average precision and recall of more than of 75% on 12 REST APIs including BestBuy, Facebook, and DropBox.

Thus, the main contributions in this paper are: (1) the definition of detection heuristics for 13 REST patterns and antipatterns from the literature, namely [2,5,6,8,12,13,18], (2) the extension of SOFA framework from its early version [9] to allow the detection of REST patterns and antipatterns, and, finally, (3) the thorough validation of SODA-R approach with 13 REST patterns and antipatterns on a set of 12 REST APIs by invoking 115 REST methods from them.

The reminder of the paper is organised as follows. Section 2 briefly describes the contributions from the literature on the specification and detection of SOA patterns and antipatterns. Section 3 presents our approach SODA-R, while Section 4 presents its validation along with detailed discussions. Finally, Section 5 concludes the paper and sketches the future work.

2 Related Work

It is important to design REST (REpresentational State Transfer) APIs of quality for building well-maintainable and evolvable RESTful systems. In the literature, the concept of patterns and antipatterns are well-recognised as the means to evaluate various design concerns in terms of quality. Despite of the presence of some REST patterns and antipatterns defined recently by the SOA (Service Oriented Architecture) community, the methods and techniques for their detection are yet to propose.

Indeed, there are few books [2, 5, 6] that discuss a number of REST patterns. In addition, a number of online resources [8, 12, 13, 18] by REST practitioners provide a high-level overview of REST patterns and antipatterns and discuss how they are introduced by developers at design-time. Beyond those contributions, however, the detection of patterns and antipatterns require a concrete approach, to support their rigorous analysis, which is still lacking in the current literature.

For instance, Erl in his book [5] discussed 85 SOA patterns related to service design and composition. Erl *et al.* [6] also explained the REST and RESTful service-orientation, and discussed seven new REST patterns, thus in total, the catalog defines 92 SOA patterns. Daigneau [2] introduced 25 design patterns for SOAP (Simple Object Access Protocol) and RESTful services related to the service interaction, implementation, and evolution. Moreover, various online resources [8,12,13,18] defined a limited number of REST antipatterns related to API design with simple examples. All those books and online resources discussed (1) the solutions to recurring design problems (*i.e.*, patterns) or (2) the bad design practices (*i.e.*, antipatterns), but none of them discussed their detection.

A few contributions are available on the detection of SOA patterns and antipatterns for various SOA standards/styles, *e.g.*, SCA (Service Component Architecture) [3, 9–11] and Web services [14]. For example, Demange *et al.* [3] performed the detection of five SOA patterns on SCA systems relying on a rule-based approach to specify patterns using metrics. Di Penta *et al.* [14] followed an alternative model-checking approach for the detection of SOA patterns, where the authors built models from the SOAP messages exchanged among services.

For the detection of SOA antipatterns, we proposed contributions on diverse SOA technologies relying on different techniques. To summarise them here: we proposed the first approach [9] for the specification and detection of SOA antipatterns in SCA systems. This approach, called SODA (Service Oriented Detection of Antipatterns), relies on rule cards, *i.e.*, set of rules using static and dynamic metrics. Later, Nayrolles *et al.* [10] applied association rules to capture antipatterns from the execution traces of SCA systems and gained improvement in the detection accuracy. An extensive validation of the SODA approach in [11] with the largest SCA system (*i.e.*, FraSCAti [17]) further confirmed the extensibility of the rule-based language and the detection accuracy of the rule cards.

To the best of our knowledge, the detection of REST patterns and antipatterns, in the literature deserves yet to receive attention. As a continuous effort to investigate diverse SOA technologies with the goal of detecting REST patterns and antipatterns, we focus, in this paper, on analysing the REST APIs, both statically and dynamically.

3 The SODA-R Approach

We propose the SODA-R approach (Service Oriented Detection for Antipatterns in REST) for the detection of REST patterns and antipatterns. Figure 1 shows the two different steps of SODA-R.

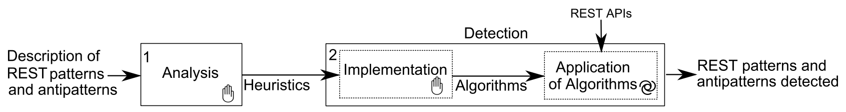
REST APIs
Detection
Description of REST patterns and antipatterns
1
Analysis
Heuristics
2
Implementation
Algorithms
Application of Algorithms
REST patterns and antipatterns detected

Fig. 1: The SODA-R approach.

Step 1. Analysis of Patterns and Antipatterns: This manual step involves analysing the description of REST patterns and antipatterns to identify the relevant properties that characterise them. We use these properties to define detection heuristics. *Step 2. Detection of Patterns and Antipatterns:* This semi-automatic step involves the manual implementation of detection algorithms based on the heuristics defined in the previous step. Later, we automatically apply these detection algorithms on a set of REST APIs and get the list of detected patterns and antipatterns.

The next sections detail the analysis of `REST` patterns and antipatterns, the implementation of detection algorithms, and the application of the detection algorithms on `REST APIs`. The validation of `SODA-R` is discussed in Section 4.

3.1 Analysis of Patterns and Antipatterns

For the definition of heuristics, we perform a thorough analysis of `REST` patterns and antipatterns by studying their descriptions and examples in the literature [6, 13, 18, 19]. This analysis helps us to identify the static and dynamic properties relevant to each `REST` pattern and antipattern. A static property is a property that is defined on a `RESTful` service and is obtained statically (*i.e.*, before invoking the service) by analysing the `HTTP` request header and body. For instance, the `HTTP` request headers `Accept` and `Cache-Control` and their corresponding values, respectively used to set the resource formats requested by the clients and to set the caching preferences, correspond to static properties.

A dynamic property, on the other hand, is obtained after making a service call to access a resource and can be found in the response headers and bodies, at runtime. For instance, the `HTTP` response headers `Location` and `Status` and their corresponding values, respectively used to set the new location by servers and to indicate the current context and status of the action performed by the server on a client request, correspond to dynamic properties. Table 1 shows the relevant static and dynamic properties for each pattern and antipattern, which we use and combine in the following to define detection heuristics.

Table 1: Relevant static and dynamic properties of patterns and antipatterns.

REST Antipatterns	**REST Patterns**	**Static Properties**	**Dynamic Properties**
Breaking Self-descriptiveness	–	*request-header fields*	*response-header fields*
Forgetting Hypermedia	Entity Linking	*http-methods*	*entity-links*; `Location`
Ignoring Caching	Response Caching	`Cache-Control`	`Cache-Control`; `ETag`
Ignoring MIME Types	Content Negotiation	`Accept`	`Content-Type`
Ignoring Status Code	–	*http-methods*	*status*; *status-code*
Misusing Cookies	–	`Cookie`	`Set-Cookie`
Tunneling Through GET	–	*http-method*; *request-uri*	–
Tunneling Through POST	–	*http-method*; *request-uri*	–
–	End-point Redirection	–	`Location`; *status-code*
–	Entity Endpoint	*end-points*; *http-methods*	–

Detection Heuristics of REST Antipatterns and Patterns: Using the static and dynamic properties, we define detection heuristics of `REST` patterns and antipatterns. Figures 2 and 3 show the detection heuristics defined for the *Forgetting Hypermedia* antipattern and *Entity Linking* pattern, respectively.

```
FORGET-HYPER-MEDIA(response-header, response-body, http-method)
    links[] ← EXTRACT-ENTITY-LINKS(response-body)
    if(http-method = GET and length(links[]) = 0) or
    (http-method = POST and ("Location:" ∉ response-header.getKeys() and
     length(links[]) = 0))) then
        print "Forgetting Hypermedia detected"
    end if
```

Fig. 2: Heuristic of *Forgetting Hypermedia* antipattern.

Forgetting Hypermedia [18] is a `REST` antipattern that identifies the absence of *entity links* in the response body or header. In general, for the `HTTP GET` requests, the *entity links* are provided in the response body, hence, checking the absence of links in the response body (*i.e.*, the size of the array containing the entity-links, *links*[], is zero) is sufficient (line 3, Figure 2). As for the `HTTP POST` requests, usually the server provides a *location* in the response header or *links* in the response body. Therefore, it is sufficient to look for the absence of `Location` in the response header (line 4, Figure 2) or the absence of links in the response body (line 5, Figure 2) to detect *Forgetting Hypermedia* antipattern. The corresponding pattern, *Entity Linking* [6] (Figure 3) refers to a `REST` service that provides entity links to follow in their response bodies or headers. We put the detection heuristics for the seven other `REST` antipatterns and four `REST` patterns on our web site[4].

```
ENTITY-LINKING(response-header, response-body, http-method)
  links[] ← EXTRACT-ENTITY-LINKS(response-body)
  if(http-method = GET and length(links[]) ≥ 1) or
   (http-method = POST and ("Location:" ∈ response-header.getKeys() or
   length(links[]) ≥ 1))) then
      print "Entity Linking detected"
  end if
```

Fig. 3: Heuristic of *Entity Linking* pattern.

Heuristics are more suitable, in particular for the detection of `REST` patterns and antipatterns, because they are more intuitive. Moreover, the engineer's knowledge and experience on `REST` patterns and antipatterns play a key role in defining heuristics.

3.2 Detection of Patterns and Antipatterns

In this section, we detail the detection of `REST` patterns and antipatterns. We show the different implementation and application steps in Figure 4.

Step 2.1 Implementation: From the heuristics defined in the previous step (in Section 3.1), we manually implement their corresponding detection algorithms. These algorithms are thus conform to detection heuristics that use and combine static and dynamic properties. We implement also the service interfaces for invoking `REST` services, and later to analyse their static and dynamic properties. These interfaces written in JAVA contain a set of methods mapped to respective `HTTP` requests for all `REST APIs` from their online documentations (see Table 4). The `REST API` online documentations comprise of (1) a list of resources, (2) a list of actions to perform on these resources, (3) the `HTTP` requests with entity end-points, and (4) a list of parameters for each `HTTP` request. However, the `REST` service providers barely expose machine-readable documentations, *i.e.*, they mostly rely on `HTML` or `PDF` documentations.

Step 2.2 Dynamic Invocation: After we getting JAVA interfaces for `REST APIs`, we implement the `REST` clients to invoke each service by providing the correct parameter lists. The `REST` clients must conform to the `API` documentations. During the detection time, we dynamically invoke the methods of service

[4] http://sofa.uqam.ca/soda-r/

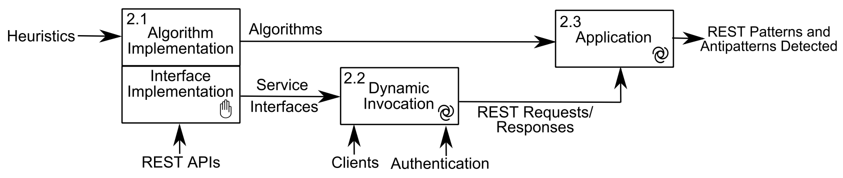

Fig. 4: Steps of the detection of REST patterns/antipatterns (Step 2 in Figure 1).

interfaces. From REST point of view, invocation of a method refers to performing an action on a *resource* or on an *entity*. For some method invocations, clients require to authenticate themselves to the servers. For each authentication process, we need to have a user account to ask for the developer credentials to the server. The server then supplies the user with the authentication details to use every time to make a signed HTTP request. For instance, YouTube and DropBox support OAuth 2.0 authentication protocol to authenticate their clients. At the end of this step, we gather all the requests and responses from the clients and servers, respectively, for further analysis.

Step 2.3 Application: For the application, we rely on the underlying framework SOFA (Service Oriented Framework for Antipatterns) [9] that enables the analysis of static and dynamic properties specific to REST patterns and antipatterns. We automatically apply the heuristics in the form of detection algorithms on the requests from the clients and responses from the servers, gathered in the previous step. In the end, we obtain a list of detected REST patterns and antipatterns while identifying the suspicious elements from the requests and responses, *i.e.*, the properties in Table 1.

From its initial version in [9], we further developed the SOFA framework to support the detection of REST patterns and antipatterns. SOFA itself is developed based on the SCA (Service Component Architecture) standard [4] and is composed of several SCA components. SOFA framework uses FraSCAti [17] as its runtime support. We added a new *REST Handler* SCA component in the framework. The *REST Handler* component supports the detection of REST patterns and antipatterns by (1) wrapping each REST API with an SCA component and (2) automatically applying the detection heuristics on the SCA-wrapped REST APIs. This wrapping allows us to introspect each request and response at runtime by using an IntentHandler. The intent handler in FraSCAti is an interceptor that can be applied on a specific service to implement the non-functional features, *e.g.*, transaction or logging. When we invoke a service that uses an IntentHandler, the service call is interrupted and the intent handler is notified by calling the invoke(IntentJoinPoint) method. This interruption of call enables us to introspect the responses of an invoked REST service.

The extended SOFA currently supports the detection of eight REST antipatterns and five REST patterns. In its early version, it supported the detection of 13 SCA-specific antipatterns [11] and five SOA design patterns [3].

4 Validation

To show the robustness of SODA-R approach, accuracy of our detection heuristics, and performance of the implemented algorithms, we performed experiments with five REST patterns and eight REST antipatterns on a set of 12 REST APIs.

4.1 Hypotheses

We define three hypotheses to assess the effectiveness of our `SODA-R` approach.
H_1. Robustness: *The `SODA-R` approach is robust.* This hypothesis claims that our `SODA-R` approach is assessed rigorously on a large set of `REST APIs` and with a set of different `REST` patterns and antipatterns.
H_2. Accuracy: *The detection heuristics have an average precision of more than 75% and a recall of 100%, i.e., more than three-quarters of detected patterns and antipatterns are true positive and we do not miss any existing patterns and antipatterns.* Having a trade-off between precision and recall, we presume that 75% precision is acceptable while our objective is to detect all existing patterns and antipatterns, *i.e.*, 100% recall. This hypothesis claims the accuracy of the defined detection heuristics and the implemented detection algorithms.
H_3. Performance: *The implemented algorithms perform with considerably a low detection times, i.e., on an average in the order of seconds.* Through this assumption, we support the performance of the implemented detection algorithms.

Table 2: List of eight `REST` antipatterns.

Breaking Self-descriptiveness: `REST` developers tend to ignore the ***standardised headers***, ***formats***, or ***protocols*** and use their own customised ones. This practice shatters the self-descriptiveness or containment of a message header. Breaking the self-descriptiveness also limits the ***reusability*** and ***adaptability*** of `REST` resources [18].

Forgetting Hypermedia: The ***lack*** of ***hypermedia***, *i.e.*, ***not linking resources***, hinders the state transition for `REST` applications. One possible indication of this antipattern is the ***absence*** of `URL` ***links*** in the ***resource representation***, which typically restricts clients to follow the links, *i.e.*, limits the dynamic communication between clients and servers [18].

Ignoring Caching: `REST` clients and server-side developers tend to ***avoid*** the caching capability due to its complexity to implement. However, caching capability is one of the principle `REST` constraints. The developers ignore caching by setting ***Cache-Control: no-cache*** or ***no-store*** and by not providing an ***ETag*** in the ***response header*** [18].

Ignoring `MIME` Types: The server should represent ***resources*** in various formats *e.g.*, ***xml, json, pdf***, etc., which may allow clients, developed in diverse languages, a more flexible service consumption. However, the server side ***developers*** often intend to have a ***single representation*** of resources or rely on their ***own formats***, which limits the resource (or service) ***accessibility*** and ***reusability*** [18].

Ignoring Status Code: Despite of a rich set of defined ***application-level status codes*** suitable for various contexts, `REST` developers tend to ***avoid*** them, *i.e.*, ***rely*** only on ***common ones***, namely 200, 404, and 500, or even use the ***wrong or no*** status ***codes***. The correct use of status codes from the classes ***2xx, 3xx, 4xx, and 5xx*** helps clients and servers to communicate in a more semantic manner [18].

Misusing Cookies: Statelessness is another `REST` principle to adhere—***session state*** in the server side is ***disallowed*** and any ***cookies violate*** `RESTfulness` [7]. Sending ***keys*** or ***tokens*** in the ***Set-Cookie*** or ***Cookie*** header field to server-side session is an example of misusing cookies, which concerns both ***security*** and ***privacy*** [18].

Tunneling Through `GET`: Being the most fundamental `HTTP` method in `REST`, the *`GET`* method ***retrieves*** a resource identified by a `URI`. However, very often the developers ***rely only*** on *`GET`* method to perform any kind of actions or operations including ***creating***, ***deleting***, or even for ***updating*** a resource. Nevertheless, `HTTP GET` is an inappropriate method for any actions other than ***accessing*** a ***resource***, and does not match its ***semantic purpose***, if improperly used [18].

Tunneling Through `POST`: This anti-pattern is very similar to the previous one, except that in addition to the URI the ***body*** of the `HTTP POST` request may ***embody operations*** and ***parameters*** to apply on the resource. The ***developers*** tend to ***depend*** only on `HTTP` *`POST`* method for ***sending any*** types of ***requests*** to the server including ***accessing***, ***updating***, or ***deleting*** a resource. In general, the proper use of `HTTP POST` is to ***create*** a server-side resource [18].

4.2 Subjects

We define heuristics for eight different `REST` antipatterns and five `REST` patterns from the literature. Tables 2 and 3 list those `REST` antipatterns and patterns collected from the literature, mainly [6, 8, 13, 18, 19]. In Tables 2 and 3, we put the relevant properties for each antipattern and pattern in ***bold-italics***.

4.3 Objects

We use some widely-used and popular `REST APIs` for which their underlying `HTTP` methods, service end-points, and authentication details are well documented online. Large companies like BestBuy, Facebook, or YouTube provide self-contained documentations with good example sets. Table 4 lists the 12 `REST APIs` that we analysed in our experiments.

Table 3: List of five `REST` patterns.

Content Negotiation: This pattern supports ***alternative resource representations***, *e.g.*, in ***json***, ***xml***, ***pdf***, etc. so that the service consuming becomes more flexible with ***high reusability***. Servers can provide resources in ***any standard format*** requested by the clients. This pattern is applied via standard `HTTP` ***media types*** and adhere to *service loose coupling* principle. If not applied at all, this turns into ***Ignoring*** `MIME` ***Types*** antipattern [6].

End-point Redirection: The ***redirection*** feature over the Web is supported by this pattern, which also plays a role as the means of ***service composition***. To redirect clients, servers send a new ***location*** to follow with one of the ***status code*** among ***301***, ***302***, ***307***, or ***308***. The main benefit of this pattern is—an ***alternative service*** remains ***active*** even if the requested service end-point is not sound [6].

Entity Linking: This pattern enables ***runtime communication*** via ***links*** provided by the server in the ***response body*** or via ***Location:*** in the ***response header***. By using ***hyper-links***, the servers and clients can be ***loosely coupled***, and the clients can ***automatically find*** the ***related entities*** at ***runtime***. If not properly applied, this pattern turns into ***Forgetting Hypermedia*** antipattern [6].

Entity Endpoint: Services with single ***end-points*** are too coarse-grained. Usually, a client requires at least ***two identifiers***: (1) a ***global*** for the ***service*** itself and (2) a ***local*** for the ***resource or entity*** managed by the service. By applying this pattern, *i.e.*, using ***multiple end-points***, each ***entity (or resource)*** of the incorporating service can be ***uniquely identified*** and ***addressed*** globally [13].

Response Caching: Response caching is a good practice to ***avoid sending duplicate requests*** and ***responses*** by caching all response messages in the ***local*** client machine. In opposed to ***Ignoring Caching*** antipattern, the ***Cache-Control:*** is set to any value other than ***no-cache*** and ***no-store***, or an ***ETag*** is used along with the ***status code 304*** [6].

Table 4: List of 12 `REST APIs`.

REST APIs	Online Documentations
Alchemy	alchemyapi.com/api/
BestBuy	bbyopen.com/developer/
Bitly	dev.bitly.com/api.html
CharlieHarvey	charlieharvey.org.uk/about/api/
DropBox	dropbox.com/developers/core/docs/
Facebook	developers.facebook.com/docs/graph-api/
Musicgraph	developer.musicgraph.com/api-docs/overview/
Ohloh	github.com/blackducksw/ohloh_api/
TeamViewer	integrate.teamviewer.com/en/develop/documentation/
Twitter	dev.twitter.com/docs/api/
YouTube	developers.google.com/youtube/v3/
Zappos	developer.zappos.com/docs/api-documentation/

4.4 Process

We defined the heuristics for five `REST` patterns and eight `REST` antipatterns, and implemented their detection algorithms. We also implemented the Java interfaces and clients for the 12 `REST APIs`. Through the clients, we invoked a total set of 115 methods from the service interfaces to access resources and got the responses from servers. Then, we applied the detection algorithms on the REST requests and responses and reported any existing patterns or antipatterns using our SOFA framework. We manually validated the detection results to identify the true positives and to find false negatives. The validation was performed by two professionals who have knowledge on REST and were not part of the experiments. We provided them the descriptions of REST patterns and antipatterns and the sets of all requests and responses collected during the service invocations. We used precision and recall to measure our detection accuracy. Precision concerns the ratio between the true detected patterns/antipatterns and all detected patterns/antipatterns. Recall is the ratio between the true detected patterns/antipatterns and all existing true patterns/antipatterns.

4.5 Results

Table 5 presents detailed detection results for the eight REST antipatterns and five REST patterns listed in Tables 2 and 3. Table 5 reports the patterns/antipatterns in the first column followed by the analysed REST APIs in the following twelve columns. For each REST API and for each pattern/antipattern, we report: (1) the total number of validated true positives with respect to the total detected patterns/antipatterns by our algorithms, *i.e.*, the precision, in the first row and (2) the total number of detected true positives with respect to the total existing true positives, *i.e.*, the recall, in the following row. The last two columns show the average precision-recall and the average detection time for each pattern/antipattern. The detailed results on all the test cases, *e.g.*, 115 methods from 12 REST APIs, are available on our web site[4].

4.6 Overview on the Results

Figure 5 shows the bar-plots of the detection results for the eight antipatterns and five patterns on the 12 REST APIs. In this section, we present an overview on the results.

REST developers are most likely to use their own header fields, data formats, and protocols, which limit the comprehension and reusability of REST APIs. For example, among more than 80 instances of detected *Breaking Self-descriptiveness* (BSD) antipattern: Facebook (29 instances), DropBox (12 instances), BestBuy (12 instances), and Twitter (10 instances) were mostly using customised header fields, data formats, and protocols. Also, *Forgetting Hypermedia* (FH) antipattern was detected in Facebook (8 instances) and DropBox (10 instances) APIs. Moreover, *Ignoring MIME Types* (IMT) antipattern was detected in Twitter (10 instances) and YouTube (9 instances) APIs. Among the less frequent antipatterns, *Ignoring Status Code* (ISC, 2 instances) and *Misusing Cookies* (MC, 3 instances) were not significantly observed among the 115 test cases.

As for REST patterns, *Content Negotiation* (CN, 70 instances) and *Entity Linking* (EL, 62 instances) were most frequently applied by REST developers. *Content Negotiation* pattern supports the ability to represent REST resources in diverse formats (implemented by REST developers) as requested by the clients. *Entity Linking* pattern facilitates clients to follow links provided by the servers. Furthermore, some APIs also applied *Response Cashing* (RC, 13 instances) and *End-point Redirection* (ER, 2 instances) patterns.

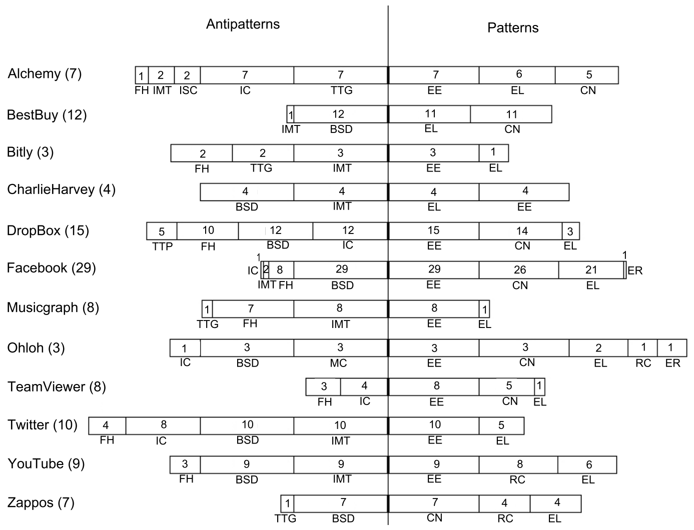

Fig. 5: Bar-plots of the detection results for eight antipatterns and five patterns. (`APIs` are followed by the number of method invocations in parentheses. The acronyms correspond to the pattern/antipattern name abbreviation and the number represents their detected instances.)

Overall, `APIs` that follow patterns tend to avoid corresponding antipatterns and *vice-versa*. For example: BestBuy and Facebook are found involved respectively in 0 and 8 instances of *Forgetting Hypermedia* antipattern; however, these `APIs` are involved in 11 and 21 corresponding *Entity Linking* pattern. Moreover, DropBox, Alchemy, YouTube, and Twitter `APIs` had 27 instances of *Ignoring Caching* antipattern, but they were involved in 8 instances of the corresponding *Response Cashing* pattern. Finally, we found Facebook, DropBox, BestBuy, and Zappos `APIs` involved in only 3 instances of *Ignoring `MIME` Types* antipattern, which conversely are involved in more than 55 instances of corresponding *Content Negotiation* pattern.

In general, among the 12 analysed `REST APIs` with 115 test cases and eight antipatterns, we found Twitter (32 instances of four antipatterns), DropBox (40 instances of four antipatterns), and Alchemy (19 instances of five antipatterns) are more problematic, *i.e.*, contain more antipatterns than others (see Figure 5). On the other hand, considering the five `REST` patterns, we found Facebook (49 instances of four patterns), BestBuy (22 instances of two patterns), and YouTube (15 instances of three patterns) are well designed *i.e.*, involve more patterns than others (see Figure 5).

4.7 Details of the Results

In this section, we discuss three detection results in detail, obtained in our experiments as presented in Table 5.

`REST` developers tend to rely on their own customised headers, formats, and protocols, and thus introduce *Breaking Self-descriptiveness* antipattern. The analysis on the 12 `REST APIs` shows that developers used non-standard header fields and protocols in most APIs including BestBuy, DropBox, Facebook, and Twitter. For example, Facebook used `x-fb-debug` and `x-fb-rev` header fields, which are mainly used to track a request id for their internal bug management purpose. Similarly, we found DropBox using the `x-dropbox-request-id` and Twitter using `x-tfe-logging-request-*` and `x-xss-protection` header fields. In general, the designers and implementers often distinguish the standardised and non-standardised header members by prefixing their names with "x-" (a.k.a. *eXperimental*). Indeed, the "x-" convention was highly discouraged by the Internet Society in RFC822 [15]. The manual validation reveals that all our detection was true positives and we reported all existing non-standard header fields and protocols (except two in DropBox where the manual validation considered them as common practice). This leads to the precision of 100% and the recall of 98.21% for this detection.

Table 5: Detection results of the eight REST antipatterns and five REST patterns obtained by applying detection algorithms on the 12 REST APIs.

(Test Methods)REST API	**(7)Alchemy**	**(12)BestBuy**	**(3)Bitly**	**(4)CharlieHarvey**	**(15)DropBox**	**(29)Facebook**	**(8)Musicgraph**	**(3)Ohloh**	**(8)TeamViewer**	**(10)Twitter**	**(9)YouTube**	**(7)Zappos**	**precision(p)-recall(r)**	**(115) Total**	**Precision-Recall(%)**	**Detection Time**
REST Antipatterns																
Breaking Self-	0/0	12/12	0/0	4/4	12/12	29/29	0/0	3/3	0/0	10/10	9/9	7/7	p	86/86	100%	21.31s
descriptiveness	0/0	12/12	0/0	4/4	12/14	29/29	0/0	3/3	0/0	10/10	9/9	7/7	r	86/88	98.21%	
Forgetting	1/1	0/0	2/2	0/0	9/10	8/8	7/7	0/0	3/3	4/4	2/3	0/0	p	36/38	94.58%	19.54s
Hypermedia	1/1	0/0	2/2	0/0	9/9	8/8	7/7	0/0	3/3	4/4	2/2	0/0	r	36/36	100%	
Ignoring	7/7	0/0	0/0	0/0	12/12	1/1	0/0	1/1	4/4	8/8	0/0	0/0	p	33/33	100%	18.99s
Caching	7/7	0/0	0/0	0/0	12/12	1/1	0/0	1/1	4/4	8/8	0/0	0/0	r	33/33	100%	
Ignoring	2/2	1/1	3/3	4/4	0/0	2/2	8/8	0/0	0/0	10/10	9/9	0/0	p	39/39	100%	19.39s
MIME Types	2/2	1/1	3/3	4/4	0/0	2/2	8/8	0/0	0/0	10/10	9/9	0/0	r	39/39	100%	
Ignoring	1/2	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	p	1/2	50%	21.22s
Status Code	1/2	0/0	0/1	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	r	1/3	25%	
Misusing	0/0	0/0	0/0	0/0	0/0	0/0	0/0	3/3	0/0	0/0	0/0	0/0	p	3/3	100%	19.1s
Cookies	0/0	0/0	0/0	0/0	0/0	0/0	0/0	3/3	0/0	0/0	0/0	0/0	r	3/3	100%	
Tunneling	5/7	0/0	0/2	0/0	0/0	0/0	0/1	0/0	0/0	0/0	0/0	0/1	p	5/11	17.86%	28.26s
Through `GET`	5/5	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	r	5/5	100%	
Tunneling	0/0	0/0	0/0	0/0	5/5	0/0	0/0	0/0	0/0	0/0	0/0	0/0	p	5/5	100%	28.64s
Through `POST`	0/0	0/0	0/0	0/0	5/5	0/0	0/0	0/0	0/0	0/0	0/0	0/0	r	5/5	100%	
REST Patterns																
Content	5/5	11/11	0/0	0/0	14/14	26/26	0/0	3/3	5/5	0/0	0/0	7/7	p	71/71	100%	19.63s
Negotiation	5/5	11/11	0/0	0/0	14/14	26/26	0/0	3/3	5/5	0/0	0/0	7/7	r	71/71	100%	
Entity	6/6	11/11	1/1	4/4	3/3	21/21	1/1	2/2	1/1	5/5	6/6	4/4	p	65/65	100%	19.90s
Linking	6/6	11/11	1/1	4/4	3/3	21/21	1/1	2/2	1/1	5/5	6/7	4/4	r	65/66	98.81%	
End-point	0/0	0/0	0/0	0/0	0/0	1/1	0/0	1/1	0/0	0/0	0/0	0/0	p	2/2	100%	20.36s
Redirection	0/0	0/0	0/0	0/0	0/0	1/1	0/0	1/1	0/0	0/0	0/0	0/0	r	2/2	100%	
Entity	1/1	0/0	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	0/0	p	10/10	100%	23.06s
Endpoint	1/1	0/0	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	0/0	r	10/10	100%	
Response	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	8/8	4/4	p	13/13	100%	19.23s
Caching	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0	0/0	8/8	4/4	r	13/13	100%	
Average													p	369/378	89.42%	21.43s
													r	369/374	94%	

Any RESTful interaction is driven by *hypermedia*—by which clients interact with application servers via URL links provided by servers in resource representations [7]. The absence of such interaction pattern is known as *Forgetting Hypermedia* antipattern [18], which was detected in eight APIs, namely Bitly, DropBox, Facebook, and so on (see Table 5). Among 115 test cases, we found more than 30 instances of this antipattern. Moreover, REST APIs that do not have this antipattern well applied the corresponding *Entity Linking* pattern [6], *e.g.*, Alchemy, BestBuy, and Ohloh, which is a good practice. This observation suggests that, in practice, developers sometimes do not provide hyper-links in resource representations. As for the validation, we detected 38 instances (36 were manually validated) of this antipattern; therefore, we have an average precision of 94.58% and a recall of 100%. For *Entity Linking* pattern, the manual validation confirmed 66 instances whereas we detected a total of 65 instances, all of which were true positives. Thus, we had an average precision of 100% and a recall of 98.81%.

Caching helps developers implementing high-performance and scalable REST services by limiting repetitive interactions, which if not properly applied violates one of the six REST principles [7]. REST developers widely ignore the caching capability by using *Pragma: no-cache* or *Cache-Control: no-cache* header in the requests, which forces the application to retrieve duplicate responses from servers. This bad practice is known as *Ignoring Caching* antipattern [18]. In contrast, the corresponding pattern, *Response Caching* [6] supports response cacheability. We detected six REST APIs that explicitly avoid caching capability, namely Alchemy, DropBox, Ohloh, and so on (see Table 5). On the other hand, cacheability is supported by YouTube and Zappos, which were detected as *Response Caching* patterns. The manual analysis of requests and responses also confirmed these detections, and we had an average precision and recall of 100%.

4.8 Discussion on the Hypotheses

In this section, we discuss the hypotheses defined in Section 4.1.

H_1. Robustness: To validate the SODA-R approach, we performed experiments on 12 REST APIs including well-known Facebook, BestBuy, DropBox, Twitter, and YouTube REST APIs. We analysed 115 methods in the form of HTTP requests from these APIs and applied detection algorithms of eight common REST antipatterns and five REST patterns on them. For each request among 115, we analysed individual request headers and bodies, and the corresponding response headers and bodies. With such an extensive evaluation and validation, we support our hypothesis on the robustness of our SODA-R approach.

H_2. Accuracy: As shown in Table 5, we obtained an average recall of 94% and an average precision of 89.42% on all REST APIs and for all test cases. The precision ranges from 17.86% to 100%, while we obtained a recall between 25% and 100% for all REST patterns and antipatterns. Thus, with an average precision of 89.42% and a recall of 94%, we can positively support our hypothesis on the accuracy of our defined heuristics and implemented detection algorithms.

H_3. Performance: The total required time includes: (i) the execution time, *i.e.*, sending REST requests and receiving REST responses (ranges from 19.1s to 24.55s) and (ii) the time required to apply and run the detection algorithms on the requests and responses (ranges from 0.004s to 4.312s). Each row in Table 5 (last column) reports the total required detection time for a pattern or an antipattern, which varies from 19.1s to 28.64s. We performed our experiments on an Intel Core-i7 with a processor speed of 2.50GHz and 6GB of memory. The detection time is comparatively very low (on an average 3% of the total required time) than the execution time. With a low average detection time of 21.43s, we can positively support our hypothesis on performance.

4.9 Threats to Validity

As future work, we plan to generalise our findings to other `REST APIs`. However, we tried to minimise the threat to the *external validity* of our results by performing experiments on 12 `REST APIs` by invoking and testing 115 methods from them. The detection results may vary based on the heuristics defined for the `REST` patterns and antipatterns. *Internal validity* refers to the effectiveness of our approach and the framework. We made sure that every invocation receives responses from servers with the correct request `URI` and the client authentication done while necessary. Moreover, we tested all the major `HTTP` methods in `REST`, *i.e.*, `GET`, `DELETE`, `PUT`, and `POST` on resources to minimise the threat to the internal validity. Engineers may have different views and different levels of expertise on `REST` patterns and antipatterns, which may affect the definition of heuristics. We attempted to lessen the threat to *construct validity* by defining the heuristics after a thorough review of existing literature on the `REST` patterns and antipatterns. We also involved two professionals in the intensive validation of the results. Finally, the threats to *reliability validity* concerns the possibility of replicating this study. To minimise this threat, we provide all the details required to replicate the study, including the heuristics, client requests, and server responses on our web site[4].

5 Conclusion and Future Work

`REST` (REpresentational State Transfer) is now a popular architectural style for building Web-based applications. `REST` developers may apply design patterns or introduce antipatterns. These `REST` patterns and antipatterns may respectively: (1) facilitate and hinder semantically richer communications between clients and servers or (2) ease and cause difficult maintenance and evolution.

This paper presented the `SODA-R` approach (Service Oriented Detection for Antipatterns in REST) to define detection heuristics and detect `REST` patterns and antipatterns on `REST APIs`. The detection of patterns and antipatterns in `REST APIs` requires an in-depth analysis of their design, documentation, invocation, and authentication. We applied `SODA-R` to define the detection heuristics of five common `REST` patterns and eight `REST` antipatterns. Using an extended `SOFA` framework (Service Oriented Framework for Antipatterns), we performed an extensive validation with 13 `REST` patterns and antipatterns. We analysed 12 `REST APIs` and tested 115 methods, and showed the accuracy of `SODA-R` with an average precision of 89.42% and recall of 94%.

In future work, we want to replicate `SODA-R` on other `REST APIs` and methods with more `REST` patterns and antipatterns. Currently, the literature on `REST` antipatterns and patterns is at its infancy, we intend to enrich the catalog of antipatterns and patterns by thoroughly investigating a large sets of `REST APIs` available online.

Acknowledgements The authors thank Abir Dilou for initiating the study. This study is supported by NSERC (Natural Sciences and Engineering Research Council of Canada) and FRQNT research grants.

References

1. Bennett, K., Layzell, P., Budgen, D., Brereton, P., Macaulay, L., Munro, M.: Service-based Software: The Future for Flexible Software. In: Proceedings of Seventh Asia-Pacific Software Engineering Conference. (2000) 214–221
2. Daigneau, R.: Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. Addison-Wesley (November 2011)
3. Demange, A., Moha, N., Tremblay, G.: Detection of SOA Patterns. In Basu, S., Pautasso, C., Zhang, L., Fu, X., eds.: Service-Oriented Computing. Volume 8274 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2013) 114–130
4. Edwards, M.: Service Component Architecture (SCA). OASIS, USA (April 2011)
5. Erl, T.: SOA Design Patterns. Prentice Hall PTR (January 2009)
6. Erl, T., Carlyle, B., Pautasso, C., Balasubramanian, R.: SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST. The Prentice Hall Service Technology Series from Thomas Erl. (2012)
7. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis (2000)
8. Fredrich, T.: RESTful Service Best Practices: Recommendations for Creating Web Services (May 2012)
9. Moha, N., Palma, F., Nayrolles, M., Conseil, B.J., Guéhéneuc, Y.G., Baudry, B., Jézéquel, J.M.: Specification and Detection of SOA Antipatterns. In: Service-Oriented Computing. Volume 7636 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (November 2012) 1–16
10. Nayrolles, M., Moha, N., Valtchev, P.: Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces. In: 20th Working Conference on Reverse Engineering. (October 2013) 321–330
11. Palma, F., Nayrolles, M., Moha, N., Guéhéneuc, Y.G., Baudry, B., Jézéquel, J.M.: SOA Antipatterns: An Approach for their Specification and Detection. International Journal of Cooperative Information Systems **22**(04) (2013)
12. Pautasso, C.: RESTful Service Design, Available Online: http://dret.net/netdret/docs/soa-rest-www2009/ (April 2009)
13. Pautasso, C.: Some REST Design Patterns (and Anti-Patterns), Available Online: http://www.jopera.org/node/442. (October 2009)
14. Penta, M.D., Santone, A., Villani, M.L.: Discovery of SOA Patterns via Model Checking. In: 2Nd International Workshop on Service Oriented Software Engineering: In Conjunction with the 6th ESEC/FSE Joint Meeting. IW-SOSWE '07, New York, NY, USA, ACM (2007) 8–14
15. RFC2822: Internet Message Format by Internet Engineering Task Force. Technical report (2001)
16. Rodriguez, A.: RESTful Web Services: The Basics. IBM. (November 2008) White paper.
17. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.B.: A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. Software: Practice and Experience **42**(5) (May 2012) 559–583
18. Tilkov, S.: REST Anti-Patterns, Available Online: www.infoq.com/articles/rest-anti-patterns (July 2008)
19. Tilkov, S.: RESTful Design: Intro, Patterns, Anti-Patterns, Available Online: http://www.devoxx.com/ (December 2008)
20. Vinoski, S.: Serendipitous Reuse. IEEE Internet Computing **12**(1) (January 2008) 84–87