Generating REST API Specifications through Static Analysis

Ruikai Huang
Georgia Institute of Technology
Atlanta, Georgia, USA
rkh@gatech.edu

Manish Motwani*
Georgia Institute of Technology
Atlanta, Georgia, USA
motwanim@oregonstate.edu

Idel Martinez
Georgia Institute of Technology
Atlanta, Georgia, USA
imartinez@gatech.edu

Alessandro Orso
Georgia Institute of Technology
Atlanta, Georgia, USA
orso@cc.gatech.edu

ABSTRACT

Web Application Programming Interfaces (APIs) allow services to be accessed over the network. RESTful (or REST) APIs, which use the REpresentation State Transfer (REST) protocol, are a popular type of web API. To use or test REST APIs, developers use specifications written in standards such as OpenAPI. However, creating and maintaining these specifications is time-consuming and error-prone, especially as software evolves, leading to incomplete or inconsistent specifications that negatively affect the use and testing of the APIs. To address this problem, we present Respector (REST API specification generator), the first technique to employ static and symbolic program analysis to generate specifications for REST APIs from their source code. We evaluated Respector on 15 real-world APIs with promising results in terms of precision and recall in inferring endpoint methods, endpoint parameters, method responses, and parameter attributes, including constraints leading to successful HTTP responses or errors. Furthermore, these results could be further improved with additional engineering. Comparing the Respector-generated specifications with the developer-provided ones shows that Respector was able to identify many missing endpoint methods, parameters, constraints, and responses, along with some inconsistencies between developer-provided specifications and API implementations. Finally, Respector outperformed several techniques that infer specifications from annotations within API implementations or by invoking the APIs.

KEYWORDS

REST APIs, OpenAPI specifications, documentation, static analysis

1 INTRODUCTION

The REpresentation State Transfer (REST) architecture has emerged as the main go-to approach for designing web APIs [10]. Because REST lacks a standard way of describing REST APIs, which makes development and testing challenging, the OpenAPI Initiative created the *OpenAPI specification (OAS)* [33], a vendor-neutral, portable, and open specification for REST APIs. With significant backing from industries such as Google, Microsoft, and IBM, OAS has become the de facto standard for describing REST APIs.

Prior research indicates that developers often fail to write and maintain specifications for REST APIs [37, 40, 41], and the APIs in production may therefore differ from their specification. While there exist many API specification generation techniques (e.g., AppMap [38], Swagger Inspector [55], ExpressO [48], Springfox [44], springdoc-openapi [52], ApiCarv [59]), these techniques have limited applicability and require developers to perform manual work, such as adding to the API source code technique-specific annotations or manually deploy the API and invoke all its endpoints. Further, such techniques typically produce relatively simple specifications that developers have to manually enhance. For example, a recent technique (ApiCarv [59]) generates OASs describing only HTTP methods and endpoint paths without describing all possible path parameters, parameter constraints, and responses.

This paper presents Respector (REST API specification generator), the first technique that employs static and symbolic program analysis to generate specifications for REST APIs from their implementations in an automated way. Given a REST API implementation as input, Respector produces an OAS as output by performing a set of steps. First, it determines the REST framework used by the API and performs static analysis to identify the API's endpoint methods. For each method, Respector then gathers the method's metadata (method URI, HTTP method, response type and status code(s)) and extracts the method's parameters. It then performs symbolic analysis to identify and add to the specification the conditions under which the method returns a success, an error, or terminates with an uncaught exception. Additionally, Respector identifies within the methods externally visible variables that are used before being defined (i.e., their value is externally provided) and/or written within the method (i.e., their value can be used by other methods) and uses this information to determine dependences between methods. Finally, Respector produces an OAS for the REST API using the information extracted by the analysis.

The OASs produced by Respector are richer than traditional OASs because they can also describe (1) parameters encapsulated in request bodies and defined using controller class fields, (2) parameter constraints that cause successful/erroneous API invocations, (3) responses (status code and schema) implemented for the endpoint method, and (4) dependencies between endpoint methods

*Also with Oregon State University, Corvallis, Oregon, USA.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639137

through global variables. This additional information can be useful for web developers using the API and for developers and testing tools when verifying the API implementation.

To validate our approach, we developed a Respector prototype that generates OAS v3.0 specifications for Java-based REST APIs developed using two popular REST frameworks: Spring Boot [50] and Jersey [22]. We then evaluated Respector on 15 real-world, open-source APIs, and answered the following research questions:

RQ1: Can Respector generate accurate specifications? For the APIs we considered, Respector generated specifications with, on average, 100% precision and 98.6% recall in inferring endpoint methods, 100% precision and 94.4% recall in inferring endpoint parameters, 100% precision and 92.6% recall in inferring responses, and 95.6% precision and 50.0% recall in inferring parameter constraints. Further, Respector accurately detected a total of 4,806 inter-dependencies across 100 endpoint methods.

RQ2: How do Respector-generated specifications compare with developer-provided specifications? For the APIs we evaluated, the Respector-generated specifications contained 228 endpoint methods, 2,795 parameters, 15 constraints, and 502 responses missing from the developer-provided specifications. Respector also identified 4 constraints that were inconsistent with the developer-provided specifications and were confirmed by the developers.

RQ3: How does Respector compare with alternative state-of-the-art API specification generation techniques? For the APIs we considered, four existing techniques (AppMap [38], Swagger Core [54], springdoc-openapi [52], and SpringFox [44]) failed to detect all specification components detected by Respector. The techniques detected, on average, 75% endpoint methods, 66% parameters, 18% constraints, and 64% responses detected by Respector.

The main contributions of this paper are:

- Respector, the first static-analysis-based approach for generating OASs from REST API implementations.
- An implementation of Respector that supports two Java REST frameworks: Spring Boot [50] and Jersey [22].
- An evaluation of Respector on 15 real-world REST APIs that shows that (1) Respector can effectively and automatically generate REST API specifications from API implementations, (2) Respector-generated specifications can reveal inconsistencies between developer-provided specifications and API implementations, and (3) Respector-generated specifications are more accurate than those generated by other existing API specification-generation techniques.
- A replication package for our empirical evaluation, available at https://archive.softwareheritage.org/browse/origin/https://github.com/nntzuekai/Respector.

2 MOTIVATING EXAMPLE

REST APIs use HTTP methods (GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS, and TRACE) to expose endpoints that perform CRUD (Create, Read, Update, Delete) operations on resources. The APIs are implemented using REST frameworks [39], such as Spring Boot, Jersey, Reslet, and Grails. Figure 1 shows a partial implementation of the *GET /entity-networks* endpoint in Senzing [11] API using the Jersey [22] framework. The implemented class and method use framework-specific annotations and libraries to specify paths,

```
1  import javax.ws.rs.*
2  @Path("/")
3  public class EntityGraphServices implements ServicesSupport {
4  @GET @Path("entity-networks")
5  public SzEntityNetworkResponse getEntityNetwork(
6  @DefaultValue("1000")@QueryParam("maxEntities") int maxEntities, ...){
7    // check for consistent entity IDs
8    try {
9    ...
10   if (buildOut < 0) { throw this.newBadRequestException(GET, uriInfo,
           timers,"Build out must be zero or greater: " + buildOut); }
11   if (maxEntities < 0) { throw this.newBadRequestException( GET, uriInfo,
           timers,"Max entities must be zero or greater: " + maxEntities); }
12  } catch (Exception e) {
13     e.printStackTrace();
14     throw this.newBadRequestException(GET,uriInfo,timers, e.getMessage());
15  }
16   try {
17     this.enteringQueue(timers);
18     String rawData = provider.executeInThread(() -> {
19       this.exitingQueue(timers);
20       ...
21       // parse the raw data
22       return sb.toString();
23     });
24     ...
25     // construct the response
26     SzEntityNetworkResponse response = this.newEntityNetworkResponse(GET,
             200, uriInfo, timers, entityNetworkData);
27     // if including raw data then add it
28     if (withRaw) response.setRawData(rawData);
29     // return the response
30     return response;
31   } catch (Exception e) {
32     throw this.newInternalServerErrorException(GET, uriInfo, timers, e);
33   }
34 }}
```

Figure 1: Partial view of the implementation of *GET /entity-networks* endpoint in Senzing API using Jersey framework.

methods, parameters, and responses (lines 2, 4, 6, 26 in Figure 1). APIs implement checks on request parameters and return a successful response (2XX) if the request is valid and they perform the desired operation successfully, otherwise, they return a response indicating a malformed request (4XX) or a server error (5XX). For example, on lines 10–15 in Figure 1, the endpoint method checks if the values of *buildOut* and *maxEntities* parameters are positive and responds with a bad request if they are not. The API performs desired operation and returns a successful/unsuccessful response if it completes it (lines 16–30 in Figure 1) or returns a server error response (line 32 in Figure 1) if it encounters any errors. Respector generates OAS by statically analyzing such API implementations.

In OpenAPI 3.0, each endpoint is specified using *path* and an HTTP method, and developers can define multiple methods for one path. We denote the combination of path and method an *endpoint method*. To operate on a resource, endpoint methods use parameters or request bodies. OpenAPI provides keywords to specify parameter attributes (name, location, data type, properties (e.g., format, default, example), and constraints (e.g., minimum, maxLength, required)). The constraints restrict parameter values that will yield valid HTTP responses. For example, Figure 2 shows a partial view of the difference between developer-provided and Respector-generated OAS of the *GET /entity-networks* endpoint method implemented in Figure 1. Figure 2 (lines 9-25) shows one parameter, *buildOut* that the endpoint method accepts along with their properties and constraints. Parameters are categorized into four types (path, query, header, cookie) based on their location (e.g., line 13 in Figure 2), and HTTP requests vary based on the location. After completing a request,

```
--- Senzing_API_Developer.json
+++ Senzing_API_Respector.json
 "/entity-networks": {
   "get": {
-    "description": "This operation finds ...",
+    "description": "",
-    "operationId": "findEntityNetwork",
+    "operationId": "33",
     "parameters": [ {
         "name": "buildOut",
-        "description": "The maximum number of degrees...",
+        "description": "",
         "in": "query",
         "required": false,
         "schema": {
           "type": "integer",
-          "format": "int8",
+          "format": "int32",
           "default": 1,
           "minimum": 0,
-          "maximum": 100
+          "exclusiveMinimum": false
     }},
       ...
     ],
     "responses": {
       "200": {
-        "description": "Successful response",
+        "description": "OK",
         "content": {
           "application/json": {
             "schema": {
-              "$ref": "#/components/schemas/SzEntityNetworkResponse"
+              "type": "object",
+              "title": "com.senzing.api.model.SzEntityNetworkResponse",
+              "properties": {}
             }},
       ...
     },
+    "x-endpoint-constraints": {
+      "$ref": "#/components/x-endpoint-constraints/~1entity-networks/get"
}}}
```

Figure 2: Partial view of the difference between developer-provided and Respector-generated OpenAPI specification for *GET /entity-networks* endpoint method in Senzing API.

APIs return an HTTP response with a status code and optional body or message (e.g., lines 26–39 in Figure 2), where the status codes range from 1XX-5XX [20].

Comparing Respector-generated OAS with the developer provided one (Figure 2) reveals that while Respector specification matches most entities in the developer provided one, it detects a few inconsistencies in the parameter properties and constraints indicating that API implementation differs from developer specification. For example, for *buildOut* parameter, developers specify *format* as 8-bit integer (line 17), whereas Respector detects it to be 32-bit (line 18) from implementation (line 6 in Figure 1). Similarly, developers specify *maximum* as 100 (line 21) but Respector detected no such constraint. Such inconsistencies can negatively impact the use and testing of APIs. We submitted bug reports to verify these inconsistencies with API developers and they confirmed that Respector-generated OAS is correct. Furthermore, using AppMap [38], an existing technique to generate OAS failed to generate specification for this endpoint method and 25 other methods in the Senzing API (details described later in Section 4.2.3). Overall. for many APIs, Respector detected many endpoint methods, parameters, constraints, and responses that were missing in developer-provided and auto-generated specifications but were implemented in the API source.

```
"x-endpoint-constraints": {
"/entity-networks": {
"get": {
  "valid-path-conditions": [""],
  "global-reads": { //global variables read by the endpoint
    "g23": {
      "name": "UTC_ZONE",
      "location-details": {
        "$ref": "#/components/x-global-variables-info/23"
      }},
    ...
  },
  "global-writes": { //global variables written by the endpoint
    "g26": {
      "name": "FACTORY",
      "assigned-values": [ "new com.senzing.api.model.SzEntityData$Factory
            (new com.senzing.api.model.SzEntityData$DefaultProvider())"],
      "location-details":{"$ref":"#/components/x-global-variables-info/26"
            }
    },
    ...
  }
}}},
"x-global-variables-info": { //global vars read/written by any endpoint
"g10": {
  "name": "FACTORY",
  "id": 10,
  "defining-class": "com.senzing.api.model.SzFlaggedRecord",
  "locations-of-static-assignments": ["line 144, com.senzing.api.model.
        SzFlaggedRecord"]
},
...
},
"x-endpoint-interdependence": {
"g10": {
  "location-details": {
    "$ref": "#/components/x-global-variables-info/10"
  },
  "read-by": {
    "em9": {
      "$ref": "#/paths/~1reevaluate-entity/post"
    },
    ...
  },
  "written-by": {
    "em1": {
      "$ref": "#/paths/~1data-sources~1{dataSourceCode}~1records~1{
            recordId}/put"
}}},
...
}
```

Figure 3: Partial Respector-generated enhanced OpenAPI specification for the *GET /entity-networks* endpoint method.

Thus, Respector-generated specifications can complement/improve developer-provided and auto-generated specifications.

Additionally, to assist API developers in verifying and validating their API implementation, Respector enhances generated OAS with: (1) parameter constraints that cannot be represented using OpenAPI keywords and (2) interdependent endpoint methods based on data dependency (reads/writes to global variables in the API source) using our-defined keywords extending the OpenAPI 3.0. For example, Figure 2 (line 41) points to the enhanced OAS shown in Figure 3 that has three parts: (1) constraints on endpoint parameters, global variables, and a combination of both (lines 1–21 in Figure 3), (2) global variables accessed by endpoint methods along with their defining classes and static assignments (lines 22–30 in Figure 3), and (3) endpoint methods interdependent through global variables (lines 31–47 in Figure 3). Respector detected 72 interdependencies between endpoint methods in the Senzing API (Section 4.2.1). For example, *POST /reevaluate-entity* and *PUT /data-sources/ {dataSourceCode}/records/{recordId}* are interdependent (lines 37–51 in Figure 3) through the global variable *FACTORY* (lines 23–28 in

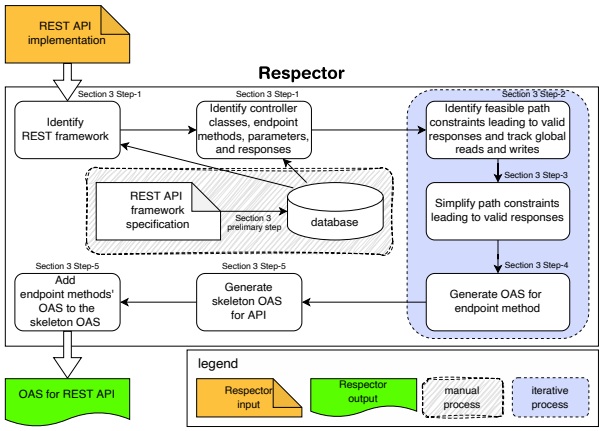

Figure 4: Overview of the Respector approach.

Figure 3). This information can be extremely useful to improve automated API testing by generating different operation sequences of the interdependent endpoint methods to test the API.

3 THE RESPECTOR APPROACH

This section details the steps of our approach (Figure 4).

Preliminary Step: REST Framework database creation. The documentation of annotation-based REST API frameworks describe: (1) packages implementing annotations and handling of HTTP requests and responses, (2) framework-specific annotations to specify controllers, methods, operations, and parameters, and (3) library methods/objects to access request body parameters and define HTTP status codes. We create a database of these patterns by studying the documentation of two widely used Java-based frameworks: Spring Boot [35, 51] and Jersey [12, 13]. The resulting database stores package name patterns, annotation semantics, and library objects/methods for request body parameters, and response creation. The database includes 2 class annotations, 14 method annotations, 11 parameter annotations, 33 library methods, 101 library objects for response creation, and 3 library methods for accessing request body parameters. **This step is a one-time effort** and took the authors less than two days for the two frameworks considered. Respector uses this database to detect these patterns in API source code to extract necessary information for generating specifications. The database does not need to be frequently updated as framework specifications do not change as often as their implementations.

Step-1: Identifying controller classes, endpoint methods, parameters, and responses. Respector infers the API's framework from the imported library members and the annotations used in the API source using patterns stored in the framework database. For example, from *import javax.ws.rs.** (line 1 in Figure 1) and annotations *@Path*, *@GET*, and *@QueryParam* (lines 2, 4, 6 in Figure 1) Respector infers that Senzing API uses Jersey framework. Next, Respector uses Algorithm 1 to extract controller classes, endpoint methods, parameters, and responses. The algorithm takes as input API class files and framework database, and outputs a data structure storing all information required to generate specification.

Extracting controller classes. To extract controller classes, Respector scans all API classes and detects the ones using framework-specific class annotations. For each such class, it extracts the URI

Algorithm 1: Extracting controller classes, endpoint methods, parameters, and responses

```
   Input: Compiled class files (CF), Framework database (DB)
   Output: Data structure storing controller classes and their associated methods,
           parameters, and responses (CC)
1  CC ← ∅;                                   // initialize set of controller classes
2  for each class C in CF do
3      if isAnnotated(C, DB) then
4          C.paths ← getClassURI(C, DB);
5          for each method M in class C do
6              if isAnnotated(M, DB) then
7                  M.metadata ← getMethodInfo(M,DB);
                   // get endpoint method URI, HTTP method, response type
                   amd status code(s)
8                  for each parameter P of method M do
9                      if isAnnotated(P, DB) then
10                         P.metadata ← getParameterInfo(P,DB);
                               // get parameter name, location, type,
                           default value, and required attributes
11                         M.addParameter(P);
                             // store parameter info of endpoint method
12                     else if isRawRequestBody(P, DB) then
13                         ReqP ← getReqParam(M, P, DB);
14                         M.addParameters(ReqP)      // store parameters'
                             info specified in raw request body
15                     else
16                         ReqSchema ← getSchema(P);
17                         M.addParameters(ReqSchema)          // store
                             parameters' converted from request body
18                 retType ← getReturnType(M.metadata);
19                 M.responseSchema ← getSchema(retType);
20                 C.addEndPointMethod(M);
21         for each field F in class C do
22             if isAnnotated(F, DB) then
23                 F.metadata ← getFieldInfo(F, DB);
24                 C.addFieldParams(F);
25         CC.add(C);
26 manageSubResources(CC);                          // resolve endpoint URIs
27 Procedure getReqParam(M, P, DB)
28     RM ← reachableFrom(M);                      // methods invoked by M
29     reqParams ← ∅;
30     for each method M in RM do
31         for each call-site C in M do
32             if callSiteAccessesParam(C, P, DB) then
33                 reqParams.add(extractBodyParam(C, P, DB));
34     return reqParams;
35 Procedure getSchema(Type)
36     if isUserDefinedClass(Type) then
37         schema ← {};
38         for each field F in Type do
39             schema.append(getSchema(F));          // recursive call
40         return schema;
41     else
42         return convertIntoOpenAPISchema(Type);
```

bound to it from the annotation (line 2–4 in Algorithm 1). For example, Respector identifies the controller class *EntityGraphService* from the annotation *@Path("/")* (line 2 in Figure 1) and records the URI path (/) bound to the controller class.

Extracting methods. For each identified controller class, Respector scans all its methods to detect those using framework-specific method annotations (e.g., *@Path* and *@GET* in line 4 in Figure 1). For each such method, Respector extracts the method's path and HTTP method from its annotations, as well as the return type and response status codes (e.g., *200*, *InternalServerError* from lines 26 and 32 in Figure 1) using the framework information stored in the database and the method's return type (e.g., in Jersey, the status code of an endpoint returning *null* response should be 204) (lines 5–7 in Algorithm 1).

Extracting parameters. For each identified method, Respector scans all its parameters to identify those using framework-specific parameter annotations (e.g., *@QueryParam* in line 6 in Figure 1). For each such parameter, Respector extracts its *name*, *location*, *type*,

Algorithm 2: Identifying URI paths for endpoint methods in REST APIs that use sub-resources.

```
   Input: Data structure storing controller classes and their associated methods, parameters,
          and responses (CC)
   Output: URI paths to access endpoint methods using sub-resources
1  Procedure manageSubResources(CC)
2      linkSuperResources(CC);
3      for each controller class C in CC do
4          for each endpoint method M in C do
5              SR ← getSuperResources(M);
6              M.metadata ← identifyFullPaths(M, SR);
7  Procedure linkSuperResources(CC)
8      for each endpoint method M in CC.getAllEndpointMethods() do
9          if returnType(M) in CC then
               // if endpoint method returns an instance of controller
               class
10             C ← returnType(M);
11             for each method M_C in C.getEndPointMethods() do
12                 M_C.addSuperResource(M);
13             CC.removeEndpointMethod(M);
14 Procedure identifyFullPaths(M, SR)
15     URIprefixes ← ∅;
16     C_M ← definingClass(M);
17     URIprefixes ← URIprefixes ∪ C_M.paths;
18     for each method M_P in super resources SR do
19         SR_P ← getSuperResources(M_P)
20         URIprefixes ← URIprefixes ∪
21         identifyFullPaths(M_P, SR_P);                    // recursive call
22     if URIprefixes = ∅ then URIprefixes ← {""};
23     methodPaths ← M.getURIPaths();
24     if methodPaths = ∅ then methodPaths ← {""};
25     fullPaths ← ∅;
26     for each path prefix P_i in URIprefixes do
27         for each method path P_j in methodPaths do
28             fullPaths.add(Concatenate(P_i, P_j));
29     return fullPaths
```

default, *format* and *required* attributes from the annotations (lines 8–10 in Algorithm 1). Unlike existing OAS generation techniques, Respector also detects parameters encapsulated in request bodies instead of using annotations. For this, Respector applies specific procedures based on whether the request body's class is implemented by framework (e.g., *WebRequest* from Spring Boot, *HttpServletRequest* from Jakarta Servlet [17]) or is user-defined and deserialized by framework to fetch parameters (e.g., using @ModelAttribute in SpringBoot [35] and Entity Providers in Jersey [13]). Respector identifies the type by checking it against the framework-specific classes stored in the database. If the type is framework-specific, Respector executes the *getReqParam* procedure (lines 12–13 in Algorithm 1) to identify all methods that are directly or indirectly called by this method and checks if any of those methods invoke any framework-specific library methods (also stored in the database) to access the request body parameters. Respector extracts those parameters from method invocations (line 14 in Algorithm 1). If the type is user-defined, Respector invokes the *getSchema* procedure (lines 15–16 in Algorithm 1) that deserializes the class to extract its fields in the form of OpenAPI schema and records them as the endpoint method's parameters (line 17 in Algorithm 1). Finally, Respector extracts parameters defined using controller class fields (lines 21–24 in Algorithm 1). For example, APIs using Jersey can define *path* parameters as controller class fields. For this, Respector scans through the fields of all controller classes and extracts the *name*, *location*, *type*, *default*, and *required* attributes of the ones using framework-specific parameter annotations. Respector records this information for each controller class (line 25 in Algorithm 1).

Extracting response schema. To extract response, Respector executes the *getSchema* procedure on the return type of the detected method encoded in its metadata (line 18-19 in Algorithm 1). If the return type is a user-defined class, Respector *recursively* executes the *getSchema* procedure on all the fields of that class until the type can be described using OpenAPI data types [43] (lines 36–40 in Algorithm 1). Once the return type can be described using OpenAPI data type, Respector generates OpenAPI schema for it (line 42 in Algorithm 1) and records this information for the detected method.

Extracting indirect paths. To express relationships, APIs may use nested resource URLs [56], where a request to access an endpoint method is routed through a controller class that does not encapsulate that method. We call the paths to access the nested resources *indirect paths*. For example, in the nested URL path */books/1/ratings* the controller class defining *book* resource returns a collection of *rating* resources (defined in a different controller class) that belong to the *book* resource with an *id* of 1. We call an endpoint method that can be invoked from other controller classes the *sub-resource* of those classes. To extract the correct paths bound to sub-resources, Respector uses the *manageSubResources* procedure (described in Algorithm 2) as a post-processing step (line 26 in Algorithm 1). When using sub-resources, the path to access an endpoint method is the concatenation of its *super-resource*s (other endpoint methods that can invoke this endpoint) and its own path. Respector uses Algorithm 2 to detect *full paths* to access the endpoint method invoked as a sub-resource by other endpoint methods. For this, Respector first links all super resources of all endpoint methods (line 2 in Algorithm 2) using the *linkSuperResources* procedure in Algorithm 2. Next, for each endpoint method, Respector identifies its super-resources (line 5 in Algorithm 2) and uses them to compute all possible full paths that can access the endpoint method. For this, Respector uses the *identifyFullPaths* procedure (line 5 in Algorithm 2) that uses the path of the endpoint method and those of its super-resources to recursively compute all possible full paths that can access the endpoint method. As multiple paths bound to the same endpoint method use different operations and parameters, Respector generates separate specification for each full path. After extracting endpoint methods, parameters, and responses Respector derives parameter constraints in Step-2.

Step-2: Identifying feasible paths and path constraints leading to successful responses and tracking read/write accesses to global variables. To derive parameter constraints that lead to successful or valid responses, Respector symbolically analyzes feasible paths starting from the entry point of the method to the statement that returns a response or throws an uncaught exception. This process generates path constraints (PC) and records any reads/writes made to global variables, which is used to infer interdependency between methods as described later in Step-5. A PC expresses constraints on the symbolic variables that must be satisfied for execution to reach a specific point in the program. Every time the execution follows a branch whose predicate involves symbolic values, the PC is suitably updated. A PC is represented as a conjunction of constraints ($c1 \wedge c2 \wedge ... \wedge cn$), where each ci is a constraint on one or more symbolic variables. The PCs for paths ending in successful responses are the conditions that must hold to get successful responses. For all other paths, the PCs denote conditions that lead to unsuccessful responses. For example, there are two PCs starting from the entry point (line 7) and ending in the uncaught exception (line 14) in Figure 1:

Algorithm 3: Symbolically analyzing endpoint method to derive constraints leading to valid responses

Input: Endpoint method (M), Inter-procedural control flow graph (ICFG), Framework database (DB)
Output: Path constraints leading to valid responses (ValidPC), Path constraints leading to invalid responses (InvalidPC), Global variables read by endpoint method (GlobalReads), Global variables written by endpoint method (GlobalWrites)

```
1  ValidPC ← ∅; InvalidPC ← ∅; GlobalReads ← ∅; GlobalWrites ← {};
2  SymStore ← ∅;                          // stores symbolic vars and values
3  currPC ← ∅;                            // stores current path constraints
4  NodeStack ← {}; // stack to store the snapshot of traversal state at a
     branching node
5  while DFSTraverse(ICFG(M)) do
6      N ← current node in traversal;
7      currPath ← current path in traversal;
8      if N is an Assignment node then
9          N_RHS ← updateSymStore(N_LHS, N_RHS);
10         SymStore[N_LHS] ← N_RHS;
11         if N_LHS is a global variable then
12             GlobalWrites[N_LHS].add(N_RHS);
13         if N_RHS is a global variable then
14             GlobalReads.add(N_RHS);
15     else if N is a Branch node then
16         NodeStack.push(currPath, N, SymStore, currPC);
17         childNode, cond ← getFeasibleBranch(N, currPC);
18         cond' ← substituteSymVars(cond, SymStore);
19         currPC = currPC ∧ cond';
20         continueTraverse(childNode);
21     else if N is a Return node then
22         if N returns a response then
23             currPath.statusCode ← getResponseCode(N, currPath, DB);
24             if isValid(currPath.statusCode) then
25                 ValidPC.add(currPath, currPC);
26             else
27                 InvalidPC.add(currPath, currPC);
28         currPath, N, SymStore, currPC ← backtrack(NodeStack);
29         continueTraverse(N);
30     else if N throws an exception then
31         if N has catch block then
32             currPath, N, SymStore, currPC ← backtrackWithException(e,
                 currPath, NodeStack);
33             continueTraverse(N);
34         else
35             InvalidPC.add(currPath, currPC);
```

PC-1: *buildOut<0*

PC-2: *!(buildOut<0)* ∧ *maxEntities<0*

Both PCs describe conditions that lead to unsuccessful responses.

Algorithm 3 describes this process. To perform this analysis, Respector constructs an inter-procedural control flow graph (ICFG) of the API source using Soot [57], as a one-time pre-processing step. The algorithm takes as input an endpoint method (M), the ICFG, and the framework database (DB) to compute: (1) *ValidPC*, the set of path constraints leading to successful responses, (2) *InvalidPC*, the set of path constraints leading to unsuccessful responses, (3) *GlobalReads*, the set of global variables read by the endpoint method, and (4) *GlobalWrites*, a map of global variables written by the endpoint method containing the set of values assigned to them. Respector traverses the ICFG in a depth-first manner starting from the entry point of the method until reaching a statement that either returns an HTTP response or exits the method due to an uncaught exception.

During traversal, Respector maintains (1) *SymStore*, a map to store the values of symbolic variables for endpoint parameters, global and local variables, (2) *currPC*, a list to store the PCs of the current path, and (3) *NodeStack*, a stack to store the traversal state at branching nodes to backtrack after traversing a branch.

When Respector encounters an assignment node, it recursively substitutes the assigned value (RHS) in the assignment using *SymStore* and adds the updated assignment node to *SymStore* (lines 8–10 in Algorithm 3). If the assignment contains invocations to methods that are defined in external libraries, Respector does not analyze them. If the assigned variable (LHS) is a global variable, Respector records its value in *GlobalWrites*. If the assigned value (RHS) uses a global variable, Respector adds that global variable to *GlobalReads* (lines 11–14 in Algorithm 3).

When Respector encounters a branch node, it first saves the current path, ICFG node, *SymStore*, and the path constraints in *NodeStack* (line 16 in Algorithm 3). It then uses Z3 SMT solver [16] to identify a *feasible branch*, which is a non-visited branch whose condition does not conflict with the collected PCs (line 17 in Algorithm 3). Filtering out non-feasible branches significantly reduces the search space. For the feasible branch, Respector recursively substitutes the variables in the condition using the *SymStore*, adds the updated condition to the path constraints (lines 18–19 in Algorithm 3), and continues the traversal along the feasible branch (line 20 in Algorithm 3).

When Respector encounters a return node, it extracts the response status code and determines whether it maps to a successful or unsuccessful response using the framework database (lines 21–23 in Algorithm 3). It then adds the current path and PCs to *ValidPC* or *InvalidPC*, accordingly (line 24–27 in Algorithm 3). To backtrack the traversal after reaching a return node, Respector uses *NodeStack* to find the last branching node with at least one non-visited branch, resets the traversal state, and resumes traversal from that branching node (line 28–29 in Algorithm 3).

When Respector encounters a throw statement, it checks if a *catch* block exists in the current path by traversing the ICFG, which contains all the associated try-catch blocks. If it does, Respector jumps to the catch block and continues its traversal (lines 31–33 in Algorithm 3); otherwise, the exception is considered to lead to an unsuccessful response, and the corresponding status code is inferred using the framework database, and the current path and PCs are added to *InvalidPC* (line 35 in Algorithm 3).

To address the path explosion issue when analyzing loops and recursions, Respector ignores all back-edges by unrolling loops once and dropping paths with recursive calls. Further, Respector uses an empirically determined threshold of 5,000 on the maximum number of feasible paths per endpoint method to analyze in order to scale on large code bases. After gathering path constraints, Respector simplifies them to derive the parameter constraints required to produce valid responses as described next.

Step-3: Simplifying path constraints leading to valid responses for an endpoint method.

The OpenAPI standard describes constraints on API parameters needed for valid responses (recall Section 2). Respector uses Algorithm 4 to simplify PCs and express them using OpenAPI keywords. The algorithm takes an endpoint method (M) and a set of feasible paths and PCs leading to successful responses (*ValidPC*) as input and outputs constraints imposed on endpoint parameters (C_p) or global variables (C_g). Other than using exclusively endpoint parameters or global variables, some constraints that must hold true for successful responses are defined using a combination of endpoint parameters, global variables, and uninterpreted functions, identified as *validPathConditions*. For example, for all PCs that end with

Algorithm 4: Simplifying path constraints

Input: Endpoint method (M), Path constraints leading to valid responses ($ValidPC$)
Output: Constraints on endpoint parameters (C_p), Constraints on global variables (C_g), Constraints that are not part of C_p and C_g ($validPathConditions$)

```
1  endpointParamList ← M.getAllEndPointParameters();
2  C_p ← simplifyConstraints(endpointParamList, ValidPC, Z3)
3  endpointGlobalList ← M.getAllGlobals();
4  C_g ← simplifyConstraints(endpointGlobalList, ValidPC, Z3)
5  validPathConditions ← ∅
6  for each (P, PC) in ValidPC do
7      for each constraint c in PC do
8          if c exists in AllPaths(ValidPC) then
9              if c has multipleVars() then
10                 validPathConditions.add(c);
11             else if !c.containtsEndpointParameters() and !c.containtsGlobals() then
12                 validPathConditions.add(c);
13 Procedure simplifyConstraints(VarList, ValidPC, SMTSolver)
14     varConstraints ← {}
15     for each variable k in VarList do
16         for each (P, PC) in ValidPC do
17             varConstraints[k] ← varConstraints[k] ∨
                 extractConstraints(PC, k);
18     return simplifyIntoConjunctions(varConstraints, SMTSolver)
```

successful responses (line 30 in Figure 1), the constraint (*!(buildOut<0)* ∧ *!(maxEntities<0)*) derived from the two PCs listed in the previous step *must* hold.

Each constraint in *ValidPC* defines a set of predicates (e.g., *!(buildOut<0)*) that must *all* be true to produce a successful response. However, predicates involving the same parameter that belong to different constraints (associated with different feasible paths) do not necessarily need to be true for producing successful response as all constraints independently lead to valid responses. To simplify the parameter constraints, Respector derives them from all PCs using the *simplifyPathConstraints* procedure (line 13 in Algorithm 4). The procedure takes as input a list of endpoint parameters or global variables associated with the input method, *ValidPC*, and an SMT solver. It extracts the constraints imposed on each variable from all paths and combines them using the disjunction (∨) operator (line 15–17 in Algorithm 4). Finally, it uses Z3 SMT solver to simplify the disjunction of extracted constraints and converts them into conjunctions (∧) [30]. This ensures that all predicates in the simplified constraint are *necessary* conditions to produce successful responses, and the simplified constraint does not contain redundant predicates. For example, if there are three constraints imposed on a variable x that are extracted from three different paths, then the set of constraints is represented using the disjunction (∨) operator as: $x \leq 1 \vee x > 0 \vee x < 2$. Respector uses SMT solver to simplify and converts the set into conjunctions (∧) as: $x > 0 \wedge x \leq 1$. Respector executes the *simplifyPathConstraints* procedure twice, once each for endpoint parameters (lines 1–2 in Algorithm 4) and for global variables (line 3–4 in Algorithm 4). Finally, predicates in all simplified constraints that are defined using a combination of endpoint parameters, global variables, or uninterpreted functions are recorded separately in *validPathConditions* (line 5–12 in Algorithm 4). These predicates can assist API developers in verifying the intended behavior of their API implementation. Since *validPathConditions* contain the constraints on the inputs of an endpoint method that must hold true to produce a successful response, using these constraints, API developers can assess the implemented behavior of their APIs for different types of inputs and check whether it matches their expectations.

Step-4: Generating endpoint method specification. In this step, Respector constructs OAS for the endpoint methods using the information derived in Steps 1–3. For each method, Respector uses its metadata to generate its OAS containing its URI path, HTTP method, operationId, endpoint parameters, and responses. Respector adds parameter constraints to the method specification by first attempting to express the predicates imposing constraints on endpoint parameters (recall Step-3) using OpenAPI keywords. For this, Respector uses a pattern-matching approach that checks if a predicate is not nested, identifies the operand data type, and the operator used. Based on the type and operator, Respector converts the predicate into an OpenAPI constraint. For example, `!(buildOut<0)` is converted into *{"minimum":0, "exclusiveMinimum":false}* (line 20, 22 in Figure 2) using the inferred type (*integer*) and operators (! and <). Similarly, Respector converts `x!=null` into *{"required":true}* using the type (`null`) and the operator (`!=`). Respector derives 8 of the 15 kinds of OpenAPI 3.0 constraints [34]: *minimum*, *maximum*, *exclusiveMinimum*, *exclusiveMaximum*, *required*, *maxLength*, *minLength*, and *multipleOf*. Of the remaining 7 kinds, 4 (*pattern, minItems, maxItems, uniqueItems*) cannot be derived from the information present in the API source and 3 (*enums*, *minProperties*, *maxProperties*) require analyzing external libraries, which is not currently supported by Respector prototype.

Optionally, Respector enhances the specification with additional constraints that cannot be expressed using OpenAPI by using our extended OpenAPI keywords (recall Figure 3). For example, the constraint *codes.contains*(";") on parameter *codes* cannot be expressed using any OpenAPI keywords. Respector represents such constraints in the SMT-LIB format [9], which is both machine-processable and human-readable, and puts them under *x-valid-path-conditions* (e.g., line 4 in Figure 3). Respector populates the enhanced specification using the *GlobalReads*, *GlobalWrites*, and *validPathConditions* (from Steps 3 and 4). For each global variable read/written by the method (listed under *global-reads* and *global-writes*), Respector adds its *name*, *location-details*, imposed constraints (*global-constraints*), and assigned values (*assigned-values*).

Step-5: Generating final API specification. To produce the final OAS, Respector generates an OpenAPI 3.0 skeleton (JSON) that contains the following.

```
1  {   "openapi": "3.0.0",
2      "servers": [{ "url": "http://localhost:8080" }],
3      "info": {
4          "title": "<name of API>",
5          "version": "",
6          "description": "<name of API>"
7      },
8      "paths": {},
9      "components": {
10         "x-endpoint-constraints": {},
11         "x-global-variables-info": {}
12         "x-endpoint-interdependence": {},
13     }
14 }
```

Respector then adds all the endpoint method OASs (from Step 4) under *paths* and their enhanced specifications under *x-endpoint-constraints* (lines 1–21 in Figure 3). Next, Respector adds all the global variables accessed by any endpoint method (obtained from *GlobalReads* and *GlobalWrites*) to the *x-global-variables-info* by specifying each global variable's name, id, defining class, and locations in the API source where it is initialized (e.g., lines 22–36 in Figure 3). Finally, Respector specifies the interdependence between endpoint

methods based on reads and writes made to global variables—for each global variable. Two endpoint methods are interdependent if they read or write to the same global variable. Respector identifies all such interdependent methods and puts this information under *x-endpoint-interdependence* (lines 37–43 in Figure 3). The final OAS is OpenAPI 3.0 compliant and is ready for consumption.

4 EMPIRICAL EVALUATION

This section describes the experiment setup to evaluate Respector (Section 4.1), evaluation results in terms of the research questions asked (Section 4.2), discussion on the evaluation findings (Section 4.3), and limitations and threats to validity (Section 4.3).

4.1 Experiment Setup

This section describes the dataset, metrics, and experiment procedure we use to evaluate Respector.

Dataset: As Respector requires bytecode to generate specifications, to evaluate Respector, we collected APIs from GitHub by searching "Java REST APIs" and selecting those that use Spring Boot or Jersey, have developer-provided specification, have at least 5 stars, and compile successfully. This resulted in 8 APIs (Digdag, enviroCar, Gravitee, Kafka, cassandra, Quartz, Senzing, and Ur-Codebin). Further, we included 7 APIs from prior studies [7, 40, 41] that have developer-provided specification and compile successfully. Figure 5 lists the 15 open-source Java APIs used to evaluate Respector, which vary in size from 1K to 119K lines of code (SLOC).

Metrics: To assess Respector's accuracy, we create a ground truth for each subject API by analyzing both, its developer-provided specification and source code. Multiple authors independently analyzed the API code and specification to identify the *endpoint methods* (path, HTTP method) and their associated *parameters* (name, location, data type), *parameter constraints*, and *responses* (successful status code and return type). At the end of the analysis, the authors reconciled their findings to create the ground truth. Following a recent study [59], we then compare the generated specifications with the ground truth to compute *precision* (correctly identified entities over total identified entities) and *recall* (correctly identified entities over total correct entities) in inferring: (1) endpoint methods, (2) parameters of the detected endpoint methods, (3) constraints on the detected parameters, and (4) responses for the detected endpoint methods. We manually inspect the accuracy of inferred interdependencies by verifying them in the API source.

Experiment Procedure: We used Respector to generate specifications for 15 APIs from their compiled classes that took 22 min per API, on average, with median time of 15.97 seconds. We manually inspect the accuracy of Respector-generated specifications by comparing them against the ground truth. We also compare the generated specifications with developer-provided ones to identify the entities detected and missed by Respector. As there exists no static-analysis-based API specification generation techniques, we compared Respector with four existing techniques AppMap [38], Swagger Core [54], springdoc-openapi [52], and SpringFox [44], which are closest to Respector because they use either developer-written tests or annotations in API code to generate specifications. While AppMap generates OAS from the information gathered by executing API tests, the other three techniques (Swagger Core, springdoc-openapi, and SpringFox) infer framework-specific and technique-specific annotations at runtime to generate OAS. Note that Unlike Respector, all four techniques *require* running the API or its tests to generate specifications. We attempted to generate specifications using the existing techniques by modifying the API's build configuration and deploying them locally on our server. All experiments were run on a server with two 2.53GHz Intel Xeon CPUs, 240 GB RAM, and Ubuntu 20.04 operating system.

REST API (short name)	Framework	SLOC	Description
Digdag (Digdag) [14]	Jersey	54.8K	API for workload automation system to manage task pipelines
enviroCar (enviroCar) [18]	Jersey	22.7K	API to track and analyze road traffic and driving behavior
Features-Service (Features-Service) [42]	Jersey	1K	API for managing products Feature Models
Gravitee.io (Gravitee) [25]	Jersey	118.8K	API management tool
Kafka REST Proxy (Kafka) [31]	Jersey	19.4K	API to manage data on Kafka cluster
Management API for Apache Cassandra (cassandra) [15]	Jersey	10.5K	API for managing distributed, wide-column store, NoSQL database management system
RESTCountries (RESTCountries) [5]	Jersey	1.3K	API to get information about countries
Senzing (Senzing) [11]	Jersey	30.9K	API for a platform used for entity resolution
CatWatch (CatWatch) [32]	Spring Boot	4K	API to fetch user's GitHub data and makes it accessible via REST API
CWA Verification Server (cwa) [1]	Spring Boot	2.1K	Verification API for the Corona-Warn-App
OCVN (OCVN) [23]	Spring Boot	24.6K	API to import Vietnam public procurement data into Open Contracting Data Standard (OCDS) NoSQL storage
Ohsome (Ohsome) [27]	Spring Boot	7.3K	API for analyzing OpenStreetMap history data
ProxyPrint (ProxyPrint) [47]	Spring Boot	5.5K	API for managing printshops and consumers
Quartz Manager (Quartz) [21]	Spring Boot	2.3K	API for Quartz Scheduler
Ur-Codebin (Ur-Codebin) [19]	Spring Boot	1.2K	Backend API of website allowing users to share their code with others

Figure 5: Real-world, open-source Java REST APIs used in the evaluation. "SLOC" denotes the source lines of code.

4.2 Results

This section describes our evaluation results in terms of the three research questions we ask.

4.2.1 RQ1: Can Respector generate accurate specifications? Figure 6 depicts the precision and recall of Respector in inferring endpoint methods and their parameters, constraints, and responses.

Endpoint methods. On average, Respector achieved 100% precision and 98.60% recall across the 15 APIs analyzed, detecting 946 (99.37%) out of 952 endpoint methods. Respector failed to detect 6 methods in 2 APIs that were bound to URIs dynamically using user-defined or framework classes that create absolute URIs at runtime (5 methods were missed in Kafka because they use user-defined class *io.confluent.kafkarest.response.UrlFactory* and 1 method in Ur-Codebin that uses Spring Boot class *setFilterProcessesUrl*).

Endpoint Parameters. On average, Respector achieved 100% precision and 94.44% recall in identifying parameters across the 15 APIs, detecting 7,977 (99.25%) out of the 8,037 parameters. Respector failed to detect 60 parameters in 7 APIs because these are handled using template types, overloaded HTTP methods, or framework-specific interfaces. For example, parameters in Digdag used *JsonDeserialize* annotation (provided by an external library) that instantiates an interface to accept the parameters. In enviroCar, the missed parameters use injectable interface (e.g., *javax.ws.rs.core.UriInfo*) that provides runtime access to application and request.

Parameter constraints. On average, Respector achieved 95.59% precision and 50% recall in detecting parameter constraints across the 15 API analyzed, inferring 31 (27.93%) out of the 111 constraints. Analyzing the 80 constraints that Respector missed, we found that 58 constraints (all of which are *required:true* in Ohsome API) were

API	endpoint method			endpoint parameter			parameter constraint			endpoint response			method interdependence		
	GT	precision	recall	GT	precision	recall	GT	precision	recall	GT	precision	recall	#methods	#GV	#IP
Digdag	41	100.00%	100.00%	96	100.00%	83.33%	4	NA	0.00%	41	100.00%	100.00%	4 (9.8%)	4	0
enviroCar	128	100.00%	100.00%	351	100.00%	57.55%	12	NA	0.00%	141	100.00%	87.94%	7 (5.5%)	17	0
Features-Service	18	100.00%	100.00%	35	100.00%	100.00%	0	-	-	18	100.00%	100.00%	5 (27.8%)	1	0
Gravitee	28	100.00%	100.00%	99	100.00%	100.00%	0	-	-	26	100.00%	100.00%	12 (42.9%)	21	0
Kafka	74	100.00%	93.24%	175	100.00%	88.00%	0	-	-	108	100.00%	63.89%	31 (41.9%)	22	0
cassandra	50	100.00%	100.00%	94	100.00%	100.00%	3	NA	0.00%	60	100.00%	86.67%	48 (96.0%)	14	990
RESTCountries	27	100.00%	100.00%	35	100.00%	100.00%	12	100.00%	100.00%	25	100.00%	100.00%	26 (96.3%)	12	0
Senzing	34	100.00%	100.00%	156	100.00%	98.72%	17	82.35%	100.00%	34	100.00%	100.00%	33 (97.1%)	115	72
CatWatch	14	100.00%	100.00%	32	100.00%	100.00%	3	100.00%	100.00%	10	100.00%	100.00%	7 (50.0%)	11	0
cwa	5	100.00%	100.00%	14	100.00%	92.86%	0	-	-	7	100.00%	71.43%	5 (100.0%)	33	0
OCVN	278	100.00%	100.00%	5,002	100.00%	100.00%	0	-	-	278	100.00%	100.00%	108 (38.9%)	16	0
Ohsome	159	100.00%	100.00%	1,937	100.00%	98.81%	58	NA	0.00%	280	100.00%	93.21%	80 (50.3%)	51	3,744
ProxyPrint	75	100.00%	100.00%	150	100.00%	97.33%	0	-	-	75	100.00%	100.00%	47 (62.7%)	55	0
Quartz	14	100.00%	100.00%	3	100.00%	100.00%	0	-	-	14	100.00%	100.00%	10 (66.7%)	5	0
Ur-Codebin	7	100.00%	85.71%	14	100.00%	100.00%	2	100.00%	100.00%	7	100.00%	85.71%	2 (28.6%)	16	0
Average	63.47	100.00%	98.60%	535.80	100.00%	94.44%	8.50	95.59%	50.00%	74.67	100.00%	92.59%	28.33 (54.3%)	26.20	320.40
Total	952	–	–	8,037	–	–	111	–	–	1,120	–	–	425 (44.6%)	393	4,806

GT: ground truth. *NA*: Respector could not detect any constraints. *#GV*: total number of global variables detected. *#IP*: count of interdependent endpoint method pairs.

Figure 6: Evaluating the accuracy of Respector-generated specifications in extracting the endpoint methods, parameters, constraints, responses, and method interdependencies for 15 REST APIs.

implemented using functions that Z3 could not resolve (e.g., Respector could not derive the constraint for *groupByKey* parameter because Z3 could not interpret function *splitParamOnComma* in the assignment `String[] groupByKey=inputProcessor.splitParamOnComma(inputProcessor.createEmptyArrayIfNull(servletRequest.getParameterValues("groupByKey")));`). The remaining 22 constraints were implemented with methods using runtime reflection or lambda functions invoked dynamically, and therefore cannot be statically analyzed. For example, Respector missed *minItems:4* constraint for parameter *bbox* in *GET /measurements* endpoint method of enviroCar API because its implementation uses user-defined *BoundingBox* class with 4 fields that are validated by Jersey at runtime. Analyzing the 3 incorrectly inferred constraints, we found that these occur in Senzing API whose code contains many nested conditionals leading to an excessive number of feasible paths exceeding Respector's threshold. Therefore, the constraints derived from analyzed paths were not *necessary* conditions to get valid responses.

Endpoint Responses. On average, Respector detected responses with 100% precision and 92.59% recall across the 15 APIs analyzed, detecting 1,038 (92.68%) out of the 1,120 responses. Analyzing the 82 responses in 6 APIs which Respector missed, we found that these use user-defined classes or third-party libraries to return asynchronous responses (e.g., *GET /v3/clusters* in Kafka uses *AsyncResponses* class), which Respector could not statically analyze.

Method interdependence. Figure 6 ("method interdependence") shows that Respector detected 425 (44.6%) of the 953 endpoint methods in the 15 APIs that read/write to 393 global variables. 100 of these 425 methods were inferred to be interdependent based on data dependency through some global variable. Respector detected a total of 4,806 interdependencies across these 100 methods.

> Respector generates accurate specifications with, on average, 100% precision and 98.6% recall in inferring endpoint methods, 100% precision and 94.4% recall in inferring parameters, 95.6% precision and 50% recall in inferring parameter constraints, and 100% precision and 92.6% recall in inferring responses. Further, it accurately detects 4,806 interdependencies across 100 endpoint methods (RQ1).

API	endpoint method		endpoint parameter		parameter constraint		endpoint response	
	Res	Dev	Res	Dev	Res	Dev (conflict)	Res	Dev
Digdag	41	39	80	72	0	0 (0)	41	36
enviroCar	128	77	202	84	0	0 (0)	124	44
Features-Service	18	18	35	35	0	0 (0)	18	7
Gravitee	28	26	99	99	0	0 (0)	26	21
Kafka	69	41	154	101	0	0 (0)	69	0
cassandra	50	50	94	94	0	0 (0)	52	25
RESTCountries	27	10	35	20	12	0 (0)	25	0
Senzing	34	34	154	150	14	13 (4)	34	34
CatWatch	14	6	32	28	3	1 (0)	10	6
cwa	5	5	13	4	0	0 (0)	5	3
OCVN	278	190	5,002	2,834	0	0 (0)	278	160
Ohsome	159	135	1,914	1,608	0	0 (0)	261	123
ProxyPrint	75	71	146	44	0	0 (0)	75	67
Quartz	14	10	3	3	0	0 (0)	14	6
Ur-Codebin	6	6	14	6	2	2 (0)	6	4
Total	946	718	7,977	5,182	31	16 (4)	1,038	536

Dev: the count of entities extracted by Respector (*Res*) and present in developer-provided specifications.

Figure 7: Comparing Respector-generated and developer-provided OpenAPI specifications for 15 REST APIs.

4.2.2 RQ2: Do Respector-generated specifications cover behavior missed by developer-written specifications? The *Res* columns in Figure 7 show the total numbers of endpoint methods and their associated parameters, constraints, and responses extracted by Respector and the *Dev* columns depict the count of ones present in the developer-provided specifications. In total, developers missed specifying 228 endpoint methods, 2,795 parameters, 15 constraints, and 502 responses detected by Respector. Using Respector, we also found 4 conflicting parameter constraints (*conflict* in Figure 7) between developer specifications and API implementations. We submitted bug reports for these 4 conflicts, and developers have confirmed that Respector-generated OAS is correct.

Analyzing the entities missed in the developer-provided specifications, we found that the missing endpoint methods use sub-resources (recall *Extracting indirect paths* in Section 3 Step-1), which lead to a multitude of endpoints, while the missing parameters were encapsulated in request bodies, which are hard to manually enumerate and therefore were probably missed by developers. For example, developers specified endpoint methods *GET /tracks/{track}/measurements* and *GET /measurements/{measurement}* but missed *GET /tracks/{track}/measurements/{measurement}* in *enviroCar* API. Analyzing missed responses, we found that developers either missed describing response schema (e.g., in RESTCountries, developers missed schema for all 10 responses) or successful responses with non-200 status codes (e.g., in Kafka, developers missed

all 41 responses with 204 status code). Respector also detected a few missing 200 responses (e.g., response of *POST /apis/apiId/deployments* endpoint method in Gravitee API is missed by developers).

> Respector identifies 228 endpoint methods, 2,795 parameters, 15 constraints, and 502 responses missed by developer-provided specifications, and 4 parameter constraints that were inconsistent with the developer specifications (RQ2).

4.2.3 RQ3: How does Respector compare with alternative state-of-the-art API specification generation techniques? While generating OASs for the 15 APIs (Section 4.1), AppMap failed for 7 APIs because it has limitations in recording API's test execution (cassandra, Kafka), API tests did not involve any requests and responses (Digdag), or the API tests failed (enviroCar, Gravitee, RESTcountries, OCVN). Because AppMap does not support generating constraints and responses in specifications, those were not present even in APIs for which AppMap generated OASs.

While generating OASs for the 15 APIs using Swagger Core [54] (for Jersey APIs), springdoc-openapi [52] and SpringFox [44] (for Spring Boot APIs), we attempted to deploy and run the 15 APIs locally. We failed to deploy 8 APIs (Digdag, enviroCar, Gravitee, Kafka, cassandra, Senzing, ProxyPrint, and Quartz) because of missing documentation on setting up databases, authentication failures, and configuration to run the API similar to the prior studies [36, 61].

Figure 8 lists the 10 APIs for which at least one of the four existing techniques generated OAS. For each API, the figure shows the number of endpoint methods, parameters, constraints, and responses extracted by Respector and their respective counts inferred by the four techniques. For example, AppMap detected only 6 out of 34 endpoint methods in Senzing API as it could not record Parameterized JUnit Tests testing the other 28 methods. For REST-Countries, AppMap failed to generate OAS because the API tests failed. Overall, AppMap worked for 8 APIs detecting 118 (36.3%) out of 325 endpoint methods, 81 (3.5%) of the 2,311 parameters, and *none* of the 31 constraints and 726 responses detected by Respector because AppMap does not support inferring constraints and responses. Analyzing the reason why AppMap failed to detect all the methods and parameters revealed that developer-written tests missed testing requests using those methods and parameters.

While Swagger Core can generate OAS only for Jersey APIs, springdoc-openapi and SpringFox can generate OAS only for Spring Boot APIs. Swagger Core generated OASs for 2 out of 3 Jersey APIs, detecting 40 (88.9%) out of 45 methods, 69 (98.6%) out of 70 parameters, 2 (16.7%) out of 12 constraints, and 7 (16.3%) out of 43 responses detected by Respector. springdoc-openapi generated OASs for the 3 while SpringFox generated for the 2 out 5 Spring Boot APIs. For the 3 APIs on which springdoc-openapi worked, it detected 173 (96.6%) out of 179 methods, 1,936 (98.8%) out of 1,960 parameters, 1 (20%) out of 5 constraints, and 270 (97.5%) out of 277 responses detected by Respector. For the 2 APIs on which SpringFox worked, it detected 349 (79.9%) out of 437 methods, 4,442 (64.2%) of the 6,916 parameters, and 421 (78.1%) out of 539 responses detected by Respector. Analyzing why Swagger Core, springdoc-openapi, and SpringFox missed endpoint methods, parameters, constraints, and responses (detected by Respector), we found that some were implemented in non-annotation-based approaches (e.g., parameters encapsulated in request bodies as mentioned in Section. 3, Step-1, *Extracting parameters*). Furthermore, these techniques also missed some entities that are implemented using annotations (e.g., springdoc-openapi failed to detect the endpoint method *GET /statistics/contributors* in CatWatch API even though it uses Spring Boot annotation *RequestMapping* to specify the path). We suspect that happens due to either conceptual limitations or potential bugs in their implementation and we have created bug reports for such scenarios.

> Respector outperforms four state-of-the-art OAS generation techniques that detected only on average, 75.43% endpoint methods, 66.28% parameters, 18.35% constraints, and 63.97% responses detected by Respector (RQ3).

4.3 Discussion

In this section, we discuss the three main causes that make Respector generate imprecise specifications or miss generating them. First, Respector fails to generate all the specifications when endpoint methods use classes or methods that are out-of-scope of the analysis. For example, Respector exhibits lower recall in detecting parameters and responses in the enviroCar and Kafka APIs because their endpoint methods invoke methods that are external to the API code. Further, SpringFox allows API developers to create a configuration file that lists additional parameters that are bound to endpoint methods at runtime [45]. SpringFox can thus generate these parameters while Respector fails as it does not analyze the configuration files. As developers are aware about these parameters, their tests also include them and therefore AppMap is also able to detect them. Second, Respector may generate incorrect constraints when an endpoint method has too many nested conditionals such that the total number of paths exceeds the preset threshold to handle the path explosion problem. This occurred for the Senzing API whose endpoint methods has $> 2^{20}$ paths. Third, Respector fails to extract constraints that cannot be expressed using the SMT solver's vocabulary, e.g., in Ohsome, Respector failed to generate constraints because it could not represent string operation *splitParamOnComma* using Z3. Our evaluation shows that these limiting scenarios occur less frequently in practice and overall, Respector shows promising results. Further, some of these limitations (e.g., analyzing external libraries) can be addressed by additional engineering efforts.

4.4 Limitations and Threats to Validity

Respector inherits the limitations of static analysis, which include path explosion when analyzing endpoints with many nested conditionals and being unable to generate specifications when the APIs handle endpoints dynamically. Further, when API implementations use specific Java features such as type erasure or interfaces, Respector's precision drops because the information required to generate specifications (e.g., data types) is lost during compilation. Finally, Respector prototype depends on what the Z3 solver and Soot implementations support and does not analyze code in external libraries that prevents Respector in generating all the specifications.

	endpoint method					endpoint parameter					parameter constraint					endpoint response				
API	Respector	AM	SC	SD	SF	Respector	AM	SC	SD	SF	Respector	AM	SC	SD	SF	Respector	AM	SC	SD	SF
Features-Service	18	17	18	NA	NA	35	0	35	NA	NA	0	NA	0	NA	NA	18	NA	7	NA	NA
RESTCountries	27	✠	22	NA	NA	35	NA	34	NA	NA	12	NA	2	NA	NA	25	NA	0	NA	NA
Senzing	34	8	✠	NA	NA	154	11	✠	NA	NA	14	NA	✠	NA	NA	34	NA	✠	NA	NA
CatWatch	14	5	NA	8	✗	32	14	NA	8	✗	3	NA	NA	1	✗	10	NA	NA	4	✗
cwa	5	5	NA	✗	✗	13	0	NA	✗	✗	0	NA	NA	✗	✗	5	NA	NA	✗	✗
OCVN	278	✠	NA	✗	190	5,002	✠	NA	✗	2,834	0	NA	NA	✗	0	278	NA	NA	✗	160
Ohsome	159	63	NA	159	159	1,914	56	NA	1,914	1,608	0	NA	NA	0	0	261	NA	NA	261	261
ProxyPrint	75	7	NA	✠	✠	146	0	NA	✠	✠	0	NA	NA	✠	✠	75	NA	NA	✠	✠
Quartz	14	10	NA	✠	✠	3	0	NA	✠	✠	0	NA	NA	✠	✠	14	NA	NA	✠	✠
Ur-Codebin	6	3	NA	6	✗	14	0	NA	14	✗	2	NA	NA	0	✗	6	NA	NA	5	✗
Total	630	118	40	173	349	7,348	81	69	1,936	4,442	31	—	2	1	0	726	—	7	270	421

"✗": technique could not generate API specification; "✠": API could not be run/tests failed; "NA": technique not applicable.

Figure 8: Comparing Respector with four (AppMap (AM) [38], Swagger Core (SC) [54], springdoc-openapi (SD) [52], SpringFox (SF) [44]) state-of-the-art API specification generation techniques that use API implementation to generate specifications.

We address the threat to external validity by evaluating Respector on 15 diverse real-world APIs. A recent study [24] found that having more REST case-studies to evaluate the new approach is an open challenge as running APIs on local machines for experimentation has non trivial setup costs and can take a significant amount of time to find and setup a large number of REST APIs for experimentation. This finding is consistent with our experience of creating the evaluation dataset for our study. As mentioned in Section 4.1, our selection criteria only required that APIs use Spring Boot or Jersey, have developer-provided specifications and compile successfully because the Respector prototype currently supports the Spring Boot and Jersey frameworks, and takes bytecode as input. Since these criteria focus only on general aspects of the APIs, we believe that they should not bias the results against or in favor of any specific tool we consider in our analysis. We address the threat to internal validity by multiple authors independently analyzing Respector-generated specifications' accuracy using the developer-provided specifications and the API source code, and then reconciling their analysis results. Finally, we mitigate bugs in our code by testing Respector on dummy APIs and making our artifacts available to enable replication of our results.

5 RELATED WORK

API specification format. While OpenAPI [33] is commonly used to describe REST APIs, there exist other languages such as RESTful API Modeling Language [58] and API Blueprint [2] to describe APIs in a human-readable format for which Respector can be extended.

API specification generation techniques. Several techniques (e.g. SpringFox [44], BlueBird [8], ramlo [60], Talend [49], Swagger Core [54], Swagger Inspector [55], AppMap [38], ExpressO [48], springdoc-openapi [52], ApiCarv [59]) automatically generate OASs. However, they require developers to perform additional steps to generate relatively simple specifications that developers need to manually enhance. For example, Swagger Core [54] analyzes technique-specific and Spring Boot annotations in API source at runtime to generate a simple OAS that does not describe all the responses and parameter constraints. Swagger Inspector [55] and AppMap [38] generate API specifications from the requests/responses sent/received to/from the API endpoints by manually invoking endpoints or running API tests, respectively. ExpressO [48] generates specification for JavaScript APIs using Express framework by running the APIs in an isolated environment to identify their endpoints and responses, and using Express's structure to detect parameters. ApiCarv [59] uses UI tests of APIs to generate OASs by inferring endpoints dynamically and deriving parameters from the endpoint URIs, and responses from the execution of the endpoints. Respector outperforms these techniques by generating richer OASs containing both basic and complex parameter constraints (that can and cannot be expressed in OpenAPI) and interdependent endpoint methods without requiring any manual effort. Respector also complements tools such as Postman [46], Apiary [4], Stoplight [53], Dredd [3], and EvoMaster [6], which allow users to design, build, model, test, and validate APIs using their specifications.

Code analysis for REST APIs. Prior research has explored using static analysis and symbolic execution to detect interfaces in servlets [28, 29]. However, REST APIs often have more complex request and response formats that often use structured data formats such as JSON or XML, and require more sophisticated parsing and analysis than servlets. Finally, Respector addresses all the eight limitations of existing API documentation and code analysis approaches in identifying parameter constraints [26] and improves upon the state-of-the-art of constraint extraction techniques.

6 CONCLUSION

We presented Respector, the first static-analysis-based approach for automatically generating REST API specifications from API implementations. Respector performs well in practice and can generate specifications for real-world APIs with hundreds of endpoints. Our evaluation shows that Respector can be effective at generating specifications, can find previously unknown inconsistencies in mature APIs, and can improve upon alternative state-of-the-art techniques.

DATA AVAILABILITY

All of our data, source code, and documentation to reproduce our results are available at https://archive.softwareheritage.org/browse/origin/https://github.com/nntzuekai/Respector.

ACKNOWLEDGMENTS

This work was partially supported by NSF, under grant CCF-0725202, DOE, under contract DE-FOA-0002460, and gifts from Facebook, Google, IBM Research, and Microsoft Research.

REFERENCES

[1] Deutsche Telekom AG. 2022. Corona-Warn-App Verification Server. https://github.com/EMResearch/EMB/tree/master/jdk_11_maven/em/embedded/rest/cwa-verification. [Online; accessed March-2023].

[2] Apiary. 2020. API Blueprint. https://apiblueprint.org/documentation/specification.html. [Online; accessed March-2023].

[3] Apiary. 2021. Dredd. https://github.com/apiaryio/dredd. [Online; accessed March-2023].

[4] Apiary. 2023. Apiary. https://apiary.io. [Online; accessed March-2023].

[5] APILayer. 2021. REST Countries. https://github.com/EMResearch/EMB/tree/master/jdk_8_maven/cs/rest/original/restcountries. [Online; accessed March-2023].

[6] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Trans. Softw. Eng. Methodol.* 28, 1 (2019), 37 pages. https://doi.org/10.1145/3293455

[7] Andrea Arcuri and Juan P. Galeotti. 2021. Enhancing Search-Based Testing with Testability Transformations for Existing APIs. *ACM Trans. Softw. Eng. Methodol.* 31, 1 (2021), 34 pages. https://doi.org/10.1145/3477271

[8] Djordje Atlialp and Mathias Polligkeit. 2019. BlueBird. https://github.com/KittyHeaven/blue_bird. [Online; accessed March-2023].

[9] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, Vol. 13. 14.

[10] Andrei Ciortea, Olivier Boissier, Antoine Zimmermann, and Adina Magda Florea. 2017. Give Agents Some REST: A Resource-Oriented Abstraction Layer for Internet-Scale Agent Environments. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems* (São Paulo, Brazil) *(AAMAS '17)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1502–1504.

[11] Senzing community. 2022. Senzing REST API. https://github.com/Senzing/senzing-api-server. [Online; accessed March-2023].

[12] Oracle Cooperation. 2023. Package javax.ws.rs. https://docs.oracle.com/javaee/7/api/javax/ws/rs/package-summary.html. [Online; accessed July-2023].

[13] Oracle Corporation. 2017. JSR 370: Java API for RESTful Web Services (JAX-RS 2.1) Specification. https://download.oracle.com/otn-pub/jcp/jaxrs-2_1-final-eval-spec/jaxrs-2_1-final-spec.pdf. [Online; accessed March-2023].

[14] Treasure Data. 2023. Digdag. https://github.com/treasure-data/digdag. [Online; accessed March-2023].

[15] Inc. DataStax. 2022. Management API for Apache Cassandra. https://github.com/k8ssandra/management-api-for-apache-cassandra. [Online; accessed March-2023].

[16] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[17] Jakarta EE. 2022. Jarkata Servlet. https://projects.eclipse.org/projects/ee4j.servlet. [Online; accessed May-2023].

[18] The enviroCar project. 2022. enviroCar Server. https://github.com/enviroCar/enviroCar-server. [Online; accessed March-2023].

[19] Mathew Estafanous. 2022. Ur-Codebin. https://github.com/Mathew-Estafanous/Ur-Codebin-API. [Online; accessed March-2023].

[20] R. Fielding, M. Nottingham, and J. Reschke. 2022. *HTTP Semantics*. RFC 9110. https://httpwg.org/specs/rfc9110.html

[21] Fabio Formosa. 2022. Quartz Manager. https://github.com/fabioformosa/quartz-manager. [Online; accessed March-2023].

[22] Eclipse Foundation. 2023. Eclipse Jersey. https://eclipse-ee4j.github.io/jersey/. [Online; accessed March-2023].

[23] Development Gateway. 2017. Open Contracting Vietnam (OCVN). https://github.com/EMResearch/EMB/tree/master/jdk_8_maven/cs/rest-gui/ocvn. [Online; accessed March-2023].

[24] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Transactions on Software Engineering Methodology* 33, 1 (nov 2023), 41 pages. https://doi.org/10.1145/3617175

[25] Gravitee.io. 2023. Gravitee.io API Management. https://github.com/gravitee-io/gravitee-api-management. [Online; accessed March-2023].

[26] Henk Grent, Aleksei Akimov, and Maurício Aniche. 2021. Automatically Identifying Parameter Constraints in Complex Web APIs: A Case Study at Adyen. In *43rd International Conference on Software Engineering: Software Engineering in Practice*. IEEE Press, 71–80. https://doi.org/10.1109/ICSE-SEIP52600.2021.00016

[27] GIScience Research Group and HeiGIT. 2023. Ohsome API. https://github.com/GIScience/ohsome-api. [Online; accessed March-2023].

[28] William G.J. Halfond, Saswat Anand, and Alessandro Orso. 2009. Precise Interface Identification to Improve Testing and Analysis of Web Applications. In *International Symposium on Software Testing and Analysis*. 285–296. https://doi.org/10.1145/1572272.1572305

[29] William G. J. Halfond and Alessandro Orso. 2007. Improving Test Case Generation for Web Applications Using Automated Interface Discovery. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 145–154. https://doi.org/10.1145/1287624.1287646

[30] Ruanqianqian (Lisa) Huang, Ayana Monroe, Nikolaj Bjørner, and Peli de Halleux. 2023. Z3 Documentation: Simplifiers Summary. https://microsoft.github.io/z3guide/docs/strategies/simplifiers-summary. [Online; accessed Nov-2023].

[31] Confluent Inc. 2023. Kafka REST Proxy. https://github.com/confluentinc/kafka-rest. [Online; accessed March-2023].

[32] The Zalando Incubator. 2018. CatWatch. https://github.com/EMResearch/EMB/tree/master/jdk_8_maven/cs/rest/original/catwatch. [Online; accessed March-2023].

[33] OpenAPI Initiative. 2021. OpenAPI Specification. https://spec.openapis.org/oas/latest.html. [Online; accessed March-2023].

[34] OpenAPI Initiative. 2021. Schema Properties in OpenAPI 3.0. https://spec.openapis.org/oas/v3.0.0#properties. [Online; accessed March-2023].

[35] Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad, Arjen Poutsma, Chris Beams, Tareq Abedrabbo, Andy Clement, Dave Syer, Oliver Gierke, Rossen Stoyanchev, Phillip Webb, Rob Winch, Brian Clozel, Stephane Nicoll, Sebastien Deleuze, Jay Bryant, and Mark Paluch. 2022. Spring Framework Documentation. https://docs.spring.io/spring-framework/reference/. [Online; accessed March-2023].

[36] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2023. Adaptive REST API Testing with Reinforcement Learning. In *International Conference on Automated Software Engineering (ASE)*. arXiv:2309.04583

[37] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. In *International Symposium on Software Testing and Analysis (ISSTA)*. 289–301. https://doi.org/10.1145/3533767.3534401

[38] Elizabeth Lawler, Kevin Gilpin, Brian Kelly, Dustin Byrne, Dan Warner, Alan Potter, Rafal Rzepecki, Laurent Christophe, Ty Paulhus, and Adam Trotta. 2022. AppMap. https://appmap.io/. [Online; accessed March-2023].

[39] Hongjun Li. 2011. RESTful Web service frameworks in Java. In *International Conference on Signal Processing, Communications and Computing (ICSPCC)*. 1–4. https://doi.org/10.1109/ICSPCC.2011.6061739

[40] Alberto Martin-Lopez, Andrea Arcuri, Sergio Segura, and Antonio Ruiz-Cortés. 2021. Black-Box and White-Box Test Case Generation for RESTful APIs: Enemies or Allies?. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 231–241. https://doi.org/10.1109/ISSRE52982.2021.00034

[41] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2022. Online Testing of RESTful APIs: Promises and Challenges. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 408–420. https://doi.org/10.1145/3540250.3549144

[42] Javier Mu noz Ferrara. 2016. Features Model MicroService. https://github.com/EMResearch/EMB/tree/master/jdk_8_maven/cs/rest/original/features-service. [Online; accessed March-2023].

[43] OpenAPI Initiative. 2023. OpenAPI 3.0 Data Types. https://swagger.io/docs/specification/data-models/data-types/. [Online; accessed March-2023].

[44] Marty Pitt, Dilip Krishnan, and Adrian Kelly. 2020. SpringFox. https://github.com/springfox/springfox. [Online; accessed March-2023].

[45] Marty Pitt, Dilip Krishnan, and Adrian Kelly. 2020. SpringFox Documentation: Docket Spring Java Configuration. http://springfox.github.io/springfox/docs/snapshot/#docket-spring-java-configuration. [Online; accessed Nov-2023].

[46] Postman. 2023. Postman. https://www.postman.com. [Online; accessed March-2023].

[47] ProxyPrint. 2016. proxyprint-kitchen. https://github.com/EMResearch/EMB/tree/master/jdk_8_maven/cs/rest/original/proxyprint. [Online; accessed March-2023].

[48] Alessandro Romanelli, Souhaila Serbout, and Cesare Pautasso. 2022. ExpressO: From Express.js implementation code to OpenAPI interface descriptions. In *European Conference on Software Architecture (ECSA)*. Springer.

[49] Talend S.A. 2023. Talend. https://www.talend.com. [Online; accessed March-2023].

[50] Spring. 2022. Spring Boot. https://spring.io/projects/spring-boot. [Online; accessed March-2023].

[51] Spring. 2023. Spring Framework 6.0.11 API. https://docs.spring.io/spring-framework/docs/current/javadoc-api/. [Online; accessed July-2023].

[52] springdoc. 2023. springdoc-openapi. https://github.com/springdoc/springdoc-openapi. [Online; accessed March-2023].

[53] Stoplight. 2023. Stoplight. https://stoplight.io/studio. [Online; accessed March-2023].

[54] Swagger. 2023. Swagger Core. https://github.com/swagger-api/swagger-core. [Online; accessed March-2023].

[55] Swagger. 2023. Swagger Inspector. https://inspector.swagger.io/. [Online; accessed March-2023].

[56] David Tang. 2021. Nested Resource URL Paths and Relationship Links. *Pro Ember Data: Getting Ember Data to Work with Your API* (2021), 87–91.

[57] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java Bytecode Optimization Framework. In

CASCON First Decade High Impact Papers (Toronto, Ontario, Canada). USA, 214–224. https://doi.org/10.1145/1925805.1925818

[58] RAML Workgroup. 2020. RAML. https://raml.org. [Online; accessed March-2023].

[59] Rahulkrishna Yandrapally, Saurabh Sinha, Rachel Tzoref-Brill, and Ali Mesbah. 2023. Carving UI Tests to Generate API Tests and API Specification. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1971–1982. https://doi.org/10.1109/ICSE48619.2023.00167

[60] Kamil Zasada and Michał Myśliwiec. 2016. ramlo. https://github.com/PGSSoft/ramlo. [Online; accessed March-2023].

[61] Man Zhang and Andrea Arcuri. 2023. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 144 (sep 2023), 45 pages. https://doi.org/10.1145/3597205