Server Interface Descriptions for Automated Testing of JavaScript Web Applications

Casper S. Jensen
Aarhus University
Denmark
semadk@cs.au.dk

Anders Møller
Aarhus University
Denmark
amoeller@cs.au.dk

Zhendong Su
University of California, Davis
USA
su@cs.ucdavis.edu

ABSTRACT

Automated testing of JavaScript web applications is complicated by the communication with servers. Specifically, it is difficult to test the JavaScript code in isolation from the server code and database contents. We present a practical solution to this problem. First, we demonstrate that formal server interface descriptions are useful in automated testing of JavaScript web applications for separating the concerns of the client and the server. Second, to support the construction of server interface descriptions for existing applications, we introduce an effective inference technique that learns communication patterns from sample data.

By incorporating interface descriptions into the testing tool Artemis, our experimental results show that we increase the level of automation for high-coverage testing on a collection of JavaScript web applications that exchange JSON data between the clients and servers. Moreover, we demonstrate that the inference technique can quickly and accurately learn useful server interface descriptions.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Languages

Keywords

Web applications, automated testing, interface descriptions

1. INTRODUCTION

Many modern web applications run in browsers as HTML-embedded JavaScript programs that communicate with a server. The JavaScript code reacts to user events and asynchronously sends HTTP requests to the server for updating or retrieving data. The response from the server is used for example to dynamically modify the HTML page. With this so-called Ajax style of structuring web applications, the server mostly acts as a central database seen from the client's point of view. The server interface comprises a collection of operations, identified by URLs, that accept input and produce output typically in XML or JSON data formats.

Some web service providers have public APIs, such as Google, Twitter, and Facebook, that are well documented and used by many client applications, for example in mashups that each use small parts of different APIs. In contrast, in many other web applications, the server-side and the client-side are developed in conjunction within the same organization. In such web applications, the programming interface of the server is often not described in a formal way, if documented at all. This can make it difficult to modify or extend the code, even for small web applications. More concretely, we have observed that it limits the possibility of applying automated testing on the JavaScript code in isolation from the server code and database contents.

It is well known that precisely specified interfaces can act as contracts between the server code and the client code, thus supporting a clean separation of concerns and providing useful documentation for the developers. In this work, we show that having formal descriptions of the programming interfaces of the server code in Ajax web applications is instrumental when conducting automated testing of the JavaScript code in such applications. In addition, we present a technique for automatically learning server interface descriptions from sample data for pre-existing web applications.

As an example, consider the JavaScript code in Figure 1, which is part of a web application that manages attendance lists for meetings. When the function `goto_page` is called, an Ajax request is

```
1  function goto_page(id, q) {
2    jQuery.ajax(GET_PAGE_URL + '?page=' + id +
3                '&query=' + q,
4      {'dataType': 'json',
5       'success': function(result) {
6          populate_table(result);
7      }});
8  }
9
10 function populate_table(attendees) {
11   var table = $('#attendees')
12   table.html('');
13   for (i = 0; i < attendees.length; i++) {
14     var a = attendees[i];
15     var style = '';
16     if (a.checkedin) {
17       style = ' style="background-color: #B6EDB8;"';
18     }
19     ahtml = '<tr id="row' + a.id + '"' + style + '>' +
20       '<td><b>' + a.name + '</b> - ' +
21          a.email + '<br/>' +
22          a.department + '</td>' +
23       '<td><a href="#" onclick="info(' + a.id + ')">' +
24         '[info]</a> ' +
25       '<a href="#" onclick="checkin(' + a.id + ')">' +
26         '[checkin]</a> ' +
27       '<a href="#" onclick="del(' + a.id + ')">' +
28         '[delete]</a>' +
29       '</tr>';
30     table.append(ahtml);
31   }
32 }
```

Figure 1: A typical example of Ajax in JavaScript.

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
ACM 978-1-4503-2237-9/13/08
http://dx.doi.org/10.1145/2491411.2491421

```
[{"id": 6451, "name": "Fred", "email": "fred@cs.au.dk",
  "department": "CS", "checkedin": true},
 {"id": 4358, "name": "Joe", "email": "joe@imf.au.dk",
  "department": "IMF", "checkedin": false}]
```

Figure 2: Example JSON response for the Ajax interaction from Figure 1.

sent to the server via the jQuery library.[1] This request takes the form of an HTTP GET request with a specific URL and the parameters `page` and `query`. The `dataType` value `'json'` on line 4 indicates that the response data is expected to be formatted using JSON, a widely used format because it integrates well with JavaScript.[2] When the response data arrives, the function `populate_table` is called via line 6. By inspecting that function (lines 10–32) we see that the JSON data is expected to consist of an array of objects with specific properties: `id`, `name`, `email`, `department`, and `checkedin`. Moreover, their values cannot be arbitrary. For example, the `checkedin` property is used in a branch condition, so it probably holds a boolean value, and the other properties appear to hold strings that should not contain special HTML characters, such as < or &, since that could lead to malformed HTML when inserted into the page on line 30. Figure 2 shows an example of an actual JSON response that may appear.

In this example—as in many JavaScript web applications in general—the interface between the server and the client is not made explicit. As a consequence, the server code and the client code become tightly coupled, so it becomes difficult to change either part without carefully checking the consequences to the other part. For instance, the server code could safely omit the `checkedin` property when the value is `false` without breaking the client code, since `a.checkedin` on line 16 would then evaluate to `undefined`, which is coerced to `false`, however, the necessity for such subtle reasoning makes the application fragile to modifications. Also, the client code implicitly assumes that escaping of special HTML characters has been performed on the server, but this may not have been communicated to the server programmer.

One aim of our work is to advocate the use of formal interface descriptions as contracts between the client code and the server code. In the example above, an interface description could specify what are valid request parameters and the details of the response data format, such that the server code and the client code to a larger extent can be developed separately. Interface descriptions are the key to solve a substantial practical problem that we have observed in our work related to the tool Artemis that performs automated testing of JavaScript web applications [2]: It can be difficult to set up servers and populate databases to be able to test the client-side JavaScript code. Moreover, an automated tester, that focuses on testing the JavaScript code and has a black-box view on the server, is often not able to produce high coverage tests within a given time budget. With interface descriptions, we can automatically construct mock servers that can be integrated into such an automated tester in place of the real servers.

To illustrate this idea, consider again the example from Figure 1. If we wish to apply automated testing to the JavaScript code, two approaches could be considered at first: (1) We could ignore the server and simply assume that any response is possible to any Ajax request. Automated testing could then reveal that the JavaScript code will throw a runtime exception if the response data is not an array or if the array contains a null value (on line 13 and line 16, respectively), and malformed HTML would be generated if the object properties contain special HTML characters. However, this does not imply that there are errors in the JavaScript code—implicitly it may be the server's responsibility to ensure that the Ajax response does not contain such values. (2) Alternatively, we could use a live server with realistic database content. This would eliminate the problem with false positives in the first approach. However, two drawbacks arise: first, it requires deep insight into the application to be able to provide realistic database content [6]; second, the testing capabilities become fixed to that particular database content, which may limit the coverage of the client code. Interface descriptions give us another alternative: (3) With a description of what requests the server accepts and the responses it may produce, an automated testing tool such as Artemis becomes able to focus on testing the JavaScript code on meaningful inputs.

To alleviate the burden of writing interface descriptions for pre-existing applications, we additionally propose an automated learning technique. Our hypothesis is that practically useful interface descriptions can be created using only sample request and response data. The sample data can be obtained by users exercising the functionality of the application without requiring detailed knowledge of the server code. This makes the learning technique independent of the specific programming languages and frameworks (PHP, JSF, ASP.NET, Ruby, etc.) that may be used on the server and thereby be more generally applicable.

The idea of using interface description languages (IDLs) to specify the interfaces of software components has proven successful in many other contexts. Prominent examples in related domains include Web IDL for the interface between browsers and JavaScript application code [10], WSDL for web service communication [8], and OMG IDL for interprocess communication with CORBA [24]. Nevertheless, IDLs are still not widely used in the context of client-server interactions in Ajax web applications, despite the existence of languages, such as WADL [13]. We suspect that one reason is that writing the interface descriptions is a laborious task. To this end, our work is the first to propose an automatic technique to learn interface descriptions for Ajax web applications.

In summary, our contributions are as follows:

- We first introduce a simple Ajax server interface description language, named *AIL*, inspired by WADL (Section 2). This language can describe HTTP operations involving JSON and XML data as commonly seen in Ajax web applications.
- We demonstrate how the interface descriptions can be incorporated into automated testing of JavaScript web applications to be able to test client code without involving live servers. Specifically, we extend the automated testing tool Artemis with support for AIL (Section 3) by introducing a generic mock server component that is configured using AIL descriptions.
- We provide an algorithm for learning AIL descriptions of Ajax web applications through dynamic analysis of network traffic between clients and servers (Section 4).
- We experimentally evaluate our approach by investigating how AIL descriptions affect the code coverage obtained by Artemis with our extensions and by comparing the inferred AIL descriptions with manually crafted ones (Section 5). Our results show that (1) by using the descriptions, Artemis can obtain as good coverage with the mock server as with real servers and manually populated databases and (2) the learning algorithm is capable of producing useful AIL descriptions.

Testing Ajax applications is recognized as a difficult problem [20, 22] and interface descriptions have proven useful for testing classical web applications [1, 15–18, 21], but no previous work has combined interface descriptions and testing of Ajax applications. Related work on interface description languages, learning algorithms, and automated testing is discussed in Section 6.

[1] `http://jquery.com/`
[2] `http://json.org/`

```
URL http://www.example.org
GET news/read():@items.json
GET author(name:*):@author.json
GET users/login(user:*, pwd:*):@token.json
POST news/send(token:@token.json, items:+@item.json):void
POST users/create(token:@token.json, user:*, pwd:*):void
```

Figure 3: An example AIL description.

In this paper, we use the term Ajax [12] in a broad sense, covering different technologies for client-server communication in JavaScript-based web applications. In current web applications this typically involves the XMLHttpRequest[3] API, but our general approach in principle also encompasses the more recent WebSocket[4] API. The data being transmitted may involve different formats including XML and JSON that are supported by AIL, although our current learning algorithm and experiments focus on JSON.

2. AN INTERFACE DESCRIPTION LANGUAGE FOR AJAX

Our first step is to design a formal language, *AIL* (Ajax server Interface description Language), for describing the interfaces of servers in Ajax-style web applications. The communication in such web applications consists of HTTP client-server interactions where the JavaScript code running on an HTML page in a browser sends requests and receives responses. An HTTP request contains a method (typically GET or POST), a URL path, and a number of name-value parameter pairs. For simplicity, we abstract away the other information in the HTTP requests, such as the protocol and headers. We design AIL as a simple language that concisely captures the essence of WADL [13] and integrates with JSON.

An AIL description consists of a base URL and a collection of *operation descriptors*, each of the form

request : *response*

where *request* is a pattern that describes a set of possible HTTP requests, and *response* describes the possible responses that may be generated by the server for those requests. Within an AIL description, the request patterns must be disjoint in the sense that every possible HTTP request can match at most one of the patterns, which ensures a deterministic behavior.

An AIL description establishes a contract between the clients and the server: The clients are responsible for ensuring that each request matches one of the request patterns, and it is the server's responsibility that the response matches the corresponding response schema. Below we describe the syntax and matching semantics of request patterns and response schemas.

Figure 3 shows an AIL description (without JSON Schema files) for a simple JSON news server that makes five operations available for JavaScript applications. The first three operations provide access to news items, author information, and authentication. The last two operations can be used for submitting news items to the server and for registering new users. All operations use HTTP and JSON. The description refers to external JSON Schema files that specify the data formats involved in the operations. Such an AIL description evidently characterizes the structure of the operations that are supported by the server while abstracting away from the actual data being transmitted at runtime.

The initial version of AIL supports two kinds of data formats: JSON and XML. AIL simply relies on JSON Schema[5] and RELAX NG[6] for describing the structure of data.

[3] `http://www.w3.org/TR/XMLHttpRequest/`

[4] `http://www.w3.org/TR/websockets/`

[5] `http://json-schema.org/`

[6] `http://relaxng.org/`

```
POST comment/delete/comment_id:*() : @delete.json

POST project_id:*/issues/issue_id:*/set/title/
     value/value:*() : @set_title.json

POST project_id:*/scrum/add/task/for/story/story_id:*/
     mode/issue(task_name:*) : @add_task.json
```

Figure 4: Parts of the AIL description of *The Bug Genie*.

A request pattern consists of an HTTP *method* (`GET`, `POST`, etc.), a *path*, and a comma-separated list of *parameters*:

method path(*parameters*)

A path consists of *path fragments* separated by '/', each being a string or a parameter. Each parameter has the form *name*:*cardinality datatype*, where *cardinality* is either absent (meaning precisely one occurrence), '?' (optional) or '+' (zero or more).

A *datatype* is written as a constant string (e.g. `"start"`), the wildcard '*', the keyword `void`, a reference to an external JSON Schema file (e.g. `@token.json`), a reference to a RELAX NG schema file (e.g. `@person.rng`) or a type defined in such a file (e.g. `@types.rng#person`), or a list of datatypes separated by '|'. The datatypes of parameters that occur in paths are restricted to simple string types, such as numerals or string enumerations, and the special datatype `void` is never used in request patterns.

A datatype matches strings in the obvious way: a constant string matches that string and no others, * matches any value, `void` is used for describing the empty response, a reference to a schema type matches according to the semantics of JSON Schema and RELAX NG, respectively, and '|' represents union. An HTTP request matches a request pattern if each constituent matches. A response pattern is simply a datatype with matching defined accordingly. We omit the precise semantics of pattern matching due to the limited space, but the intuition should be clear from this brief description.

The example shown in Figure 4 is a part of an AIL description of the application *The Bug Genie*.[7] This application uses REST-style naming where some parameters appear in the path, not in the HTTP request body or the query string. The responses use JSON in both application; we omit the details of the associated JSON schemas.

The AIL language as presented above is capable of expressing the basic properties of server interfaces. One straightforward extension is to support other data formats, such as, HTML, plain text, or JSONP (JSON with padding), credentials for HTTP Basic authentication, and request content types (i.e. MIME types). In some situations it can also be useful both for documentation and testing purposes to describe error responses, that is, non-"200 OK" HTTP response codes, and HTTP content negotiation. For now, AIL cannot describe temporal properties, for example that operation A must be invoked before operation B, simply because such properties have not been significant in any of the web applications we have studied. Another possible extension is support for WebSockets, which unlike HTTP involves connection-oriented communication and thereby does not fit directly into the simple request-response model.

3. USING SERVER INTERFACE DESCRIPTIONS IN AUTOMATED TESTING

Server interface descriptions are not only useful for documenting the server interface for the client programmer; they also make it possible to test the client code in isolation from the server code, which provides new opportunities for practical automated testing. In Section 3.1 we give a brief overview of the Artemis tool from earlier work by Artzi et al. [2], with a focus on the complications caused by Ajax communication. In Section 3.2 we show how a new

[7] `http://www.thebuggenie.com/`

mock server component can exploit AIL descriptions to improve the level of automation.

3.1 Automated Testing with Artemis

A JavaScript web application is driven by events, such as the initial page load event, mouse clicks, keyboard presses, and timeouts. Event handlers are executed single threaded and non-preemptively. A test input to an application is thus given by a sequence of parameterized events. Of particular relevance here are the events that are triggered by Ajax response where the event parameter contains the HTTP response data from the server.

Figure 5 shows a use of the XMLHttpRequest API[8], which provides low-level Ajax functionality (in contrast to the example in Figure 1 that uses the jQuery library). The call to `x.send` on line 45 initiates the request to the server, in this case an HTTP GET request to the relative URL `news/read`, which matches the AIL description in Figure 3. An event handler for receiving the response is set up on line 36. When the response content has finished loading, `x.readyState` will have the value 4, and the event handler function is called. If the response status code is 200 the response content is then available as a text string in `x.responseText`. For this example, the challenge for an automated tester is how to produce meaningful server response data that will thoroughly exercise the response event handler including the `showItems` function being called on line 39.

The Artemis tool uses feedback-directed random testing of JavaScript web applications to produce high-coverage test inputs. That is, it executes the application on a test input and monitors the execution to collect interesting information that can be used to generate new test inputs to improve coverage. The heuristics used for collecting information, producing new test inputs, and prioritizing the work list of test inputs are described by Artzi et al. [2], and we here focus on the interactions with the server.

Although our goal is to test the JavaScript code, not the server, we face the problem that the JavaScript code in Ajax-style applications closely depends on the server. As discussed in Section 1 it is often difficult to populate the server database with useful data that is required to ensure high coverage of the JavaScript code. A simple example is line 16 in Figure 1, where both branches can only be covered if the server database contains a nonempty list of attendees, whereof at least one is marked as `checkedin` and another is not—no matter how other events, such as mouse clicks, are being triggered in the browser. On top of this, even a well populated database may not suffice. Reaching one part of the JavaScript code may require certain values in the database where another part may require different values, so *multiple* database snapshots may be necessary to enable high coverage of the JavaScript code, which makes the burden even higher.

Yet another problem for automated testing appears when important server responses can only be triggered by request values that are practically impossible to guess by the testing tool. Consider for example the operation `users/login` for the server described in Figure 3. A successful response requires the client to provide a valid user name and password, which is (hopefully) impossible to guess, so a considerable part of the client code will remain unexplored. A common workaround is to ask the user for help in such situations [3]. The consequence of these problems is that "automated" testing may require a considerable manual effort.

We observe that when testing client code, many execution paths require data from the server that is structurally well-formed but not necessarily semantically valid. As an example, for testing the `populate_table` response handler function in Figure 1, we do not need server response data that contains actual attendee names and email addresses, but we do need JSON data with a certain structure. This observation allows us to use AIL descriptions instead of actual servers and live data for testing client code.

[8] `http://www.w3.org/TR/XMLHttpRequest/`

```
33 function getNewsItems() {
34   var x = new XMLHttpRequest();
35   x.open("GET", "news/read");
36   x.onreadystatechange = function() {
37     if (x.readyState == 4)
38       if (x.status == 200) {
39         showItems(x.responseText);
40       } else {
41         alert("An error occurred :-(");
42       }
43     }
44   };
45   x.send(null);
46 }
```

Figure 5: A simple use of XMLHttpRequest to perform an Ajax call to get news items from the server from Figure 3.

3.2 Extending Artemis with an AIL Mock Server

To alleviate the problems described above, we have extended the Artemis tool with a mock server component that is configured by an AIL description. Whenever the JavaScript program under test initiates Ajax communication, the mock server intercepts the request such that the actual server is never contacted during the testing.

Given an HTTP request, the mock server performs the following steps: (1) It searches through the AIL description to find an operation descriptor with a request pattern that matches the HTTP request. If one is found, a random response that matches the corresponding response datatype is prepared; otherwise, the response to the client is a generic "404 Not Found". (2) The response data is then sent to the test input generator component in Artemis, which will subsequently produce new test inputs that include an Ajax response event containing the response data.

As result, we obtain a nondeterministic model of how the server may behave according to the AIL description, and the JavaScript code can be explored without the need of a real server and database.

Now, several observations can be made. First, using the mock server solves the problem of populating databases since it automatically returns a wide range of possible responses, as specified by the AIL description. This means that the client code is effectively tested on a variety of structurally meaningful inputs. The response data generated by the mock server may of course not be semantically valid, but as argued above, structurally correct response data will suffice for testing many properties of client code. This approach also elegantly handles the issue with the `users/login` operation mentioned above: the mock server component will skip the actual password check and automatically produce a meaningful response representing successful login.

Second, our approach makes it easy to model the asynchronous nature of Ajax, which is a source of intricate race errors [25, 28]: Even though the AIL mock server component only produces a single response for each request in step 1, the Artemis test input generator component may create multiple new inputs in step 2 to test different event orders.

Third, the construction of responses in step 1 may not be entirely random. We can exploit the existing feedback mechanism in Artemis such that information that has been collected by Artemis in previous executions of the JavaScript code with different test inputs can guide the selection of the new Ajax response data. Specifically, Artemis dynamically collects constants that are used in each event handler [2], and these constants are often good candidates for values in new event parameters, such as the Ajax response data.

4. AUTOMATIC LEARNING OF AIL DESCRIPTIONS

We have shown that AIL offers a simple, formal mechanism for documenting client-server communications in Ajax-style web applications and that AIL descriptions are useful in automated testing of the client code. However, despite the many advantages of having server interface descriptions, constructing such descriptions for pre-existing applications can be a nontrivial task. To support the construction of AIL descriptions, we show how to automatically learn descriptions from samples of client-server communication traffic through a black-box, dynamic approach. This approach has been chosen for generality and independence of particular server technologies used. We imagine that such a learning algorithm can be used when a development team realizes that their web applications have grown from being small and manageable to become larger and more difficult to maintain without proper separation of concerns and without the ability to apply automated testing techniques. Automated learning makes it easier to retrofit server interface descriptions to existing applications, thereby supporting automated testing for the further development of the applications.

We assume that the AIL descriptions being used in automated testing as described in Section 3 have been written manually or with support from the learning algorithm. The automatically generated descriptions may naturally require some manual adjustments since they are generated on the basis of sample data.

We first introduce our learning problem. The *input* I denotes a finite set of concrete HTTP request and response pairs $\langle r, s\rangle$. The *output* d denotes an AIL description that expresses a possibly infinite relation $[\![d]\!]$ of request and response pairs. Since we perform black-box learning, we assume that sufficient samples are available for learning.

Given a set of input samples, there are many valid AIL descriptions that "generalize" it. Thus, it is important to define which AIL descriptions are the most desired. At the high level, we want a learned AIL description to closely match the server programmer's view of the interface—a set of independent operations each with its own meaning and purpose. The central challenge is to identify these operations from the given observations without any white-box knowledge of the server and client.

To guide our learning algorithm, we specify the following desirable properties that a learned description should have:

completeness: The input I is covered by the learned AIL description d, i.e. $I \subseteq [\![d]\!]$.

disjointness: The request patterns of d must be disjoint.

precision: d should be as close to I as possible, i.e. $[\![d]\!] \setminus I$ should be small. We say that d_1 is *more precise* than d_2 iff $[\![d_1]\!] \setminus I \subseteq [\![d_2]\!] \setminus I$.

conciseness: d should be small. We say that d_1 is *more concise* than d_2 iff $|d_1| \leq |d_2|$, where $|d|$ denotes some appropriate notion of the size of an AIL description d.

With these properties in mind, we devise an algorithm to learn AIL descriptions from input samples. Our algorithm has two phases: *data clustering* and *pattern generation*. The data clustering phase is the key step, organizing the input samples into distinct clusters such that each cluster corresponds to a "likely" operation descriptor, and these together form an AIL description with the aforementioned properties. Once the appropriate clustering has been determined, the pattern generation phase transforms the clusters into actual AIL descriptions and JSON schemas. This last step is straightforward and will not be described in this paper due to space constraints.

For the clustering phase we make two observations. First, identifying responses that are structurally similar can be a good starting point. For example, two JSON values that have the same object structure but contain different strings or numbers can be considered "similar" and hence likely belong to the same operation. Second, we can infer important information for clustering from the path fragments and parameters that occur in the request data. As an example, consider requests to the first operation from Figure 4:

```
POST comment/delete/comment_id:*() : @delete.json
```

A request consists of path fragments and GET/POST parameters, which we will denote features. The features for this operation are three path fragments, i.e. the constant strings `comment` and `delete` and a comment ID value. These can be divided into *key features*, which are characterized by having relatively few possible values that together identify the operation for the request, and *non-key features*, with a higher number of possible values, which do not identify operations. For this particular operation, the key features are the first two, i.e. `comment` and `delete`, and we can expect that our sample data will contain a higher number of comment ID values than the number of distinct operations.

These observations motivate us to further divide the clustering phase into two steps: (1) construct an initial clustering by considering only the response data and grouping the responses into distinct clusters with respect to their response types (Section 4.1); and (2) restructure the clustering using request data by identifying the likely key features (Section 4.2). After the clustering phase, we construct AIL descriptions that satisfy the completeness property by ensuring that each sample is associated with a cluster and giving the cluster a request pattern and a response pattern that match all samples in the cluster.

4.1 Response Data Clustering

We first cluster the input set I using HTTP response data. Although AIL can describe both XML and JSON data, we describe our algorithm for JSON, which is the most widely used data interchange format for Ajax web applications. A JSON response is a JavaScript data structure containing primitive values (strings, numbers, booleans, and null), objects, and arrays.

For each request and response pair $\langle r, s\rangle \in I$, the response s contains JSON data. We map s to its *type abstraction*:

- a *primitive value* is mapped to its respective primitive type (e.g. `String`, `Number`, `Boolean`, or `Null`);
- an *object value* $\{p_1 : v_1, \ldots, p_k : v_k\}$ is mapped to a record type $\{p_1 : t_1, \ \ldots, \ p_k : t_k\}$ by replacing each object property value with its type, where t_i denotes the type of the value v_i; and
- an *array* $[v_1, \ldots, v_k]$ is mapped to a union type $\cup_{i=1}^{k} t_i$, where t_i denotes the type of v_i.

We now cluster all sample pairs from I according to structural equivalence of the response type abstractions. For example, the five sample responses shown in Figure 6 will be clustered together into three clusters. The first two samples have the same type abstraction {`id` : `Number`, `name` : `String`, `stories` : `Number`} and are thus grouped together, while the next two contains an additional property, resulting in the type abstraction {`id` : `Number`, `name` : `String`, `email` : `String`, `stories` : `Number`} and their own cluster. Similarly, the type abstraction of the last response {`id` : `Number`, `title` : `String`} does not match the first or the second cluster, so it will be placed in a third cluster.

4.2 Request Data Clustering

Using the distinction between key and non-key features, we want our learning algorithm to construct request patterns in which key features are represented using constant strings, and non-key features are represented using wildcards. However, deciding on the division between key and non-key features may require restructure of the clustering to ensure that the disjointness property is satisfied.

```
⟨GET author?name=alice,
 {"id": 1, "name": "Alice", "stories": 10}⟩
⟨GET author?name=bob,
 {"id": 2, "name": "Bob", "stories": 12}⟩
⟨GET author?name=charlie,
 {"id": 3, "name": "Charlie",
  "email": "charlie@example.org", "stories": 1}⟩
⟨GET author?name=eve,
 {"id": 3, "name": "Eve",
  "email": "eve@example.org", "stories": 1}⟩
⟨POST news/read,
 [{"id": 1, "title": "News 1"},
  {"id": 2, "title": "News 2"}]⟩
```

Figure 6: Example request and JSON response pairs, $\langle r, s \rangle$, for two different operations.

In the example shown in Figure 6, the first four responses are initially put into two clusters. If the `name` parameter is classified as a key feature, then we need to split the two clusters into four, one for each sample. On the other hand, if it is classified as a non-key feature, then we need to merge the two clusters into one to ensure disjointness. To generate the desired request patterns using constant strings and wildcards, this example shows that we may need to *merge* clusters together, using a wildcard, or *split* them into separate clusters, using constant strings.

To describe in more detail how we merge and split clusters, we first introduce some additional terminology. As stated, each path fragment and parameter of a request is a *feature*. The set of features in a request forms its *signature*, denoted by S. As an example, a request with URL `foo/bar` and a parameter `baz=1` has the signature $\{\texttt{\#0}, \texttt{\#1}, \texttt{\#baz}\}$ where path fragments are identified by their positions in the URL and parameters are identified by their names. Since we wish to construct one request pattern for each cluster, we first split clusters that contain requests with different signatures. Request patterns that are constructed from clusters with different signatures are trivially disjoint. Next, we need to decide on a suitable partition of S into key and non-key features, corresponding to constant strings and wildcards, respectively.

There are two obvious extremes when selecting the partition: (1) assign wildcards to all features, thereby merging all clusters with the same signature into a single one, which is likely to be highly imprecise, and (2) assign constant strings to all features, thereby splitting all clusters into singletons (i.e. simply the input set I), which is neither concise nor very useful. These two extremes relate to operation descriptions being concise and precise respectively, which are conflicting requirements that we must reconcile.

Our algorithm is given in Figure 7. Let D be initial response data clustering D from Section 4.1. For each signature S in D, the algorithm iterates over all clusters D' that match S. It then iterates over all possible partitions ρ that divide S into key and non-key features, selecting the partition with the *minimal cost* with respect to a cost function C. This partition is used to restructure the clusters D. The end result, after iterating over all signatures, is D restructured in accordance with its request data. What remains is to define the cost function $C(\rho, D')$, where ρ is a partition of S into key and non-key features and D' is a set of clusters with the same signature.

Recall our observation that clustering based on response types typically yields a good baseline clustering. Thus, we favor partitions that result in the smallest number of splits and merges compared to the baseline clustering. This strategy is further supported by the other observation that key features have few unique values, so our goal is to find a partition that leads to the smallest number of splits and merges.

```
function REQUESTDATACLUSTERING(D)
    for all S in SIGNATURES(D) do
        D' ← FINDCLUSTERSWITHSIGNATURE(D, S)
        c_min ← ∞
        ρ_min ← null
        for all ρ in PARTITIONS(S) do
            c ← C(ρ, D')
            if c < c_min then
                c_min ← c
                ρ_min ← ρ
            end if
        end for
        RESTRUCTURE(D, ρ_min)
    end for
end function
```

Figure 7: The request data clustering algorithm.

We define the cost $C(\rho, D')$ as the total number of splits and merges necessary to get from D' to the restructured clustering. Intuitively, a least cost partition helps avoid merging too much, for precision, and avoid splitting too much, for conciseness. In case of a tie, we choose a partition that minimizes the number of wildcards.

To illustrate the cost calculation, consider the two initial clusters that were created from the first four sample responses in Figure 6. Those two clusters are a result of different response structures, however, we cannot ensure disjointness of the request patterns without a reorganization. Both clusters have the signatures $S = \{\texttt{\#0}, \texttt{\#name}\}$ corresponding to the `author` URL path fragment and the `name` parameter. The cost function is applied to all possible partitions ρ of S, in this example the following four partitions:

1. Neither `#0` nor `#name` is considered a key feature, causing the two clusters to be merged at a cost of 1 into a cluster with request pattern `*?name=*`.
2. Only `#0` is a key feature, which also causes a single merge operation, hence the cost is 1, but the resulting cluster now has request pattern `author?name=*`.
3. Only `#name` is a key feature, which means that the two clusters are split into four singleton clusters at a total cost of 2, resulting in the four request patterns `*?name=alice`, `*?name=bob`, `*?name=charlie`, and `*?name=eve`.
4. Both `#0` and `#name` are key features, which also results in four singleton clusters at a total cost of 2, but the request patterns are now `author?name=alice`, `author?name=bob`, `author?name=charlie`, and `author?name=eve`.

We choose the second partition since it has minimal cost and minimal number of wildcards.

Finally, we have constructed clusters with the desired properties. Each cluster can be turned into an AIL operation descriptor, as hinted earlier. Its request pattern is generated from the employed partition ρ, and JSON schemas for the response patterns are generated from the type abstractions of the response samples in the cluster. The close connection between JSON schemas and the type abstraction we uses for response data leads to a straightforward construction.

5. EVALUATION

We have argued that server interface descriptions can provide separation of concerns, which enables testing of JavaScript code in isolation from the server code. When conducting automated testing of the JavaScript code, the use of AIL and a mock server removes the burden of setting up actual servers with appropriate database contents. To find out how this may influence other aspects of automated testing, we first consider the following research questions:

Benchmark	LOC	Client Framework	Server Language
simpleajax	79	jQuery	Python (Django)
resume	244	Flapjax	Python
globetrotter	347	jQuery	Java (JWIG)
impresspages	558	jQuery	PHP
buggenie	3,716	Prototype	PHP
elfinder	6,724	jQuery	PHP
tomatocart	8,817	Prototype	PHP
eyeos	17,629	jQuery	PHP

Figure 8: Benchmark applications.

Q1. How is the running time of automated testing affected when replacing the real server by the mock server for a fixed number of test inputs?

Q2. Does the use of AIL in place of live servers affect the code coverage obtained by automated testing?

To evaluate how our learning algorithm from Section 4 can be useful when creating AIL descriptions for existing applications, we consider two additional research questions:

Q3. To what extent is the learning algorithm capable of producing AIL descriptions that correctly describe the servers in actual JavaScript web applications?

Q4. How much effort is required for producing request and response data for the learning algorithm, and how fast is the learning algorithm?

To answer these questions we have implemented three tools:[9] (1) a web proxy for recording the HTTP communication between clients and servers, (2) the learning algorithm from Section 4, which reads the data recorded by the web proxy and outputs AIL descriptions and JSON schemas, and (3) the AIL mock server for Artemis.

We have collected 8 benchmark applications that use JavaScript for their client-side logic and Ajax for communicating between the client and the server, and where the source code for the entire application has been available, including the server code: *simpleajax* is a small home-built test application for event registrations; *resume*[10] is an application management system; *globetrotter*[11] is a travel application system; *impresspages*[12] is a CMS system; *elfinder*[13] is an online file explorer; *buggenie*[8] is a project management tool that we also used as example in Section 2; *tomatocart*[14] is an e-commerce platform; and *eyeos*[15] is an online desktop environment.

Figure 8 contains a list of the applications together with the number of lines of JavaScript code (excluding frameworks), the framework they use for JavaScript if any, and the language or framework used on the server side.

Our experiments are performed on a 3.1GHz i5 machine with 4GB of memory.

5.1 Using AIL in Automated Testing

To be able to answer Q1 and Q2 we run Artemis on our benchmark applications using various configurations: *EmptyDB* with real servers but with empty databases, *FullDB* with real servers where the databases are populated with realistic data, *Random* with a fully generic mock server that accepts all requests and produces random JSON responses, and *AIL* with the mock server using the AIL description.

[9] Our tools are available at `http://www.brics.dk/artemis/`

[10] old version of `https://resume.cs.brown.edu/cs/`

[11] `https://services.brics.dk/java/globetrotter/`

[12] `http://www.impresspages.org/`

[13] `http://elrte.org/elfinder`

[14] `http://www.tomatocart.com/`

[15] `http://eyeos.org/`

Benchmark	AIL	FullDB	Init	EmptyDB	FullDB	Random	AIL
simpleajax	25s	26s	22	30	62	60	62
resume	67s	77s	12	105	108	14	113
globetrotter	102s	94s	10	-	180	17	205
buggenie	104s	180s	662	-	1,322	1,138	1,308
elfinder	162s	152s	571	1,236	1,337	665	1,366

Figure 9: Execution time for Artemis with a budget of 100 test input executions, and code coverage obtained by Artemis with a budget of 300 test inputs.

The database contents used in the FullDB configuration are selected as snapshots obtained when we exercised the applications to collect sample request and response pairs. For the AIL configuration, we use manually constructed AIL descriptions, or equivalently, descriptions that were produced by the automated learning and subsequently manually adjusted to properly model the servers. Three of the larger benchmark applications are unfortunately beyond the capabilities of the latest version of Artemis for reasons that are unrelated to AIL and Ajax communication, so our experiments are conducted on the remaining five applications.

The execution time of Artemis depends on a number of factors, one of course being the time budget, which is expressed as a maximum number of test input executions. Other factors are the specific data that the JavaScript application receives from the server in Ajax interactions and the response time of the server. Replacing the live server with a mock server affects the two latter factors. Responses that are randomly generated from the AIL response patterns may trigger long running loops in the JavaScript code, however, the work performed by the mock server is presumably simpler than that of the real server in most cases.

The first columns in Figure 9 show the total running time of Artemis with the two configurations AIL and FullDB using a budget of 100 test input executions for each application. This gives an answer to Q1: for these applications, the running time is not affected notably by the AIL mock server.

The remaining columns show the code coverage (measured as number of lines of JavaScript code) for 300 test input executions of each application using all four configurations. The extra column, Init, shows the coverage obtained by loading the application without triggering any events, which can be viewed as a baseline for the coverage comparisons. The *globetrotter* and *buggenie* applications have not been tested with empty databases since this did not make sense for those cases. (Please note that the LOC column in Figure 8 should not be compared with the coverage numbers in Figure 9, since the latter only include lines with executable code.)

We observe that the use of the AIL mock server yields similar coverage results as when using the real servers populated with realistic data, which partially answers Q2.

For *globetrotter*, *elfinder*, and *resume*, coverage is slightly improved when using AIL. In each case, the increased coverage is caused by conditions in the JavaScript code that are only triggered with specific Ajax response data, for example an empty array or a certain boolean value somewhere in a JSON structure. These are examples of application behavior that depend on the precise contents of the server database, as discussed in Section 3. In *globetrotter*, for example, the program state describes a travel application that can be at different workflow stages. The mock server quickly generates JSON responses that cover all the workflow stages, while the FullDB configuration only manages to cover a single one.

The lower code coverage for *buggenie* is caused by an animation not being triggered in the AIL configuration due to the heuristics used internally by Artemis. For *elfinder*, a few lines are reached with FullDB but not with the AIL configuration. The data in this application contains a tree-like structure of files and directories that are linked through hash and parent hash values that refer to each

other. This invariant cannot be expressed with the current design of AIL, so the mock server is not able to produce the right response.

Several additional observations can be made from the coverage numbers. The EmptyDB, FullDB and AIL measurements show higher coverage than Init, demonstrating that we actually test additional functionality besides simply loading the page. Interestingly, the Random measurements for *resume*, *globetrotter*, and *elfinder* show considerably less coverage, which demonstrates that meaningful response data is important. In all cases, this is caused by the initialization of the web applications depending on correctly structured Ajax responses. As expected, populating the database (i.e. FullDB) results in higher or equal coverage than using the empty database (i.e. EmptyDB).

Although we did not expect to find bugs in the benchmark applications, the use of the AIL mock server revealed one in *resume* that was not found with any of the other configurations. The bug is triggered by a sequence of events that involve sending an empty array to the server and back to the client ending up in `obj.values` at the following snippet of code where it leads to a runtime error:

```
var ln =
  A({href:'javascript:undefined'},
  ''+obj.values[0]['number']+' - '
  +obj.values[obj.values.length-1]['number']);
```

This example illustrates how unclear assumptions between client and server developers can end up creating errors in the applications.

In other situations, similar unclear assumptions do not cause errors but lead to fragile code that may break in future revisions made by programmers who are not aware of subtle invariants that must be satisfied. The use of AIL in Artemis also revealed such a situation. The *elfinder* application contains the following snippet of code where `dir` and `files` originate from an Ajax response:

```
while (dir && dir.phash) {
  dir = files[dir.phash]
}
```

The purpose of this code is to traverse a directory structure where files are represented in an array indexed by file hash values. Running Artemis with the AIL configuration discovered that if this data structure contains loops then the `while` loop never terminates. The required invariant—that the data structure sent in the Ajax response never contains such loops—is not documented in the application source code. AIL is not expressive enough to capture such invariants, but one could argue anyway that the application would be more robust if its correctness did not depend on such intricate invariants involving the server state.

Concluding these experiments, our answer to Q2 is that the use of AIL leads to good coverage compared to using a server with an empty database, a server with a populated database, or a mock server that generates random responses. The experiments also pointed us to examples where correctness of the applications depends on subtle, undocumented invariants.

5.2 Automated Learning of AIL Descriptions

To obtain the training data for the learning algorithm, we install and exercise each application by manually clicking on visual elements and entering data into forms for a few minutes, while the web proxy monitors all Ajax communication. This is done without detailed knowledge of each application and entirely without looking at the server code. This gives us between 70 and 611 sample request and response pairs, depending on the amount of information exchanged.

We now run the learning algorithm on the data obtained for each application, which in each case takes less than a second. The request data clustering process described in Section 4.2 performs altogether 18 splits and 43 merges when searching for the partitions with the minimal cost. This results in a total of 130 AIL operation descriptors and 9,550 lines of JSON schema—all generated automatically.

Benchmark	Samples	Sampling	Learning	Match	$1{\rightarrow}N$	$N{\rightarrow}1$
simpleajax	70	3m	74ms	5	0	0
resume	128	9m	111ms	12	3	0
globetrotter	97	8m	84ms	4	1	0
impresspages	179	6m	224ms	5	1	0
buggenie	210	6m	118ms	4	3	0
elfinder	181	6m	124ms	11	3	0
tomatocart	370	8m	153ms	22	6	1
eyeos	611	6m	213ms	22	0	0
Total				85	17	1

Figure 10: Number of sample request and response pairs used for AIL learning, time used for collecting sample data and learning AIL descriptions, and results from comparing the learned AIL descriptions with the manually written ones.

Figure 10 shows the amount of sample data, the time used for collecting the sample data, and the time used by the AIL learning algorithm for each application. From these numbers we can give a rough answer to Q4: the effort required for using automated AIL learning is clearly manageable, compared to the time otherwise spent developing the web applications.

Producing AIL descriptions is of course not enough; they also need to capture the actual patterns of the Ajax communication. Recall from Section 4 that the constructed AIL description is complete by construction, relative to the training data. However, the training data may not cover the entire application, which might result in incomplete AIL descriptions where some operations supported by the server are missing in its AIL description. Another potential source of mismatches between automatically constructed AIL descriptions and manually written ones is that the learning algorithm may not be sufficiently precise or concise (using the terminology from Section 4). Furthermore, as there is no canonical "best" AIL description for a given Ajax server, we must settle for a subjective baseline for comparison, which we decide to construct as follows: For each benchmark application, we manually write an AIL description based on an inspection of the source code for the server part of the application. This process can take hours, but this work is of course only required to be able to measure the quality of the learning algorithm in our experiments.

Next, we need a measure of the difference between the automatically constructed AIL descriptions and the manually constructed ones. The first aspect of this measure is how the individual operation descriptions match between the two. Figure 10 also shows the results of this comparison. The Match column counts the number of learned operations that map directly to the actual server operations, while $1{\rightarrow}N$ and $N{\rightarrow}1$ count the number of server operations that map to multiple learned operations and vice versa, which indicate mismatches between the two descriptions. A second aspects is to what extent the individual datatypes of parameters and response patterns differ between the two descriptions.

We get a total of 85 matches, 17 occurrences of $1{\rightarrow}N$, and a single $N{\rightarrow}1$. The high number of matches is already encouraging, but a closer inspection of the other cases reveal that they are relatively benign. In all the $1{\rightarrow}N$ cases, a simple manual merging of the relevant operation descriptors suffices to resolve the discrepancy. This is acceptable since the learned AIL description is only intended as a starting point for the programmer, as an alternative to writing the AIL description from scratch. An example is `delete` operation in *resume*, which can be called both with and without an `id` parameter resulting in different responses, causing the learning algorithm to produce two separate AIL operation descriptors. An-

other example of a $1 \rightarrow N$ case appears in *buggenie*. In this case, a specific server operation `runIssueRevertField` performs multiple tasks and dispatches internally, based on a parameter `field`, in a way where one may argue that the AIL description produced by the learning algorithm, where these sub-operations are divided into separate descriptors, is in fact just as good as the manually constructed one.

The single $N \rightarrow 1$ case appears in *tomatocart* and is caused by our merging heuristic being slightly too aggressive. Two operations for deleting images and setting default images, respectively, both take an `id` parameter and return a trivial response, and the operations are only distinguished by the value of an `action` parameter. The similarity of the data causes the two operations to be merged by our current heuristic.

Regarding the quality of the inferred datatypes in request and response patterns, we notice that many of our benchmarks use JSON in responses but not in requests. For request patterns, the main question then is whether wildcards are introduced appropriately. The learning algorithm needs at least two distinct values of a path fragment or parameter to conclude that it is not constant. For example, *resume* represents session IDs in parameters, so the training data must involve multiple sessions. Incompleteness of our sample data in some cases results in missing wildcards, however, this is easy to adjust manually after the learning phase.

Most JSON response data in the benchmark applications is built using arrays and simple objects with fixed collections of properties. For these common cases the learning algorithm is able to generate JSON schemas correctly. Differences between the JSON schemas constructed by the learning algorithm and the manually constructed ones are mostly due to incomplete sample data. However, we observed two interesting cases—in *impresspages* and *globetrotter*, respectively—that could be improved. Some responses in *impresspages* have a recursive structure of objects and arrays. More specifically, the data represents a list of page and directory objects where each directory object itself contains a list of page and directory objects. Our current learning algorithm is not able to produce the desired concise JSON schema. In *globetrotter*, a specific JSON structure contains information about a list of countries. Each country is represented by an object where the country name appears as a property name, not as a property value, which causes the learning algorithm to view each country object as being distinct.

Based on these experiments, our answer to Q3 is that the learning algorithm is able to produce AIL descriptions that are reasonably close to manually constructed ones. This suggests that automated learning can be a good starting point for creating AIL descriptions for pre-existing applications, and that sufficient sample data can be obtained by someone who is familiar with the functionality of the applications but does not have knowledge of the server code.

5.3 Threats to Validity

A possible threat to validity of our experimental results is that the manually constructed AIL descriptions that we use as baseline for the comparisons have been made by ourselves without expert knowledge of most of the benchmark applications. More solid results could perhaps be obtained by performing user studies with the developers of the applications. Also, our benchmark applications do not reflect all possible uses of Ajax and JSON, and they may not be representative of typical usage patterns although we have striven toward including a wide variety of applications.

6. RELATED WORK

Our work touches on several areas of work on interface descriptions, automated testing of web applications, and learning algorithms.

6.1 Interface Descriptions for Separation of Concerns

The idea of design-by-contract is a fundamental principle in modern software engineering for separating the concerns of individual software components. Even in web programming, which often involves dynamic scripting languages both on the client and the server, interface description languages play an important role: Similar to AIL, WSDL [8] allows description of operations and their argument and return types, however, WSDL is tailored to XML-based web services and has no support for JSON, and we are not aware of uses of WSDL for describing server interfaces in Ajax-style JavaScript web applications. As mentioned in Section 2, AIL is by design conceptually closer to the language WADL [13], although AIL has a compact non-XML syntax and supports JSON. The Web IDL language is used for describing the API that web browsers provide to JavaScript programs [10] for accessing the HTML DOM and other parts of the browser state, however, unlike AIL, each Web IDL description is common to all JavaScript web applications and cannot describe the interfaces of individual Ajax servers. Web IDL has its roots in the OMG IDL interface definition language for CORBA [24].

Interface descriptions have also been proposed for HTML form-based web applications without JavaScript. The WebAppSleuth methodology by Fisher et al. [18] works by submitting requests to a server and analyzing the responses to infer parts of its interface. The resulting interface descriptions are related to AIL descriptions but tailored to HTML forms, not JSON or XML. Each form is described by its set of mandatory and optional fields together with simple value constraints and dependencies between the fields.

The extensive survey by Alalfi et al. [1] covers many modeling methods used in web site verification and testing, but without JavaScript and Ajax. To our knowledge, the only existing work involving interface descriptions for Ajax communication is that by Hallé et al. [16]. They propose a contract language based on interface grammars, linear temporal logic, and XPath expressions for specifying the order of HTTP interactions that exchange XML data in long-running sessions. We believe the data formats of requests and responses are more important in typical Ajax applications than restrictions on the order of operations, so we have chosen to ignore the temporal aspect in our first version of AIL. Their paper discusses how the contracts can be used for runtime monitoring. They also ask the important question *"who should write the contracts?"* To this end, we take the approach of using machine learning on sample execution traces, as explained in Section 4.

A range of well-documented web services that fit into the design of AIL can be found at Google's APIs Explorer website.[16] The interface descriptions for those web services are only made available as online documentation for programmers, not using any interface description language, which makes them less accessible to, for example, automated testing tools.

6.2 Automated Testing of Web Applications

Besides the Artemis tool [2] that we discussed in Section 3.1, we are aware of a few other tools for automatically testing JavaScript web applications. The Kudzu tool by Saxena et al. [26] performs automated testing by a combination of symbolic execution with a string constraint solver for value space exploration and random exploration of the event space, whereas Artemis uses a more lightweight feedback-directed approach. The state-based testing technique by Marchetto et al. [20, 21] builds finite-state machines that model Ajax web pages from concrete execution traces. As in our approach, a subsequent manual validation or refinement step is

[16] `http://code.google.com/apis/explorer/`

required to ensure that the extracted model is correct before the model can be used for automated testing. The key difference to our approach is that the models in state-based testing describe the DOM states of the JavaScript execution, not the interactions with the server. A closely related tool is Crawljax by Mesbah et al. [22, 23] that also aims to derive models of the user interface states of Ajax applications and use these models as a foundation for testing. AJAX Crawl by Duda et al. [9] similarly performs dynamic exploration of Ajax applications, but for the purpose of crawling the applications by search engines, not aiming at testing.

A common limitation of Kudzu, Crawljax, and AJAX Crawl is that the exploration of the JavaScript applications is done with little focus on the client-server communication, simply using live servers, which leads to the problems discussed in Section 3.1 about how to properly populate the server databases. On top of this, most tools, with Artemis as an exception, do not restore the server database state after each test input execution, which affects testing reproducibility.

The JSConTest tool by Heidegger and Thiemann performs random testing for JavaScript programs that are annotated with type contracts [17]. These function annotations play a similar role as AIL descriptions, but at the level of function calls rather than Ajax client-server interactions. Due to the JavaScript-oriented design of JSON Schema that we use in AIL, it is natural that the basic contract language in JSConTest has similar expressiveness. However, JSConTest also supports function types, which are not relevant for client-server exchange of JSON or XML data. Another difference is that JSConTest permits user-defined contracts written in JavaScript, which might be useful to consider for a future version of AIL to address the limitations identified in Section 5.

Several tools have been developed for automatically testing server-based web applications. The Apollo tool by Artzi et al. [3] and the tool by Wasserman et al. [27] perform directed automated random testing for PHP code, but JavaScript is not considered. With our proposal of using a server interface description language for separating the concerns of server code and client code, we have so far focused on testing the client code, but an interesting direction of future work is to develop testing or analysis techniques that can also check whether the servers fulfill their part of the contracts.

Elbaum et al. [11] have proposed the use of user session data for black-box testing of web applications. They record concrete user sessions and replay the sessions in various ways to test the server code, not aiming for testing client code and not involving explicit server interface descriptions.

The WAM tool by Halfond and Orso [14, 15] automatically discovers interfaces of web applications using static analysis or symbolic execution of the server code. The interfaces are subsequently used in automated testing, similar to our approach, although WAM considers classical web applications without Ajax and JSON. The notion of interfaces used by WAM is similar to that in WebAppSleuth. WAM is restricted to Java Servlets, unlike our approach, which is in principle independent of the languages and frameworks used on the server.

6.3 Learning Algorithms

The learning algorithm presented in Section 4 has been developed specifically for AIL, but related algorithms exist for other domains.

WebAppSleuth [18], which we also mentioned in Section 6.1, uses learning techniques to identify interfaces of server-based web applications that receive HTML forms. That approach does not involve learning the structure of server response data, and a single server operation is considered at a time, while our learning algorithm needs to work for multiple operations.

The latest version of WAM [14] likewise uses a learning algorithm to produce interface descriptions. The WAM algorithm operates on path constraints constructed through symbolic execution of the server code, which differs from our learning algorithm that is based on sample request and response data and has a black-box view on the server. Furthermore, WAM does not consider response types, unlike our learning algorithm.

The clustering problem that we face in Section 4 is related to the work by Broder et al. on clustering web pages that are syntactically similar [5]. Their approach is to define a distance measure between two web pages, using a distance threshold to cluster similar pages. This approach could be transferred to JSON responses and our learning algorithm, but we found the results from initially clustering only entirely equal structures to be sufficient for our purposes.

We are not aware of existing work on JSON Schema inference. The closest related work has been centered around DTD and XML Schema inference. This problem is described by Chidlovskii as being reducible to grammar inference [7]. Others improve on this line of work [4], however, the differences between XML and JSON make their algorithms unsuitable for JSON Schema.

7. CONCLUSION

Server interface descriptions for Ajax-style web applications enable separation of concerns of the server code and the client code. The AIL language has been designed to capture the essence of the existing proposals WADL and allow concise description of the basic properties of server operations, in particular involving JSON data. Our experimental validation suggests that the expressiveness of AIL suffices for typical Ajax communication patterns, but also that it might be useful in future work to add support for user-defined contracts to specify more fine-grained invariants.

One key contribution is that we demonstrate that server interface descriptions are useful in automated testing. No previous work has combined server interface descriptions with testing of Ajax applications. Our experimental results show that this approach can improve the level of automation by eliminating the need for carefully populated databases on the servers, while maintaining the quality of the testing of the client code. Another key contribution of our work is the automated learning algorithm that can produce server interface descriptions from sample request and response data. The experiments show that AIL learning can be performed with a modest effort, and that the resulting descriptions are a good starting point when programmers wish to construct AIL descriptions for pre-existing web applications.

In addition to the suggestions about possible extensions of AIL, several directions of future work appear. As an alternative or supplement to our AIL learning approach that has a black-box view on the server, it would be interesting to infer or validate AIL descriptions by static or dynamic analysis of the server code for the most widely used server web frameworks. Additionally, AIL may also be useful for static analysis of JavaScript applications to enable more precise reasoning of Ajax interactions than currently possible. Specifically, we wish to incorporate AIL into the JavaScript analysis tool TAJS [19].

Acknowledgements

This work was supported by Google, IBM, and The Danish Research Council for Technology and Production. The last author acknowledges partial support from U.S. NSF grants 0917392 and 1117603. We thank Simon Holm Jensen and Kristoffer Just Andersen for their contributions to the Artemis tool used in the experimental evaluation.

8. REFERENCES

[1] M. H. Alalfi, J. R. Cordy, and T. R. Dean. Modelling methods for web application verification and testing: state of the art. *Software Testing, Verification & Reliability*, 19(4): 265–296, 2009.

[2] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *Proc. 33rd International Conference on Software Engineering*, May 2011.

[3] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 36(4):474–494, 2010.

[4] G. J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. In *Proc. 17th International Conference on World Wide Web*, 2008.

[5] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks*, 29 (8-13):1157–1166, 1997.

[6] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weber. A framework for testing database applications. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, August 2000.

[7] B. Chidlovskii. Schema extraction from XML: A grammatical inference approach. In *Proc. 8th International Workshop on Knowledge Representation meets Databases*, 2001.

[8] R. Chinnici, J.-J. Moreau, A. Ryman, and S. W. (editors). Web services description language (WSDL) version 2.0, June 2007. W3C Recommendation. `http://www.w3.org/TR/wsdl20/`.

[9] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou. AJAX Crawl: Making AJAX applications searchable. In *Proc. 25th International Conference on Data Engineering*. IEEE, April 2009.

[10] C. M. (editor). Web IDL, February 2012. W3C Working Draft. `http://www.w3.org/TR/WebIDL/`.

[11] S. G. Elbaum, G. Rothermel, S. Karre, and M. F. II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3): 187–202, 2005.

[12] J. J. Garrett. Ajax: A new approach to web applications, 2005. `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`.

[13] M. Hadley. Web application description language, August 2009. W3C Member Submission. `http://www.w3.org/Submission/wadl/`.

[14] W. G. J. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *Proc. International Symposium on Software Testing and Analysis*. ACM, July 2009.

[15] W. G. J. Halfond and A. Orso. Improving test case generation for web applications using automated interface discovery. In *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, September 2007.

[16] S. Hallé, T. Bultan, G. Hughes, M. Alkhalaf, and R. Villemaire. Runtime verification of web service interface contracts. *IEEE Computer*, 43(3):59–66, 2010.

[17] P. Heidegger and P. Thiemann. Contract-driven testing of JavaScript code. In *Proc. 48th International Conference on Objects, Models, Components, Patterns*, June 2010.

[18] M. F. II, S. G. Elbaum, and G. Rothermel. Dynamic characterization of web application interfaces. In *10th International Conference on Fundamental Approaches to Software Engineering*, March 2007.

[19] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2011.

[20] A. Marchetto, F. Ricca, and P. Tonella. A case study-based comparison of web testing techniques applied to AJAX web applications. *International Journal on Software Tools for Technology Transfer*, 10(6):477–492, 2008.

[21] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *Proc. 1st International Conference on Software Testing, Verification, and Validation*, April 2008.

[22] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):3:1–3:30, 2012.

[23] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering*, 38(1):35–53, 2012.

[24] Object Management Group, Inc. Common object request broker architecture (CORBA) specification, version 3.2, November 2011. `http://www.omg.org/spec/CORBA/3.2/Interfaces/PDF`.

[25] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[26] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, D. Song, and F. Mao. A symbolic execution framework for JavaScript. In *Proc. 31st IEEE Symposium on Security and Privacy*, May 2010.

[27] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *Proc. ACM/SIGSOFT International Symposium on Software Testing and Analysis*, July 2008.

[28] Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proc. 20th International Conference on World Wide Web*, 2011.