Browserless Web Data Extraction: Challenges and Opportunities*

Ruslan R. Fayzrakhmanov, Emanuel Sallinger, Ben Spencer, Tim Furche, Georg Gottlob
Department of Computer Science, University of Oxford
Oxford, UK
name.surname@cs.ox.ac.uk

ABSTRACT

Most modern web scrapers use an embedded browser to render web pages and to simulate user actions. Such scrapers (or wrappers) are therefore expensive to execute, in terms of time and network traffic. In contrast, it is magnitudes more resource-efficient to use a "browserless" wrapper which directly accesses a web server through HTTP requests, and takes the desired data directly from the raw replies. However, creating and maintaining browserless wrappers of high precision requires specialists, and is prohibitively labor-intensive at scale. In this paper, we demonstrate the principal feasibility of automatically translating browser-based wrappers into "browserless" wrappers. We present the first algorithm and system performing such an automated translation on suitably restricted types of web sites. This system works in the vast majority of test cases and produces very fast and extremely resource-efficient wrappers. We discuss research challenges for extending our approach to a general method applicable to a yet larger number of cases.

CCS CONCEPTS

• **Information systems → Site wrapping**; **Deep web**; *Information extraction*;

KEYWORDS

web data extraction; scraping; deep web; HTTP; AJAX

ACM Reference Format:
Ruslan R. Fayzrakhmanov, Emanuel Sallinger, Ben Spencer, Tim Furche, Georg Gottlob. 2018. Browserless Web Data Extraction: Challenges and Opportunities. In *Proceedings of The Web Conference 2018 (WWW 2018).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3178876.3186008

1 INTRODUCTION

"The Web is the largest database" is a sentence one sometimes hears. This statement is, of course, wrong: Web data relevant to most applications is distributed over heterogeneously structured websites, usually does not come with a schema, and cannot be directly queried, except by manual keyword search. Given that many companies and institutions need to access outside data for better decision making [33], they have to rely on automated Web data extraction programs, also known as *wrappers*. They use wrapper generators that produce wrappers which continuously or periodically extract information from relevant websites and store this information in a highly structured format in a local database. In fact, Web data extraction is nowadays heavily and proficuously used by various branches of industry. Electronics retailers, for example, are interested in the daily prices offered by their competitors, as are hotels and supermarket chains. International construction firms automatically extract tenders from hundreds of websites. Other sectors have adopted Web data extraction as part of their core business. Among those are flight search engines and media intelligence companies.

Since around the year 2000, sophisticated semi-automated visual and interactive tools have been developed, which allow users to define wrappers via visual point-and-click actions. Examples are tools such as *STALKER* [36] and *Lixto* [3]. Other advanced semi-automatic tools are, for example, *import.io*, Mozenda [35], FMiner [15], iMacros [21], Visual Web Ripper [46] , and the BODE system [45]. The OXPath data extraction language [18, 25] enriches XPath with simulated user interaction, and node and form field selection based on visual features. OXPath is the target language of the fully automated high-precision visual-clue based wrapper generator DIADEM [16]. In DIADEM the knowledge for extracting data from websites belonging to an application domain (e.g., real-estate) is provided in form of Datalog rules. Using such rules and a set of URLs as input, the DIADEM system autonomously extracts data from sites belonging to this domain.

Advantages of visual-clue based data extraction. Most modern data extraction tools, including those mentioned, use visual clues and visual interaction in the wrapper generation process and rely on graphical and geometric concepts such as the *distance* between two rendered elements [24, 26]. For example, there is a rule in the DIADEM knowledge base which, in simplified form, says the following: *The closest text chunk below or above an input field on a Web page is (with high probability and in absence of better information) the explanatory label of this field.* The use of visual clues and "page geography" often leads to more precise and more robust wrappers [13]. The possibility of defining, modifying, and testing wrappers in a visual fashion has relieved wrapper designers from the tedious task of having to decipher the HTML code of each target web page, and of writing sequential programs that act on it. The semi-automatic wrapper generation process for complex websites, inclusive of testing, now typically takes a couple of hours at most, rather than a couple of days using traditional methods.

Drawbacks of visual-clue based data extraction. While infinitely beneficial for *wrapper design*, visual-clue based wrappers, referred to as ***visual wrappers***, come with an awful drawback at execution time: The natural approach, which is currently in use, is to render the page internally via an embedded browser. This *rendering* tasks consists of (a) parsing the web page and building a DOM tree (b) applying the CSS rules and (c) executing JavaScript, which is typically the most computationally expensive part, often having wide-ranging effects on the page's DOM.

*This work is supported by the EPSRC programme grant VADA: Value Added Data Systems – Principles and Architecture, no. EP/M025268/1. Georg Gottlob is also employed at TU Wien. Tim Furche is also with Wrapidity & Meltwater, London, UK.

This paper is published under the Creative Commons Attribution 4.0 International (CC BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.
WWW 2018, April 23–27, 2018, Lyon, France
© 2018 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC BY 4.0 License.
ACM ISBN 978-1-4503-5639-8/18/04.
https://doi.org/10.1145/3178876.3186008

A coarse analysis of the relative computational cost for visual wrappers in OXPath shows that the browser initialization phase takes about 13% of the runtime, the rendering phase about 85%, and the actual extraction about 2% [18]. Initialization and rendering jointly require approximately 50 times more time than the actual data extraction. For visual-clue based wrappers written in other programming languages than OXPath, this disproportion is similar or even worse. Companies engaged in massive data extraction suffer sorely under this huge runtime overhead. For example, financial companies, flight search companies or news services have hundreds of wrappers working continually, both in their own data centers and on rented cloud resources. A factor of 50 runtime overhead leads to huge additional costs. Moreover, the implied slow-down of real-time extraction tasks, such as extracting the current price of a flight, often leads to user dissatisfaction. For these and other critical real-time scraping tasks, such as data access in automated option trading, visual wrappers are simply too slow.

HTTP wrappers. To mitigate such runtime problems, many companies resort to replacing their most heavily used wrappers by alternative hand-programmed wrappers which we refer to as ***HTTP wrappers***. An HTTP wrapper is a browserless wrapper that interacts directly with a remote web server by sending HTTP requests (similar to those a browser would issue) and by analyzing the received replies without rendering them, picking the desired data directly from the raw content. In our experiments, we observed that HTTP wrappers are on average 23.8 times faster than the original visual wrappers, even excluding browser initialisation (see Tab. 1). Another advantage of HTTP wrappers is that they usually make only those server requests that are really necessary for obtaining the desired data, and skip a large number of requests that a browser would usually make to load irrelevant images, fonts, CSS style sheets, JavaScript code, text, and ads. Around 96.8% of the Internet traffic generated by visual wrappers (already ignoring images) is totally useless for obtaining the desired data and can be elided in a corresponding HTTP wrapper.

In order to build such HTTP wrappers, highly-qualified specialists initially perform the typical user interaction through a browser and analyze the HTTP traffic between the browser and the server. Once they have understood the browser-server interaction, which usually consists of a series of back-and-forth data exchange steps, and after they have identified the precise flow of the relevant data items, they write a parametrized HTTP wrapper that simulates the browser-server interaction obtaining, for each set of input parameters, the desired output data from the remote server without any rendering. Note that writing HTTP wrappers by hand is an extremely complicated task. Even specialists typically require several days to develop and test a new HTTP wrapper. Moreover, HTTP wrappers are usually less robust than visual-clue based wrappers and require more maintenance in case of structural website changes.

Key idea underlying this work. It is frustrating that for many scraping applications one has to go back to old-fashioned manual programming, and the data extraction community is in sore need of new methods which avoid the drawbacks of the above two types of wrappers altogether. As the authors have learnt from multiple contacts with scraping-intensive industries, the dichotomy of wrapper types and the implied necessity of choosing between one or another set of drawbacks is a major pain point. It would be fantastic if there were a method which combined all the advantages of both visual wrapping and HTTP wrapping, without sharing the disadvantages of either method. As will be reported in this paper, we discovered that this is indeed possible, and we will provide evidence of this. Our key innovative idea is very simple to formulate, but, as we will see later, represents a great technical challenge:

> **Key idea**: Transform modern visual-clue based wrappers automatically into browserless HTTP wrappers. Use the former at wrapper design time and the latter at runtime.

Of course such an automated transformation should be "intelligent" and produce HTTP wrappers which perform only the necessary steps and suppress useless requests that load pictures, ads, and other irrelevant data. This approach would then combine the advantages of relatively simple visual wrapper generation with the fast execution time, real-time wrapping suitability, low cost, and reduced Internet traffic of HTTP wrappers. Clearly, with the proposed new approach, wrapper maintenance, too, is simpler than with hand-made HTTP wrappers. We observed, that the transformation approach achieves a level of wrapper robustness that is not worse than the robustness level of visual wrappers. The "compilation" approach thus combines all advantages of visual-clue wrapping and HTTP wrapping. It follows that the proposed HTTP wrapper generation approach, when successful, has none of the disadvantages mentioned for either of the wrapping paradigms. This assumes of course that the transformation itself is robust, which is a key research challenge.

To our knowledge, nobody has ever attempted to construct an automatic transformation from visual wrappers into HTTP wrappers. This is not astonishing, given that it is a difficult endeavor which combines several hard research challenges. The main contributions of this work and its structure are detailed as follows:

- **Problem definition, challenges, and limitations** (Section 2). We open up the new area of automated HTTP wrapper generation from visual-clue based wrappers. We present a number of pitfalls that induce interesting research challenges, and set the limitations we assume in this paper.
- **The FastWrap approach** (Section 3). We present the first approach for automatically generating HTTP wrappers from visual-clue based wrappers. This includes novel techniques for identifying dependencies between browser-server interactions through the concept of "dependency graph". This graph is obtained by new methods of generalizing browser-server interactions based on their similarity, and a number of other technical tools. We present both the high-level approach, as well as a concrete algorithm. Complex analysis tasks are performed in relatively little time: The translation process takes less than one minute on average.
- **Evaluation** (Section 4). We present a comprehensive evaluation of the implementation of our approach, proving the principal feasibility of automatically generating HTTP wrappers. Encouragingly, even considering the limitations on the scope of the approach, the results are rather promising: The system successfully generates an HTTP wrapper in 79.2% of the test cases. Our evaluation shows that these HTTP wrappers are on average 23.8 times faster, use 31.3 times less network traffic, and send 71.4 times fewer requests than the original interactive wrappers executed through Firefox with images disabled.
- **Related work and research directions** (Sections 5 and 6). After discussing related work, we identify research directions arising from the challenges and limitations described in Section 2, as well as from our evaluation in Section 4.

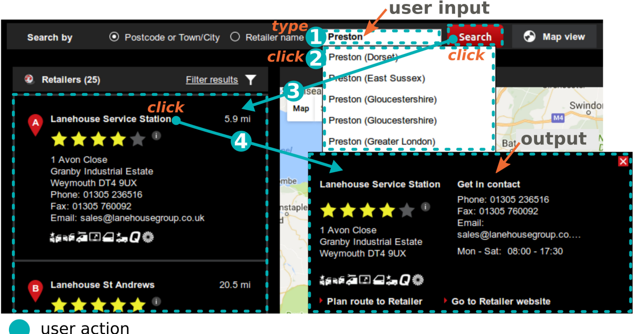

Figure 1: The four-step interaction for extraction from Vauxhall with the user input, "Preston"

2 PROBLEM DEFINITION, CHALLENGES, AND LIMITATIONS

Visual wrappers targeting complex Web applications require the simulation of specific sequences of user interactions either to get to a certain state of a current Web page (e.g., to get detailed information about a product loaded by AJAX requests) or to reach a page with relevant data (e.g., sending a filled in Web form or navigating to a specific page). These interactions are enriched with input and output instructions, parametrizing such interactive wrappers, e.g., to define that a certain form field should receive a location as input and that a particular part of the search result page to be extracted is a phone number. The parametrization is necessary for an intensive large-scale data extraction.

Visual wrappers mimic user actions by firing DOM events such as `click` or `keypress` in the integrated Web page rendering engine. In Fig. 1, we see an example of a visual wrapper used for extracting details of car dealerships. This scenario is represented by a sequence of four user interactions: ❶ entering search keywords (an input parameter), ❷ selecting a relevant suggestion from the autocomplete list, ❸ clicking the search button, which displays a list of corresponding car dealers, and ❹ clicking a link to a more detailed view which shows the target output data.

We can observe two levels of interaction: the *user-browser interaction* and the HTTP-based *browser-server communication*. On the first level, the data extraction scenario is defined by a sequence of interactions with the rendered graphical interface. Simulated user actions (e.g., ❶–❹ illustrated in Fig. 1) trigger the invocation of HTTP request-response exchanges (*HTTP interactions*) as well as the execution of the JavaScript code parsing and modifying the data obtained from the server. In our example in Fig. 1, the visual wrapper execution will produce a sequence of *HTTP traces* as schematically illustrated in Fig. 2 (and which will be explained in Sec. 3). For example, the user action ❶, entering input string into the search form, generates an autocomplete list by triggering the HTTP request-response II. The latter is an AJAX request returning a JSON object from the server, which is converted by client's JavaScript into a snippet of HTML code and integrated into the rendered web page.

The **main goal and problem** that we target in this paper is how to eliminate the first level of interaction (along with the browser) and conduct web data extraction exclusively using HTTP interactions, corresponding to the second level. As a **second goal and problem**, we also want to minimize the number of HTTP interactions by eliminating those of them which do not lead to HTTP responses with desired data.

Limitations. We now describe the limitations that we deliberately set on the wrappers and websites we consider. Two are on the visual wrappers, and two are on the websites our approach applies to. All four are important limitations, giving rise to interesting research challenges, and are discussed in Sec. 6.

Single target page. While the considered visual wrapper may visit arbitrarily many pages during execution (and the navigation may be arbitrarily complex), we only consider visual wrappers where the final extracted data is contained in a single Web page, which we call the target page. This limitation is not essential, as it does not touch upon the major critical points in browserless web data extraction. In fact, it is conceptually not hard to generalize the method to wrappers that take data from multiple pages. Moreover, note that overwhelming number of wrappers used in various industrial applications are single-target page wrappers (for example, in application areas that we have considered so far, these are more than 80%). Most wrappers that extract data from multiple pages do this because of next-page links, which are simple to recognize and handle automatically, but are not considered here for the avoidance of irrelevant overhead and to concentrate on the essential.

Flat records. For simplicity we only consider visual wrappers which extract flat data records as output. That is, we allow no nesting of records, no optional fields, and no compound data values. We believe that this restriction can be easily lifted by state-of-the-art structure identification techniques.

Service with reproducible interactions. We also here exclude unstable servers, i.e., web servers that reply to the same request with considerably different content in the response. This limitation is mitigated by the fact that according to our observations the vast majority are related to dynamically generated session-ids: When repeating the request with the same session-id, the server can return an error message. For session-ids, special approaches are possible, as will be discussed in Sec. 6. In some web domains the content can change frequently according to the data in the backend database (e.g., products in a shop, or flight details). In our approach we assume that the content does not change during the HTTP wrapper generation process.

No Google Maps Geocoding API. We do not support websites which use the Google Maps API. More specifically, we disallow websites that use the Google Maps API as a necessary part of the wrapper's execution. The deeper reason for this is that corresponding requests may use different parameters in different traces. In fact, this API encodes query values into the parameter names. However, there is an obvious solution to the problem: To carefully analyze the code to generate a bespoke, generic HTTP wrapper for interacting with the Google Maps service.

A wrapper is called **compliant** if it satisfies the four wrapper limitations given above.

3 THE FASTWRAP METHOD

In this section, we introduce FastWrap, our approach and algorithm for automatically generating HTTP wrappers. The algorithm takes as input a visual wrapper, referred to as the *input wrapper*, along with its input parameters, and produces as output a corresponding browserless HTTP wrapper, referred to as the *output wrapper*. The output wrapper extracts the same data as the input

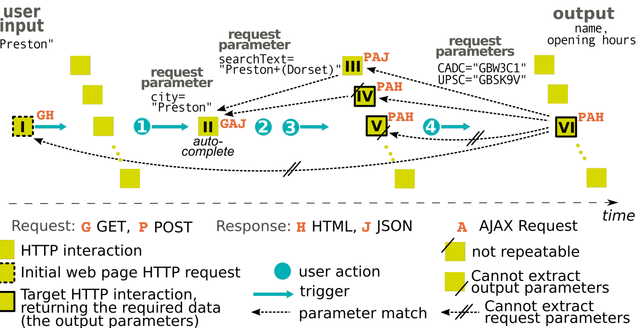

Figure 2: A schematic representation of an HTTP trace for the Vauxhall example with the user input, "Preston". HTTP interactions are ordered chronologically. It also illustrates parameter matches and the relations between user actions and triggered HTTP interactions.

wrapper by directly interacting with the website's servers and does not require loading and rendering the web page.

The FastWrap system implementing our approach works by executing the input wrapper with its input parameters to collect and analyze the sequence of request-response exchanges (referred to as *HTTP traces* or simply *traces*) between the input wrapper and the web server. Specifically these exchanges stem from a rendering engine (as used in, e.g. Lixto [19]), whether or not it is part of a browser (as used in, e.g. Mozenda [35] or OXPath [18]). HTTP traces also include data extracted by the input wrapper (so-called *output records*). This data is used to identify *target HTTP interactions* (or *target interactions*) with the desired output records in their responses. Furthermore, the provided records are used to generate *data selectors* to extract the data from the content and transform them into a structured representation. If *input parameters* are necessary (e.g., for web form filling), HTTP traces also include these. They are used to correctly induce the output wrapper which extracts the same data as the input wrapper. Wrapper creation is thus solely based on the direct analysis of these HTTP traces. This makes our approach applicable to a wide range of visual wrappers and avoids tying it to any particular system.

3.1 Overview of the Approach

The overall FastWrap approach can be intuitively explained via the following four main steps. A more formal definition of some of the terms used will be given in Sec. 3.2. The core algorithm is given in Sec. 3.3 and some further technical details in Sec. 3.4.

Step 1: Execute the input wrapper on sample input parameters and collect the corresponding HTTP trace for each run, along with the input parameters and output records. In the special case where input wrapper has no input parameter execute the input wrapper a single time.

In Step 1, the input wrapper is executed several times with different input parameters (e.g., "Preston", "Southampton", and "Walton" for the input parameter `city`), producing a corresponding number of HTTP traces. A graphical representation of a trace with user input "Preston" is illustrated in Fig. 2. The actual trace seen by our algorithm is a log of HTTP request-response messages (in this case obtained from the OXPath wrapper execution engine), where each response may contain HTML, JSON, XML or other formats.

Step 2: Analyze the traces in order to identify those HTTP interactions that contain the target records. Initially, the target records are the output records, while for subsequent recursive invocations, they are provided by Step 4.7.

In our example, interaction VI is one of the interactions that carries the required output data. In subsequent invocations, Step 2 will analyze interactions I, II, III, IV, and V. This way the traces are analyzed staring from the output records and leading back towards the input parameters. The advantage of this approach is that only interactions which are causally relevant to the output will be considered, while other, irrelevant interactions (e.g., downloading ads) are disregarded. In our example, all interactions (represented by rectangles) without numbers are such irrelevant interactions.

Step 3: Group similar HTTP interactions containing the target records from different traces into the same *equivalence class*. Two interactions are equivalent if they share exactly the same parameter names (possibly with different parameter values).

In our example, recall that Fig. 3 shows only a single trace for one possible input parameter, namely for Preston. Other such traces exist for other input parameters, such as "Southampton", "Walton", etc., where, for instance, instead of interaction I, II, III, we will have interactions I′, II′, III′ (for Southampton) and I″, II″, III″ (for Walton). Step 3 now groups I, I′ and I″ into the same equivalence class, and does the same for the other interactions.

Note that in our simple example, the grouping looks rather trivial. In more complex cases, the traces for different inputs may of course differ, and the exact notion of equivalence that induces the equivalence classes is important, which we shall discuss later.

Step 4: For each equivalence class c do:

Step 4.1: Generate a *data selector* able to extract the required records (which were identified in Step 2). That is, generate a single data selector that, for each interaction e of equivalence class c, extracts the required record from the HTTP response of interaction e. If a data selector cannot be generated, skip Steps 4.2–4.8 and return to Step 4.

It is very often the case that extractors for the same parameter can be found in multiple equivalence classes. For example, interaction VI needs the parameters `CADC` and `UPSC`. These are in fact obtainable from four different classes of interactions, namely I, III, IV and V. Yet, interactions I and V contain all possible `CADC`'s, and it is not possible for a selector to "magically" guess the right one. Thus, the selector generation fails for (the equivalence class of) interactions I and V, respectively. However, the selector can be easily built from interaction III or IV, for which the selector generation succeeds. Note that for interaction VI, the algorithm generates an XPath-based data selector, as the response is HTML. In other cases, the responses are, for example, JSON-formatted, in which case it would generate JSONPath. Sometimes transformations of the extracted data are required, in which case they are recognized by our algorithm and applied.

Step 4.2: Ensure that all HTTP interactions from the equivalence class are repeatable. That is, re-execute each of the requests, and check whether the selector still selects the same data. Otherwise, skip Steps 4.3–4.8 and return to Step 4.

Step 4.3: For the equivalence class c, classify the parameters into constant and variable parameters. A constant parameter has the same value over all interactions in c; a variable parameter has two or more such values.

In our example, parameters `CADC` and `UPSC` (from the body of the HTTP POST requests) are variables, as they occur with different values in different traces (such as `GBW3C1`, `GBW771`, `GBC678` for `CADC`). At

the same time there are also constant parameters, such as `x-brand` which is always `vauxhall` and `x-language` which is always `en` (from the URL's query string).

Step 4.4: With query probing techniques, eliminate superfluous variable parameters: those which can be omitted without affecting the data extracted from the HTTP response by the data selectors derived in Step 4.1.

In principle, the algorithm could also remove constant parameters when they are unnecessary. However, removing variable parameters is most important; this reduces the number of dependencies to be satisfied and therefore the algorithm's search space.

Step 4.5: Synthesize an abstract HTTP interaction, that is, an HTTP request which contains placeholders, which are to be instantiated with the values of the variable parameters. These will be the nodes of the dependency graph.

For our example, Fig. 3 shows the abstract HTTP interactions generated from the (concrete) HTTP interactions in Fig. 2. For instance, the equivalence class of VI corresponds to VI.

Step 4.6: Find variable parameters in c whose values come directly from the input parameters.

For example, the parameter city in interaction II can be obtained from the input. Note that in this case the parameter literally occurs in the input (i.e., we have the string "Preston" as a parameter value, and the exact same string "Preston" as the input), but in other cases transformations may be required, as in Step 4.1.

Step 4.7: For each variable parameter v that cannot be obtained from the input, go to Step 2 recursively. Let the target records be all parameter values of v. To avoid recursive loops, remove all interactions from the equivalence class of c from the trace. From the recursive call, we get one dependency graph D_v for each such variable parameter v, or find that no such dependency graph can be obtained for v.

For example, for abstract interaction VI, the variable parameter `CADC` cannot be obtained from the input parameters. Therefore, we recursively go to Step 2, using the values of `CADC` as the target records, i.e., the set {`GBW3C1`, `GBW771`, `GBC678`}.

For this recursive call, all interactions in the equivalence class of VI are no longer considered. This recursive call returns a "local" dependency graph $D_{\mathtt{CADC}}$ (which will ultimately become a subgraph of the overall dependency graph constructed by the algorithm). For the variable parameter `UPSC`, it yields a dependency graph $D_{\mathtt{UPSC}}$. In fact, these two graphs both have the form: II←III.

Note that, through this recursion, the final wrapper is not required to keep the HTTP requests in chronological order but may rearrange them to obtain a more efficient wrapper.

Step 4.8: If for some variable parameter v a dependency graph D_v could not be computed, skip this step. Otherwise, create a new dependency graph D by merging all "local" dependency graphs D_v. Add to D the abstract interaction of c as well as edges from c to each D_v and c. Return D.

In our example, for abstract interaction VI, we obtained the graphs II←III in the previous step, thus the result of merging is exactly the graph II←III. The algorithm adds to it the abstract interaction VI, as well as two edges, from VI to III, one labeled with the variable `CADC`, the other with `UPSC`. This is illustrated in Fig. 3.

This was a simplified explanation of the algorithm. As can be seen in Fig. 3, other parts of the HTTP wrapper are generated apart from those discussed. However, it highlights the central concepts.

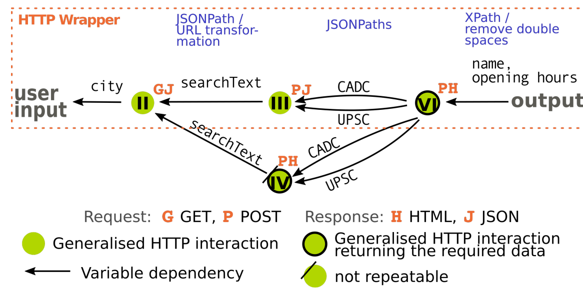

Figure 3: The overall dependency graph of generalised HTTP interactions for the Vauxhall example (with disconnected nodes omitted) and corresponding HTTP wrapper

3.2 Formal Definition of Main Concepts

We now give formal definitions of some of the concepts we have informally introduced in Sec. 3.1, and which are used by the FastWrap algorithm. For space reasons, we cannot give a detailed description of all the data structures used.

Definition 3.1. An *HTTP interaction* (or simply an *interaction*) represents an HTTP request-response exchange as a tuple $(U, V, \mathcal{P}, B)$, where U is a parametrized URL (a URL without its query part and with values of segments split by "`/`", removed); V is an HTTP verb; and $\mathcal{P} = \{\rho_1, \rho_2, \dots\}$ is a set of HTTP request parameters, each of which are pairs $\rho_i = (n_i, v_i)$ specifying parameter names and values. A *parameter name* is a pair (o, s) of an *origin* and *origin-specific name*. Origin is one of "header", "path", "query", or "post". □

An **HTTP trace** is a sequence of HTTP interactions observed during a single execution of a visual wrapper. It can either be collected by a proxy server or directly from the input wrapper's interpreter. A trace additionally contains the visual wrapper's input data (e.g. used for form filling) and the extracted output data. If an interaction of an HTTP trace contains all the extracted output data in its response, it is called a *target interaction*.

Definition 3.2. An **abstract HTTP interaction** is an equivalence class of HTTP interactions. Two interactions are *equivalent* if they have the same *signatures* and the same *response content type*. The *signature* of an HTTP interaction is the set of parameter names of the HTTP interactions (i.e., ignoring the parameters' values). The response content type is the MIME-type of the response (e.g., `text/html` or `application/json`).

A parameter name (or simply, a parameter) in an abstract HTTP interaction is either *constant* or *variable*. Let V be the set of all values for a parameter in the original HTTP interactions. If this set V contains a single element, the parameter is called *constant*, otherwise, it is called *variable*. □

Conceptually, an abstract HTTP interaction represents an equivalence class as defined above. In the implementation, it is an HTTP interaction *template* $(U, V, \mathcal{P}, B)$ with empty response body B and where values for variable parameters are removed from $\mathcal{P}$.

Definition 3.3. A **dependency graph** is a DAG whose nodes are abstract HTTP interactions, where each edge is annotated by the tuple (n, s, t) where n is a parameter name, s is a *data selector* and t is a *value transformer*. A *data selector* is an expression (e.g., XPath, JSONPath, or regular expression) for extracting data from the HTTP response content. A *value transformer* is a function which converts the data extracted from the response of one interaction into the format required by the dependent interaction. Nodes may be labeled

as *target interactions*. In addition, the graph contains special nodes *input* and *output*, each annotated with parameter names. □

Fig. 3 shows a dependency graph for the Vauxhall example. It contains all abstract interactions from all available sets of traces, along with their dependencies. Some matches from Fig. 2 do not give rise to corresponding abstract dependencies; in particular, there are no edges between VI and V, or VI and I. This is due to the fact that V and I contain all possible values for certain parameters, and it is impossible to generate a data selector which can precisely extract only the relevant data corresponding to the given user input.

Definition 3.4. An **HTTP wrapper** is a dependency graph with the following properties:

- It always contains a single node labeled as the target interaction.
- All parameter dependencies are resolved. That is, for each abstract interaction in the graph, if it contains a parameter named n, there is an outgoing edge labeled with n. □

The HTTP wrapper can be executed straightforwardly by a depth-first traversal rooted at the single target interaction which provides the output data. Each interaction is initiated after obtaining values for all its variable parameters either from the user input or from other HTTP interactions.

3.3 The FastWrap Algorithm

Starting by identifying the final interactions, FastWrap recursively constructs abstract HTTP interactions and sets their corresponding dependency arcs. This exploration of each potential dependency corresponds to a search through the implicit dependency graph for a suitable wrapper. Once a valid wrapper is found, the search terminates. Assuming one exists, one wrapper is generated for each target node. The user or external system can then select one of those according to its own objective function, based on criteria such as precision, recall, number of interactions, or runtime.

Alg. 1 shows the core algorithm at the heart of FastWrap, starting at Step 2, after having generated the traces. Given the set of recorded HTTP traces $\mathbb{T}$, the algorithm is invoked by calling FindWrappers($\mathbb{T}$, *output*($\mathbb{T}$), *true*), where *output*($\mathbb{T}$) represents the output data for each trace.

The FastWrap algorithm consists of two functions: FindWrappers and ResolveDependencies. The former finds a) the final abstract interactions containing all required data (see line 3), executing ResolveDependencies in mode 1 ($init = true$), and b) the intermediate abstract interactions (with their dependency subgraphs), each of which provides the value for a specific variable parameter (see line 5–9), executing ResolveDependencies in mode 2 ($init = false$). Given a set of parameters (in mode 2, the set consists of a single parameter), ResolveDependencies, in turn, recursively builds an HTTP wrapper, identifying corresponding interactions with desired data in their responses and adding the appropriate dependency edges.

3.4 Implementation Details

FastWrap is implemented in Python. For data and parameter value extraction, FastWrap implements two types of data selectors: XPath and JSONPath. It currently supports a subset of XPath 1.0 for extracting from HTML and XML content. This subset includes node tests, position predicates, attribute, child and descendant axes. JSONPath is an analogous query language for JSON data[1], which is used to select data from JSON and JSONP documents. All data

[1]Our JSONPath notation is similar to http://goessner.net/articles/JsonPath/

Algorithm 1: The FastWrap Algorithm

1 **Function** FindWrappers($\mathbb{T}$, V, *init*)
  Input: A set of HTTP traces $\mathbb{T}$, a parameter V which maps traces to sets of target records, and a boolean flag *init* specifying whether it is the inital run of the algorithm.
  Output: A set of HTTP wrappers which extract the target parameters.
2 **if** $init = true$ **then** // identify target abstract interactions
3   **return** ResolveDependencies($\mathbb{T}$, V, *true*)
4 **else** // identify other abstract interactions
5   $D \leftarrow \{\}$
6   **for** $v \in V$ **do** // each $v \in V$ associated with a variable parameter is traced separately
7     $D' \leftarrow$ ResolveDependencies ($\mathbb{T}$, $\{v\}$, *false*)
8   **if** $|D'| \neq 0$ **then** $D \leftarrow D \cup D'$
9   **return** D

10 **Function** ResolveDependencies($\mathbb{T}$, V, *init*)
  Input: A set of HTTP traces $\mathbb{T}$, a parameter V which maps traces to sets of target records, and a boolean flag *init* specifying whether it is the initial run of the algorithm.
11 Find interactions $T \in \mathbb{T}$ with *values*(V) in their response (Step 2)
12 Group interactions T into a set of equivalence classes C (Step 3)
13 $D \leftarrow \{\}$ // set of intermediate or final HTTP wrappers
14 **for** $c \in C$ **do** (Step 4)
15   Generate a data selector(s) S (Step 4.1)
16   Generate a value transformation function(s) F
17   **if** S and F are successfully generated **then**
18   **if** interactions in c can be repeated and data of V can be extracted with S and F **then** (Step 4.2)
19   Identify constant and variable parameters V' in c (Step 4.3)
20   Remove *superfluous parameters* from V' of c (Step 4.4)
21   Create an abstract HTTP interaction A' for c (Step 4.5)
22   Label A' with S, F, and *name-origin*, accordingly /* add parameter tracing information */
23   **for** $v' \in V'$ **do** (Step 4.6)
24   **if** v' is a user input parameter with a value transformer F' **then**
25   Label v' with the name of the corresponding input parameter and F'
26       $V'' \leftarrow V' \setminus$ set of labelled parameters (Step 4.7)
27       $D' \leftarrow$ FindWrappers($\mathbb{T} \setminus c$, V'', *false*)
28   **if** $|V''| \neq |D'|$ **then continue** /* not all parameters can be traced for A' */ (Step 4.8)
29   Initialise a dependency graph d with A' labelled as a "target interaction"
30   **for** $d' \in D'$ **do**
31   Add d' into d
32   Add a dependency arc between a local target node A'' of d' and A' and label it with parameter tracing information
33   **if** $init = false$ **then** Remove parameter tracing information from A'', remove label "target interaction"
34       $D \leftarrow D \cup \{d\}$
35   **if** $init = false$ **then break** /* ensure that each parameter has a single dependency arc */
36 **return** D

selectors can also select sub-structures (i.e., subtrees in the case of XPath, and nested objects in the case of JSONPath) and substrings (either by position or by splitting on a specific symbol).

FastWrap supports six basic types of value transformations, which were introduced based on the analysis of various real-world sites, independent from the dataset evaluated in this experiment.

- Removal of double, leading, and trailing spaces.
- Extracting substrings and concatenation with constant strings.
- Space conversions such as space-to-plus/minus symbols.
- URL en-(de-)coding with newer [5] and older [6] standards.
- Replacing HTML escape symbols.
- Changing the precision of real numbers.

We also support two specific composite transformations:

- Space-to-plus followed by URL encoding.
- URL encoding, followed by string concatenation.

4 EVALUATION

In this section, we conduct an extensive evaluation of the FastWrap approach on different types of websites. For the experiments we implemented the FastWrap system in Python. It executes visual wrappers and analyses HTTP traces to build corresponding HTTP wrappers. We selected OXPath, a state-of-the-art web data extraction system, as a visual wrapper system for our experiments.

4.1 Choice of Test Cases

In total, we considered 130 wrappers, out of which 101 (78%) were compliant (according to the definition from Sec. 2). 45% of the 29 rejected wrappers had unreproducible interactions carrying relevant data, and the other 55% invoked the Google Maps API to obtain necessary data (in particular, to obtain geographic coordinates for the location provided as user input). The compliant wrappers (78% of the original 130) span a variety of application domains and feature various degrees of difficulty. In particular, we consider (A) more complex examples involving web form interaction and (B) simpler cases without web form interaction.

Type A test cases. Type (A) is used to test the success rate of our approach in converting wrappers interacting with web forms with different input parameters (which is necessary for Deep Web data extraction) into their HTTP wrapper counterparts. For this class of test cases, we tested (i) 12 OXPath wrappers for US restaurant chains from the DIADEM dataset [17]; (ii) 11 OXPath wrappers for well-known car dealership websites; (iii) 12 OXPath wrappers for popular websites, mostly from the retail domain, with two or three simulated user interactions (e.g., `click` and `type`), each of which triggers HTTP requests (often via AJAX calls) to deliver the content necessary for subsequent interactions (we call this category A-multisteppers); (iv) 25 OXPath wrappers for randomly chosen web pages with web forms (referred to as A-RND). The random sites were chosen by randomly sampling URLs from the Common Crawl [10] search index dataset, which includes around 3 billion web pages. As there were no existing wrappers for classes (ii), (iii) and (iv), new OXPath wrappers were manually written. Wrappers from category (A) are typical search-based wrappers, which fill web forms and extract data from the first result page (which may involve AJAX requests to load additional data). Interestingly, test set (iv), i.e., random URLs turned out to be much simpler and easier to deal with than categories (i)–(iii), which showed rather homogeneous behavior. We thus group (i), (ii) and (iii) into the single class A-RCM (restaurants, cars, and multisteppers). Hence class (A) consists of two subclasses: A-RCM and A-RND.

Type B test cases. Type (B) comprises 41 additional OXPath wrappers for US restaurant chains from the DIADEM dataset. In contrast to the restaurant wrappers from set (i), these wrappers do not require filling web forms and do not take input parameters from a user. Thus, for this class of test cases it is enough to analyze a single HTTP trace to identify any target HTTP interactions. This approach is applicable to any wrapper which uses a single, linear navigation and does not depend on input data, even if that navigation requires multiple steps or page downloads.

4.2 Experimental Setup

Each experiment consists of two phases: (1) execution of the visual wrapper and (2) generation of the HTTP wrapper.

For phase 1, OXPath was executed with images disabled. It was extended with a proxy server module to listen to HTTP request-response exchanges and record them as HTTP traces, including

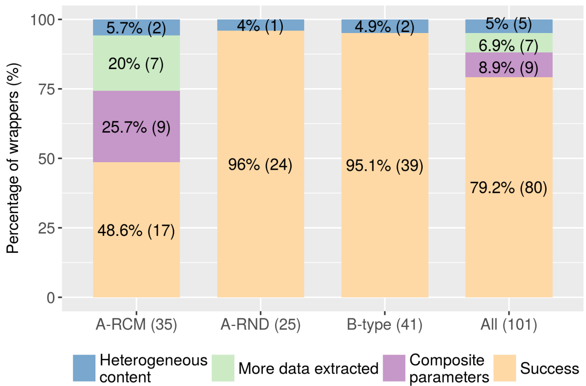

Figure 4: Overall results

input parameters and extracted data. Each wrapper of type (A) (i.e., A-RCM and A-RND) was executed eight times with different input parameters selected manually. Wrappers of type (B) do not require input and were executed once each.

In phase 2, the stored HTTP traces from phase 1 were used to induce an HTTP wrapper and test it. Wrappers of type (A) used five of the eight traces for induction (i.e., building an HTTP wrapper). The three remaining traces are used to evaluate the correctness and performance of the generated HTTP wrapper. Wrappers of type (B) were induced and evaluated on a single trace from phase 1.

4.3 Experimental Results

In this section, we will first discuss the overall success rate (lower part of Fig. 4) of the FastWrap approach. Then we compare the performance of the generated output wrappers to the performance of the original input wrappers (Tab. 1). Finally, we discuss the characteristics of the generated wrappers in more detail (Tab. 2). Sec. 4.4 presents a detailed failure analysis which allows us to draw important conclusions for necessary future research to bring this approach to industrial strength (upper part of Fig. 4).

Overall results. Overall, our experimental evaluation showed a high success rate of 79.2%. However, interestingly, it demonstrated quite different performance on the test sets. In particular, HTTP wrappers were successfully generated for 48.6% of A-RCM wrappers, 96.0% for A-RND, and 95.1% for B-type (lower part of Fig. 4, the rest of the figure will be discussed later). The average time required by FastWrap to generate a wrapper is 44.9 seconds. For the B-type wrappers, which is the simpler category, the average is 2.1 seconds. Due to the larger number of interactions, and in particular more complex data structures in the HTTP responses (and thus in the traces), together with a somewhat larger data volume, the average generation time is 137.9 seconds for A-RCM. The wrapper generation time is shown in Tab. 2, whose other columns will be discussed further below.

Performance comparison. As can be seen in Tab. 1, all generated HTTP wrappers considerably outperform the respective OXPath wrappers on all test sets. The resource usage of the generated wrappers was orders of magnitude lower, which confirms our initial expectation that “cutting off” the browser would lead to a significant benefit. The average runtime of the HTTP wrappers is only 4.2% of the runtime of their visual wrapper counterparts. The generated HTTP wrappers use only 3.2% of the data traffic, and perform just 1.4% of the HTTP interactions. Interestingly, while the number of HTTP interactions in the test cases A-RND and B-type is

Table 1: Comparative analysis of OXPath and HTTP wrappers

Wrappers	OXPath wrappers			HTTP wrappers			Comparison		
	Time† (ms)	Data‡ (KB)	Interactions	Time (ms)	Data (KB)	Interactions	Time (%)	Data (%)	Interactions (%)
A-RCM	33 780	4151	90.0	1145	134	1.5	3.4	3.2	1.7
A-RND	22 299	2839	79.7	1314	76	1.0	5.9	2.7	1.3
B-type	23 879	1523	38.6	837	69	1.0	3.5	4.6	2.6
All	26 917	3013	76.9	1139	96	1.1	4.2	3.2	1.4

† without browser initialization
‡ without loading images

Table 2: Characteristics of generated HTTP wrappers

Wrappers	Generation◇ (s)	Content formats (%)	Value transformers♦ (%)	Ignored variable parameters (%)	Stably correct models (%)	Successful evaluations (%)
A-RCM	137.9	JSON: 52; HTML: 45; XML: 3	**n**: 77; **u**: 14; **p**: 6; **s**: 3	16.7	100.0	100.0
A-RND	33.1	HTML: 91; JSON: 9	**n**: 52; **u**: 33; **s**: 10; **c**: 5	8.7	90.9	97.0
B-type	2.1	HTML: 92; JSON: 5; XML: 3	–	0.0	100.0	100.0
All	44.9	HTML: 76; JSON: 22 XML: 2	**n**: 68; **u**: 21; **s**: 5; **p**: 4; **c**: 2	13.8	94.9	98.2

◇ HTTP wrapper generation from HTTP traces
♦ **n** – no transformation, **u** – URL en-(de-)code, **p** – precision, **s** – substring, **c** – composite

always 1 (which is 1.3% resp. 2.6% of the original number of interactions), for test cases A-RCM it is 1.5 (1.7%) on average. This mainly stems from the subclass A-multisteppers, and in particular eight wrappers, seven of which require two interactions in our HTTP wrapper, and one which requires three (namely our running example, Vauxhall). In these cases, multiple interactions are required due to additional transformations that are not computed on the client side, because they need data from the server or require server-side functionality, such as a database lookup. This is, for example, the case for lexus.fr, where it is necessary to transform input postcodes to geographic coordinates. For our running example Vauxhall, the wrapper needs to transform location names to the variables `CADC` and `UPSC` (as mentioned before) representing car dealerships.

Detailed analysis of relevant wrapper features. In Tab. 2 we analyze interesting features of the generated HTTP wrappers. We classify a specific evaluation test as *successful* if both precision and recall are 100%, i.e., the data extracted by the HTTP wrapper is identical to the data extracted by the input wrapper. We say that a wrapper is ***stably correct***, if it successfully passes all tests of phase 2 (see Sec. 4.2). As we see, all wrappers in class A-RCM and B-type are stably correct. For A-RND, we see that 90.9% are stably correct and in total 97% of individual tests are passed. The reason for this is related to deviations in HTML structures, which require either richer test sets or more powerful data selectors. According to our evaluation, 13.8% of variable parameters of HTTP interactions can be omitted without affecting the quality of data extraction. Interestingly, many web applications actively use such parameters (16.7% for A-RCM), which indicates the importance of query probing for identifying an irrelevant set of parameters to reduce the dependency graph and increase the chance of resolving all dependencies. It is also worth noting that A-RCM's web applications use JSON (52% of content processed by the HTTP wrappers) to deliver relevant data considerably more often than web applications from other categories which, in turn, more often use HTML snippets (91% for A-RND and 92% for B-type). In Tab. 2, we omit value transformations applied to the output data as they are always related to removing redundant spaces. For input and intermediate parameters for HTTP interactions, frequently no transformation (68%), or only URL encoding-decoding (21%) are required for transforming values between requests.

4.4 Discussion and Lessons Learned

Our experiments confirm that our approach is feasible. Beyond the criteria mentioned in Sec. 2 for compliant wrappers, we can identify further critical points where future research is needed. The overall numbers are an indicator that the approach presented in this paper is principally feasible. Still, it is useful to try to understand what the reasons for the 20.8% failure rate are (see the fourth bar in Figure 4). As illustrated in Fig. 4, most of the issues are encountered in A-RCM with wrappers written for more complex web applications (51.4% of the A-RCM wrappers could not be translated). There are three main causes for the failures: 1) composite parameters, 2) more data extracted, 3) heterogeneous content.

Composite parameters. This problem covers 8.9% of All test cases and 25.7% of A-RCM. A variable parameter (i.e., a parameter of the abstract HTTP request) can be a string-serialization of multiple values. It can represent a tuple (as in "`lat|long|locality`"), an array (e.g. "`lat-long,lat-long,...`" representing a set of geometric points), or a complex structure (such as a GET parameter containing a URL-encoded XML document). Furthermore, the structure of the value may not be obvious from its string serialization without domain knowledge; for example if the value is an address constructed from different attributes, such as "72-353 Highway 111". Thus, composite parameters might require the analysis of regularities in the string-serialized representation, and application of domain specific methods, or generic techniques such as NLP.

More data extracted. The second common cause of failure (6.9% of All test cases or 20% of A-RCM) is related to extracting a superset of the data extracted by the original wrapper. For example, a car search website may return car models from all production years, while a client-side JavaScript filter may restrict the models shown to a user to those of the current year. Another example is that for design reasons (so as not to overwhelm the user with too many

results), a website may choose to display only a limited number of results, while the data received by the client actually contains much more data. Another example we encountered is from the flight search domain, in which JavaScript filters some results returned from the server before displaying them on a web page. The filter analyses attributes of the search result to eliminate those with lower scores, reflecting their trustworthiness or relevance. This also motivates further research into the identification of such JavaScript filters and methods to mimic them. Both machine learning and program analysis may be promising approaches. Note that for certain applications, returning more data is in fact a benefit.

Heterogeneous content. Multiple levels of embedding of different content types were regularly encountered in our test cases, i.e., in 5% of All, 5.7% of A-RCM, 4% of A-RND, and 4.9% of B-type. For example, a typical website will return an HTML, JSON, or XML document, which is no problem for our approach; whereas some sites use a single response which is a snippet of HTML, where the desired data is embedded within a serialized JSON object embedded within JavaScript source code within the HTML. Desired data can also be encoded into a string-serialized snippet of HTML represented as a value in a JSON object.

We believe that these problems can be largely overcome by further targeted research and will give further details in Sec. 6. Note that interestingly, if we had excluded composite parameters a priori, then the success rate would grow to 87% of the compliant cases. This shows the huge potential of improvements further research could bring here.

5 RELATED WORK

To the best of our knowledge, this is the first work that specifically addresses the problem of automatically transforming visual wrappers into HTTP wrappers, other than a different prototype tool described in [39]. However, there are some closely related areas that serve to inform our approach or delimit its goals.

Most supervised web data extraction tools provide visual interfaces for the definition of visual wrappers ([15, 19, 21, 25, 35, 45, 46]). Such wrappers are then executed typically using a live browser. For this work, we focus on visual wrappers written in OXPath, as it outperforms previous commercial and open source systems in both time and memory [18].

Web automation and some AJAX crawlers share the simulation of user interaction with visual wrappers. In many web automation systems, this simulation is to test the correct behavior of a web application [1, 2, 8, 28, 31]. Our approach would be applicable to "test scripts" produced by such systems and could be used to verify that the backend produces the intended results. This can be achieved by converting tests into their HTTP counterparts. However, as our intent is to *avoid* interaction with the application's frontend, it is obviously not suited to test that part of the application. In contrast, when considering crawling of heavily scripted AJAX sites, our approach could be used for finding shortcuts to specific nodes in the state-flow graphs as produced by such AJAX crawlers [11, 12, 34].

The techniques and aims presented in this paper share some similarity with techniques for the analysis of HTTP logs [47] or click streams [32]. However, that similarity is only on the surface, as we focus on observing one or more interactions with a single backend, without any control over that backend. In contrast, HTTP log analysis usually assumes many users accessing a single, instrumented backend. Also, as far as we are aware, none of the works on HTTP log analysis have considered the problem of discovering the underlying backend API (as it is usually assumed to be known).

6 DIRECTIONS FOR FURTHER RESEARCH

The main goal of future work is certainly to increase the number of cases where a visual wrapper can be automatically transformed into an HTTP wrapper. Through our analysis we now know better what needs to be done to achieve this goal. We now re-examine the cases that we excluded upfront in Sec. 2 as well as those failures observed in Sec. 4.4 to formulate directions for future research.

Shallow JavaScript analysis. Value transformations of parameters transferred between HTTP interactions may vary from simple text transformations (e.g., merging two strings into one) to complex functions. In the case of complex transformations, which go beyond basic numerical or string-based changes, shallow JavaScript analysis is required (e.g., knowledge-based, heuristic methods for variable and value tracing). These transformations can often be automatically induced, however in certain cases partial execution of relevant pieces of JavaScript code is actually necessary. To identify relevant pieces of JavaScript code transforming values, approaches from the field of concolic testing (e.g., Jalangi [43]) can be used.

Composite parameter handling. Our evaluation in Sec. 4 demonstrated the importance of being able to analyze composite parameters of HTTP requests. One approach to handle them is to parse their structures directly. Another is to identify the atomic values contained in complex parameters by the analysis of discrepancies between HTTP interactions of the same equivalence class. Techniques from data matching [7, 22, 23, 44] may apply.

Heterogeneous content (e.g., mixtures of HTML, XML, JSON). This poses yet another extraction problem related to locating identified parameters within HTTP replies. There are various existing approaches from the web data extraction and information extraction areas. However, they target specific formats, such as XML, HTML [9, 14, 20, 29, 37] and plain or formatted texts [27, 41]. Thus, a development of a hybrid approach is necessary for dealing with multiple formats at once, creating appropriate *heterogeneous data extractors*. Moreover, in some cases, data is actually contained inside of JavaScript code. This needs further research.

Non-reproducible interactions. Sometimes interactions cannot be reproduced due to the session-ids or access tokens being required. Apart from shallow JavaScript analysis [38, 40, 42], research in the area of web automation may contribute to automatically obtain such tokens on behalf of a user [1, 2, 31].

HTTP interactions with varying sets of parameters. Sometimes, alike HTTP interactions from different executions of a visual wrapper can have different sets of variable parameters. We frequently faced this problem with the Google Maps service. This requires more advanced approaches to identifying similar HTTP requests that are able to deal with such a high degree of variability.

Multiple target pages and complex navigation scenarios. Allowing extraction from different web pages might require the analysis of linear navigation paths through the result pages or more complex navigation patterns. Examples of such difficult cases are conditional navigation, as well as unbounded loops in navigation for which more research is needed.

Wrapper optimization. Identification of the most efficient and robust wrapper can also be represented as an optimization problem, minimizing the overall cost of the wrapper generation and execution processes. This has potential connections to web service composition [4, 30, 48].

REFERENCES

[1] Shaon Barman, Sarah Chasins, Rastislav Bodík, and Sumit Gulwani. 2016. Ringer: web automation by demonstration. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 748–764.

[2] Alberto Bartoli, Eric Medvet, and Marco Mauri. 2012. Recording and replaying navigations on AJAX web sites. In *International Conference on Web Engineering*. Springer, 370–377.

[3] Robert Baumgartner, Oliver Frölich, and Georg Gottlob. 2007. The Lixto Systems Applications in Business Intelligence and Semantic Web. In *The Semantic Web: Research and Applications, 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007, Proceedings*. 16–26.

[4] Amina Bekkouche, Sidi Mohammed, Benslimane Marianne, Chouki Tibermacine, Fethallah Hadjila, and Mohammed Merzoug. 2017. QoS-aware optimal and automated semantic web service composition with user's constraints. *Service Oriented Computing and Applications* (2017), 1–19.

[5] Tim Berners-Lee, Roy Fielding, and Larry Masinter. 2005. *Uniform Resource Identifier (URI): Generic Syntax*. Standard RFC 3986. The Internet Society (ISOC) / Internet Engineering Task Force (IETF).

[6] Tim Berners-Lee, Larry Masinter, and M. McCahill. 1994. *Uniform Resource Identifier (URI)*. Standard RFC 1738. Network Working Group.

[7] Philip A. Bernstein, Jayant Madhavan, and Erhard Rahm. 2011. Generic Schema Matching, Ten Years Later. *PVLDB* 4, 11 (2011), 695–701.

[8] Jeffrey P. Bigham, T. Lau, and J. Nichols. 2009. Trailblazer: enabling blind users to blaze trails through the web. In *Proceedings of the 13th international conference on Intelligent user interfaces*, Vol. 09. ACM, 177–186.

[9] Michal Ceresna. 2005. *Supervised Learning of Wrappers from Structured Data Sources*. PhD Thesis. Vienna University of Technology.

[10] CommonCrawl 2018. Common Crawl. (2018). Retrieved Feb 14, 2018 from http://commoncrawl.org/

[11] Mustafa Emre Dincturk, Suryakant Choudhary, Gregor von Bochmann, Guy-Vincent Jourdan, and Iosif Viorel Onut. 2012. A statistical approach for efficient crawling of rich internet applications. In *Web Engineering*, Marco Brambilla, Takehiro Tokuda, and Robert Tolksdorf (Eds.). Springer, Berlin, Heidelberg, 362–369.

[12] Cristian Duda, Gianni Frey, Donald Kossmann, Reto Matter, and Chong Zhou. 2009. AJAX Crawl: Making AJAX applications searchable. In *Proceeding of the IEEE 25th International Conference on Data Engineering (ICDE '09)*. IEEE, Washington, DC, USA, 78–89.

[13] Ruslan R. Fayzrakhmanov. 2015. Models and Approaches for Web Information Extraction and Web Page Understanding. In *The Evolution of the Internet in the Business Sector: Web 1.0 to Web 3.0*, Pedro Isaías, Piet Kommers, and Tomayess Issa (Eds.). IGI Global, Chapter 2, 25–50.

[14] Emilio Ferrara, Pasquale De Meo, Giacomo Fiumara, and Robert Baumgartner. 2014. Web data extraction, applications and techniques: A survey. *Knowledge-Based Systems* 70 (2014), 301–323.

[15] FMiner 2018. FMiner. (2018). Retrieved Feb 14, 2018 from http://www.fminer.com

[16] Tim Furche, Georg Gottlob, Giovanni Grasso, Omer Gunes, Xiaonan Guo, Andrey Kravchenko, Giorgio Orsi, Christian Schallhart, Andrew Sellers, and Cheng Wang. 2012. DIADEM: domain-centric, intelligent, automated data extraction methodology. In *Proceedings of the 21st International Conference Companion on World Wide Web (WWW '12 Companion)*. ACM, New York, NY, USA, 267–270.

[17] Tim Furche, Georg Gottlob, Giovanni Grasso, Xiaonan Guo, Giorgio Orsi, Christian Schallhart, and Cheng Wang. 2014. DIADEM: Thousands of Websites to a Single Database. *PVLDB* 7, 14 (2014), 1845–1856.

[18] Tim Furche, Georg Gottlob, Giovanni Grasso, Christian Schallhart, and Andrew Sellers. 2013. OXPath: A language for scalable data extraction, automation, and crawling on the deep web. *VLDB Journal* 22, 1 (2013), 47–72.

[19] Georg Gottlob, Christoph Koch, Robert Baumgartner, Marcus Herzog, and Sergio Flesca. 2004. The Lixto Data Extraction Project: Back and Forth Between Theory and Practice. In *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '04)*. ACM, New York, NY, USA, 1–12.

[20] Andrew W. Hogue and David R. Karger. 2005. Thresher: automating the unwrapping of semantic content from the World Wide Web. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*. 86–95.

[21] iMacros 2018. iMacros. (2018). Retrieved Feb 14, 2018 from http://imacros.net

[22] Ekaterini Ioannou, Nataliya Rassadko, and Yannis Velegrakis. 2013. On Generating Benchmark Data for Entity Matching. *J. Data Semantics* 2, 1 (2013), 37–56.

[23] Hanna Köpcke and Erhard Rahm. 2010. Frameworks for entity matching: A comparison. *Data and Knowledge Engineering* 69, 2 (2010), 197–210.

[24] Iraklis Kordomatis, Christoph Herzog, Ruslan R. Fayzrakhmanov, Bernhard Krüpl-Sypien, Wolfgang Holzinger, and Robert Baumgartner. 2013. Web object identification for web automation and meta-search. In *3rd International Conference on Web Intelligence, Mining and Semantics, WIMS '13, Madrid, Spain, June 12-14, 2013*. 13.

[25] Jochen Kranzdorf, Andrew Sellers, Giovanni Grasso, Christian Schallhart, and Tim Furche. 2012. Visual OXPath: Robust Wrapping by Example. In *Proc. of WWW*. 369–372.

[26] Bernhard Krüpl-Sypien, Ruslan R. Fayzrakhmanov, Wolfgang Holzinger, Mathias Panzenböck, and Robert Baumgartner. 2011. A versatile model for web page representation, information extraction and content re-packaging. In *Proceedings of the 2011 ACM Symposium on Document Engineering, Mountain View, CA, USA, September 19-22, 2011*. 129–138.

[27] Nicholas Kushmerick. 2003. Finite-State Approaches to Web Information Extraction. *Information Extraction in the Web Era* 2700 (2003), 77–91.

[28] Tessa Lau, Julián Cerruti, Guillermo Manzato, Mateo Bengualid, Jeffrey Bigham, and Jeffrey Nichols. 2010. A conversational interface to web automation. *Proceedings of the 23nd annual ACM symposium on User interface software and technology* (2010), 229–238.

[29] A. Lemay, J. Niehren, and R. Gilleron. 2006. Learning n-Ary Node Selecting Tree Transducers from Completely Annotated Examples. *International Colloquium on Grammatical Inference (ICGI 2006)* 4201 (2006), 253–267.

[30] Angel Lagares Lemos, Florian Daniel, and Boualem Benatallah. 2016. Web Service Composition: A Survey of Techniques and Tools. *ACM Comput. Surv.* 48, 3 (2016), 33:1–33:41.

[31] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa A. Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the 2008 Conference on Human Factors in Computing Systems, CHI 2008, 2008, Florence, Italy, April 5-10, 2008*. 1719–1728.

[32] Jun Liu, Cheng Fang, and Nirwan Ansari. 2014. Identifying user clicks based on dependency graph. *2014 23rd Wireless and Optical Communication Conference, WOCC 2014* (2014).

[33] Jorn Lyseggen. 2017. *Outside Insight: Navigating a World Drowning in Data*. Penguin Books Limited. 336 pages.

[34] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-based Web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)* 6, 1 (2012), 1–30.

[35] Mozenda 2018. Mozenda. (2018). Retrieved Feb 14, 2018 from http://www.mozenda.com

[36] Ion Muslea, Steven Minton, and Craig A. Knoblock. 1999. A Hierarchical Approach to Wrapper Induction. In *Agents*. 190–197.

[37] Adi Omari, Sharon Shoham, and Eran Yahav. 2017. Synthesis of forgiving data extractors. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (WSDM '17)*. ACM, New York, 385–394.

[38] Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. 735–756.

[39] Richard Penman. 2016. *Web Data Extraction Optimization: From User Interaction To Web Server Communication*. MSc Thesis. University of Oxford.

[40] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. 1–12.

[41] Sunita Sarawagi. 2008. Information extraction. *Foundations and Trends in Databases* 1, 3 (2008), 261–377.

[42] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. 513–528.

[43] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 488–498.

[44] Wei Shen, Jianyong Wang, and Jiawei Han. 2015. Entity linking with a knowledge base: Issues, techniques, and solutions. *IEEE Transactions on Knowledge and Data Engineering* 27, 2 (2015), 443–460.

[45] Jui Yuan Su, Der Johng Sun, I. Chen Wu, and Lung Pin Chen. 2010. On design of browser-oriented data extraction system and the plug-ins. *Journal of Marine Science and Technology* 18, 2 (2010), 189–200.

[46] Visual Web Ripper 2018. Visual Web Ripper. (2018). Retrieved Feb 14, 2018 from http://visualwebripper.com

[47] Guowu Xie, Marios Iliofotou, Thomas Karagiannis, Michalis Faloutsos, and Yaohui Jin. 2013. ReSurf: Reconstructing Web-Surfing Activity From Network Traffic. *Proc. IFIP Networking Conference* (2013), 1–9.

[48] Yuhong Yan and Min Chen. 2013. Anytime QoS-aware service composition over the GraphPlan. *Service Oriented Computing and Applications* 9, 1 (2013), 1–19.