User-friendly and Extensible Web Data Extraction

Tomáš Novella   *tomasnovella@gmail.com*

Department of Software Engineering – Charles University
Prague, Czech Republic

Irena Holubová   *holubova@ksi.mff.cuni.cz*

Department of Software Engineering – Charles University
Prague, Czech Republic

Abstract

Creation of web wrappers is a subject of study in the field of web data extraction. Designing a domain-specific language for a web wrapper is a challenging task, because it introduces trade-offs between expressiveness of a wrapper's language and safety. In addition, little attention has been paid to execution of a wrapper in a restricted environment.

In this paper we present a new wrapping language – Serrano – that has three goals: (1) ability to run in a restricted environment, such as a browser extension, (2) extensibility to balance the tradeoffs between expressiveness of a command set and safety, and (3) processing capabilities to eliminate the need for additional programs to clean the extracted data. Serrano has been successfully deployed in a number of projects and provided competitive results.

Keywords: web data extraction, safe execution, restricted environment, web browser extension.

1. Introduction

Since the dawn of the Internet, the amount of available information has been steadily growing every year. Email, social networks, knowledge bases, discussion forums – they all contribute to the rapid growth of data. These data are targeted for human consumption, therefore, the structure tends to be loose. Although humans can easily make sense of unstructured and semi-structured data, machines fall short and have a much harder time doing so. Automation of data extraction therefore gives companies a competitive edge: instead of time-consuming and tedious human-driven extraction and processing, they become orders of magnitude more productive, which leads to higher profits and more efficient resource usage.

With the advent of new web technologies, such as AJAX [4], and the rise of the Web 2.0 [9], simple raw manipulation of HTML [3] proved no longer sufficient. As a result, extraction tools have started being bundled with an HTML layout rendering engine, or have been built on top of a web browser to be able to keep up with modern standards. Extraction tools have evolved to be more user-friendly; many came with a wizard – an interactive user interface – that allowed for convenient generation of wrappers. All this evolves in the direction to increase wrapper maintainability, which helps to take on incrementally larger tasks. Major challenges facing the tools available currently on the market are as follows:

- *Data manipulation* Tools, even the recent ones, provide only a restricted way of data manipulation, such as data trimming and cleaning. These tasks are often delegated to separate tools and modules, which may be detrimental to wrapper maintenance, considering it leads to unnecessary granularization of a single responsibility, since there have to be additional programs that process the data that are pertinent to the given wrapper.

- *Extensibility* With the rapid evolution of web technologies, many tools soon become ob-

solete due to the inability to easily extend the tool to support modern technologies.

- *Execution in restricted (browser) environment* New execution environments have emerged, which gives rise to novel applications of data extraction. Examples include web browser extensions (in-browser application), which help to augment the user browsing experience. These environments are restricted in terms of programming languages they execute and system resources. Besides, script execution safety is another concern.

In this paper we propose a novel data extraction language, Serrano, which deals with all the three mentioned problems. In Section 2 we overview the related work. In Section 3 we introduce the Serrano language. Section 4 showcases the user stories, Section 5 discusses the advantages of Serrano and Section 6 concludes.

2.   Related Work

Inspired by [18] in this paper we define a *web wrapper* as a procedure for seeking and finding data, extracting them from web sources, and transforming them into structured data. The exact definition of a wrapper varies and it is often interchanged with the definition of the *extraction toolkit* [7], a software extracting, automatically and repeatedly, data from websites with changing contents, and that delivers extracted data to a database or another application. Toolkits are often equipped with a GUI that features an internal WebView that represents a tab in a browser to facilitate wrapper generation. Typically, a user manipulates with the web inside the Web-View in order to obtain the desired data. User actions are recorded as DOM [1] events, such as form filling, clicking on elements, authentication, output data identification, and a web wrapper is generated. This wrapper can run either in the toolkit environment, or separately packaged with a wrapper execution environment. After the execution additional operations may be implemented, such as data cleaning [20], especially when information is collected from multiple sources. Finally, extracted data are saved in a structured form in a universal format, such as XML [2], JSON [10], or into a database.

One of the first endeavors to classify Web Data Extraction toolkits [18] proposed a taxonomy for grouping tools based on the main technique used by each tool to generate a wrapper. Tools were divided into six categories: languages for wrapper development (e.g., TSIMMIS [15]), HTML-aware tools (e.g., W4F [21]), NLP-based tools (e.g., RAPIER [8]), wrapper induction tools (e.g., WIEN [17]), modeling-based Tools (e.g., NODoSE [5]) and ontology-based tools (e.g., DIADEM [12]). Most wrappers combine two or three of underlying techniques for locating data in the documents to compensate for their deficiencies. We can distinguish regular expression-based approaches (e.g., W4F), tree-based approaches (e.g., OXPath [13]), declarative approaches (e.g., Elog [6]), spatial reasoning (e.g., SXPath [19]), and machine-learning based approaches (e.g., RAPIER). In [11], the authors identify and provide a detailed analysis of 14 enterprise applications of data extraction.

3.   Serrano Language

This section examines and explains why Serrano was designed the way it was. For a complete in-depth specification, the reader is referred to the official language specification[1]. The source codes as well as playground projects can be found on Github[2] and are written in Javascript.

Gottlob [14] presented four desiderata that would make an ideal extraction language:

1. *Solid and well-understood theoretical foundation* Serrano uses jQuery[3] selectors, a su-

---

[1]https://github.com/salsita/Serrano/wiki/Language-Spec
[2]https://github.com/salsita/Serrano/tree/master/serrano-library
[3]https://jquery.com/

perset of CSS selectors, for locating elements on the web page. These technologies have been studied in depth along with their limitations and computational complexity. Serrano wrapper is a valid JSON and every command corresponds to a Javascript command.

2. *A good trade-off between complexity and the number of practical wrappers that can be expressed* One of Serrano's cornerstones is extensibility. Currently, the language can only locate elements by CSS selectors and simulate mouse events. Nevertheless, the command set can be easily extended so that a larger range of wrappers can be expressed.

3. *Gentle learning curve* Many Serrano commands have the same name and arguments as their Javascript counterparts.

4. *Suitability for incorporation into visual tools* Selector identification is a task already handled by browsers in the Developer Tools extension. There is no obvious obstacle that would prevent us from incorporating selected Serrano commands into a visual tool.

In order to make a language easy to integrate with Javascript, we leveraged JSON. In contrast to other data transmission formats, such as XML, JSON has been strongly integrated into Javascript, which eliminates the need of additional helper libraries for processing. In Serrano, both the wrapper and the result are valid JSON objects. This makes them convenient to transform and manipulate: they can be passed around via AJAX, or altered via Javascript directly, since they are represented by a built-in object type. Moreover, Javascript libraries such as Lodash[4] further extend object manipulation capabilities.

To deal with extensibility, Serrano has separated the command set and allows to create custom commands. Examples of such extension are commands for document editing which makes Serrano, to the best of our knowledge, the first data extraction as well as data editing language used in the browser. With a simple extension of a command set, we can allow Serrano to manipulate the native Javascript window object, manage user credentials[5] or change the page location. This offers expressive power and control beyond the range of most extraction tools.

Wrapper maintainability is another design goal. Powerful commands, such as conditions and type checks, make it possible to write verification inside the wrapper.

3.1. Type System

Serrano type system inherits from the Javascript type system. It supports all types that are transferable via JSON natively; that is `number`, `string`, `boolean`, `undefined`, `null` and `object` as well as some additional Javascript types, such as `Date` and `Regexp`.

3.2. Scraping Directive

The basic building block is called a *scraping directive*. It represents a piece of code that evaluates to a single value. There are 3 types of scraping directives: *command*, *selector* and *instruction*.

Command

Commands are the core control structure of Serrano. As such, they appear similar to functions in common programming languages; in that they have a name and arguments. However, their use is much broader. Serrano has commands such as `!if` for conditions, logical commands such as `!and`, `!or`, commands for manipulation with numbers and arrays of numbers, such as `!+`, `!-`, `!*`, `!/` etc. Elevating the strength of commands and making them the central control

---

[4] `https://lodash.com/`
[5] `http://w3c.github.io/webappsec-credential-management/`

structure is the cornerstone of flexibility and extensibility: all control structures are of the same kind and adding/removing these structures is a part of an API. Although some languages, such as Selenium IDE[6], make it possible to write plugins and extensions of default command sets[7], we did not find any wrapping language that allows to add and remove any command control structure arbitrarily.

Syntactically, a command is a JSON array, where the first element has a string type denoting the command name followed by arguments (the rest of the array).

Below, we present an example of the `!replace` command with three string arguments.

```
["!replace", "hello world", "hello", "goodbye"]
```

In this example, `!replace` is the command name, which has three arguments, namely `hello world`, `hello` and `goodbye`. This command returns a new string based on an old string (supplied as a first argument) with all matches of a pattern, be it a string or a regular expression (second argument) replaced by a replacement (third argument). Finally, the command returns the string `goodbye world`.

Raw arguments

Command arguments, unless stated explicitly otherwise, have *implicit evaluation*. That means, when an argument of a command is another scraping directive, it is first evaluated and only then the return value supplied. However, this behavior is not always desired. Because of this, the command specification determines which arguments should be *raw* (not processed). An example of such a command is the `!constant` command, that takes one raw argument and returns it. Had the argument not been specified as raw, the constant command would return a string `hello mars`.

```
["!constant", ["!replace", "hello  world", "world", "mars"]]
⇒ ["!replace", "hello world", "world", "mars"]
```

Implicit foreach

By default, commands have a so-called *implicit foreach*. That means, when the first argument of the command is an array, the interpreter engine automatically loops through the array, and applies the command to each element, returning a list of results. It is also known as the *map* behavior. Conversely, when a command does not have an implicit foreach, the first argument is passed as-is, despite being an array.

An example illustrates two commands. Command `!upper` has the implicit foreach enabled. Thus, it loops through the array of strings and returns a new array containing the strings in upper case. The second command `!at` has the implicit foreach functionality disabled; therefore is selects the third element in the array. (Had it been enabled for `!at`, the command would return the third letter of each string, namely the following array `["e", "o", "r", "u"]`.)

```
// implicit foreach enabled for !upper
["!upper", ["!constant", ["hello", "world"] ] ] ⇒ ["HELLO", "WORLD"]

// implicit foreach disabled for !at
["!at", ["!constant", ["one", "two", "three", "four"] ], 2] ⇒ "three"
```

Selector

A *selector* is used for extracting specific elements from the web page. It is denoted as a single-item array, containing only one element (of type string) that is prefixed with one of the characters

---

$, =, ~ and followed by a string indicating the selector name. This selector name is treated as a CSS selector (more precisely, a jQuery selector[8]).

From the low-level perspective, a selector is syntactic sugar for the `!jQuery` command (which takes one or two arguments and evaluates them as a jQuery selector command[9]) and the different kinds of selectors are syntactically "desugarized" as follows:

Dollar sign This is the basic type of selectors. `["$selector"]` is treated as `["!jQuery", "selector"]` which internally invokes the `$("selector")` jQuery method.

Equal sign Selectors that start with this sign, i.e., `["=selector"]` are treated as an *instruction* `[ ["$selector"], [">!call", "text"] ]`. The important thing is, that after selecting specific elements, the text value of the selector is returned, which internally corresponds to invoking a jQuery `text()` method on the result.

Tilde sign This sign infers that type conversion of a selector result to a native Javascript array is imposed. By definition, `["~selector"]` is treated as `[["$selector"], [">!arr"]]`.

Most wrapping languages (including Selenium IDE language and iMacros[10]) enable to target elements by CSS selectors. Those languages also support other forms of element addressing such as XPath queries. SXPath language enables addressing elements by their visual position. Serrano does not support those additional methods and in the first version we decided to implement the support for CSS selectors, since they are more familiar to web stack developers than other methods. Nevertheless, we consider adding further methods of element addressing a valuable future prospect.

Instruction

An *instruction* is a sequence of commands (and selectors), that are stacked in an array one after another. Similarly to the UNIX pipe (`|`), the output of the previous command can be supplied as the first argument of the following. This functionality is enforced by the addition of an optional *greater than* sign at the beginning of a command name or a selector. In that case, the supplied argument is called the *implicit argument*. Otherwise, the result of the previous command is discarded. The example below illustrates three examples of upper casing the `hello` string. The first directive is an instruction that constructs a `hello` string and passes it to the `!upper` command. The second directive is a direct command and the third one first constructs the `goodbye` string, but because the `!upper` method is not prefixed with the greater than sign, it is discarded and the command runs only with its explicitly stated `hello` argument. The last directive throws an error, since the `!upper` command is expecting one argument and zero arguments are supplied.

```
[["!constant", "hello"], [">!upper"]]
["!upper", "hello"]
[["!constant", "goodbye"], ["!upper", "hello"]] ⇒ "HELLO"
[["!constant", "goodbye"], ["!upper"]] ⇒ Error
```

3.3. Scraping Query and Scraping Result

Scraping directives are assembled into a higher logical unit that defines the overall structure of data we want to extract. In other words, a scraping query is a finite-depth key-value dictionary

---

[8] https://api.jquery.com/category/selectors/
[9] http://api.jquery.com/jquery/
[10] http://imacros.net/

where for each key, the value is the scraping directive or another dictionary. The example below showcases a scraping query.

```
{ title: [["$h2"], [">!at", 2], [">!lower"]],
  time: {
    start: [["$.infobar [itemprop='datePublished']"],
            [">!attr", "content"], [">!parseTimeStart"]]
    end: // another scraping directive
  }
}
```

Once the Serrano interpreter starts interpretation of a scraping query, it recursively loops through all the keys in an object and replaces the scraping directive with respective evaluated values. E.g., if the interpreter runs in context of a fictional movie database page, scraping query above will evaluate to a *scraping result* that looks like this.

```
{ title: "The Hobbit",
  time: {
    start: "8:00pm"
    end: "10:41pm"
  }
}
```

The structure provides two main advantages over wrappers in other languages: (1) The pivotal part of the Serrano wrapper are the data, and a quick glance at a scraping query reveals what data are being extracted and what the instructions that acquire them are. Wrapping languages such as internal Selenium IDE language or iMacros are instruction-centric, that is, the wrapper is described as a sequence of commands, where some commands happen to extract data. Languages, such as Elog also do not reveal immediately the structure of the extracted data. (2) A scraping query consists of scraping directives. If one directive throws an error, it can be reported, and the processing continues with the following directive in the scraping query. In tools, such as Selenium IDE, the data-to-be-extracted are not decoupled, so a failure at one point of running the wrappers halts the procedure.

3.4. Scraping Unit

A *scraping unit* roughly corresponds to the notion of a web wrapper. It is a higher logical unit that specifies when the scraping is to begin as well as what actions need to be executed prior to data extraction. The reason is that often scraping cannot start immediately after the page loads. When scraping from dynamic web pages, we might be awaiting certain AJAX content to load, some elements to be clicked on etc. These waits are referred to as *explicit waits*.

Some languages, such as TSIMMIS, do not expect that some content is not ready immediately after the page has loaded. Other languages, such as iMacros, also consider the page ready right after the load[11] but also provide a command to wait for a given period of time[12].

We have separated the waiting prescription into the scraping unit instead of mixing it with the wrapper itself to make the wrapper more clear and separate the tasks. A certain disadvantage of our approach might be the fact, that for more complex wait instructions (e.g., scraping intertwined with waiting) we also have to mix them, which creates a disorderly wrapper.

Because the execution can be delayed or ceased (if the element we are waiting for will not appear), interpretation of the scraping unit returns a Javascript Promise. A Promise is an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is yet incomplete.

---

[11]`http://wiki.imacros.net/FAQ##Q:_Does_the_macro_script_wait_for_the_page_to_fully_finish_loading.3F`
[12]`http://wiki.imacros.net/WAIT`

3.5. Page Rules

Sometimes we want to execute different wrappers and run actions on a single web page. *Page rules* is an object, that associates scraping units and scraping actions with a web page. To our best knowledge, no wrapping language has this functionality and users have to manage the wrappers and actions manually. Thus Serrano also has the role of a "web data extraction manager", where it manages which wrapper should be executed on a given page.

The page rules object has two properties, *scraping* and *actions*, that serve for specification of scraping units and actions, respectively. A valid rules object must have at least one of these properties non-empty. The scraping property contains either a single scraping unit, or a key-value pair of scraping units and their names. Serrano then enables the user to execute the scraping unit by the name. Similarly, an action can either be a scraping action (which is a special type of a scraping directive) or a key-value pair of named actions.

3.6. Document Item and Global Document

Each page rules object needs to be associated with the respective URL or a set of URLs so that, at the visit of a web page in the browser, Serrano is able to find the most suitable rules object. The associating object is called a *document item* and it has the following four properties: the *domain*, then either a *regexp* (a regular expression) that matches the URI, or a *path* which is the URN, and finally the *rules* object. Multiple document items may match the given URL. In that case, we select the match with the highest priority.

The priority is given to every document. The most important criterion is the "length" of a domain. This is determined by the number of dots in the URL. E.g., `scholar.google.com` has a higher level of specification than `google.com` and thus it has higher priority. The next criterion for priority is determined by other fields. The *regexp* field has higher priority than the *path* field. Both fields are optional and they cannot be used in a single document item simultaneously. The lowest priority has a document item with the domain attribute set to `*`. This domain item is also referred to as the *default domain item* and matches all URLs.

Finally, an array of document items forms a *global document* and it is the top-level structure that encapsulates all the data in Serrano. With the Serrano API, we usually supply this global document and the engine chooses the matching page rules.

3.7. Command Set

One of the leading ideas behind Serrano is to create an extensible language that extracts and processes the extracted data. The aim is to completely eliminate the need for middleware processing that is dependent on a given wrapper. Therefore, we consider extraction and subsequent data processing as one responsibility and find valuable to couple these tasks together. As a consequence, Serrano wrapper creators are capable of extracting and cleaning the data, all in one script. To accomplish this, the resulting command set must be rich – the extracted data often undergo complex transformations in order to be unified. These commands constitute the core library of Serrano.

The rest of this section provides an overview of most important commands and illustrates useful use cases. The full list can be found in the Language Specification[13].

Conditions and Logical Predicates

Ability to change the control flow is one of the distinguishing features of Serrano. Using conditions, Serrano can decide which information to scrape and how to transform it during runtime.

---

[13]`https://github.com/salsita/Serrano/wiki/Language-Spec`

Commands that contribute to this category are divided into:

- *Branching commands.* The main representative is the `!if` command with optional *else* branch. The first argument is a predicate, which is a scraping directive that returns a Boolean result.

- *Existence tests* Commands, such as `!exists` or `!empty` and their logical negations `!nexists`, `!nempty` enable us to test if a given structure exists (is not `undefined` or `null`) and whether the array contains any elements, respectively.

- *Comparison tests* serve for comparing two integers. Commands in this category are: `!lt, !gt, !le, !ge, !eq, !neq` and are directly translated to <, >, <=, >=, ==, !==, respectively.

- *Compound conditions* include `!and` and `!or` commands and their `!all` and `!any` aliases. They help to group multiple single predicates into compound predicates.

- *Result filtering* is a means for reducing an array of results to only those items that pass a filtering criterion. For this purpose we define the `!filter` command that takes an argument in the form of an array and on each array item it evaluates the *partial condition* that is the second argument to `!filter` command. By partial condition we mean that the condition which is the argument of the `!filter` command should use argument chaining, i.e. should be evaluated on each tested item of the filtered array.

Arithmetics

Arithmetics is especially useful when we need to add offsets to dates, or do other minor calculations. There are four commands `!+, !-, !*, !/` that cover the basic operations with numbers. The commands have two operands and work on both numbers and arrays of numbers. If both operands are arrays of the same length, the operation is executed "per partes". Otherwise, `NaN` is returned.

Text Manipulation

Among the myriad of commands, we list the most important ones: `!lower, !upper, !split, !trim, !replace, !join, !concat, !splice,` and `!substr`. The behavior is identical to their Javascript counterparts; details are provided by the official specification.

DOM Manipulation

Serrano has been recently enriched with DOM manipulation capabilities on top of data extraction. To manipulate the DOM we can use `!insert`, `!remove` and `!replaceWith` commands, which are identical to their jQuery counterparts.

The `!insert` command takes three arguments: first one has to be a selector, followed by the string `"before"` or `"after"` to denote where the insertion is to be done, and the final argument is the text to be inserted.

```
[ "!insert", [ "$p:first" ], "before", "<h2>Hello John!</h2>" ]
```

The third variable may also be a template string enclosed by `{{` and `}}`. Names of interpreted variables are either plain names, or refer to nested properties using standard Javascript dot notation. The object with template values is supplied when the scraping is initiated.

```
[ "!insert", [ "$p:first" ], "before", "<h2>Hello {{person.name}}!</h2>" ]
```

The `!remove` command takes one argument – the selector that is to be removed from the DOM. Finally, `!replaceWith` is used for replacing selected elements with a new content. It takes two arguments, the selector and the HTML definition of a new content.

4. User Stories

Serrano has proven its applicability in a number of real-world projects. Below, we pick three and discuss how Serrano has benefited them.

4.1. Magneto Calendar

Magneto[14] is a cloud-based calendar system that enables creation of meetings and to-dos from any web page and adding them to Google or Microsoft Exchange calendar. It also extracts key information for the corresponding events and stores it with the items. If the user visits a website that contains information suitable for a calendar event and clicks on the Magneto button (see Figure 1), a browser action window appears with extracted information of the event. To achieve this goal, Magneto uses custom-page wrappers, along with the default wrapper.

There were two main reasons for rewriting the rules in Serrano: (1) As the project expanded, the number of web sites and their respective wrappers became harder to maintain and manage. (2) Updating the whole extension every time a single wrapper is updated is stultifying to the user and bandwidth-consuming.

Separation and outsourcing the rules into Javascript would run into several problems, most important of which is *safety*. Javascript is a general-purpose language and allowing to execute arbitrary Javascript code in the extension would create a potential security hole. Furthermore, downloading and executing remote Javascript violates the conditions of the most application stores, for the same reason. Hence, the application could not be placed there. Usage of another wrapping language would also be problematic. Wrappers that were already written in Javascript involved processing of the scraped information, such as cleaning of the selected data from HTML tags, date processing etc.

When rewriting wrappers into Serrano, we identified common functionality across the wrappers and created new commands, including `!convert12hTo24h` which was used to convert the time of an event into a 24-hour clock, since some web sites use a 12-hour format. Further helper commands include `!textLB` (LB stands for line break) that appends a new line symbol after specific tags, such as `<div>`, `<p>`, `<br>`, `<hr>`. Another command was `!cleanupSel` for removing the tags and the superfluous white spaces from the selected text.

Next, we identified parts of the wrappers that required higher expressive power than Serrano had. We created commands that encapsulate this functionality and they work as black boxes. That is, the functionality that requires higher expressive power is encapsulated within the commands without granting the language higher expressive power. These constructs include while loops, exceptions etc.

Another challenge for Serrano was understanding of the date of an event on Facebook. Facebook, for user convenience, describes dates in various formats depending on when it is going to occur. Valid descriptions of a date include: *May 24, next Thursday, tomorrow* etc. Our workaround involved creating a `!parseFacebookDate` command, which was a raw copy and paste of the complex function featuring in the former Javascript wrapper. After some time, Facebook coupled additional microdata [16] with the event, so this command was removed.

After the replacement[15] of Javascript wrappers with Serrano scraping units, both maintainability and maintenance were increased.
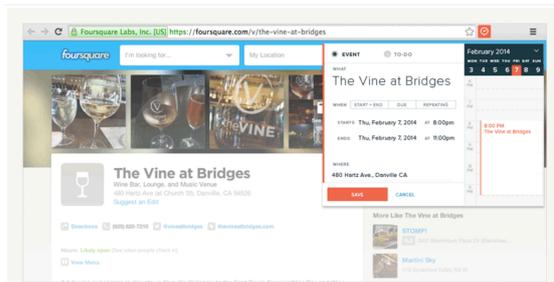
---

[14]`https://magneto.me/welcome/about-us.html`
[15]`https://github.com/salsita/Serrano/tree/master/magneto/scraping-units`

Figure 1. Magneto interface, when user clicked on the Magneto browser action button

Figure 2. Excerpt of the search results for "Effective Java" augmented by MyPoints extension

4.2. MyPoints

MyPoints[16] is a shopping rewards program that runs affiliate programs with 1900+ stores. It motivates people to make a purchase at associated stores to earn points, which can be then transformed into discount coupons and subsequently redeemed. Serrano was used in beta version of the extension. On websites with search results of a search engine, MyPoints extension injects a text informing the potential shopper about the amount of points they can earn, as shown in Figure 2. Moreover, when the user proceeds to a checkout in the store, it automatically fills in the input field with an available coupon. To serve this purpose, the commands for DOM manipulation[17] were added.

4.3. Video Downloader

In this case, an extension was built into a modified version of Opera browser.[18] The purpose of Video Downloader (VD) is to facilitate download of a currently played video and to enable one to eventually watch it offline. To accomplish this, VD identifies videos on the websites – either by recognizing the domain, or the player if the video is embedded – and attaches a small button that is displayed when user hovers over the video. VD applies Serrano rules for both player element and player content identification. Specifically, an instruction for player identification returns a player element, which is then supplied to the second Serrano instruction for download address identification.

During the implementation of the extraction rules we encountered two challenges. The first was that the video player element only needed to be extracted when it had a class attribute with `off-screen` value. This was achieved by extending the command set with `!if`. The second challenge was caused by the fact that some players use different forms of video embeddings. For example, Youtube uses both `<object>` and `<embed>` tags for embedding a video in an external source. However, Serrano was able to deal with this by conflating these elements in one selector.

5. Discussion

In this part we explain the motivation behind choosing Serrano in the above-mentioned projects. In all the projects we were limited to extraction within a browser extension which had to be safe and able to extract the required data. Along with Serrano we considered two alternatives: pure Javascript and in-browser wrappers such as iMacros/Selenium. These tools were primarily designed for writing UI tests, hence we refer to them as *testing tools*.

---

[16] http://mypoints.com/

[17] https://github.com/salsita/Serrano/wiki/Language-Spec##dom-manipulation

[18] The browser vendor wishes to remain undisclosed.

Technology	Safety	Learning Curve	Extensibility
Serrano	Volatile	Gentle	Good
Javascript	Low	None	Not Needed
Testing Tools	Fixed	Steep	None

Table 1. Key attributes of the technology

Technology	Easy tasks	Medium tasks	Complex tasks
Serrano	Easy	Medium	Very Hard
Javascript	Medium	Medium	Hard
Testing Tools	Easy	Medium/Hard	Impossible

Table 2. Suitability of the technology for a given project scope

In Table 1 we compare the technologies in terms safety, learning curve and extensibility. Regarding safety, executing arbitrary Javascript code poses a serious security risk. Testing tools are safe depending on the capabilities of their default command set. In terms of a learning curve for web developers, Javascript is the best option followed by Serrano, which uses CSS selectors and has very similar functions to Javascript. Testing tools have a very specific API suited for the scope they were designed for. Extensibility-wise, Javascript is a Turing-complete language with no need for extension of capabilities. Serrano has an expressive power determined by the command set. The basic set only supports CSS selectors but it can be theoretically extended to support everything Javascript does (by e.g. addition of an `!eval` command which would evaluate pure Javascript). Wrapping languages of testing tools are not extensible to the best of our knowledge.

In Table 2, we discuss the feasibility of the technology with regard to the scope of the task. For easy tasks with minimum scraping logic, both Serrano and testing tools score well, thanks to a built-in library of functions that make scraping easy, as opposed to native Javascript which requires a lot of code and libraries. In medium complexity tasks, Javascript is closing in due to its expressiveness. And very complex tasks are impossible to manage with testing tools since their command sets are impossible to extend. Writing very complex wrappers is admittedly more difficult in Serrano than in Javascript, but it is not impossible, since the command set can be extended to arbitrary expressive power.

6. Conclusion

The aim of our research was to create a web data extraction tool that could work in a restricted environment. We implemented a novel language, Serrano, which championed extensibility of the command set and separation of concerns. That helped to eliminate the need for any accompanying software further transformating and processing of the extracted data. Extensibility also works the other way – the command set can be reasonably restricted so that the wrappers will only be able to extract and process data to the extent they are allowed to. Deployment in real-world projects has proven the durability of the language as well as significance of the goals. Each project we faced contributed to broadening of the command set confirming its extensibility.

Despite the advantages, there still remain a few steps that can be taken to further improve the language. E.g., creation of a toolkit with a GUI, outsourcing wrapper creation and then dynamically downloading and updating them, or building a database of command sets so that the users of Serrano could find appropriate commands to personalize the language functionality.

Acknowledgements

Bibliography

1. *Document Object Model (DOM)*. W3C, 2005. http://www.w3.org/TR/REC-DOM-Level-1/cover.html.

2. Extensible Markup Language (XML) 1.0 (Fourth Edition), 2006. http://www.w3.org/XML/.

3. A vocabulary and associated APIs for HTML and XHTML, 2016. https://www.w3.org/TR/html5/.

4. *AJAX*. Mozilla Developer Network, 2017. https://developer.mozilla.org/en/ajax.

5. B. Adelberg. NoDoSE – a tool for semi-automatically extracting structured and semistructured data from text documents. *ACM Sigmod Record*, 27(2):283–294, 1998.

6. R. Baumgartner, S. Flesca, and G. Gottlob. The Elog web extraction language. In *LPAR*, pages 548–560. Springer, 2001.

7. R. Baumgartner, W. Gatterbauer, and G. Gottlob. Web data extraction system. In *Encyclopedia of Database Systems*, pages 3465–3471. Springer, 2009.

8. M. E. Califf and R. J. Mooney. Bottom-up relational learning of pattern matching rules for information extraction. *JMLR*, 4:177–210, 2003.

9. G. Cormode and B. Krishnamurthy. Key differences between Web 1.0 and Web 2.0. *First Monday*, 13(6), 2008.

10. D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. JSON.org, 2006.

11. E. Ferrara, P. De Meo, G. Fiumara, and R. Baumgartner. Web data extraction, applications and techniques: A survey. *Knowledge-based systems*, 70:301–323, 2014.

12. T. Furche, G. Gottlob, G. Grasso, O. Gunes, X. Guo, A. Kravchenko, G. Orsi, C. Schallhart, A. Sellers, and C. Wang. DIADEM: domain-centric, intelligent, automated data extraction methodology. In *WWW '12*, pages 267–270. ACM, 2012.

13. T. Furche, G. Gottlob, G. Grasso, C. Schallhart, and A. Sellers. OXPath: A language for scalable data extraction, automation, and crawling on the deep web. *The VLDB Journal*, 22(1):47–72, 2013.

14. G. Gottlob and C. Koch. Monadic datalog and the expressive power of languages for web information extraction. *JACM*, 51(1):74–113, 2004.

15. J. Hammer, J. McHugh, and H. Garcia-Molina. Semistructured Data: The TSIMMIS Experience. In *ADBIS '97*, pages 22–22, 1997.

16. I. Hickson. HTML microdata, 2011. http://www.w3.org/TR/microdata/.

17. N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1):15–68, 2000.

18. A. H. Laender, B. A. Ribeiro-Neto, A. S. da Silva, and J. S. Teixeira. A brief survey of web data extraction tools. *ACM Sigmod Record*, 31(2):84–93, 2002.

19. E. Oro, M. Ruffolo, and S. Staab. SXPath: extending XPath towards spatial querying on web documents. *Proceedings of the VLDB Endowment*, 4(2):129–140, 2010.

20. E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

21. A. Sahuguet and F. Azavant. Building intelligent web applications using lightweight wrappers. *Data & Knowledge Engineering*, 36(3):283–316, 2001.