FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Tomáš Novella

Web Data Extraction

Department of Software Engineering

Supervisor of the master thesis:	doc. RNDr. Irena Holubová, Ph.D.
Study programme:	Computer Science
Study branch:	Theoretical Computer Science

Prague 2016

Title: Web Data Extraction

Author: Tomáš Novella

Department: Department of Software Engineering

Supervisor: doc. RNDr. Irena Holubová, Ph.D., department

Abstract: Creation of web wrappers (i.e programs that extract data from the web) 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 restricted environment. In this thesis, we present a new wrapping language – Serrano – that has three goals in mind. (1) Ability to run in 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 encouraging results.

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

First and foremost, I would like to express my deepest gratitude to Irena Holubová, Ph.D – a supervisor par excellence – for scrunitizing my thesis and teaching me to become a better writer.

Next, I'd like to extend my appreciation to RNDr. Roman Kašpar, a true Salsitista, who helped me with the programming part by giving me priceless feedback and teaching me discipline to become a better programmer.

Special thanks goes to Miroslav Kratochvíl for always having time to discuss my ideas and giving me constructive critique and providing me with invaluable advice and to Veronika Machalová for meticulously proofreading this work and showing me that articles and commas are important.

Contents

1. Introduction

Since the dawn of the Internet, the amount of information available has been steadily growing every year. Email, social networks, knowledge bases, discussion forums – they all contribute to 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. Finding relevant information on the web and subsequent transformation into structured data is a challenge that web data extraction tackles. Structured data can then be processed by a computer to distill and interlink information, generate statistics, etc.

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.

Common business applications of web data extraction include, among other things, opinion mining, web application testing, web harvesting and competitive intelligence cultivation [1]. Apart from those, semantic web creation (transformation into linked data) greatly benefits scientists; social network mapping is useful for confirming hypotheses on population and web accessibility aids visually impaired users to take advantage of Internet.

Web data extraction has been dealt with from the very beginning of the existence of the WWW. Back then, web wrappers (i.e., programs that extract the desired data) were written in general-purpose languages like Java or Python. Soon afterwards, domain-specific languages have been created. Authors of these languages tried to identify the essence of an extraction task and create a convenient syntax. Although this has made the languages slightly less expressive, it has also made them more concise and the wrappers became easier to maintain. With the advent of new web technologies, such as AJAX [2], and the rise of the Web 2.0 [3], simple raw manipulation of HTML [4] proved no longer sufficient. As a result, extraction tools have started being bundled with an HTML layout rendering engine, or were 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 users to generate wrappers more easily. All this evolves in the direction to increase wrapper maintainability, which helps to take on incrementally larger tasks. Major challenges the tools available in the market currently face 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 obsolete. One of the main culprits is the inability to easily extend the tool to support current technologies.

- *Execution in restricted (browser) environment.* Finally, 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.

On these grounds we propose a novel data extraction language – Serrano, which deals with the problems discussed above.

1.1 Outline of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 contains a brief introduction of the Web. It outlines the relating main concepts and technologies and briefly defines terms that will be used throughout this thesis.

Chapter 3 discusses the taxonomy related to data extraction. It defines web wrappers and web data extraction and identifies its main responsibilities. It is followed by three classifications that allow us to effectively categorize every wrapper language and toolkit. Every category contains several examples of languages and their short description.

Chapter 4 introduces the motivation for a new language, discusses alternative solutions and outlines how we address their shortcomings.

Chapter 5 describes Serrano and its usage. It is not, however, a complete language specification; rather a concise highlight of the cornerstones.

Chapter 6 discusses the implementation aspect of Serrano.

Chapter 7 reviews the uses cases. Serrano has been successfully used in a number of commercial projects and this chapter showcases some of them.

Chapter 8 is the conclusion of this thesis and the potential further work is outlined.

2. Preliminaries

This chapter provides an overview of the World Wide Web, discusses key technologies and analyzes their relevance to data extraction. Is serves as an introduction to terms we will be using in the following chapters.

2.1 Web Documents

The World Wide Web (WWW, or simply the Web) is described as “an information space where documents and other web resources are identified by URLs, interlinked by hypertext links, and can be accessed via the Internet.”[1].

Lately, the purpose of the web documents (sometimes also called web pages) has shifted from presenting data to creating robust *web applications* and the so-called *deep web*.

Web documents can be in various MIME [5] types (document types). They can be free and unstructured (e.g. plain text), binary, or structured (RSS [6], XML [7]). However, the majority are semi-structured and implemented by three core technologies for web content production, namely HTML, CSS and Javascript.

In the remainder of this section, we provide a quick description of web documents and outline core technologies that prove helpful for the web data extraction. We then dedicate a section to Web applications and describe the challenges that complex applications pose to extraction.

2.1.1 HTML

The prevalent language used for composing web documents is HTML [4], which stands for the Hypertext Markup Language. This language enables us to introduce almost arbitrary structure into the document by marking parts of the document with special HTML tags. HTML can be easily transformed to XML which allows us to exploit the complete stack of technologies developed for XML manipulation. Some of these technologies are:

DOM The Document Object Model (DOM [8]) is a convention for representing HTML and XML documents as *labeled ordered rooted trees* where labels represent HTML tags. That way, we can exploit relationships between HTML nodes.

XPath is a language with the primary goal of locating elements in an XML document. The first version, XPath 1.0 [9], is described as “a language for addressing parts of an XML document, designed to be used by both XSLT [10] and XPointer [11]” and although it supports more, for example, arithmetics and string and boolean expressions, these features are kept to the minimum. It leverages[2] DOM and provides the ability to navigate through the tree and select nodes by a variety of criteria. XPath 1.0 lacks the expressive power of the first order logic and thus, several extensions

[1]`https://www.w3.org/Help/#webinternet`

[2]`https://www.w3.org/TR/DOM-Level-3-XPath/xpath.html`

have been developed and studied [12]. By version 2.0 [13], the specification has widely expanded and has introduced a new type model as well as added variables and loops to enhance its expressive power to enable us to express more complex relationships between nodes.

2.1.2 CSS

HTML leverages the Cascading Style Sheets [14] (CSS) to position the HTML elements and create a visually pleasing layout for the document. CSS makes it possible to prescribe style rules, such as font and background color to affect certain elements identified by CSS selectors [15]. Selectors provide us, alternatively to XPath, a way to select and target elements. The concept of CSS selectors has later been borrowed and reused in other technologies. A popular Javascript library, jQuery [16], extends them with pseudoselectors and thus creates central means to access elements within a web page [17].

2.1.3 Javascript

Javascript [18] is a dynamic programming language embedded in most modern web browsers and designed to interoperate with the DOM and CSS. It can listen to both user and DOM events, communicate with other web resources in the background via AJAX [2] and manipulate the DOM. Due to the differences of Javascript DOM implementation across the web browsers, many developers prefer jQuery selectors to native API.

Javascript has become an essential part of the majority of web pages and is a cornerstone of rich internet applications[3].

2.2 Web Application

Web application[4] is a server-client application that uses a browser as a client.

Even though a web application runs in the browser and strictly speaking, is a web document powered by the same technologies as a web document, there is a fundamental difference in paradigms between a web document and an application.

First hallmark of a web application is the notion of the state, i.e. accessing the same URL might render different documents. State is usually kept via cookie files and with Javascript.

Furthermore, navigation inside an application does not necessarily correspond with the change of the URL. A growing number of Web applications is placed on a single URL, the so-called Single Page Applications[5]. The DOM of these pages is generated on-the-fly by Javascript, depending on content retrieved by AJAX and the user actions.

Moreover, the user actions are not only triggered by clicking anchors (represented by `<a>` tags in the HTML structure), but by a great variety of events[6] done with arbitrary document part.

[3]`http://www.w3.org/TR/wai-aria/`

[4]`http://webtrends.about.com/od/webapplications/a/web_application.htm`

[5]`http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php`

[6]`https://developer.mozilla.org/en-US/docs/Web/Events`

2.2.1 Challenges for Information Extraction

Web Applications are by definition very complex and for correct functionality they require rendering engines that support the newest standards. Consequently, web data extractors need to support them as well to successfully conduct extraction from web applications.

2.2.2 Deep Web

The Deep Web are parts of the WWW whose contents are not indexed by a search engine. Examples are documents that require authentification (login), such as email services, social networks, private forums, et cetera.

These documents often are suited only for individuals and there is often deliberate effort by the service providers to make this content harder to access by programs.

2.3 Web Browser

Web Browsers are specialized programs that enhance web browsing experience, such as navigation between documents and visual document representation, by implementing up-to-date standards for web technologies. They are a place where web applications run.

Common internet browsers, including Chrome[7] and Firefox[8], support additional features unspecified by standards, such as *browser extensions*.

2.3.1 Browser Extension

A browser extension[9] is a small program – generally written in HTML, Javascript and CSS – that can extend functionality of a browser as well as improve the browsing experience. This is achieved by injection of mostly scripts into several contexts of a web browser. Below we describe the main contexts that are available for Chrome extensions.

1. *Web page context* is created whenever a new document is loaded. Javascript scripts that are injected into this context are referred to as *content scripts* and can read and modify the DOM of the current document.

2. *Background context.* Scripts executed there are loaded on browser start, can use advanced API functions and have no access to DOM.

3. *Popup context* is created on user request. Browsers that support extensions contain a *browser action bar* – generally located on the right of the address bar – which contains an extension-specific action. When user clicks on the action, a new context(in a new popup page) is created and forms a web document.

Scripts from different context can communicate via message passing[10].

[7]`https://www.google.com/chrome/`
[8]`https://www.mozilla.org/firefox/`
[9]`https://developer.chrome.com/extensions`
[10]`https://developer.chrome.com/extensions/messaging`

3. Data Extraction

This chapter studies data extraction. First, due to the lack of a unified terminology, we make our own definitions of a web wrapper and discuss its responsibilities. We also investigate the lifecycle of a wrapper, i.e. stages, that every wrapper has. After that, we dive into web data extraction toolkits, which are complete system suites for wrapper management. We finish the chapter by reviewing categorizations of web wrappers by three different aspects:

- Laender's and Riveriro-Neto's taxonomy [19], which organizes the wrappers and toolkits based on the main method used for wrapper generation.
- Our own taxonomy, based on techniques of element identification in [1].
- Real-world applications of extraction tools, also rooted in [1].

Remark. Terms *extraction tool*, *wrapping tool* or *scraping tool* are blanket terms for a program that may be a wrapper or a wrapping toolkit.

3.1 Web Wrapper and its Responsibilities

A wrapper is a specialized program that identifies data of interest and maps them to a suitable format. The name comes from the database community, where a wrapper is a software component that queries data and converts them from one model to another. The exact definition of a wrapper varies across literature and is often interchanged with the definition of the extraction toolkit. To facilitate a more profound analysis of wrappers, we use our own definition, inspired by [19].

Definition. A *web wrapper* is a procedure for seeking and finding data, extracting them from Web sources, and transforming them into structured data.

This definition is especially useful, because it provides us with a guideline for addressing different phases of wrapper design. Wrappers and toolkits usually use this terminology when discussing their functionality. Responsibilities are as follows.

1. *Data retrieval.* In the early nineties, a typical web wrapper managed to follow links by sending GET and POST requests and it sufficed to access majority of the web. In the presence of Web Applications, the Deep Web and Single Page Applications it however proves insufficient, and it is imperative that a web wrapper be able to simulate user interaction with high fidelity. The majority of interactions with a Web Application constitutes mouse clicks and filling forms.

2. *Extraction from Web sources.* In this phase, the wrapper locates information within the document. Depending on the structure of the text, it may use algorithms for natural language processing [20] (prevalently for unstructured text) or may exploit the DOM, for structured and semi-structured data.

3. *Transformation and mapping into structured data.* Most languages only support simple transformations, such as data trimming and the heavy lifting is left to the toolkit. Occasionally, the language supports data cleaning, deduplication and data merging. A flagship format for structured data is XML, as most wrappers and tools use, but there are other possibilities, such as JSON or CSV[1].

3.2 Lifecycle of a Wrapper

A lifecycle of a wrapper constitutes 3 phases which outline the key design decisions that had to be tackled by an inventor of a wrapper.

3.2.1 Wrapper Building

A wrapper can either be produced manually, induced or generated using machine learning. When produced manually, the programmer may use their expertise in the domain to tailor the wrapper. However, writing wrappers manually may be a tedious and time-consuming task, all the more when a website changes frequently. Wrapper induction deals with this issue by allowing users to specify the wrapper via toolkit. Best depiction is macro recording: user crawls the website and executes actions the wrapper should do. This approach makes wrapper generation more feasible, especially when a large number of domains are to be used for data extraction. Third option is to generate the wrapper automatically, making use of supervised learning. The system is fed with training samples and results from which it attempts to deduce the extraction rules. Unfortunately, this technique requires a huge amount of supervised training samples and the learned rules are not transferable to different domains. For more complex websites, this system often fails to find accurate rules completely. Many present wrappers are built by a combination of these techniques and are referred to as *hybrid wrappers*.

3.2.2 Wrapper Execution

In this step, a wrapper is being executed. The execution environment varies and depends on the sophistication of the system. Simple approaches implement their own parser that identifies the data, e.g. Deixto [21] toolkit uses a Perl script and custom libraries. More robust approaches rely on a DOM library without having a specific browser attached to it, or use a web view API available in their environment to access advanced browser capabilities, such as cookie storage, CSS box model[2], etc.

3.2.3 Wrapper Maintenance

For a long time, the importance of this phase has been undervalued. Badly maintanable wrapper results in higher human engagement, thus higher maintenance costs over time. To address this issue, Kushmerick [22] defined a concept of

[1]https://tools.ietf.org/html/rfc4180

[2]`https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Box_Model/Introduction_to_the_CSS_box_model`

wrapper verification. The idea is that during the wrapper execution, the system assesses if data have been extracted correctly, or if some data are missing. Moreover, attempts for *automatic wrapper adaptation* [23] have been made. These works rely on the idea, that despite structural alterations of the page, the properties of data, like string lengths or patterns remain unchanged.

3.3 Web Data Extraction Toolkit

Definition. Web Data Extraction Toolkits, or Web Data Extraction Systems refer to a comprehensive suite of programs and services that constitute tools for web wrapper lifecycle management, usually presented as single piece of software.

Remark. In the literature, the definition and the terminology are non-uniform. For example, according to [24] a Web Data Extraction Toolkit is “a software extracting, automatically and repeatedly, data from Web pages with changing contents, and that delivers extracted data to a database or some other application”. Interestingly, this definition introduces three aspects that largely overlap with our definition. These aspects are as follows.

1. Automation.
2. Data transformation
3. Use of extracted Data

Toolkits are often equipped with 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 WebView in order to obtain the desired data. User actions are recorded as DOM 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 [25], especially when information is collected from multiple sources.

Ultimately, extracted data are saved in a structured form in a universal format, such as XML, JSON, or into a database.

3.4 Laender’s Taxonomy

To our best knowledge, one of the first endeavors to classify Web Data Extraction toolkit comes from Laender et al. [19], who 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, HTML-aware tools, NLP-based tools, Wrapper Induction tools, Modeling-based Tools and Ontology-based tools.

Similarly to other categorizations, it is not rigid and definite, therefore toolkits usually fall into multiple classes. In the remainder of this section we briefly discuss these categories.

3.4.1 Languages for Wrapper Development

In addition to general-purpose programming languages such as Perl or Java, specially designed programming languages have been developed to assist wrapper writers to create wrappers more efficiently. Some of the examples include Minerva, TSIMMIS, Jedi, as well as a number of languages explored in Section 3.5.

3.4.2 HTML-aware Tools

Languages that rely on the inherent structure of HTML are grouped here. These tools turn the document into a parsing tree, following the creation of extraction rules automatically, or semi-automatically. Some representative examples include W4F [26], XWRAP [27] and RoadRunner [28] and Lixto [29].

3.4.3 NLP-based Tools

Natural language processing (NLP) comprises techniques used for processing free documents (i.e. in natural language). NLP-based toolkits apply techniques such as filtering, part-of-speech tagging, lexical and semantic tagging to establish relationships between phrases and sentence elements. Henceforth, extraction rules can be derived studying these relationships. Representative tools based on such an approach are RAPIER [30], WHISK [31] and SRV [32].

3.4.4 Wrapper Induction Tools

Typically, extraction rules are generated by feeding the toolkit with training examples. The main distinction between these tools and the NLP-based tools is that they do not rely on linguistic constraints, but rather on formatting features found in the document. This makes these tools more suitable for documents with inherent structure such as HTML documents. Examples of this group of toolkits include WIEN [33], SoftMealy [34] and STALKER [35].

3.4.5 Modeling-based Tools

This category consists of tools that for a pre-defined domain model (structure of objects of interest) try to locate conforming data on the web pages and extract them. Tools that adopt this approach are NODoSE [36] and DEByE [37].

3.4.6 Ontology-based Tools

Tools in this category use a given application ontology (description of key terms and their relations) to locate data constants in the web page. These constants then help with object construction and that help with the automatic wrapper generation. The advantage of this approach is, that the wrapper is less sensitive to minor structural changes of the web page. The pioneer effort comes from Data Extraction Group [38] and the most recent effort, to our best knowledge, is DIADEM [39].

3.5 Techniques of Element Identification

This section discusses the underlying techniques wrappers can use for locating data in the documents. Most wrappers combine two or three of them to compensate for their deficiencies. In some cases, the techniques overlap and there is no clear distinction.

3.5.1 Regular Expression-based Approach

Regular expressions (regexps) is a concise domain-specific language for finding patterns in text. When using this technique, we view the document, or segment of a document as plain text, i.e. we do not assign any special importance to HTML tags. One of the advantages is the ability to search for patterns even in unstructured text, like emails and articles, which makes it a viable tool applicable in wide contexts. Furthermore, regexps are relatively fast and expressively equivalent to L4 grammars, which makes them a feasible solution for most cases. Alternatively, they are usually not the best option for data extraction of semi-structured and structured data, because they ignore the additional information provided by the markup. Due to the rigidity of regexp wrappers they tend to break even at minuscule changes in the DOM, which introduces maintenance costs.

W4F [26] is an example of a regular expression-based toolkit. In its architecture, it identifies and separates the web wrapper responsibilities (retrieval, extraction and mapping) and keeps these layers independent and minimalistic. Consequently, it lacks data transformation capabilities and leaves that part to a separate program. Wrapper generation is aided by a *wizard* procedure, that allows users to annotate the document directly in the web page, and generates regular expression-based rules. Finally, expert users can make manual wrapper optimizations on the resulting rules. Extraction rules include *match* a *split* commands which are specified by a regular expression and also include simple DOM manipulation commands.

3.5.2 Tree-based Approach

This approach makes use of the tree-based character of structured and semi-structured document, like HTML or XML, by building the DOM. Afterwards, it enables navigation by exploring node relationships inside the document. Since the node addressing via DOM is more likely to be conserved than regular expressions, building wrappers this way lowers maintenance costs. Here, we adopt the techniques mentioned in Section 2.1, namely XPath and CSS selectors. On the other hand, building the DOM is computationally relatively expensive and does not completely eliminate the possibility of a wrapper breakdown after a structural change in the document.

An example of a wrapping language that uses this approach is OXPath [40], which is based on XPath 1.0 and enriches it with navigation and extraction capabilities. It introduces a concept of *declarative navigation* which turns scraping into a two-step process:

1. choose relevant nodes and

2. apply a particular action to them

The language adds four extensions to XPath, namely:

- CSS properties. For lightweight visual navigation, computed style of rendered HTML is exposed. For this, a new axis for accessing CSS DOM properties is introduced. Selection based on visibility is also possible. For example, `//span.price[style::color='red']` selects spans of class price that are rendered red.

- User actions, such as click and form filling. For example, `//field(2)/click` clicks on the third input field on the page.

- Bounded and unbounded navigation sequences, and the Kleene star. This helps to deal with navigation over paginated data (such as the "Next button"). For example, `(//a#next/{click /})*//h1` selects all first level headings on on first page as well as the pages that can be reached by clicking on the *a* element with ID=next.

- (Hierarchical) data extraction. OXPath contains a special directive for extracting nested data, that can be stored as XML or in a relational database. Example, `//div.result/span[1]:<price=string(.)>` selects all prices from the second span and stores them in variable `price`.

To indicate how OXPath actually works, we provide an actual example for which this wrapper is asked to extract bibliographical data from `http://scholar.google.com` for the search results for query `world wide web`. Figure 3.1 shows how it is done along with the OXPath code.

OXPAth has been employed in a number of cases, such as the tool DIADEM [39], which transforms unstructured web information into structured data without human supervision.

Despite its usage, the language exhibits a number of weaknesses, such as incapability to make even trivial transformations on the extracted data(for example, splitting the address, etc), therefore it requires further post-processing.

OXPath uses the HtmlUnit[3] Java browser engine. However, its rendering differs from current browsers and fails for certain heavily scripted cases. At the time of writing this thesis, OXPAth is being ported to Selenium WebDriver which enables it to access the actual rendering of a page in current browsers.

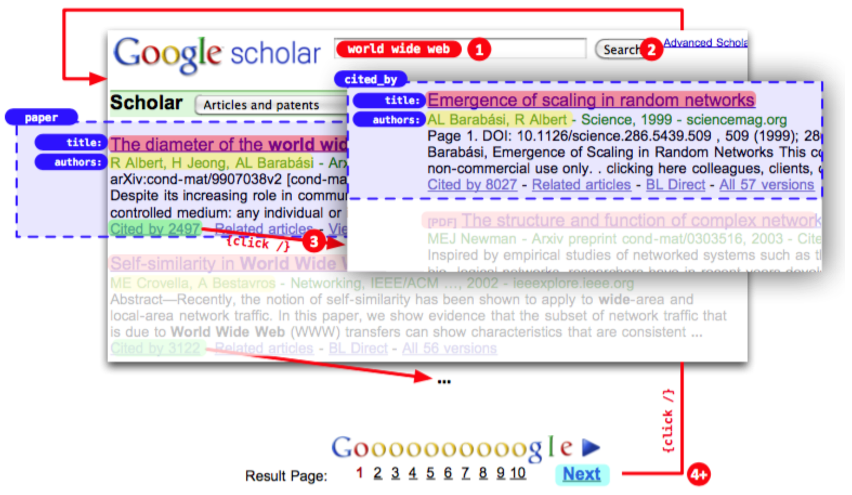
3.5.3 Declarative Approach

In general, a wrapper is considered declarative, if there is "a clear separation of extraction rules from the computational behavior of the wrapping engine" [42].

Two representative approaches that fall into this category are the rule-based prolog-like approach, exploited in Elog and the database-like approach represented by WebOQl.

Elog [43] is a logic-based language that is derived from Monadic Datalog [44], which is in turn derived from Datalog, a syntactic subset of Prolog. Elog views the document as a labeled and ordered (finite) tree and uses relations, such as

[3]`http://htmlunit.sourceforge.net/`



```
doc("scholar.google.com")/
descendant::field()[1]/{"world wide web"}❶
 /following::field()[1]/{click/}❷
 /(//a[contains(string(.),'Next')]/{click/})*❹
 //div.gs_r:<paper>
  [.//h3:<title=string(.)>]
  [.//*.gs_a:<authors=substring-before(.,' - ')>]
  [.//a[.~'Cited by']/{click /}❸
 //div.gs_r:<cited_by>[.//h3:<title=.>]]
```

Figure 3.1: OXPath extracts bibliographical data from `http://scholar.google.com` [41]

nextSibling and *firstChild* for navigation. Expressive strength equals to monadic second-order logic[4], hence more expressive than XPath. In contrast to MSO however, it is easy to build a GUI on the top Elog.

Standardly, the rules have a following format:

$$New(S, X) \leftarrow Par(, S), Ex(S, X), \phi(S, X)$$

where S is the parent instance variable (in terms of which the filter is defined in the visual specification process), X is the pattern instance variable, $Ex(S, X)$ is an extraction definition atom, and $\phi(S, X)$ is a (possibly empty) set of condition atoms. New and Par are pattern predicates. A standalone example to illustrate a real-world task is provided in Figure 3.2. This language made its way to the commercial world through the Lixto [45] toolkit, which provides a GUI and is powered by Elog.

Other examples of declarative languages are the ones developed in the database community such as Florid [46] and WebOQL [47]. Both thus share similarities with SQL.

WebOQL is an example of a language that views the Web as a database that can be queried using declarative language. The main data structure are *hypertrees* which are ordered arc-labeled trees that have two types of arcs, internal and external. Internal arcs are used to represent structured objects and external

[4]MSO is a restriction of a second-order logic in which only quantification over unary relations is allowed.

```
 tablesq(S, X)  ←  document("www.ebay.com/", S), subsq(S, (.body, []), (.table, []), (.table, []), X),
                   before(S, X, (.table, [(elementtext, item, substr)]), 0, 0, _, _), after(S, X, .hr, 0, 0, _, _)
  record(S, X)  ←  tableseq(_, S), subelem(S, .table, X)
 itemdes(S, X)  ←  record(_, S), subelem(S, (*.td. * .content, [(a, , substr)]), X)
   price(S, X)  ←  record(_, S), subelem(S, (*.td, [(elementtext, \var[Y].*, regvar)]), X), isCurrency(Y)
    bids(S, X)  ←  record(_, S), subelem(S, *.td, X), before(S, X, .td, 0, 30, Y, _), price(_, Y)
currency(S, X)  ←  price(_, S), subtext(S, \var[Y], X), isCurrency(Y)
```

Figure 3.2: Elog Extraction Program for Information on eBay

represent references (i.e., hyperlinks) among objects. The main construct is the familiar *select-from-where* so the query usually reads well and conveys the request in an intuitive form. Like many similar languages it suffers from a common limitation: lack of support for exploiting more complex document structure.

3.5.4 Spatial Reasoning

The idea here is that web pages are human-oriented and the visual arrangement of content in the web pages provides humans with additional cues, such as proximity of related elements. These visual cues tend to change less frequently than the layout.

Moreover, most real-world documents, often use sophisticated positioning and deep nesting of elements to create an appealing layout. Thus, even if the structure is visually simple, the underlying HTML may not be. This creates a conceptual gap between document and layout structure. Typical problems are incurred by the separation of document structure and the ensued spatial layout, whereby the layout often indicates the semantics of data items. E.g., the meaning of a table cell entry is most easily defined by the leftmost cell of the same row and the topmost cell of the same column. In real-world Web pages, such spatial arrangements are frequently hidden in complex nestings of layout elements — corresponding to intricate tree structures that are conceptually difficult to query.

One representative example is SXPath [48], which is an abbreviation for Spatial XPath. The language is a superset of XPath 1.0, it builds on Rectangular algebra, where each visible element is represented by a *minimum bounding rectangle* (MBR) which is the minimum rectangle that surrounds the contents of the given visible element.

With that, it introduces SDOM (spatial DOM) that enriches DOM with spatial relationships between MBRs, such as the *contains* and *contained* and *equals* relations.

Design-wise, it extends XPath with two navigation primitives.

- *Spatial Axes* are based on topological and rectangular cardinal relations that allow selecting document nodes that have a specific spatial relation with regard to the context node.

- *Spatial Position Functions* make use of spatial orderings among document nodes, and allow for selecting nodes that are in a given spatial position with regard to the context node.

The advantages of this approach include increased maintainability of wrappers and more intuitive usage, because the queries reflect what the user sees on the web page, not what the underlying structure is. On the other hand, the premise may be false and usage of spatial reasoning should be delegated to the domain expert.

3.5.5 Machine Learning-based Approach

Machine learning-based approaches rely on the training session where the system acquires the domain expertise. It requires a training step, where the system obtains the tagged training data. Subsequently, it deduces the wrapping rules. Machine learning-based approaches usually employ techniques from natural language processing.

RAPIER [30] (Robust Automated Production of Information Extraction Rules) uses pairs of sample documents and filled templates to induce pattern-match rules that directly extract fillers for the slots in the template. Each rule consists of a pre-filler, filler, and post-filler pattern that match the preceding of the text, the text itself and the following part. Each pattern can impose constraints upon the text, such as the exact word match, the POS (part-of-speech) tag and the semantic class (that is, available synonyms).

On the one hand, Rapier rules are flexible, because they are not restricted to contain a fixed number of words but, on the other hand, it is hard to recognize what rules are actually useful to perform data extraction. To this purpose, a learning algorithm has been developed to find effective rules and this algorithm is based on Inductive Logic Programming.

The main strength of this system is that the resulting rules do not require prior parsing or subsequent processing.

WHISK [31] is a tool designed for versatility. It handles a wide variety of document types, both unstructured (free) texts, structured XML data and mixed content. Generation of extraction rules is similar to regular expressions with slight differences, e.g. asterisk is not greedy but is limited to the nearest match. In structured texts, the rules specify a fixed order of relevant information and the labels or HTML tags that delimit strings to be extracted. For free text, initial syntactic analysis and semantic tagging are employed to recognize the domain objects and to annotate the text. Annotation is done by inserting markup, similar to XML to identify parts of speech a special constructs. After a successful match, the rule is reapplied to the remaining text.

Example of a WHISK rule is

```
 Pattern:: * (Digit) ‘ BR’ * ‘$’ (Number)
 Output:: Rental {Bedrooms $1} {Price $2}
```

and when applied on text

```
 Capitol Hill { 1 br twnhme. fplc D/W W/D.
Undrgrnd pkg incl $675.<br />  3 BR, upper flr
of turn of ctry HOME. incl gar, grt N. Hill
loc $995. (206) 999-9999 <br>
```

it returns:

```
 Rental Bedrooms 1 Price $675
 Rental Bedrooms 3 Price $995
```

This approach performs well, when the text is straightforward, vocabulary is uniform and conversely, has trouble when the surrounding context is highly variable and hence identification of patterns is problematic.

3.6 Applications of Data Extraction

In [1], the authors identify and provide a detailed analysis of 14 enterprise applications of data extraction, namely: Context-Aware Advertisting, Customer Care, Database Building, Software Engineering, Competitive Intelligence, Web Process Integration, Web Application Testing, Comparison Shopping, Mashups, Opinion Mining, Citation Databases, Web Accessibility, Main Content Extraction and Web Experience Archiving.

Most of these applications take form of a standalone program. However, tools that belong to *Web Application Testing* category can run within the browser as a browser extension. In the remainder of this section, we inspect two representative tools from the category, iMacros and Selenium, along with the underlying extraction language they use.

3.6.1 Selenium

Selenium[5] was created with the aim of browser automation. Selenium offers two main services, Selenium IDE[6], which is a browser extension to Firefox and Selenium WebDriver[7], which is a collection of bindings to programming languages, so that the user is able to work with the data after processing.

Users can either write custom macros that consist of a sequence of commands to be executed, or simply press the "Start recording" button and manually perform the actions in the browser. The extension will record the action and generate a wrapper. Wrapper consists of a sequence of commands to be executed. Apart from basic commands for navigation, such as form filling, clicking on the buttons and following the links, it is also equipped with testing related functionality, like asserts for a concrete value in a given selector. This tool provides front-end web testers with unparalleled services to the point that it has become a de facto standard in the web testing stack. Although the tool provides good example of handling browser navigation and automaton, it does not provide support in other areas, such as data extraction.

Selenium Web Driver offers bindings to various programming languages and enable us to use the commands from selenium to write our of wrapper in the languages. Naturally, these wrappers, since written in a general-purpose language, can extract data.

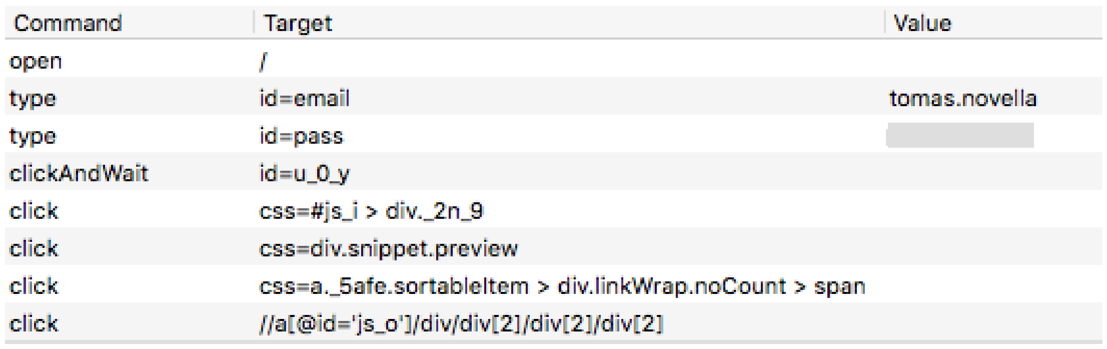
Example

We examined this tool by a real-world example, where we logged into Facebook and selected the second most recent conversation. The source of the macro for Selenium IDE is found in Figure 3.3.

[5]urlhttp://www.seleniumhq.org/

[6]`https://addons.mozilla.org/en-US/firefox/addon/selenium-ide/`

[7]`http://www.seleniumhq.org/projects/webdriver/`



Command	Target	Value
open	/	
type	id=email	tomas.novella
type	id=pass	[redacted]
clickAndWait	id=u_0_y	
click	css=#js_i > div._2n_9	
click	css=div.snippet.preview	
click	css=a._5afe.sortableItem > div.linkWrap.noCount > span	
click	//a[@id='js_o']/div/div[2]/div[2]/div[2]	

Figure 3.3: A series of commands that leads to logging into `https://www.facebook.com` and choosing the second most recent conversation

3.6.2 iMacros

iMacros[8], formerly known as iOpus, strongly resembles Selenium and is in fact targeting a very similar audience. It even provides a website that lists the distinctions between iMacros and Selenium [9]. The use cases differ from Selenium and are as follows:

1. Browser Automation
2. Web Testing
3. Data Extraction

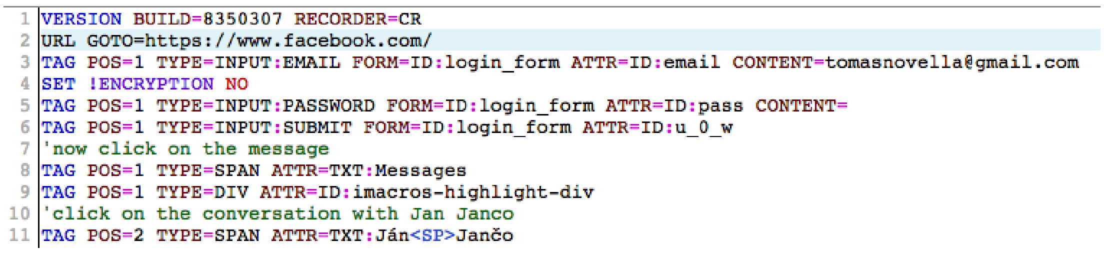
Browser Automation and Web Testing

In addition to functionality provided by Selenium, it offers a possibility to extract data into variables and to navigate through more complex dynamic pages, supporting sliders (`http://maps.google.com`) and drag-and-drop functionality. Identification of the elements on the page is either by XPath, CSS selectors, or by the element's type, positions and attributes. On top of this, is offers functionality to detect and intercept built-in browser dialogs, such as the download dialog and native Javascript dialogs. A useful advanced feature is the image recognition functionality. It relies on rendering of the image and using advanced neural networks it can identify the element, even if it has moved, or has changed color.

Data Extraction

This is the main differentiator; in this domain, we can select a variable to which the data will be extracted. Plain-text and CSV output formats are supported. One of the most significant drawbacks is the lack of any structure in the extracted data, it is a simple flat list of data.

[8]`http://imacros.net/`
[9]`http://wiki.imacros.net/Selenium`



```
VERSION BUILD=8350307 RECORDER=CR
URL GOTO=https://www.facebook.com/
TAG POS=1 TYPE=INPUT:EMAIL FORM=ID:login_form ATTR=ID:email CONTENT=tomasnovella@gmail.com
SET !ENCRYPTION NO
TAG POS=1 TYPE=INPUT:PASSWORD FORM=ID:login_form ATTR=ID:pass CONTENT=
TAG POS=1 TYPE=INPUT:SUBMIT FORM=ID:login_form ATTR=ID:u_0_w
'now click on the message
TAG POS=1 TYPE=SPAN ATTR=TXT:Messages
TAG POS=1 TYPE=DIV ATTR=ID:imacros-highlight-div
'click on the conversation with Jan Janco
TAG POS=2 TYPE=SPAN ATTR=TXT:Ján<SP>Jančo
```

Figure 3.4: A series of commands that leads to logging into `https://www.facebook.com` and choosing a conversation with Jan Janco.

Example

Again, we tried to log into Facebook and select the second conversation. Automatic recording broke down, but after some manual effort we made a macro in Figure 3.4.

3.7 Our Contribution

In this thesis, we propose what we believe is an advancement in in-browser data extraction: a wrapper that can be executed in a browser extension environment. In complement to the testing tools above, we set a goal to augment the user experience.

In addition, we make an adjustment to the classification of applications: we introduce an additional category, *Web Browsing Experience Augmentation*.

4. In-browser Web Wrapping

In the previous chapter, we concluded that in-browser data extraction is an unexplored facet of the web data extraction. Browsers and browser extensions provide an opportunity to personalize and integrate existing services and result in augmentation of the web browsing experience.

Browser extensions are popular. Most downloaded extensions these days have reached tens of millions of users, for example Adblock has about 40 million users[1] and according to Chrome extension usage statistics[2], approximately 33% of users have at least one extension installed.

Numerous extensions actively extract data from the web pages to improve the user experience. Extension usage statistics are generally a private information and few browser vendors disclose them. Therefore, most popular extensions (in regards to number of downloads) are hard to determine. Below, we list several extensions with millions of active users that extract data from the web pages and use them.

- Adblock Plus[3], an extension for blocking unsolicited advertisements. That is accomplished either by blocking the request that loads the ad content, or by hiding elements on the webpage.
- Grammarly[4] is an extension that looks for grammar and spelling errors made by user while typing text to `textarea` fields.
- Antivirus extensions, by antivirus vendors, e.g. Avast[5], generally scan for links and potentially malicious content.
- LastPass[6] is a password management service, i.e. it collects, saves and autofills passwords into certain web pages and frees the users from remembering difficult passwords.

To sum up, browser extensions are a relevant and important sector for web data extraction. In the following sections, we analyze the specifics of in-browser data extraction and discuss alternative approaches that lead us to creation of a new wrapping tool – Serrano.

4.1 Specifics of In-browser Extraction

Majority of tools we reviewed were not adapted for in-browser extraction. This restricted environment has the following specifics.

[1] According to the data on Google Play
[2] `http://blog.chromium.org/2010/12/year-of-extensions.html`
[3] `https://adblockplus.org/`
[4] `https://www.grammarly.com/`
[5] `https://chrome.google.com/webstore/search/avast?hl=en`
[6] `https://lastpass.com/`

Support of Web Standards Since the extraction takes place in the browser, the language can benefit from an up-to-date rendering engine and CSS box model as well as Javascript engine, which enable it to render even the most complex web pages reliably.

Accessibility to Private Data In-browser scraping tools usually have an access to all cookies and sessions saved in a browser instance. For example, when sending an AJAX request from extension environment, the browser appends automatically all the relevant cookies. Therefore, the wrappers can access the same things as a user and the user has full control of what the script can access.

Scalability and Distributed Networks With a large user-base, in-browser scrapers can jointly scrape substantial amount of data, including the Deep web, and either distribute them or synchronize with the central authority. In this case, it is essential to consider the privacy aspects of the matter. Potential solutions have been investigated in fields, such as Differential privacy [49], which investigates techniques for querying collected information while minimizing the chances for record identification.

4.2 Approaches to In-browser Data Extraction

This section discusses alternative approaches that are being used for data extraction in extension and in browser.

4.2.1 Javascript

Most extensions implement the data extraction-related functionality in plain Javascript. This approach, although in small scale manageable, quickly becomes clunky and cumbersome as the project scales (that is, large amount of page-specific wrappers are required), for the following reasons.

Violation of the Single Responsibility Principle [50]. First of the SOLID[7] principles, it is one of the key principles of software design. It states that "one object should have only one reason for a change". Some of the practical applications include the separation of the web page semantic structure (HTML) from visual appearance (CSS), or separation of the data representation (model), visual layout (view) and the controller in the MVC [51] architecture. Here, by designing a separate language for data-extraction tasks, we can decouple responsibilities and design a language that avoids the boilerplate code that is inevitable in Javascript.

Safety. Having a separate language with reduced expressive power makes the language safer to use. For example, it makes it possible to delegate wrapper creation to third-party developers and eliminate the risk of execution of malicious code.

[7]`http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`

4.2.2 Domain-specific Language

Another approach is to use one of the domain-specific languages. In Section 3.6, we discussed the in-browser tools that use a specialized language. These languages more or less mitigate the downsides of using Javascript, although they introduce other challenges discussed in the remainder of this section.

Integration and Embeddability The wrappers are often not first-class objects; that is, they cannot be dynamically loaded and unloaded. On top of that, tools often do not integrate with Javascript easily, which prevents their use in a browser extension and in formation of an extraction layer that can communicate with the extension environment.

Extensibility and Adaptability With the constantly changing face of the web and continuous adoption of new technologies, it is crucial that the data extraction be easily extendable with new functionality. Otherwise, the technology becomes soon dysfunctional and obsolete. Moreover, with a wide range of extraction tasks, different functionality is appreciated in various contexts. Since the choice of an appropriate tool in browser extension context is limited, the wrapper writers are often reluctant to resort to Javascript.

Learning Curve Designers of extraction languages usually have to invent their own commands and structures from scratch, which require further documentation. However, the increasing level of complexity makes it hard to familiarize new team members with the language. For this reason, many tools feature a GUI, which makes wrapper generation easier.

Developer tools We believe that novel languages usually contain bugs and glitches, which make them an unsuitable candidate for pioneering application in projects. One way to mitigate this is to equip the language with developer tools so that the users can debug the problem and create an alternative, workable solution.

Maturity Wrapping technologies in browser extensions have been around only recently and are generally less mature than their outside-the-browser counterparts. By reinventing the wheel and not building on sound foundations, novel approaches are often condemned to failure.

Licensing Several tools are licensed under proprietary licenses, which hinder further development of the language. Moreover, commercial projects often cannot use languages under licenses, such as GPL. When it comes to the the future prospect of the tool, licensing is significant differentiator.

5. Proposed Language

In this chapter, we propose a new data extraction language – Serrano. We explain the design choices and contrast the language with its competitors. We finish the chapter with a description of the language grammar and structural breakdown, features with examples. For a complete in-depth specification, the reader is advised to study the official language specification[1].

5.1 Serrano Language Design Overview

This section examines and explains why Serrano was designed the way it was. It mentions Gottlob's [44] four requirements for a good and durable data extraction language and also addresses the issues of other domain-specific languages from Section 4.2.2.

5.1.1 Gottlob's Requirements

Gottlob presented four desiderata that would make an ideal extraction language. These attributes are following.

Solid and well-understood theoretical foundation. Serrano uses jQuery selectors – which are a superset of CSS selectors – for locating elements on the web page. These technologies have been studied in depth along with their limitations and computational complexity. CSS selectors are well-known for web developers and simple, thus require minimum learning time for newcomers. Regarding their expressive power, they are weaker than XPath 2.0 (which has the expressive power of at least first-order logic) or Elog (equivalent to monadic second-order logic). Empirically, CSS selectors can handle most of the extraction tasks at hand, although eventual addition of other technologies is considered.

Serrano wrapper is a valid JSON and every command directly corresponds to a Javascript command.

A good trade-off between complexity and the number of practical wrappers that can be expressed. One of our language'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 scale of wrappers can be expressed. The command set of most languages is immutable, although some, such as Selenium offer a way to add new commands[2]. Extensions in Serrano are especially easy to make, because enriching a command set corresponds to merging a new command object with the old command set object.

[1] `https://github.com/salsita/Serrano/wiki/Language-Spec`
[2] `http://www.seleniumhq.org/docs/08_user_extensions.jsp`

Easiness to use as a wrapper programming language. Many Serrano commands have the same name and arguments as their Javascript counterparts, therefore wrapper programmers can get up to speed in no time.

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.

5.1.2 Other Domain-specific Languages' Issues

This section addresses issues outlined in Section 4.2.2 and explains how Serrano deals with them.

Integration and Embeddability In order to make a language easy to integrate with Javascript, we leveraged JSON [52]. In contrast of other data transmission formats, such as XML, JSON has been strongly integrated into Javascript, which eliminated the need of additional helper libraries for processing. In Serrano, both the wrapper and the result are valid JSON objects. This makes them very 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[3] further extend object manipulation capabilities.

Extensibility and Adaptability To deal with the issue, Serrano has separated the command set and allows to create own commands. Example of such extension are commands for document editing, which makes Serrano, to our best knowledge, the first data extraction as well as data editing language used in the browser. With simple extension of commands, we can allow Serrano to manipulate the native Javascript window object, manage user credentials[4] or change the page location. This offers expressive power and control beyond the range of most extraction tools.

Wrapper maintainability is another design goal of the language. Powerful commands, such as conditions and type checks, make writing verification inside the wrapper possible and straightforward.

Learning Curve A steep learning curve of a new language may discourage its adoption. Due to this fact, we decided to make a transition to Serrano as smooth and effortless as possible, by adopting features directly from Javascript, with familiar naming schemes and rationales. The wrapper writers can thus use their knowledge of Javascript to become productive immediately. Serrano uses jQuery framework under the hood and exposes numerous native Javascript functions. For better readability, Serrano is inspired by Dataflow programming [53] and the Lisp [54] programming language and makes control flow explicit, which makes the program structure more intuitive [55]. We support UNIX pipes "`|`", which helps

[3]`https://lodash.com/`

[4]`http://w3c.github.io/webappsec-credential-management/`

us to link the output of one command with the input of another. In Serrano, we denote a pipe with a “`>`” sign.

Developer Tools Since Serrano itself is a library written in Javascript, it can use Developer Console of the browser for debugging. Additionally, it offers playground for further experimentation. Complex wrappers can be disassembled into individual instructions and debugged incrementally. Supplementary extensions have been written to make the wrapper creation easier.

Maturity Although the language itself is new, the technologies in the background are several years old and field-tested.

Licensing Serrano is licensed under the MIT license[5], which makes it an ideal candidate for eventual third-party development. This license allows incorporation and usage of the language in both commercial and non-commercial purposes as well as further modifications of the code base.

5.2 Structure Breakdown

This section outlines the structure of a Serrano script. We describe it bottom-up, starting with the most basic primitives and work our way up to the global document. Since the Serrano wrapper is written in valid JSON, most control structures take form of a Javascript array and object.

5.2.1 Type System

Serrano type system inherits from the Javascript type system. It supports all the 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`.

5.2.2 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, a command, a selector and an instruction.

Command

Commands are the core control structure of Serrano. As such, they appear similar to functions in common programming languages; in a sense 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 structure is the cornerstone of flexibility and extensibility: all control structures

[5]`https://github.com/salsita/Serrano/blob/master/LICENSE.txt`

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

Syntactically, a command is a JSON array, where the first element has a string type denoting the command name which is 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 our example, `!replace` is a 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). Example of such a command is the `!constant` command, that takes one raw argument and returns it. Had the argument not be specified as raw, the constant command would return a string `hello mars`. More detailed explanation of the `!constant` command is done in Section 5.3.5

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

Implicit foreach. By default, commands have what we call *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 on each element, returning a list of results. It is also known as the *map* behavior. Conversely, when a command does not have implicit foreach, the first argument is passed as-is, even despite being an array.

An example illustrates two commands. The first 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"]
```

[6]`http://www.seleniumhq.org/docs/08_user_extensions.jsp#chapter08-reference`

```
// 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[7]).

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[8]) 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 with 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) 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 a 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

[7]https://api.jquery.com/category/selectors/
[8]http://api.jquery.com/jquery/

implicit argument. Otherwise, the result of the previous command is discarded. The example below illustrates three examples of upper casing the `hello` string. First scraping directive is an instruction that constructs a `hello` string and passes it into the `!upper` command. Second scraping directive is a direct command and third example 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. Last example 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
```

(Scraping) action is a scraping directive, that manipulates the DOM. It may trigger some event (e.g. simulate a mouse click on an element), or insert/remove a part of the DOM.

Formal scraping directive definition

Below we provide a formal definition of a scraping directive.

```
scrap-directive = <selector> |
                  <command> |
                  <instruction>

instruction = '[' ( <selector> | <command> )
    [ ',' <chained> ] ']'

selector = '[' <selector-name> ']'

selector-name = ( '' | '>' ) +
    ( '$' | '~' | '=' ) + <string>

command = '[' <command-name> [ ',' <args> ] ']'

command-name = ( '' | '>' ) + '!' + <string>

args = <arg1> [ ',' <args> ]

arg1 = (
         <string> |
         <number> |
         <object> |
         <array>  |
            <selector> |
            <command>  |
            <instruction>
        )

chained = ( <selector> | <instruction> )
    [ ',' <chained> ]
```

5.2.3 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 where for each key, the value is the scraping directive or another dictionary. 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 several advantages over wrappers in other languages.

1. *Data centrism.* The central part of the Serrano wrapper are the data, and a first glance at a scraping query reveals what data are being extracted and what are the instructions that acquire them. 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. *Fault Tolerance.* 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.

5.2.4 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 more often that not, scraping cannot start immediately, after the page loads. When scraping from dynamic web pages, we might be waiting for certain AJAX content to load, some elements to be clicked, etc. These wait are referred to as *explicit waits*.

Some languages, such as [56] to 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[9] but also provide a command to wait for a given period of time[10].

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. 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 badly readable 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.

Specifically, a scraping unit contains three optional and one required property. The properties are the following.

Result is the mandatory property. It is either a scraping query or a scraping directive.

WaitActionsLoop is an array of `waits` (elements to be waited for until they appear) and *actions* to be executed. The loop is blocking, i.e. the execution stops until the element is accessible. If the element is still missing after the defined period, the execution of the scraping unit is terminated. It is mutually exclusive with the *wait* and `actions` properties.

Wait is an object that constists of a string jQuery selector and optional waiting time (the default is 2 seconds).

Actions is the array of scraping actions that are executed in specified order.

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

[10]`http://wiki.imacros.net/WAIT`

Temp is an optional dictionary of temporary variables. The `<key>` of this dictionary is a string and a `<value>` is a scraping directive. Each key-value pair is internally translated into the following scraping directive.

```
["!setVal", <value>, <key> ]
```

An example of a scraping unit follows. This unit first waits for the element of class `competition_results` to appear, then it clicks on the element of ID `more_results` and finally it scrapes the competition results and returns them in a Promise.

```
{
  wait: {
    name: ".competition_results"
  },

  actions: [
    [["$#more_results"], [">!call", "click"]]
  ],

  result: ["$.competition_results"]
}
```

In the following, we provide a formal grammar description of a scraping unit.

```
scrap-unit = {

waitActionsLoop: [ ( <wait> | <actions> ), ... ],
  wait: {
    // required
    name: <string: jQuery selector>,

    // optional, default: 2000, 0 means forever
    millis: <int: milliseconds to wait>
  },

  actions: [
    <instr>,
    <instr>,
    ...
  ],

  temp: {
    <name:string>: <scrap-directive>
    <name:string>: <scrap-directive>,
    ...
  },

  result: <result-type>
}

result-type = <scrap-directive> || {
  <name:string>: <result-type>,
  <name:string>: <result-type>,
  ...
}
```

5.2.5 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 languages have this functionality and users have to manage the wrappers and actions manually. Thus Serrano also has a 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 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 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. A formal definition of the grammar follows.

```
page-rules = {
  scraping: <unit-or-object>,
  actions: <action-or-object>,
};

unit-or-object = <scrap-unit> || {
    <scrap_name:string>: <scrap-unit>,
    <scrap_name:string>: <scrap-unit>,
    ...
}

action-or-object = <scrap-directive> || {
    <act_name:string>: <scrap-directive>,
    <act_name:string>: <scrap-directive>,
    ...
}
```

5.2.6 Document Item and Global Document

Each page rules object needs to be associated with the respective URL or 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 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.

Priority is given to every document. 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` signifies higher level of specification than `google.com` and thus 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 called the *default domain item* and matches all URLs.

The formal definition of the grammar follows.

```
document-item = {
  domain: <string>,
  regexp: <string>,
  path: <string>,
  rules: <page-rules>
}
```

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 always chooses the matching page rules.

The formal definition follows.

```
global-document = [
  <document-item>,
  <document-item>,
  ...
]
```

5.3 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 one responsibility and find it 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 – 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. It is not supposed to be an exhaustive description of the command set and for the full details we would like to turn the reader's attention to the Language Specification[11].

[11] `http://github.com/salsita/Serrano/wiki/Language-Spec`

5.3.1 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. 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. The following example returns "Yes", because 5 is greater that 3.

  ```
  ["!if", ["!gt", 5, 3], ["!constant", "Yes"],
    ["!constant", "No"]]
  ⇒ "Yes"
  ```

- *Existence tests.* Commands, such as `!exists, !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. Command `!exists` returns true iff the equivalent Javascript code `(arg !== undefined && arg !== null)` evaluates to true and `!empty` returns true iff the `("object" === typeof(arg)) && (("length" in arg)?(0 === arg.length) : (0 === Object.keys(arg).length))` test returns true. Below, the example command checks, if there is some element with the `myClass` class.

  ```
  [ "!nempty", [ "$.myClass" ] ]
  // ... the same, using the implicit argument
  [ [ "$.myClass" ], [ ">!nempty" ] ]
  ```

- *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. These commands help us group several single predicates into compound predicates. The example below returns true because the condition `((0 < 1) && (3 > 2))` evaluates to true.

  ```
  [ "!and", [ "!lt", 0, 1 ], [ "!gt", 3, 2 ] ]
  ```

- *Result filtering* is a means for reducing array of results to only items that pass a filtering criterion. For this purpose we define the `!filter` command that takes an argument in form of an array and on each item of the array it evaluates the *partial condition* that comes as the second argument to `!filter` command. By partial condition we mean the that the condition that is argument of the `!filter` command should use argument chaining, i.e. should be evaluated on each tested item of the filtered array. In the following example, every item of the selector is chained with the `!ge` condition and the command returns only items where it `age` is set to 18 or more.

  ```
  [ [ "~li .age" ], [ ">!filter", [ ">!ge", 18 ] ] ]
  ```

While the above example would work, more frequent use-case of filtering is that you need to filter some items of array-like object (chosen with jQuery selector), based on some properties of the items in the array. For this, we can leverage the fact that the items passed to `!filter` are jQuery object, i.e. we can access the property we need using selector chaining. In the following example, each time a person is iteratively chained with the partial condition that further selects the age and compares it with 18. Thus, only `persons` that have an age greater or equal to 18 will pass.

```
[ "!filter", [ "$.person" ],
  [ ">=.age", ">!ge", 18 ] ]
```

5.3.2 Storage

Sometimes, the extracted value needs to be reused in multiple succeeding commands. To do so, we can use temporary storage.

There are two commands for working with temporary variables `!setVal` and `!getVal`. For these commands the implicit forEach looping rule is not applied. The `!setVal` command takes two arguments, the key under which the object should be stored and the object itself. The `!getVal` command accepts one argument that denotes the key under which the value is stored in the storage.

After running the scraping query below, two items are saved to storage, `category` and `mainHeader`. Inside the scraping directive a `!setVal` command saves the intermediate result and returns it for further scraping.

```
{
  category: [["~h1"], [">!first"],
    [">!setVal", "mainHeader"],
    [">!split", ":"], [">!first"]]
}
```

5.3.3 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 arguments are numbers (Javascript, and consequently Serrano do not differentiate between integers and floats, they only have a type `number`), the result is a number, or a NaN is special cases, such as division by zero. The example below demonstrates operations on two numbers.

```
["!*", 256, 2]   ⇒ 512
["!/", 94.84, 2] ⇒ 47.42
```

If one of the operands is a number and second one is an array of numbers, the command iterates through the items of the array invokes the operation in a `(element of array, other operand)` pair. The following example saves two testing arrays in the `data` and `data2` variables into the temporary storage. Then, it adds a number 5 to an array and also does it in flipped argument order.

```
["!setVal", "data", [1,2,3,4,5]]
["!setVal", "data2", [10,20,30,40,50]]

["!+", 5, ["!getVal", "data"]] ⇒ [6,7,8,9,10]
["!+", ["!getVal", "data"], 5] ⇒ [6,7,8,9,10]
```

If both operands are arrays of the same length, operation is executed on them "per partes". Otherwise, `NaN` is returned. The following examples (assuming the `data` and `data2` contain the same values as in the previous example) subtract two arrays. In the first example, both arrays have the same length, therefore the operation is successfull. In the second example however, the lengths differ and `NaN` is returned.

```
["!-", ["!getVal", "data2"], ["!getVal", "data"]]
⇒ [9, 18, 27, 36, 45]
["!-", ["!getVal", "data2"], ["!constant", [1, 2, 3]]]
⇒ NaN
```

5.3.4 Array Manipulation

Selectors are often transformed into arrays. Array manipulation basics are covered by four commands, `!first`, `!last`, `!at` and `!len`. Intuitively, these commands enable us to access first, last and n-th command, respectively. The `!len` command returns the length of an array. The following example demonstrates this.

```
[["!constant", [0, 1, 2, 3, 4]], [">!first"]]
⇒ 0

[["!last", ["!constant", [0, 1, 2, 3, 4]] ]
⇒ 4

[["!constant", [-1, 0, 1, 2, 3, 4]], [">!at", 4]]
⇒ 3

[["!constant", [0, 1, 2, 3, 4]], [">!len"]]
⇒ 5
```

5.3.5 Value manipulation

Command `!constant` takes one argument and the result of the command is the value of the passed argument uninterpreted. Since we are using type-checks for deciding how to interpret arguments, this command is useful especially for passing arrays as values, without the engine interpreting them.

```
[["!constant", [0, 1, 2]], [">!len"]]
⇒ 3
```

Sometimes, we need to group the results of several scraping directives into an array. For this we have introduced the `!interpretArray` command, that interprets each of its argument. It is particularly useful in combination with the `!apply` command, that is explained in Section 5.3.6.

5.3.6 Prop, Call and Apply

Remark. These commands are relatively unsafe. However, in some cases they might be useful, so we advise to carefully consider the tradeoffs of adding them to the command set.

They serve for invoking arbitrary methods and accessing properties of objects. Each command takes one optional argument that determines if it is executed on an object itself, or if it loops through sub-objects and executes it on every individual element. In other words, if the last argument has the value `"inner"`, forEach looping is enabled, otherwise is it disabled.

The example below demonstrates the use of the `!prop` command. First instruction invokes the `length` property of the string `"Hello"`, which is equivalent to calling of `"Hello".length` in Javascript. The second instruction uses the loops through the array of elements of class `name` and returns the array of the respective lengths of names. Other commands work analogically. The third instruction is without the optional argument, therefore is returns the number of items the array holds.

```
[["!constant", "Hello"], [">!prop", "length"]]
⇒ 5

[[ "~.name" ], [ ">!prop", "length", "inner"] ]

[[ "~.name" ], [ ">!prop", "length"] ]
```

The `!call` command works very similarly, except instead of invoking a property, it invokes the method, and `!apply` works similarly to `!call` except the second argument provides provides an array of arguments for the method. In the following example, `!apply` function invokes the Javascipt `!replace` function with two arguments, that are regular expressions.

```
[ ["!constant", "aba"], [">!apply", "replace",
    ["!interpretArray", [ ["!regexp", "a","g"],
      ["!constant", "b"] ]]
]]
⇒ "bbb"
```

5.3.7 Text Manipulation

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

```
[["!constant", "Hello"], [">!lower"]]
⇒ "hello"
```

```
["!upper", "Hello"]
⇒ "HELLO"

["!split", "24.7.2016", "."],
⇒ ["24","7","2016"]

["!trim", "     Good job   "]
⇒ "Good job"

["!replace", "Good job", "Good", "Outstanding"]]
⇒ "Outstanding job"

["!join", ["A","P","W","B", "Dumbledore"], ". "]
⇒ "A. P. W. B. Dumbledore"

["!concat", [1, 2, 3], [2, 3, 4], [5]]
⇒ [1, 2, 3, 2, 3, 4, 5]

["!splice", [10,20,30,40,50], 1, 3]
⇒ [10, 50]

["!substr", "serrano", 4]
⇒ "ano"
```

5.3.8 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 the first one has to be the selector, followed by the string `"before"` or `"after"`, to denote where is insertion is to be done, and final argument is the text to be inserted.

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

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 the object with template values is supplied when 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 frmo the DOM. The following scraping directive (also a scraping action, in this case) removes the first paragraph from the web page.

```
[ '!remove', [ '$p:first' ] ]
```

Finally, the `!replaceWith` is used for replacing selected elements with new content. It takes two arguments, the selector and the HTML definition of a new content.

```
[ '!replaceWith', [ '$h1:first' ],
  '<h1>My new heading</h1>' ]

[ '!replaceWith', [ '$p:last' ],
  '<p>Bye, {{name}}. See you soon.</p>' ]
```

5.4 Classification of Our Wrapper

In this section, we pidgeonhole Serrano based on the classifications in the previous chapter. We presented 3 categorizations, Laender's Taxonomy, Technique taxonomy and Applications.

5.4.1 Laender's Taxonomy

According to this categorization, Serrano can be considered a Language for Wrapper Development. We acknowledge that in-browser extraction has a need for a lightweight solution, which can be separated into the library. However, in conjunction with built-in Developer Tools in a browser, Serrano can be also regarded as a very simple HTML-aware toolkit.

5.4.2 Technique Usage

Serrano command set includes commands `!replace` and `!match` for content element parsing, which makes it supportive of the regular expression-based approach. Selectors are directly translated into jQuery selectors, which are a superset of CSS selectors, which makes it support tree-based approach as well as spatial reasoning, to a certain extent.

5.4.3 Applications

The language can be used effectively as a Web Application Testing tool. In addition, we have recognized a novel application not mentioned in the original work – Web browsing experience augmentation. This is, to a large extent, dealt with in form of browser extensions.

6. Implementation

6.1 Source Code

Serrano as well as the source codes of the playground projects can be found on Github[1] and are written in ES5[2] Javascript.

All parts of code have been extensively tested and the unit test files are named `xxx.spec.js`, where `xxx` represents the name of the module.

We used three third-party libraries. JQuery was useful for providing augmented cross-platform CSS selectors implementation, Lodash served as a useful general-purpose library and Q was used because at the time of writing Promises [57] were not a part of the contemporary Javascript specification. Nevertheless, Q Promises are compatible with the modern Promise specification.

The production version of Serrano benefits from a logging service integration[3], where all the runtime warnings and errors are uploaded.

6.2 Deployment and Playground

For testing, we created two examples in form of browser extensions.

Developer's extension is in the simple-extension[4] directory and enables user to write scraping units that are being executed on the current web page.

Client-server extension is located in demo-extension-client[5] directory and is paired with a server side[6]. Testing server was written in node.js by Roman Kašpar. Here, the client extension periodically polls the server for a new global document and extracts data from the current page accordingly.

6.3 Serrano API

Users of Serrano may only want to use limited scope of functionality. For this reason we present several ways to take advantage of this library. In all cases we suppose that Serrano has been included into the source code; that is, somewhere in the file we expect the following code `var serrano = require("../path/to/serrano.js");` . Serrano may find various uses; the API allows to scrape only the scraping directive, setting the context for templates and storage (via `serrano.utils.interpret`), run the scraping unit or (un)load the whole global document. The detailed API can be found in the main javascript file[7].

[1]https://github.com/salsita/Serrano/tree/master/serrano-library
[2]https://people.mozilla.org/~jorendorff/es5.html
[3]https://www.loggly.com
[4]https://github.com/salsita/Serrano/tree/master/simple-extension
[5]https://github.com/salsita/Serrano/tree/master/demo-extension-client
[6]https://github.com/salsita/Serrano/tree/master/serrano-server
[7]https://github.com/salsita/Serrano/blob/master/serrano-library/code/js/main.js

7. Serrano User Stories

Serrano has proven its stability and applicability in a number of projects. Below, we provide a short description of the three most important projects along with the benefits Serrano has provided.

7.1 Magneto Calendar

Magneto[1] 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. Additional features[2] include shared schedules, dynamic travel times to the event (estimate based on a prediction of traffic for a given time and day) and security (by encryption).

If the user visits a website that contains information suitable for a calendar event and clicks on the Magneto button (see Figure 7.1), 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. Originally developed in Javascript, they attempt to extract information about the event. More precisely, one or multiple of the following fields: `what`, `where`, `when`, `who` and `notes`.

There were two main reasons for rewriting the rules in Serrano.

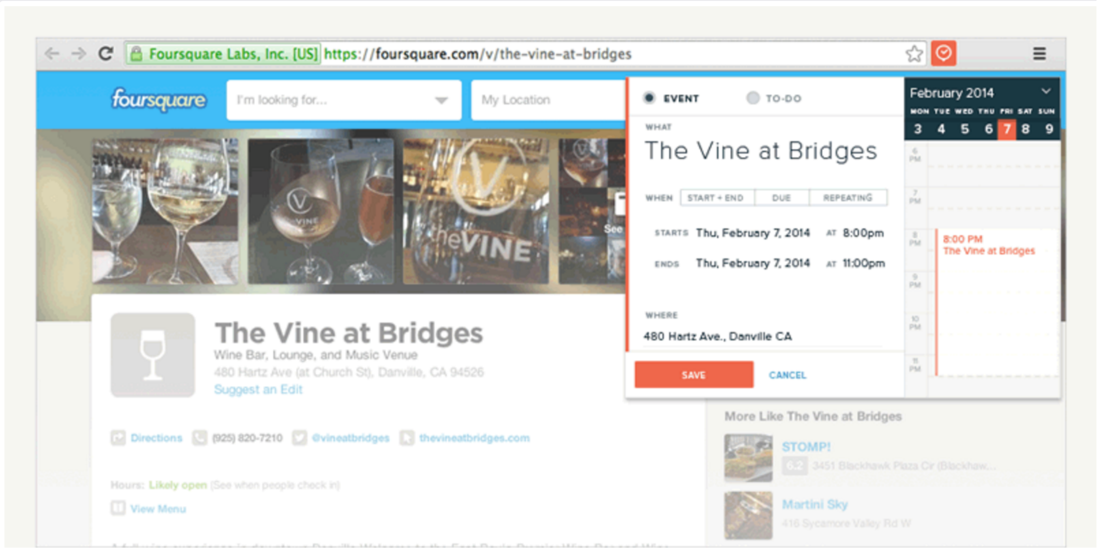
1. *Maintainability.* As the project expanded, the number of web sites and their respective wrappers became harder to maintain. Had the wrappers been outsourced to a third-party developer, number of issues would be solved. Due to the *separation of responsibilities*, the wrapper developer would not need knowledge of the rest of the extension architecture, which reduces the time of getting acquainted with the project and avoids the disclosure of the source codes.

2. *Maintenance.* Updating the whole extension every time a single wrapper is updated is stultifying to the user and bandwidth-consuming. Having the extension update the database of wrappers via AJAX would overcome this issue.

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.

Use 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.

[1]`https://magneto.me/welcome/about-us.html`
[2]`https://magneto.me/welcome/features.html`


https://foursquare.com/v/the-vine-at-bridges
foursquare
I'm looking for...
My Location
The Vine at Bridges
Wine Bar, Lounge, and Music Venue
480 Hartz Ave (at Church St), Danville, CA 94526
Suggest an Edit
Directions
(925) 820-7210
@vineatbridges
thevineatbridges.com
Hours: Likely open (See when people check in)
View Menu
More Like The Vine at Bridges
STOMP!
Martini Sky
February 2014
MON TUE WED THU FRI SAT SUN
3 4 5 6 7 8 9
8:00 PM
The Vine at Bridges

EVENT TO-DO
WHAT The Vine at Bridges
WHEN START + END DUE REPEATING
STARTS Thu, February 7, 2014 AT 8:00pm
ENDS Thu, February 7, 2014 AT 11:00pm
WHERE 480 Hartz Ave., Danville CA
SAVE CANCEL

Figure 7.1: Magneto interface, when user clicked on the Magneto browser action button on a page of a Vine bar.

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 are, for example, `!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 superflous 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.

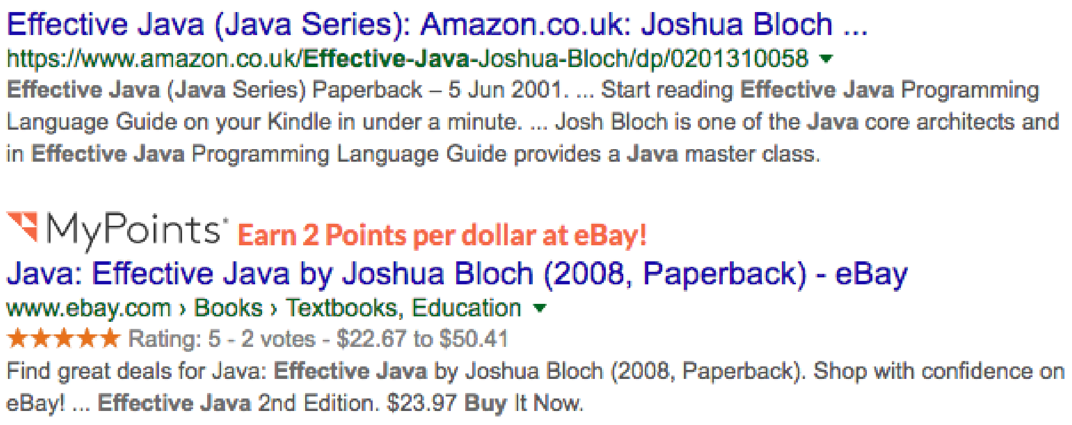
Another challenge for Serrano was understanding of the date of an event on Facebook. Facebook[3], for user convenience, describes date in various formats depending on when it is going to occur. Valid descriptions of a date include: *May 24, next Thursday, tomorrow*, etc. Our workaround involved creating 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 [58] with the event, so this command was removed.

After the replacement[4] of Javascript wrappers with Serrano scraping units, both maintainability and maintenance were increased.

7.2 MyPoints

MyPoints[5] 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,

[3]`http://www.facebook.com`
[4]`https://github.com/salsita/Serrano/tree/master/magneto/scraping-units`
[5]`http://mypoints.com/`


Effective Java (Java Series): Amazon.co.uk: Joshua Bloch ...
https://www.amazon.co.uk/Effective-Java-Joshua-Bloch/dp/0201310058
Effective Java (Java Series) Paperback – 5 Jun 2001. ... Start reading Effective Java Programming Language Guide on your Kindle in under a minute. ... Josh Bloch is one of the Java core architects and in Effective Java Programming Language Guide provides a Java master class.

MyPoints Earn 2 Points per dollar at eBay!
Java: Effective Java by Joshua Bloch (2008, Paperback) - eBay
www.ebay.com › Books › Textbooks, Education
★★★★★ Rating: 5 - 2 votes - $22.67 to $50.41
Find great deals for Java: Effective Java by Joshua Bloch (2008, Paperback). Shop with confidence on eBay! ... Effective Java 2nd Edition. $23.97 Buy It Now.

Figure 7.2: Excerpt of the search results for "Effective Java" augmented by MyPoints extension.

which can be then transformed into discount coupons and subsequently redeemed.

Serrano was used in beta version of the extension, but eventually, the extension was rewritten from scratch by a different vendor and Serrano rules were replaced. One of the possible reasons for discontinuing the use of Serrano have been the fact that at the time of deployment, Serrano was proprietary. Below, we describe the extraction-related competencies of the extension.

On websites with search results of a search engine, MyPoints extension injects text informing the potential shopper about the amount of points they can earn, as shown in Figure 7.2. 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, commands for DOM manipulation[6] were added. These commands include: `!insert`, `!remove` and `!replaceWith`.

7.3 Video Downloader (undisclosed)

In this case, an extension was built into a modified version of Opera browser. That is, the extension cannot be manipulated or separated from the browser. The browser vendor wishes to remain undisclosed, so we present a description without providing a public reference.

The purpose of Video Downloader (VD) is to facilitate download of a currently played video and enable 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

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

the command set with the `!if`. Many wrappers, such as iMacros, would not be able to alter the control flow based on a condition[7]. Typical extraction directive looks as follows.

```
{
  playerElement: [ "!if", [ [ "$#player"],
      [ ">!apply", "hasClass", [ "off-screen" ] ] ],
    [ "!constant", {selector: "#player-api"} ]
  ]
}
```

Second challenge was caused by the fact that some players use different forms of video embeddings. For example, Youtube[8] 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, as follows.

```
{
  playerUrl: [
    ["$object, embed"],
    [">!attr", "href"],
    [ ">!match", [ "!regexp", "[?&]v=([^&]+)[&]?" ] ],
    [ ">!at", 1]
  ]
}
```

[7]`http://wiki.imacros.net/FAQ#Q:_Are_there_conditional_statements_like_if..._then...else_in_the_iMacros_macro_language.3F`

[8]`http://www.youtube.com`

8. Conclusion

The goal of this thesis was to create a web data extraction tool that could work in a restricted environment (browser extension). We implemented a novel language – Serrano. Serrano championed extensibility of the command set and separation of concerns. That helped to eliminate the need for any accompanying software further transformation and processing of the extracted data. Extensibility also works the other way – the command set can be reasonably restricted so that the wrappers will be able to only 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 – which demonstrates that the language is extensible. Serrano has demonstrated good value by separating responsibilities and encouraging modular design. On top of that, Serrano introduces minimum performance overhead, due to the direct translation to Javascript.

Serrano wrappers are in JSON [52] format, which makes them easy to transfer and manipulate, for example via AJAX. The wrapper interpreter is written in Javascript, so that the interpreter can run in a browser.

Limitations. As such, Serrano performs well when the emphasis is placed on information processing. However, when more complex relationships between nodes are to be extracted, it falls short because the expressive power is limited by the expressive power of CSS selectors. A feasible solution is the addition of `!xpath` command with support of XPath. Highly dynamic web pages with run-time generated element IDs and classes also present a challenge, because they cannot be seized by a fixed selector. This is where numerous alternative approaches fail as well. Extending the language with recognition of spatial relationships could be a possible solution.

8.1 Future work

We hope that this language will become a standard tool for browser extension developers and will replace pure Javascript for data extraction. Or at least, by demonstrating the importance of extensibility and data processing within the wrapper, it will serve as a valuable learning lesson for new tools to come. In the rest, we describe some steps that can be taken to further improve the language.

Wrapping a library into a Toolkit Serrano is in the state of active development and the costs of visual tool creation outweigh the benefits due to current volatility of the command set. However, when the language becomes more stable, the creation of a Toolkit, preferably with GUI, might prove feasible and beneficial.

Wrapper databases One of the future directions can be building vast databases of continuously maintained wrappers. Outsourcing wrapper creation and then dynamically downloading and updating them can prevent the need for frequent software upgrades.

Command set databases Another direction would be to build a database of command sets - so that the users of Serrano could find appropriate commands to personalize the language functionality.

Bibliography

[1] Emilio Ferrara, Pasquale De Meo, Giacomo Fiumara, and Robert Baumgartner. Web data extraction, applications and techniques: A survey. *Knowledge-based systems*, 70:301–323, 2014.

[2] Asynchronous JavaScript. XML (AJAX). *Mozilla Developer Center: https://developer. mozilla. org/en/ajax. Access date*, 14(10), 2011.

[3] Graham Cormode and Balachander Krishnamurthy. Key differences between web 1.0 and web 2.0. *First Monday*, 13(6), 2008.

[4] Steve Faulkner Travis Leithead Erika Doyle Navara Edward O'Connor Silvia Pfeiffer Ian Hickson, Robin Berjon. A vocabulary and associated APIs for HTML and XHTML. W3c recommendation, W3C, 2016. https://www.w3.org/TR/html5/.

[5] Y Shafranovich. Rfc 4180: Common format and mime type for comma-separated values (csv) files, 2005. *Cited on*, page 67.

[6] RSS Advisory Board. RSS 2.0 Specification (2009). *URL http://www. rss-board. org/rss-specification. Accessed*, pages 02–21, 2014.

[7] J Bosak, T Bray, D Connolly, E Maler, G Nicol, CM Sperberg-McQueen, L Wood, and J Clark. W3C XML specification DTD, 1998.

[8] Gavin Nicol, Lauren Wood, Mike Champion, and Steve Byrne. Document object model (DOM) level 3 core specification. *W3C Working Draft*, 13:1–146, 2001.

[9] XML Path Language (XPath) 3.1. Technical report, W3C, 2016. https://www.w3.org/TR/xpath-31/.

[10] Michael Kay et al. Xsl transformations (xslt) version 2.0. *W3c recommendation*, 23:52–71, 2007.

[11] Steve DeRose, Eve Maler, and Ron Daniel. XML pointer language (XPointer). 2000.

[12] Maarten Marx. Conditional XPath, the first order complete XPath dialect. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 13–22. ACM, 2004.

[13] Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 data model. *W3C working draft*, 15, 2002.

[14] Selectors Level 3. Technical report, W3C, 2016. https://www.w3.org/TR/CSS/.

[15] Selectors Level 3. Technical report, W3C, 2016. https://www.w3.org/TR/selectors/.

[16] JQuery, 2016. http://jquery.com/.

[17] 2016. https://api.jquery.com/category/selectors/.

[18] ECMAScript 2015 Language Specification. ECMA-262 6th edition. Technical report, ECMA. http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf.

[19] Alberto HF Laender, Berthier A Ribeiro-Neto, Altigran S da Silva, and Juliana S Teixeira. A brief survey of web data extraction tools. *ACM Sigmod Record*, 31(2):84–93, 2002.

[20] James F Allen. Natural language processing. 2003.

[21] Fotios Kokkoras, Konstantinos Ntonas, and Nick Bassiliades. DEiXTo: a web data extraction suite. In *Proceedings of the 6th Balkan Conference in Informatics*, pages 9–12. ACM, 2013.

[22] Nicholas Kushmerick. Finite-state approaches to web information extraction. In *Information Extraction in the Web Era*, pages 77–91. Springer, 2003.

[23] Emilio Ferrara and Robert Baumgartner. Intelligent self-repairable web wrappers. In *AI* IA 2011: Artificial Intelligence Around Man and Beyond*, pages 274–285. Springer, 2011.

[24] Robert Baumgartner, Wolfgang Gatterbauer, and Georg Gottlob. Web data extraction system. In *Encyclopedia of Database Systems*, pages 3465–3471. Springer, 2009.

[25] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[26] Arnaud Sahuguet and Fabien Azavant. Building intelligent web applications using lightweight wrappers. *Data & Knowledge Engineering*, 36(3):283–316, 2001.

[27] Ling Liu, Calton Pu, and Wei Han. XWRAP: An XML-enabled wrapper construction system for web information sources. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 611–621. IEEE, 2000.

[28] Valter Crescenzi, Giansalvatore Mecca, Paolo Merialdo, et al. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, volume 1, pages 109–118, 2001.

[29] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with Lixto. In *VLDB*, volume 1, pages 119–128, 2001.

[30] Mary Elaine Califf and Raymond J Mooney. Bottom-up relational learning of pattern matching rules for information extraction. *The Journal of Machine Learning Research*, 4:177–210, 2003.

[31] Stephen Soderland. Learning information extraction rules for semi-structured and free text. *Machine learning*, 34(1-3):233–272, 1999.

[32] Dayne Freitag. Machine learning for information extraction in informal domains. *Machine learning*, 39(2-3):169–202, 2000.

[33] Nicholas Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1):15–68, 2000.

[34] Chun-Nan Hsu and Ming-Tzung Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Information systems*, 23(8):521–538, 1998.

[35] Ion Muslea, Steven Minton, and Craig A Knoblock. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4(1-2):93–114, 2001.

[36] Brad Adelberg. NoDoSE—a tool for semi-automatically extracting structured and semistructured data from text documents. In *ACM Sigmod Record*, volume 27, pages 283–294. ACM, 1998.

[37] Alberto HF Laender, Berthier Ribeiro-Neto, and Altigran S da Silva. DEByE–data extraction by example. *Data & Knowledge Engineering*, 40(2):121–154, 2002.

[38] David W Embley, Douglas M Campbell, Yuan S Jiang, Stephen W Liddle, Deryle W Lonsdale, Y-K Ng, and Randy D Smith. Conceptual-model-based data extraction from multiple-record Web pages. *Data & Knowledge Engineering*, 31(3):227–251, 1999.

[39] Tim Furche, Georg Gottlob, Giovanni Grasso, Omer Gunes, Xiaoanan Guo, Andrey Kravchenko, Giorgio Orsi, Christian Schallhart, Andrew Sellers, and Cheng Wang. DIADEM: domain-centric, intelligent, automated data extraction methodology. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 267–270. ACM, 2012.

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

[41] Giovanni Grasso, Tim Furche, and Christian Schallhart. Effective web scraping with OXPath. In *Proceedings of the 22nd international conference on World Wide Web companion*, pages 23–26. International World Wide Web Conferences Steering Committee, 2013.

[42] John W Lloyd. Practical Advtanages of Declarative Programming. In *GULP-PRODE (1)*, pages 18–30, 1994.

[43] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. The elog web extraction language. In *Logic for programming, artificial intelligence, and reasoning*, pages 548–560. Springer, 2001.

[44] Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. *Journal of the ACM (JACM)*, 51(1):74–113, 2004.

[45] Georg Gottlob, Christoph Koch, Robert Baumgartner, Marcus Herzog, and Sergio Flesca. 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*, pages 1–12. ACM, 2004.

[46] Bertram Ludäscher, Rainer Himmeröder, Georg Lausen, Wolfgang May, and Christian Schlepphorst. Managing semistructured data with florid: a deductive object-oriented perspective. *Information systems*, 23(8):589–613, 1998.

[47] Gustavo O Arocena and Alberto O Mendelzon. WebOQL: Restructuring documents, databases and Webs. In *Data Engineering, 1998. Proceedings., 14th International Conference on*, pages 24–33. IEEE, 1998.

[48] Ermelinda Oro, Massimo Ruffolo, and Steffen Staab. SXPath: extending XPath towards spatial querying on web documents. *Proceedings of the VLDB Endowment*, 4(2):129–140, 2010.

[49] Cynthia Dwork. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation*, pages 1–19. Springer, 2008.

[50] Robert C Martin. The single responsibility principle. *The Principles, Patterns, and Practices of Agile Software Development*, pages 149–154, 2002.

[51] Steve Burbeck. Applications programming in smalltalk-80 (tm): How to use model-view-controller (MVC). *Smalltalk-80 v2*, 5, 1992.

[52] Douglas Crockford. The application/json media type for javascript object notation (JSON). 2006.

[53] Tiago Boldt Sousa. Dataflow programming concept, languages and applications. In *Doctoral Symposium on Informatics Engineering*, volume 130, 2012.

[54] John McCarthy, Steve Russell, Timothy P Hart, Mike Levin, AutoLISP Arc, and Common Lisp Clojure. LISP Programming Language, 1985.

[55] Edsger W Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[56] Joachim Hammer, Jason McHugh, and Hector Garcia-Molina. Semistructured Data: The TSIMMIS Experience. 1997.

[57] Jake Archibald. Javascript promises. *Artikkeli. Luettavissa*, 2013.

[58] Ian Hickson. Html microdata. *W3C*, 24, 2011.

Appendix A - CD Contents

We provide the source code of all projects as well as the compiled program. To compile the project from the source code one has to run `npm install` in command like in its directory, followed by running `grunt`. The project appears in the `/build` directory.

The contents of the CD are following. The `compiled` directory contains the `serrano` library along with the developer's extension (`serrano-simple-extension`) and the demo client-server project (`demo-extension-client` and `demo-server`). The `sources` directory contains the source codes of all the projects from the `compiled` directory and additionally a `magneto` directory which contains the original Javascript wrapper of the Magneto project as well as the rewritten wrappers in Serrano. All source codes can be found on Github[1].

[1] `http://github.com/salsita/Serrano`