A new methodology for testing HTML5 parsing implementations

Jose Carlos Anaya Bolaños
The University of Manchester

Abstract. The Web has evolved, from plain text and images interlinked, to complex applications. In order to cope with new Web features, HTML parser implementations had to define its own way to parse and fix errors. However, there are disagreements and inconsistencies of outputs among different applications. HTML5 is the latest version of the HTML standard and the specification includes, for the first time, an algorithm for parsing and error handling. That feature aims to finally achieve full consistency and interoperability between independent parsing implementations. The document presents a plan for creating a new methodology for testing HTML5 parsing implementations. The goal of the proposed methodology is to provide tools to analyse, annotate and compare outputs from different HTML5 parsing implementations. A HTML5 parser has been developed as a prototype to build the proposed methodology.

1. Introduction

The World Wide Web Consortium (W3C) is an international organization that defines standards regarding web technologies. The mission of the W3C is to “develop protocols and guidelines that ensure the growth of the Web” [1]. Since its foundation, in October 1994, several standards have been promoted for creating, interpreting, rendering and displaying web pages.

The Hypertext Markup Language (HTML) is the most used language by web pages. It was born between 1989 and 1990 taking as a base the Standard Generalized Mark-up Language (SGML) [2]. The W3C realised of its potential, embraced it and continued to improve it. Several versions have been created since then. In October 2014 the latest version of the HTML standard, HTML5, reached the status of Recommendation (i.e., the stage of highest maturity of a standard)[3].

The forgiveness of web browsers to parse HTML led to some inconsistencies among different applications because each one parsed and fixed errors in its own way. XHTML documents can be easily parsed using an XML parser; nevertheless, those documents are restricted by a strict set of rules. The HTML5 specification includes several changes and improvements with respect to its predecessors. One of those changes is that, for the first time, the parsing process is defined as an algorithm and it includes rules for error handling. This new parsing process is a key feature of HTML5 because it ensures that every input stream of data has a well-defined output. This certainty of the input-output relation is the element that targets toward the full consistency and interoperability of independent parsing implementations.

In order to be compliant with the HTML5 specification, an HTML5 parser may be implemented with any technology, programming language or algorithm as long as it guarantees the same output as the pseudo code. Nevertheless, there are multiple factors that make testing a parsing implementation a difficult activity. Among such factors are: the nature of high error tolerance of HTML and its constant evolution, the complexity of the specification algorithm, the potential infinite different inputs, etc.
In section 2 a background research about HTML history, the HTML5 parsing process, current parsing implementations and testing methodologies is presented. Section 3 describes the project and discusses the methodology for planning and evaluating it. In the following section, the progress of the project, which includes a prototype implementing a transliteration of the parsing algorithm, is presented. Finally, the last section presents results and conclusions.

2. Background research

2.1 HTML history

The Hypertext Markup Language (HTML) was born between 1989 and 1990 as an application of the Standard Generalized Mark-up Language (SGML) [2]. The W3C was born in 1994 with the aim to increase the Web potential through standards and rapidly adopted HTML. In 1995 HTML was extended with new tags and a draft called HTML 3.0 appeared. In 1997 a stable HTML specification, named HTML 3.2, was approved by Microsoft and Netscape (the major browser vendors from that time). In spring 1998 HTML 4.0 reached the status of W3C recommendation.

HTML documents were validated against a DTD schema. A DTD schema describes the structure of a document, the legal names for elements and attributes, etc. If a document follows a schema rules, it is said to be a valid document. When a document is valid with respect to a DTD, it guarantees that the document can be parsed in a unique Document Object Model (DOM). A DOM is an interface of a data structure, represented as a tree, that allows applications to access and manipulate the structure, content and style of documents. W3C defined an specification for DOM [4].

In 1996 the W3C presented the XML specification (a subset of the SGML). XML was designed to be generic, extensible and simple to use, etc. [5]. The rules of a well formed XML document are:

- There is exactly one root element.
- Tags are correct (i.e. between “<” and “>” characters).
- Tags are properly nested.
- Attributes are unique for each tag and attribute values are quoted.
- No comments inside tags.
- Special characters are escaped.

When a non-well-formed XML document is parsed, it might produce a fatal error (known as Draconian error), and consequently the document cannot be parsed into a DOM tree by an XML parser.

With the arrival of XML, XML Schema appeared as an alternative to DTD schemas. Unlike DTD schema, XML Schema included new features such as data types, element enumerations, etc. Moreover, an XML Schema follows the XML syntax.

“The W3C believed the Web itself would eventually move to XML” [6] and, in January 2000, the XHTML 1.0 spec was adopted as a W3C Recommendation. The version 1.1 was a recommendation by May 2001. XHTML is defined as an XML application (i.e. a restricted subset of XML). The XHTML spec included three schemas (Strict, Transitional and Frameset) in order to validate a document and guarantee the uniqueness of a DOM tree.
With the schema validation and the rules for well-formedness, XHTML was against the permissive and forgiving approach of HTML. A web HTML page is expected to be rendered despite a missing closing tag or an unquoted attribute value instead of failing or displaying an error as XHTML was proposing.

Nevertheless, the W3C began to work on XHTML 2.0. Some of the W3C members were representatives of major browser vendors such as Mozilla, Apple, Google, Opera, etc. According to them, web pages were turning into something more “than text and images interconnected by links” [6]; they were becoming web applications containing dynamic content and multimedia. The first draft of the HTML5 spec (born as a proposal from Mozilla and Opera, called Web Forms 2.0) was presented in 2004 to the W3C. The draft was voted and it was rejected (8 in favour vs. 11 against). Despite the rejection, some members agreed to continue working on the project and formed the Web Hypertext Application Technology Working Group (WHATWG).

W3C continued to work in XHTML 2.0. However, in 2007 they realised that the spec proposed by the WHATWG had indeed a promising future and they asked them to work together. The drafts related to HTML were merged and renamed as HTML5. The first official draft of HTML5 appeared in January 2008. Currently the W3C and the WHATWG specifications are slightly different. The divergence began in 2012, when the W3C introduced a group of editors to organize the draft and decide what should be included in the HTML5 spec and what should be put into another specs. In the W3C recommendation they claim that, “The W3C HTML working group actively pursues convergence of the HTML specification with the WHATWG living standard” [3].

The WHATWG spec is a “living standard” named the HTML Standard [7]. Ian Hickson had been (and continues to be) the unique editor of this spec [6]. That decision was taken because web browsers are constantly experimenting with new behaviours and features. According to Hickson, “The reality is that the browser vendors have the ultimate veto on everything in the spec, since if they don’t implement it, the spec is nothing but a work of fiction” [8].

In fact, the major web browsers (Opera, Google Chrome, Apple Safari, Mozilla Firefox and Microsoft Internet Explorer) are conformant with the WHATWG HTML Standard and not the W3C HTML5 Recommendation. David Baron, a distinguished engineer from Mozilla said “When the W3C’s and WHATWG’s HTML specifications differ, we tend to follow the WHATWG one” [9].

Previous versions of HTML and XHTML did not include a parsing guide or error-handling. Each web browser vendor defined its own way to parse and fix HTML. Although “error handling is quite consistent in browsers”[10] there were inconsistencies amongst them. In order to finally end with inconsistencies, the HTML5 spec includes a parsing algorithm and error handling. Moreover, HTML5 is not an XML document, thus, is not subject to the rules for being a well formed document. The algorithm uses finite state machines and ensures that every1 input stream of data has a well-defined output.

2.2 The HTML5 parsing process
Appendix A – Overview of the HTML5 parsing process presents a flow chart of the HTML5 parsing algorithm architecture. The diagram represents a simplified overview of the parsing algorithm. The process is not as trivial and straightforward as the flow chart may suggest.

1 There are some unsupported character encodings in the spec, thus, that data cannot be parsed.
The data input is a stream of octets. The flow of the parsing process begins with the identification of the encoding of the input stream by using the **encoding sniffing algorithm**. Typically the user agent explicitly defines the encoding. When no character encoding is specified, the algorithm analyses the stream in order to try to determine the encoding. The specification discourages the use of some character encodings and suggests the use of the UTF-8 as default character encoding [3].

The next stage is the **pre-processing** of the input stream. This stage manipulates some characters and raises errors when control characters\(^2\) are encountered. After the pre-processing, the **tokenizer** consumes characters from the input data stream and produces tokens. Those tokens are then consumed by the **tree constructor**. The tree constructor creates and manipulates a DOM tree that will be the output of the parsing process.

The tokenizer state machine is composed by 69 different states and the transitions are mostly triggered by the data input. The tree constructor may also change the current state of the tokenizer. The execution of scripts may insert new characters into the input stream. The tree constructor phase is defined by 23 states and the transitions are triggered by the tokens produced by the tokenizer.

**The tokenizer**

There are six different types of tokens: character, comment, DOCTYPE, end of file, end tag and start tag. A cycle through the tokenizer will consume one or more characters and it will end by emitting one or more tokens. Most of the tokenizer states (62 out of 69) will consume and process one character from the input stream. Depending on the character value, it might be ignored, produce or emit a token (or several), cause a state transition and/or be reconsumed. The default state of the tokenizer is the *Data state* (i.e. when a token is emitted, the tokenizer will return to this state). Nevertheless, under some circumstances, the tree construction stage may change the default state. Figure 1 presents a worked example of a cycle for emitting a start tag token with one attribute.

<table>
<thead>
<tr>
<th>Input</th>
<th>Tokenizer steps</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;a href=&quot;www.manchester.ac.uk&quot;&gt;</code></td>
<td>1) <em>Data state</em> consumes a “&lt;” character. Switches to <em>tag open state</em>.</td>
</tr>
<tr>
<td></td>
<td>2) <em>Tag open state</em> consumes an “a” character. Creates a <em>start tag token</em> with value equals to “a”. Switches to <em>tag name state</em>.</td>
</tr>
<tr>
<td></td>
<td>3) <em>Tag name state</em> consumes a space. Switches to <em>before attribute name state</em>.</td>
</tr>
<tr>
<td></td>
<td>4) <em>Before attribute name state</em> consumes an “h” character. Creates an attribute for the token with name equals to “h”. Switches to <em>attribute name state</em>.</td>
</tr>
<tr>
<td></td>
<td>5) <em>Attribute name state</em> consumes an “r” character. Appends the character to the current attribute name. Keeps consuming characters and when the “=” character is consumed, it switches to <em>before attribute value state</em>.</td>
</tr>
<tr>
<td></td>
<td>6) The characters are consumed and appended to the current attribute value. When the “&gt;” character is consumed, the current <em>start tag token</em> is emitted and the tokenizer switches back to the <em>data state</em>.</td>
</tr>
</tbody>
</table>

\[\text{Figure 1 – A cycle through the tokenizer to emit a token}\]

\(^2\) The Unicode control characters have no visual representation and are used to control how text is displayed.
Recalling the previous example, if the input was the same but removing the first character, the transition to tag open state would never happened and a character token would have been emitted for each character.

The other states will attempt to consume several characters to identify character references, comments, a DOCTYPE declaration or CDATA sections. It is an attempt because the characters are consumed only if they truly represent one of the previously mentioned values. For example, a transition to the markup declaration open state is made. This state will attempt to consume characters matching DOCTYPE or [CDATA]. If there is a match, the characters are consumed, and then a transition is made. If there is no match, a transition is made without consuming the characters.

The tree construction

When the tokenizer completes a cycle, one or more tokens were generated and the tree construction machine will process the token(s). The DOM tree is manipulated in this stage. A pointer to the current node is used (initially null). The character tokens will create text nodes; comment tokens will create comment nodes; start tag tokens will produce element nodes; end tag tokens will be used for closing element nodes (i.e. the pointer to the current node is updated to point to the parent node). The machine has 23 states called insertion modes. The first state is the initial insertion mode. Figure 2 presents the cycle through the tree construction to process an end of file token (i.e. an empty string input).

<table>
<thead>
<tr>
<th>Input</th>
<th>“” (empty string)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree construction steps</td>
<td></td>
</tr>
</tbody>
</table>
1) Initial insertion mode switches to before HTML insertion mode and reprocesses the current token.  
2) Before HTML insertion mode creates an html element and appends it to the document object (DOM tree). It pushes the html element into the stack of open elements, switches to before head insertion mode and reprocesses the current token.  
3) Before head insertion mode creates a head element and appends it to the DOM tree. It pushes the head element into the stack of open elements, switches to in head insertion mode and reprocesses the current token.  
4) In head insertion mode pops the head element from the stack of open elements, switches to after head insertion mode and reprocesses the current token.  
5) After head insertion mode creates a body element and appends it to the DOM tree. It pushes the body element into the stack of open elements, switches to in body insertion mode and reprocesses the current token.  
6) After head insertion mode creates a body element and appends it to the DOM tree. It pushes the body element into the stack of open elements, switches to in body insertion mode and reprocesses the current token.  
7) In body performs some validations and finally stops parsing.

**Figure 2 – A cycle through the tree constructor to process an empty string**

The previous example depicts the simplest flow of the tree construction stage. It produces the minimal DOM tree, i.e. a DOM tree that contains only an html element (as root node) and a head and body element (as children elements of the html node).
The tree construction stage is very complex and it uses several data structures (stacks and lists), flags, pointers and persistent status (the current insertion mode). Additionally, it includes some other smaller algorithms that are used across insertion modes.

2.3 HTML5 parsing implementations

Rendering (or layout) engines are the main type of applications that require HTML parsing. Web browsers use those engines not only to parse HTML but CSS as well, execute scripts, render and display content, etc. Usually each vendor of major browsers has its own implementation of layout engines. For example Google Chrome and Opera browsers use Blink, Apple Safari browser uses WebKit, Mozilla Firefox uses Gecko, Microsoft Internet Explorer uses MSHTML (also known as Trident), etc. [10].

The new implementation of Gecko, Gecko 2 [11] implements an HTML5 parser, compliant with the spec. The parsing process is executed in a separate thread from the main UI thread to improve responsiveness from the browser. It features speculative parsing in order to parallelize the HTML parsing and the script execution, improving the performance of the rendering process.

In [12] a new browser engine called Servo is presented. It is written in Rust programming language instead of C++ as the previously commented rendering engines. It aims for taking advantage of parallel hardware and for better performance, power usage and concurrency management than other rendering engines. The authors state that “Servo must be at least as fast as other browsers at similar tasks to succeed, even if it provides additional memory safety”. It is still under development but so far they managed to make Servo faster than Gecko in the layout stage.

Apart from web browsers, there are other applications that use rendering engines such as email managers, Integrated Development Environments (IDEs), e-book readers, VoIP and videoconference applications, etc. For example Microsoft Outlook and Microsoft Visual Studio both of them use Trident, the first for rendering emails and the second one for its web page designer [13].

There are other applications that might require only a standalone HTML parser, i.e. those might not need a complex render engine as they will not execute scripts or render/display the HTML content. Among those applications are HTML debuggers, validators, reporters, web crawlers, text-mining tools, sanitizers, pretty-printers, etc.

There are several standalone HTML5 parsing implementations online, each offering different features and capabilities. Github claims to be the world’s largest code host. A search for HTML5 parser displays more than 130 repositories in more than 10 different programming languages. According to the search result in Github, the top language used is JavaScript, followed by C and then PHP.

Backed by Google and “tested on over 2.5 billion pages from Google's index”, gumbo parser [14] is the most popular and the third most forked HTML5 standalone parser available in Github. It is written in C and it claims to be fully conformant with the WHATWG spec. Moreover, it passes all the test cases from the html5lib test suite [15]. Another well positioned implementation is jsoup [16][17]. It is the most forked and the third most popular HTML5 parser in Github. It is written in java and additionally to HTML5 parsing, it features: XML and CSS parsing, pretty printing and HTML cleaning. It is conformant with the

---

3 Not always is possible to parallelize those tasks. Moreover, to take real advantage of speculative parsing, some suggestions have to be followed.
WHATWG spec. Table 1 presents the ten most popular (number of stars) standalone HTML5 parsers in Github.

In [18] a standalone, parallel HTML5 parser is presented. According to the authors, “HPar is the first pipelining and data-level parallel HTML parser”. Parallelization of the parsing algorithm is hard because there are dependencies between the tree construction and the tokenizer. Under some circumstances, a few insertion modes can modify the next tokenizer state. Additionally there are some elements that can be self-closing (for example the br element); in order to raise errors when a non-self-closing element is self-closed, the tokenizer has to wait for feedback of the tree construction.

<table>
<thead>
<tr>
<th>Name</th>
<th>Stars</th>
<th>Forks (order)</th>
<th>Language</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>google/gumbo-parser</td>
<td>3251</td>
<td>399 (3)</td>
<td>C</td>
<td>WHATWG</td>
</tr>
<tr>
<td>sparklemotion/nokogiri</td>
<td>3240</td>
<td>443 (2)</td>
<td>Ruby</td>
<td>-</td>
</tr>
<tr>
<td>jhy/jsoup</td>
<td>1878</td>
<td>646 (1)</td>
<td>Java</td>
<td>WHATWG</td>
</tr>
<tr>
<td>inikulin/parse5</td>
<td>685</td>
<td>22 (10)</td>
<td>JavaScript</td>
<td>WHATWG</td>
</tr>
<tr>
<td>aredridel/html5</td>
<td>479</td>
<td>73 (5)</td>
<td>JavaScript</td>
<td>-</td>
</tr>
<tr>
<td>html5lib/html5lib-python</td>
<td>329</td>
<td>79 (4)</td>
<td>Python</td>
<td>WHATWG</td>
</tr>
<tr>
<td>masterminds/html5-php</td>
<td>269</td>
<td>39 (6)</td>
<td>PHP</td>
<td>WHATWG</td>
</tr>
<tr>
<td>FlorianRappl/AngleSharp</td>
<td>207</td>
<td>32 (7)</td>
<td>C#</td>
<td>W3C</td>
</tr>
<tr>
<td>servo/html5ever</td>
<td>167</td>
<td>31 (8)</td>
<td>Rust</td>
<td>WHATWG</td>
</tr>
<tr>
<td>tracy-e/OCGumbo</td>
<td>150</td>
<td>26 (9)</td>
<td>Objective-C</td>
<td>WHATWG</td>
</tr>
</tbody>
</table>

Table 1 – Top ten most popular HTML5 parsers in Github

Initially, the HPar parser divides the input into chunks and each chunk is processed in parallel generating tokens and storing them in a buffer. The parsing process is speculative and it is similar to a transaction: a snapshot stores the state of the tokenizer at a given time and a flag for hazard detection is used, when the flag is true (i.e. the tokenizer state was changed by the tree constructor), a rollback has to be made (i.e. discarding some tokens and creating new ones).

To validate their parallel parser, the authors analysed over 1000 websites to find how often the tree construction stage modified the tokenizer state and they found that it was less than 0.01%. That means that the probability of a rollback is less than 0.01%. To test their implementation, they compared it against jsoup (commented previously, Table 1). HPar had a speed improvement up to 2.4 times (1.73 on average) when parsing some websites such as Facebook, YouTube, BBC, etc.

2.4 HTML5 test suites

Several of the standalone parsing implementations mentioned before use the test cases from html5lib [15]. The test suite contains more than 8000 entries detailing the input, expected output, expected number and type of errors, etc. It includes tests for parsing (i.e. tokenizer and tree construction stages), encoding, sanitizing, serializing and validating HTML5. Those test cases are generally trusted as reliable and conformant with the WHATWG spec. The html5lib project was done by four developers; nevertheless, the test suite had contributions (test cases) from several users, including developers of WebKit and Mozilla.

The W3C has its own test suite and defines it as “The Web Platform Tests Project is a W3C-coordinated attempt to build a cross-browser test suite for the Web-platform stack” [19][20]. The project is hosted in Github and it comprises test cases for the complete HTML5 spec (not only parsing but encodings, fonts,
images, media, events, etc.). It is focused on testing browsers rather than standalone parsing implementations. The WHATWG has a test suite as well [21]. It includes test cases from developers and companies: IE, Opera, Mozilla, Ian Hickson, etc. The html5lib test suite is also included.

HTML5TEST[22] is a web application that test browser support of HTML5. It runs a several tests and assigns a score. The tests cover various sections of HTML5 such as multimedia, parsing rules, device access, connectivity, performance, etc. According to the authors, they test “the official HTML5 specification, specifications that are related to HTML5 and some experimental new features that are extensions of HTML5”.

3. Project description
The aim of the project is to develop a test methodology for HTML5 parsing implementations.

The objectives are:

- Develop a spec conformant HTML5 parser.
- Write a comparative review of some parsing implementations.
- Create analytic and annotative tools that help to inspect and compare outputs from different parsers.
- Build test suite with a higher coverage of the spec than current available test suites.

3.1 Project plan
An Agile based methodology is going to be used to continue with the project. The project will be organized into two-week sprints (the sprint 0 is one week) and one month will be used for writing the dissertation. Table 2 presents the proposed project plan. Details of the first two sprints are presented after the table.

<table>
<thead>
<tr>
<th>Sprint</th>
<th>May 11-15</th>
<th>May 16-31</th>
<th>June 1-15</th>
<th>June 16-30</th>
<th>July 1-15</th>
<th>July 16-31</th>
<th>August 1-31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprint 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sprint 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sprint 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sprint 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sprint 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sprint 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dissertation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 – Project plan distributed into sprints.

- **Sprint 0.** The goal of this sprint is to complete the current parser implementation, i.e. pass all the test cases. Section 4.3 presents the status of the current prototype.
- **Sprint 1.** In this sprint it is intended to make a deep review of current parsing implementations and test suites. This review will be the base of the comparative report planned as deliverable (details in the following section). Initially, each member of the team will review one rendering engine and two standalone parsing implementations. Table 3 presents a template of the information required for the report. Moreover, is planned to download, configure and run some parsing implementations. The selected parsers will be included in the analytic tools deliverable.
Additionally, the report will contain information related to testing results (i.e. # of tests, passed, failed, type of errors, etc.). This information will be discussed and agreed with the supervisor during the next sprints.

**Sprints 2 – 5.** The tasks for these sprints will be decided while working in the previous sprints. This is because some tasks depend on obtained results and difficulties presented. Near the end of each sprint, a plan for the next one will be discussed and agreed with the project supervisor.

<table>
<thead>
<tr>
<th>General information</th>
<th>Features</th>
<th>Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Fragment parsing</td>
<td>Number of downloads</td>
</tr>
<tr>
<td>Author(s)</td>
<td>Serializer</td>
<td>Number of commits, forks, contributors, etc.</td>
</tr>
<tr>
<td>Programming language</td>
<td>CSS parsing</td>
<td>Activity (e.g. date of last commit)</td>
</tr>
<tr>
<td>Spec conformant</td>
<td>Script execution</td>
<td></td>
</tr>
<tr>
<td>Latest version</td>
<td>Manipulators</td>
<td></td>
</tr>
<tr>
<td>Type of licence</td>
<td>(e.g. minimizer, maximizer, pretty printer)</td>
<td></td>
</tr>
<tr>
<td>Type of input</td>
<td>Validator</td>
<td></td>
</tr>
<tr>
<td>(E.g. text, files, URL, etc.)</td>
<td>Sanitizer</td>
<td></td>
</tr>
<tr>
<td>Type of output</td>
<td>Other features</td>
<td></td>
</tr>
<tr>
<td>Software requirements</td>
<td>(speculative parsing, crawling, etc.)</td>
<td></td>
</tr>
<tr>
<td>Test suite</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3 – Template of information required for the comparative review.**

### 3.2 Project deliverables and evaluation

The project deliverable will be the test methodology. It will consist on the following items:

- A piece of software that implements a HTML5 parsing spec. The parser must pass all the test cases of the html5lib test suite (only the test cases related to parsing, i.e. tokenizer and tree construction).

- Additionally, the software will include a comparative report of distinct parsing implementations. This report could be qualitative (deepering into a small group of applications) or quantitative (analysing key features of a large set of applications). The report type will be decided depending on the progress of the research (i.e., results obtained in sprints). This activity is by no means a trivial task. There are several factors that have to be taken into account. Some of those factors are: the large amount of parsing implementations and the requirements that those might have, different programming languages, formats and character encodings of inputs/outputs, etc.

- A test suite.
  The aim of the test suite is to have a high coverage of the HTML5 parsing spec. In order to do that, the test suite could be a merge of some existing test suites. Another possible scenario could be an improvement of an existing test suite by adding new test cases.

### 4. Project progress

With the aim to understand and become familiar with the HTML5 parsing algorithm, two spikes and a complete prototype have been developed. Those tasks have been developed as a team (of three members).
4.1 First spike: a minimal DOM tree parser

The objectives of this first spike were:

- Identifying all the possible data inputs that are equivalent to an empty string.
- Analysing the parsing algorithm and understanding the overall process.
- Developing a program to parse the empty string equivalent inputs into the minimal DOM tree.
  (We decided to use Java as all the team members had some experience using it).
- The program should implement a transliteration of the algorithm.
- Generate a simple test suite to evaluate the parsing program.

The minimal DOM tree is the most basic representation of an HTML parsed tree. This minimal DOM tree is produced when an empty string or an equivalent input is parsed following the HTML5 parsing rules. Table 4 presents a list of the empty string equivalent inputs. The inputs are represented as regular expressions.

<table>
<thead>
<tr>
<th>Id</th>
<th>Input string (regular expression)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ε</td>
<td>Empty string</td>
</tr>
<tr>
<td>2</td>
<td>SPACE_CHAR*</td>
<td>Space characters</td>
</tr>
<tr>
<td>3</td>
<td>(&lt;ELEMENT&gt;)*</td>
<td>html, head or body start tags</td>
</tr>
<tr>
<td>4</td>
<td>(&lt;ELEMENT&gt;)*</td>
<td>html, head or body closing tags</td>
</tr>
<tr>
<td>5</td>
<td>(&lt;ELEMENT&gt;(SPACE_CHAR</td>
<td>/)<em>&gt;)</em></td>
</tr>
<tr>
<td>6</td>
<td>&lt;LATIN_LETTER (.)*</td>
<td>Unclosed start tag</td>
</tr>
<tr>
<td>7</td>
<td>(&lt;LATIN_LETTER (.)<em>&gt;)</em></td>
<td>Closing tags</td>
</tr>
</tbody>
</table>

Table 4 – List of inputs equivalent to the empty string

Note well that in a regular expression the character “.” means any character. In expressions 6 and 7 after a Latin letter there could be any number of any characters with the exception of the “>” character. This is because that symbol is used as delimiter of a tag. Some of those inputs might produce parsing errors; nevertheless they produce the minimal DOM.

An org.w3c.dom.Document object was used to store a DOM tree. In order to test the implementation, a set of inputs was generated using fuzz-testing (random valid and invalid inputs) and then the output was serialized into a string and compared against the string representing the minimal DOM tree. JUnit was used as test harness.

The first spike was really useful to understand the overall flow of the parsing process and to realize the real magnitude of the algorithm.

Some time after the spike (when testing the prototype), we realized that the table was incorrect. The input number 7 has exceptions; some closing tags produce start tags. For example the input </br> will produce a br node as child of the body node. An input </p> will produce a minimal DOM tree, but if there is a
body tag before (i.e. \texttt{<body></p>}) the tree will have a \texttt{p} node as child of the \texttt{body} node. This happens because most of the closing tags are discarded in the \textit{after head} insertion mode, but if for some reason the \textit{in body} insertion mode was reached, some closing tags can manipulate the DOM tree.

4.2 Second spike: a minimal DOM tree parser plus title elements

The objectives of the second spike were:

- Update the first spike to parse title elements.
- Update the test suite.

In order to parse title elements, more tokenizer states and insertion modes were required and new data structures were used. The most valuable learning of this spike was to understand that the test strategy was not scalable and that a new test strategy was mandatory.

The minimal DOM validation only needed a string comparison. When adding title elements, the comparison became harder because there could be any number of title elements in two different sections (head and body). To address that issue, the comparison was done using regular expressions and the input generator was updated to include random strings containing title elements. The test strategy was reasonable for the spike but it was not complete because it only validated the DOM tree structure but not the text nodes. An example of this problem of validation is presented in Figure 3.

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Input</th>
<th>Expected output (serialized DOM tree)</th>
<th>Invalid outputs (but matching the regular expression)</th>
</tr>
</thead>
</table>

Figure 3 – Example of serialized DOM trees comparison using regular expressions

4.3 A complete parser prototype

An Agile based methodology was used in order to implement a parser following the W3C HTML5 spec. The plan consisted in three two-week-sprints (i.e. 20 hours per person per week). Each sprint was considered as a milestone. In each sprint a project leader was defined to help tracking the tasks,
estimations and priorities. A backlog was used and maintained frequently. Additionally, Trello\(^4\) notes were used as well. Github was used to host the code [23].

The parser was developed with little consideration of performance, efficiency and modularity. This is because two reasons. Firstly, the parser was thought as a prototype implementing a transliteration of the spec algorithm. This was in order to check the validity and correctness of the algorithm. Secondly, as the plan was restricted in time we did not have much time to analyse completely the spec before starting to code. Some structures, functions or interfaces were done as we thought it was the best at that time. Moreover, because we followed the algorithm (as far as possible), there is non-elegant and repeated code.

The prototype has the following limitations:

- The sniffing algorithm was not implemented and the UTF-8 character encoding was used by default. There were three reasons that led to that decision: first, UTF-8 is the most widely used character encoding for websites (83.7% of websites use it, according to w3techs.com [24]). Second, UTF-8 is the suggested character encoding by the spec. Finally, the sniffing algorithm is a large a complex procedure that does not guarantee a 100% confidence in determining the character encoding. Overall, implementing the sniffing algorithm is not of high importance for understanding the spec.

- The execution of scripts was not implemented. That decision was taken for two reasons: first, the script execution is not part of the parsing process (i.e. it is part of the HTML5 spec but is a different section). The second reason is that a script can insert new data into the tokenizer. That new data could lead to a manipulation of the DOM tree by inserting, removing or modifying elements; furthermore, it could produce a change of character encoding (by inserting or modifying a meta tag).

A script execution engine is a complex and large system. Implementing a new one is beyond the scope of the project. An external engine could be used but it represented a high risk (i.e. finding an engine, checking if it was compatible with our architecture, learning how to use it, etc.).

**First sprint**

The goals of this spike were:

- Construct the architecture of the prototype.
- Define the test suite to use and build the test harness.
- Complete the tokenizer.

I was the project leader in this sprint. In order to complete the sprint, during the first week we had team meetings for discussing and programming the architecture of the system. I defined and coded a few classes (taking as a reference the code of the previous spikes) and suggested them to the team. We agreed to use that code as a base of the architecture and continued working over it.

For the second week, the tokenizer states were divided and each team member worked individually on 23 states. The division was made in order to work on related states, i.e. DOCTYPE, tags, attributes, comments, text, etc.

\(^4\) Trello is a helpful web application for managing and organizing projects. More information in https://trello.com
The test harness was built using JUnit. The test suite of html5lib was chosen due its simplicity and quantity of test cases (more than 3000 for parsing; details in Results section). By using test cases, the testing method was a dynamic white box approach. Unit tests were not used because the test cases present an input and the expected output (list of tokens and number/type of errors), i.e. an integration test of the whole tokenizer. The plan contemplated to define and develop the test harness for both, tokenizer and tree construction. Nevertheless, we decided to focus only in the tokenizer and leave the tree construction test harness for next sprints.

Once the tokenizer states were finished, testing began. We adopted an approach to individually work on fixing errors and failures specifying which test cases we were working on to avoid conflicts. At the end of the sprint we started to discuss the plan for the next sprint.

**Second sprint**

The goals of this spike were:

- Finish the architecture of the prototype.
- Build the test harness for the tree construction stage.
- Code the insertion modes and another algorithms used in the tree construction stage.

In the first week we had team meetings to complete the system architecture and we defined a general interface for all the algorithms. The complexity of the tree construction stage is higher than the complexity of the tokenizer because it includes many algorithms and data structures. Overall, there is dependency between the insertion modes and algorithms. The division of work was as follows:

- Insertion modes 1 to 8 (related to html, head and body elements).
- Insertion modes 9 to 17 (related to table elements). Algorithms related to formatting elements, adoption agency and parsing foreign content.
- Insertion modes 18 to 23 (related to templates and framesets). Algorithms related to creating and inserting nodes (elements, comments and text), parsing HTML fragments.

It was hard to find a way to divide the work minimizing code dependencies in order to avoid conflicts. We identified all the algorithms used and divided them into two sets following the spec subsections. I worked on the second set. The first set only contained insertion modes because the in body insertion mode is the largest of all the spec. Due to the complexity of that insertion mode, we all ended up helping to finish it on time.

**Third sprint**

This final sprint goal was to integrate and test the entire parser prototype. The integration was almost completed during Sprint 2 and thus we focused mainly on testing. We continued to work individually on testing and trying to fix the most errors as possible.

After the first week of the sprint, we discussed and agreed with our supervisor to pause the sprint in order to focus on writing this report. The sprint will be continued after the delivery of this report. Section 5 presents testing results of the first week of this sprint.
5. Results
The current prototype has been tested using the html5lib test suite. In the section of the tokenizer, 2112 test cases have been run with no errors or failures (Figure 4).

![Figure 4 – Test results of the tokenizer stage](image)

As mentioned in the previous section, the testing and error fixing tasks for the tree construction stage were paused in order to complete this report. Currently, a total of 1062 test cases have been run leading to 126 errors (DOM not generated by some reason, e.g. null pointers, uncaught exceptions, etc.) and 253 failures (DOM comparison failed, e.g. nodes missing, misnested content, etc.). Figure 5 presents the JUnit execution of the tests.

![Figure 5 – Test results of the tree construction stage](image)

Using percentages, the tokenizer stage is 100% completed. Meanwhile the tree construction stage is at 64.3%. Considering that the html5lib test suite guarantees a complete, WHATWG conformant HTML5 parser, the prototype is complete at 82.15%.

One important tool of the methodology we used was the backlog. User stories, priorities and estimations were registered in the backlog. With regard to the first sprint, the estimated time was of 120 hours (20 per week per person) and the real time spent was of 130 hours. One of the tokenizer states (number 69 - Tokenizing character references) caused the delay.

For sprint 2, the plan estimated 95.5 hours for the completion of the tasks. The estimation was very optimistic and at the end we completed the sprint activities in around 120 hours. The activities for sprint 3 were integration and testing. There were no estimations for this sprint besides the limit time of 120 hours. As discussed previously, after a week (60 hours), the test cases reflect that the tree construction stage is at 64.3%.

After the spikes we estimated that building a parser would take around 500 hours. Prior to beginning the prototype we reduced the estimation to 360 hours. As mentioned before, after 300 hours the parser is 82.15% completed and I am confident that we can finish it in the remaining 60 hours.

The prototype is hosted in Github, it can be accessed from [23]. It contains nearly 160 files. The first commit was made on 19 March 2015 and the last on 20 April 2015. A total of 159 commits have been done.

6. Conclusions
We, as a team, had difficulties working together at the beginning. The spikes were considered to be done in one week each one; nevertheless it took us three weeks for both spikes. There were some communication issues, frictions by different working styles and abilities and a few general disagreements.
During the spikes I felt low commitment of my teammates as I ended up writing around 70% of the code. After that we came to an agreement and modified our planning and working styles to develop the prototype.

While developing and testing the prototype of the HTML5 parser some difficulties have been presented:

- Our current implementation uses an XML DOM object to build an HTML5 DOM. Due to the permissiveness of HTML5 some characteristics cannot be fit into an XML DOM. For example, in XML an attribute name may start only with an underscore or a letter but HTML5 allow some special characters or numbers as name for attributes.

- None of us read thoroughly the spec before starting to build the prototype architecture. One tokenizer state (the last one, to our bad luck) has a different behaviour than the others. When we realised that case it was too late to rebuild the architecture and the other tokenizer states. At the end we decided to hack that state to work under the defined architecture.

- The HTML5 spec is definitely large and to some extent hard to follow. Some steps of the algorithm were too verbose for me and hard to understand and code, specially the Adoption Agency Algorithm. For example, to store the tokens generated from the tokenizer, a queue was used. Once the token was processed by the tree constructor, it was disposed. The Adoption Agency Algorithm has the capability to manipulate the DOM by moving nodes to other nodes. In order to do that, it requires storing the token that produced an element in the tree (i.e. start tag tokens). Luckily we realised that the node object has a method (setUserData) that allows associating a given object (the token, in this case) with it. Some other hacks like this were required frequently.

- This implementation was developed using the W3C recommendation. To test it, the html5lib test cases were used. Those test cases are compliant with the WHATWG spec and thus some of them fail because the WHATWG spec is continuously updated and there are some changes with respect to the W3C spec. For example, the Adoption Agency Algorithm have one single step different from the W3C spec and the WHATWG standard. That difference lead to different nesting structures under some circumstances. While checking a failing test case, it took me a couple of hours to realise the problem was the algorithm used and not the code.

- When testing the tokenizer we faced some difficulties to compare the errors and character tokens. For simplicity, when the output contains several character tokens, the test expected output presents only one token (concatenation of all the characters). Nevertheless, the errors are presented in the output in the order they were generated, e.g. {Token: “Manch”, ERROR, Token: “ester”}. Our implementation stores tokens and errors in different data structures. Thus, at the end of the process the output were two lists, one containing one error and the other containing each character of “Manchester”. As there was no way to match the prototype output with the test expected output, the error position is not validated. The error count is validated instead.

Besides the technical conclusions, I have the following personal conclusions:

- I have used for the very first time an Agile methodology: doing spikes, writing user stories, using a backlog, estimating and delivering by 2-weeks sprints, etc. It was really simple but helped me to understand the methodology and realise its potential. For example, before starting the prototype I
was doubtful on how to plan it, I was stressed because I considered it too risky and I did not know how to plan and estimate it. When I was the leader during one sprint, those techniques made me feel more confident about the progress of the prototype. The risk could not be avoided but definitely it was reduced.

- It was interesting and enjoyable to do pair programming. As a team we met to build the system architecture together and I think that helped us to complete the prototype on time.
- My skills in Java have increased, I have learned some design patterns, I have better understanding of testing methods and testing types, I have used for the very first time JUnit and overall I have understood the importance of a good testing strategy.

The project had many paths to follow and I was doubtful about which one to choose. While testing and fixing errors, I realised how useful would be a tool that facilitates the task of tracing and finding errors and comparing outputs from different sources. That was the motivation that made me choose the test methodology path.
Appendix A – Overview of the HTML5 parsing process

References


