Benchmarking and Optimisation of Engage!-based XML Parsers

Frank Groeneveld
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
f.groeneveld@student.utwente.nl

ABSTRACT

In this research, we implement two event-based specifications of XML, from which *Engage!* can then generate a parser. These parsers are then differentially tested against an existing event-based parser for XML, .NET's XMLReader. Based on the results of various tests, various possible improvements to *Engage!* have been found and suggestions on how to further improve the performance of the tool will be proposed.

Keywords

Parsing, Compilers, Event-based parsing, XML, SAX

1. INTRODUCTION

Parsing is a well explored problem and many different techniques are at our disposal [2]. Traditionally, parsers use a tree-based approach, where an in-memory tree representation is generated for the entire parsed input, even though the kinds of trees and methods of obtaining them can be very different [16]. This is a very memory-intensive process, especially in the case of large inputs, such as the OpenStreetMap database [9], which is, when uncompressed, 1352.6 GB in size at the moment of writing. On many systems, it is impossible to process such a large file as the memory available will not be able to fit the entire tree.

SAX, or Simple API for XML [5] was developed as an alternative to the classical approach, and could arguably be the first practical application of event-based parsing [15]. The idea behind this kind of parser is to process a tag as an event and then forget about it, which means that SAX is much more efficient in terms of memory usage and time needed to process the input, as no complete in-memory representation has to be generated [11].

In 2019, *Engage!* [15] was introduced as the first event-based parser generator. While a parser for an existing language, AppBuilder [10], has already been generated and tested against a tree-based parser, the technology is still very young and leaves a lot to explore. This means that there exists little documentation of the software, aside from the code [17] and the paper [15], both mostly written by V. Zaytsev, the author of *Engage!*.

Because there are no other known event-based parser generators and *Engage!* being such a young technology, little is known on how well this tool performs in practice. While it has already been shown to work in the case of AppBuilder, the performance of the parsers generated has not been tested extensively.

By differentially testing [3] an XML-parser generated by *Engage!* against a SAX-like parser, such as XMLReader [8], it can be seen how *Engage!* performs, and possible optimisations, either in the XML language specification or *Engage!* itself will become known.

XMLReader is an industrial strength event-based parser that can be used to parse XML-files in C#, in which *Engage!* was also written. It is a mature technology, and like SAX, it generates a stream of events from an XML input, rather than generating a tree from the entire input. There are however some differences [6]. Most importantly, XML follows a pull model, which means events have to be pulled from the reader, as opposed to being pushed by the parser, which is the case for SAX-based parsers and *Engage!*-based parsers.

Depending on the size and type of the XML inputs used in the tests, the loss or gain in performance between the two parsers becomes apparent, and optimisations that can be used to improve the performance of the parser, are suggested. The implementation of those suggested optimisations can improve the performance of *Engage!*, to bring it closer to an industrial strength event-based parser like SAX or XMLReader, while also providing the flexibility of allowing programmers to write their own specifications for any language desired.

In the next section, we will formulate the research questions that have been answered. We will then provide the methodology of the research and the results obtained. Finally, we present our results and conclusion.

2. RESEARCH QUESTIONS

In order to explore the performance of *Engage!*, and how well it performs compared to XMLReader, the following research questions have been asked.

1. Can XML be implemented in *Engage!*? And which language extensions, if any, would be needed?
2. Which is faster, XMLReader, or the *Engage!*-based XML parser?
    (a) In the case of small files?
    (b) In the case of realistic files?
    (c) In the case of large files?

3. Where does *Engage!* lose the most speed? What optimisations could be implemented in order to improve its performance?

3. RELATED WORK

Engage! [15, 17] is an event-based parser generator: given a parsing specification, it generates a parser that can then be used to process inputs in an event-based way. It works as follows: A specification consists of rules which have the form of triggers -> actions. The trigger is usually a token, or a token accompanied by a flag. An action is either done immediately, or executed when a different token is found.

Consider this example by Zaytsev [15, p.2]:

```
'dcl'    -> push DclBlock(types)
            where types := await* String
'char'   -> push String(n)
            where n := await number
'enddcl' -> trim String
```

This example pushes a DCLBlock containing all Strings that exist within an input that begins with 'dcl' and ends with 'enddcl'. The first rule specifies that when a 'dcl' token is reached, a DCLBlock initialized with all Strings found will be pushed to the stack. A string is constructed by adding all character tokens until a number is found. Finally, when the 'enddcl' token is found, the `trim` action is called, which stops all other ongoing actions, in this case the `await*` action, so that the event that pushes the DCLBlock is pushed.

SAX, or Simple API for XML [5], is the first practical application of event-based parsing, developed since 1997 [13]. The latest release of SAX at the moment is 2.0.2, released in 2004. Contrary to *Engage!*, in SAX the end developer does not write specifications for each scenario: any XML file can be parsed with SAX, and the events are caught and handled programmatically. It works as following, consider the following simple XML file:

```
<DocumentElement param="value">
    <FirstElement>
        Content
    </FirstElement>
</DocumentElement>
```

When this input is passed through a SAX parser, it will generate a sequence of events as following:

- XML Element start, named *DocumentElement*, with an attribute *param* equal to "value"
- XML Element start, named *FirstElement*
- XML Text node, with data equal to "Content"
- XML Element end, named *FirstElement*
- XML Element end, named *DocumentElement*

The user defines a number of callback methods that will be called when specific events occur during parsing. Consider the following example in Java:

```
public void startElement(String uri,
String localName, String qName,
Attributes attributes)
throws SAXException {
    System.out.println(qName);
}
```

This function is called every time a XML Element start event is pushed, in which case the QName, or qualified name of the XML element is printed. When parsing, SAX drops events right after the callbacks received them, and there is no way to go back. The entire document will have to be parsed again if an event was not handled correctly.

DOM, or Document Object Model [14], is an alternative to SAX, also widely used in the XML ecosystem. Unlike SAX, it is standardised as a W3C specification. Additionally, DOM does not rely on events like SAX does. It loads the entire XML document into memory to parse it into a tree-based structure. This means it is less memory efficient, but at the advantage of being easier to operate on, as the the entire document and its structure is loaded into memory.

XMLReader [8] is a reimplementation of the classic SAX, meant to be used on the .NET Framework and .NET Core (while SAX targeted JVM). It is an industrial strength event-based parser that is widely used to parse XML-files in C#. There are however some differences [6], the most important being that XMLReader uses a pull mechanism. Where SAX's push mechanism pushes events which are handled by callbacks, XMLReaders pull system gives the user control over the decision of when to pull the events, which means that there is no need to pull the entire document if only certain information is needed.

RxParse [1] is another framework for event-based SAX-style parsing for Java. It is an internal domain-specific language (essentially, a library). It ended up being out of scope of this project: there is some interest in their community in making a similar event-based library for event-based LINQ-style parsing for .NET [12], as well as some interest in making a detailed feature-wise comparison of RxParse with *Engage!* [15, p.6].

4. METHODOLOGY

Various parsers were benchmarked when solving two problems. The parsers, benchmarks and input data used are described below.

4.1 Parsers

A simple *Engage!* specification was developed in order to parse XML tags. This specification can be found in Figure 1. Based on this specification, a parser, EAX, is generated that recognises the tags in an XML document and its contents. Once the end of the input is reached, all the found tags are then returned and stored in an `EngagedXmlDoc` object, which then can be used to read the parsed XML. This parser parses the entire input, and then returns all the tags found. Then, in order to access the data parsed, one can iterate over all the returned tags.

Additionally, an additional *Engage!* specification was developed to generate a parser, EAXFind, which only returned the contents of a specified tag. This means that the parser still has to parse the entire input, but only the specific tag has to be processed afterwards, which means fewer events have to be iterated over when processing the parsed input. This specification can be found in Figure 2.

To test these parsers, XMLReader was used to parse the generated inputs.

4.2 Benchmarking

4.2.1 Unique tags

The simple *Engage!*parser and the XMLReader parser each had to count the number of unique tags present in the input, this was timed using the built in Stopwatch [7]. The

```
namespace EaxOpenClose

types
    EngagedXmlDoc;
    Name;
    TagOpen, TagClose <: TagEvent;

tokens
    ' ', '\r', '\n' :: skip
    '<', '>', '!', '/' :: mark
    string :: Id

handlers
    EOF              -> push EngagedXmlDoc(tags) where tags := pop* TagEvent
    '<'              -> lift TAG
    '<'              -> lift OPEN
    '/' upon TAG     -> lift CLOSE
    '/' upon TAG     -> drop OPEN
    Id upon TAG      -> push Name(this)
    '>' upon OPEN    -> push TagOpen(n) where n := pop Name
    '>' upon CLOSE   -> push TagClose(n) where n := pop Name
```

Figure 1. EAX (*Engage!* API for XML) specification for generating a stream of `TagOpen`/`TagClose` **objects. Online version:** https://github.com/raincodelabs/engage/blob/master/EAX/specs/OpenClose.eng

```
namespace EaxFuzzy

types
    EngagedXmlDoc;
    Name;
    TagOpen, TagClose <: TagEvent;

tokens
    ' ', '\r', '\n' :: skip
    '<', '>', '!', '/' :: mark
    'a' :: word
    string :: Id

handlers
    EOF                  -> push EngagedXmlDoc(tags) where tags := pop# TagEvent
    'a' upon TAG_CLOSE  -> drop PARSE
    'a' upon TAG        -> lift PARSE
    'a' upon TAG_CLOSE  -> drop TAG
    'a' upon TAG_CLOSE  -> drop CLOSE
    'a' upon TAG        -> drop TAG
    'a' upon TAG        -> drop OPEN
    '<'                  -> lift TAG
    '<'                  -> lift OPEN
    '<'                  -> drop CLOSE
    '/' upon TAG        -> lift CLOSE
    '/' upon TAG        -> drop OPEN
    Id upon TAG          -> push Name(this)
    '>' upon PARSE_OPEN -> push TagOpen(n) where n := pop Name
    '>' upon PARSE_CLOSE -> push TagClose(n) where n := pop Name

    '>' upon IGNORE      -> lift IGNORE
    '<'                   -> lift IGNORE
```

Figure 2. EAX (*Engage!* API for XML) specification for generating a stream of `TagOpen`/`TagClose` **objects only for tags found inside tags <a>. Online version:** https://github.com/raincodelabs/engage/blob/master/EAX/specs/Fuzzy.eng

number of tags generated and the ticks required in order to count these have then been plotted using Matplotlib [4].

4.2.2 *Search*

Both EaxFind and XMLReader had to find one tag in the input, which was timed by the same Stopwatch used for counting the unique tags. The number of tags generated, excluding the tag that had to be found, and the ticks taken before this tag could be accessed were stored, and finally plotted in the same way as the previous experiment.

4.3 Input

4.3.1 *Unique tags*

For various types and sizes, an input was randomly generated every time a test was ran. The input could either be shallow, deep, or mixed in terms of nested-ness.

To elaborate, the shallow inputs were generated by adding random tags that open and then close to the input, which is then put in one XML root node. In the case of deep inputs, every tag generated is nested inside the previous generated tag. For the mixed input, a more realistic XML input was generated, which had a maximum depth of 4 nodes, which resulted in the parser having to decide whether to transcend deeper into a node or to go to the next one. The size of the input was between 10 and 11000 randomly generated XML pairs of tags which open and close.

4.3.2 *Search*

In order to test the search performance of EaxFind and XMLReader, three types of input were generated, first, middle, and end. These inputs, of size 10 to 6200, contained one specific tag to be found, either placed first, in the middle or in the end, in order to determine how the parsers would perform in best, average and worst cases.

5. RESULTS

5.1 Unique tags

As can be seen in the first three figures (Figure 3, Figure 4, Figure 5), XMLReader outperforms the *Engage!* parser significantly. This is quite irrelevant for smaller inputs, but very inconvenient for larger inputs, especially considering many XML files contain much more than 10000 tags. Whether the input was deep or shallow did not make a noticeable difference to the performance of the parsers, but the realistic input, where XMLReader performed best, took almost twice as long to parse compared to the other two input types.

5.2 Search

As can be seen in Figure 7, regardless of the tag being placed conveniently in front or in the worst case, in the back, the *Engage!*-based parser takes a lot longer to find the tag than the XMLReader parser, of which the result of the benchmarks can be found in Figure 8. Unfortunately, the new specification does not seem to improve the performance of the *Engage!* parser much, and XMLReader greatly outperforms our parser.

The fact that the *Engage!* parser benefits so little when the tag was found in the front can be explained by the fact that the parser will still parse the entire input before pushing the result. When comparing these runs to the unique tag counting runs of the *Engage!* parser, we find that there is little to gain by reducing the number of tags that *Engage!* returns, and instead a large portion of the time used is spent parsing the input and generating the tags instead. Therefore, in order to improve the performance, the focus

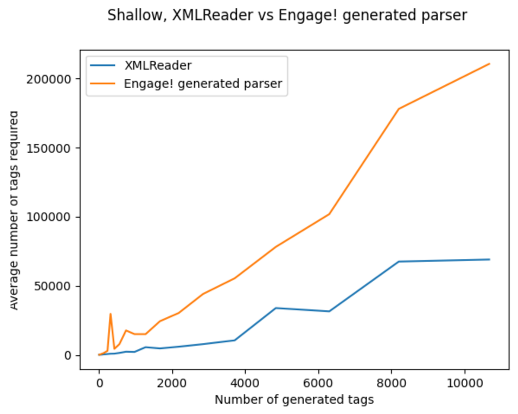

Figure 3. EAX vs SAX in the case of shallow inputs

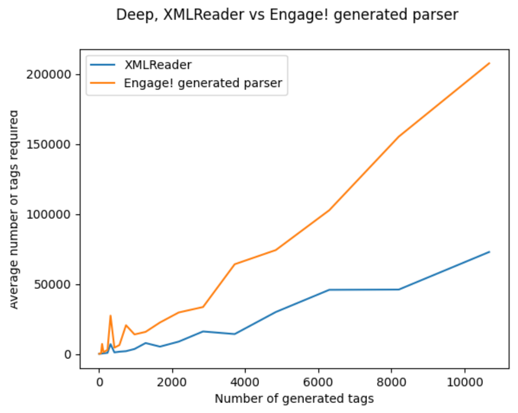

Figure 4. EAX vs SAX in the case of deep inputs

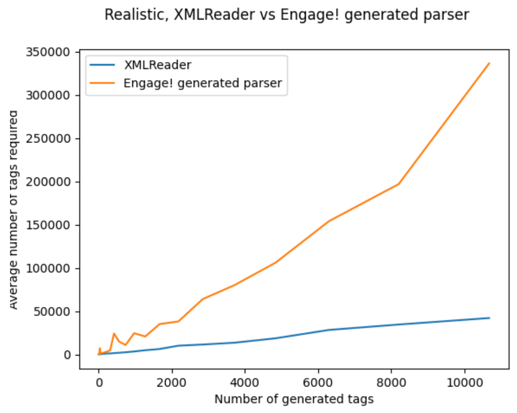

Figure 5. EAX vs SAX in the case of realistic inputs

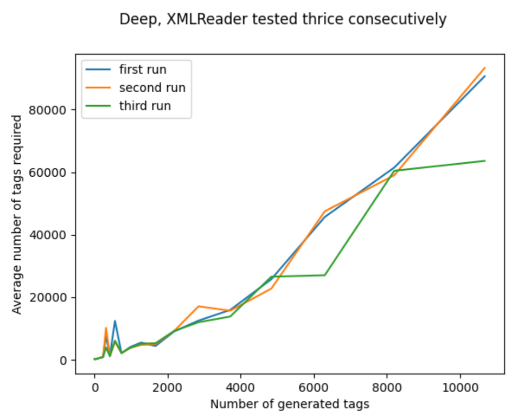

Figure 6. A comparison of three runs of the XMLReader parser on deep inputs

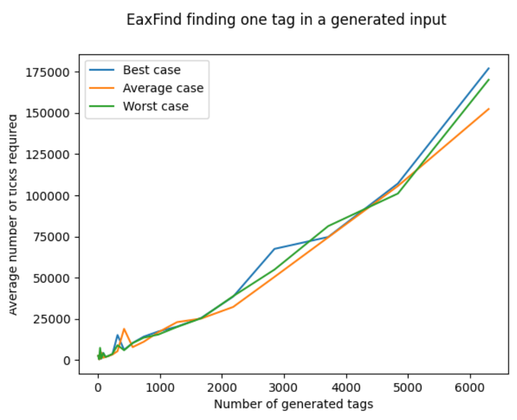

Figure 7. EaxFind finding one tag in inputs of various sizes

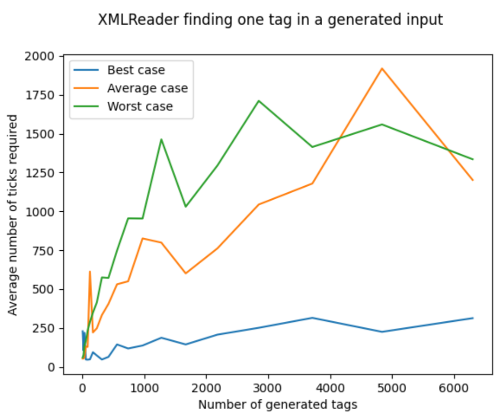

Figure 8. XMLReader finding one tag in inputs of various sizes

should be to reduce the time spent parsing as much as possible. One way to do this would be to make *Engage!* stop parsing once this is no longer necessary.

Currently, *Engage!* does not have an instruction to do so, therefore we propose to add a `stop` instruction, telling the parser to stop parsing once it is called. This could then be used to stop the parser once the desired data is parsed, so that it would not have to parse the rest of the output.

6. CONCLUSION

To conclude: It was possible to implement XML in *Engage!*. No language extensions were needed. XMLReader was significantly faster in all cases of counting unique tasks, overperforming the *Engage!*-based parser more as the input became larger. In order to make *Engage!* viable for processing larger XML inputs, certain improvements will have to be made. The improvement suggested here is to implement the stop functionality described above.

Interestingly enough, the performance of the parsers was changing a lot over various runs of the test, even when averaged over 10 runs, despite efforts to reduce the impact of the machine on the test. These efforts included making sure as few processes as possible were ran in the background, and running a few iterations of parsing that were not timed, before the actual timing was performed. Three consecutive runs of XMLreader have been timed and plotted in Figure 6. As can be seen, the parsing of the various inputs in the first two runs happened in roughly the same time, but the third run shows various anomalies compared to the other runs. This means that even with the precautions taken, the timed runs can show significant variance. Still, reliable conclusions can be drawn when looking at the graphs for the comparisons between the XMLReader parser and the *Engage!*-based parser, because the difference between these trends are so different.

7. REFERENCES

[1] A. Chen. RxParse: Reactive Parse. GitHub, https://github.com/yongjhih/RxParse, 2015.

[2] D. Grune and C. J. H. Jacobs. *Parsing Techniques — A Practical Guide*. Monographs in Computer Science. Addison-Wesley, second edition, 2008.

[3] M. A. Gulzar, Y. Zhu, and X. Han. Perception and practices of differential testing. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '19, page 71–80. IEEE Press, 2019. https://doi.org/10.1109/ICSE-SEIP.2019.00016.

[4] Matplotlib. Matplotlib. https://matplotlib.org/index.html, 2021.

[5] D. Megginson. Simple API for XML. Megginson Technologies, http://www.megginson.com/downloads/SAX/, 1998.

[6] Microsoft. Comparing xmlreader to sax reader. 2011. https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-4.0/sbw89de7(v=vs.100)?redirectedfrom=MSDN.

[7] Microsoft. Stopwatch. https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-5.0, 2019.

[8] Microsoft. Xmlreader. https://docs.microsoft.com/en-us/dotnet/api/system.xml.xmlreader?view=netcore-3.1, 2019.

[9] Openstreetmap. Openstreetmap data file. 2021. https://wiki.openstreetmap.org/wiki/Planet.osm.

[10] Raincode. The raincode tialaa compiler. https://www.raincode.com/technical-landscape/tialaa/, 2018.

[11] K. Sall. Xml family of specifications: A practical guide. 2002. ISBN-13: 978-0-201-70359-7.

[12] S. C. Taylor. Reactive Parser Combinators. Microsoft Developer Network Forums, https://social.msdn.microsoft.com/Forums/en-US/0f72e5c0-1476-4969-92da-633000346d0d/reactive-parser-combinators, 2010.

[13] J. Tigue, P. Murray-Rust, T. Bray, D. Megginson, and D. Brownell. SAX Genesis. Megginson Technologies, http://www.saxproject.org/sax1-history.html, 1998.

[14] W3C. Document Object Model (DOM) Living Standard, Jan. 2021.

[15] V. Zaytsev. Event-Based Parsing. In T. Kamina and H. Masuhara, editors, *Proceedings of the Sixth Workshop on Reactive and Event-based Languages and Systems (REBLS)*, 2019. https://doi.org/10.1145/3358503.3361275.

[16] V. Zaytsev and A. H. Bagge. Parsing in a Broad Sense. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014)*, volume 8767 of *LNCS*, pages 50–67. Springer, Oct. 2014.

[17] V. Zaytsev, M. Samy, and F. Groeneveld. *Engage!* GitHub, https://github.com/raincodelabs/engage, 2019.