XML Schema Validation Using Parsing Expression Grammars

Kimio Kuramitsu[1] **and Shin'ya Yamaguchi**[1]

[1]**GraduateSchool of Electronic and Computer Engineering, Yokohama National University**

ABSTRACT

Schema validation is an integral part of reliable information exchange on the Web. However, implementing an efficient schema validation tool is not easy. We highlight the use of parsing expression grammars (PEGs), a recognition-based foundation for describing syntax, and apply it to the XML/DTD validation. This paper shows that structural schema constraints in document type definitions (DTDs) can be validated by the converted PEGs with the linear time and constant space consumption. We study the performance of several existing PEG-based tools, and then confirm that the converted PEGs achieve a practical and even competitive level of performance under existing standard XML/DTD validators.

Keywords: Schema validation, XML/DTD, parsing expression grammars, and performance

1 INTRODUCTION

Many applications today have accepted the XML standard Bray et al. (2006) for encoding their information. Since XML is flexible enough to carry various types of information, application developers need to validate whether an XML document contains their demanded content. XML schema languages such as XML/DTDBray et al. (2006) and XML SchemaThompson et al. (2004) have been designed and are readily available for the XML validation task.

Implementing an efficient XML schema validation tool is not easy and has not been solved yet. One reason for this is that major XML schema languages including XML/DTD, XML Schema, and Relax NG, are based on regular tree automata, in which the underlying tree construction generally requires linear space the size of the XML. To make the memory consumption constant, we need another theoretical sophistication, such as in Papakonstantinou and Vianu (2000); Segoufin and Vianu (2002); Green et al. (2002); Kumar et al. (2007), but there remains a huge gap between the theory and the implementation. In part due to this gap, many computing environments, such as JavaScript on Web browsers and C on embedded systems, have no standard XML validators.

We address a translation approach to the implementation of a practical XML validator using *parsing expression grammars* Ford (2004). PEGs are a relatively new and pragmatic grammar foundation, formalized in 2004 by Ford. PEGs are expressive but can usually be implemented through simple recursive descent parsing, which offers constant memory parsing. In addition, the deterministic features of XML schema languagesBrüggemann-Klein and Wood (1998) allow us to restrict backtracking, which may eliminate the potential exponential time.

In this paper, we chiefly focus on the DTD of the XML standardBray et al. (2006). There are several reasons for our focus. First, the DTD is the first schema language of XML and has a rich history in industry; many practical XML standards have been specified in DTDs. Second, a DTD is relatively simple and fundamental, compared to other XML schema languages. Nevertheless, DTDs contain several standard schema constraints, such as data types and unique keys, thus bringing good insight when the readers apply our approach to other schema language.

In theory, a translation of DTDs into PEGs seems trivial, since DTDs can be regarded as a deterministic regular expressionMedeiros et al. (2014). In practice, however, we have found several difficulties in DTD-to-PEG conversion. Obviously, parsing expressions as well as regular expressions cannot express non-structural constraints such as ID and IDREF, which are related to the key constraint and foreign key constraint in relational schema. Accordingly, we focus on the structural constraints of DTDs.

```
<article cite="Codd70">
  <title>A Relational Model for Large Shared Data Banks</title>
  <author>
    <name> E. F. Codd </name>
    <affiliation> IBM Research </affiliation>
  </author>
</article>
```

Figure 1. An Example XML Document

```
<!ELEMENT article(title, author*)>
<!ATTLIST article cite CDATA>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author(name, affiliation)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT affiliation (#PCDATA)>
```

Figure 2. An Example DTD

Even among the structural constraints, we have the attribute-element content problem, which is derived from the unordered nature of XML attributes. Note that regular expressions cannot express the attribute-element content. In parsing expressions, we can express the unordered attribute with a permutation of attributes, but it requires a factorial ($n!$) time in n attributes. In this paper, we make an approximation by allowing the repeated occurrence of optional attributes, and avoiding the factorial time in practice. A rationale for this approximation has been examined by empirical studies on real-world DTDs.

The advantage of our PEG-based translation approach is that many existing PEG tools are readily available in various computing platforms, even where no standard XML validator exists. To study its performance extensively, we have made several DTD converters to major PEG-dialects, such as PEGTLHirsch and Frey (2014), Rats!Grimm (2006), Mouse Redziejowski (2007), LPegMedeiros and Ierusalimschy (2008), and PEGjsMajda (2015). We confirm that many PEG-based tools achieve very competitive performance, compared to standard XML/DTD validators. These results indicate that PEGs can be a practical means of validating most structural constraints of DTDs and similar XML schema languages.

The remainder of the paper is structured as follows. Section 2 is a brief introduction to DTDs, and Section 3 is a short introduction of PEGs. Section 4 presents a body of the conversion algorithm of DTDs into PEGs. Section 5 demonstrates the experimental results. Section 6 reviews related work. Section 7 concludes the paper.

2 DOCUMENT TYPE DEFINITION

A DTD is a grammar for describing the structure of an XML document. A DTD constrains the structure of an element by specifying a regular expression to which its subsequences have to conform. Figure 1 illustrates an example XML document, in which the root element (article) has two nested sub-elements (title and author) and the author element, in turn, has two nested elements. Figure 2 illustrates a DTD to which example XML document conforms to. More details on the XML specification can be found in Bray et al. (2006).

In this section, we focus on three core DTD declarations: *element type declaration* `<!ELEMENT>`, *attribute lists* `<!ATTLIST>`, and *entity declarations* `<!ENTITY>`. Note that notation declaration is another declaration of DTDs, which is not focused on in this paper, as notation declarations are used for describing references to external resources, which are usually treated as unparsed.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015

e	::=	*EMPTY*	:	no content
	|	*ANY*	:	any XML contents
	|	#PCDATA	:	parsed characters
	|	X	:	element name
	|	e, e	:	sequence
	|	$e\|e$	:	alternation
	|	$e?$	:	option
	|	$e*$	:	repetition

Figure 3. Syntax definition of DTD elements

Table 1. XML Attribute Type

Type	Description
CDATA	The value is character data
$(v_1\|v_2\|...)$	The value must be one from an enumerated list
NMTOKEN	The value is a valid XML name
ID	The value is a unique id
IDREF	The value is the id of another element

2.1 Element Declaration and Content Model

An element type declaration defines an element and its content, whose syntax is formed as:

$$\texttt{<!ELEMENT } X\ (e)\texttt{>}$$

where an element X has a content model specified by e.

The content model e is a regular expression, inductively defined as in Figure 3. The uniqueness is that the element name X plays roles in both terminals such as `'<X>'` and `'</X>'` and nonterminal referencing in its content specification. Roughly, this relation corresponds to the definition of the CFG-style production rule:

$$X \mapsto \texttt{<X>}\ e\ \texttt{</X>}$$

Due to the pair of the opening `'<X>'` and the closing `'</X>'`, XML elements are always regarded as a tree, in which each node is labeled by an element name. In general, DTDs are viewed as a *local* regular tree grammarMurata et al. (2005). The "local" property is a restricted class in tree grammars, in which two productions rules do not start with the same element name. In addition, DTDs disallow non-deterministic regular expressions, such as $(a|(a,b))$ and $(a*,a)$. These properties are also called *one-unambiguous* regular expressionsBrüggemann-Klein and Wood (1998). More importantly, these restrictions are intended to avoid backtracking and the underlying exponential time cost.

2.2 Attribute Lists

In DTDs, the structure of attributes are specified differently from that of the elements. An attribute list specifies the list of all possible attributes for a given element. The syntax of an attribute list $(a_1,...,a_i,...)$ of the element X is formed as:

$$\texttt{<!ATTLIST } X\ a_1\ t_1\ v_1\ ...\ a_i\ t_i\ v_i...\ \texttt{>}.$$

where a_i is the declared names of the attribute, t_i is its data type, and v_i is its default value. Table **??** shows the summary of XML attribute types for t_i. The attribute type CDATA/NMTOKEN represents a lexical pattern for the attribute value, while the ID/IDREF type represents non structural constraints on XML elements, similar to the key and foreign key constraints in relational schema. Table 2 shows a summary of XML value types for v_i.

It is important to note that attributes are *unordered*, and require membership validation in the set semanticsHosoya and Murata (2003); Ghelli et al. (2008). That is, we allow $a_i\ a_j$ as well as $a_j\ a_i$, and disallow the repetition $a_i\ a_i$. Since regular expressions in general cannot recognize such unordered

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015

Table 2. XML Value Type

Value	Description
`#REQUIRED`	The attribute is required
`#IMPLIED`	The attribute is optional
`#FIXED` value	The attribute value is fixed as value

Table 3. PEG Operators

PEG	Type	Proc.	Description
' '	Primary	5	Matches text
[]	Primary	5	Matches character class
.	Primary	5	Any character
A	Primary	5	Non-terminal application
(p)	Primary	5	Grouping
$p?$	Unary suffix	4	Option
$p*$	Unary suffix	4	Zero-or-more repetitions
$p+$	Unary suffix	4	One-or-more repetitions
$\&p$	Unary prefix	3	And-predicate
$!p$	Unary prefix	3	Negation
$p_1 p_2$	Binary	2	Sequencing
p_1 / p_2	Binary	1	Prioritized Choice

concatenation, the developers of the XML validator usually abandon formal validation of the membership requirement. Instead, they rely on the two-stage implementation; that is, they count the attributes to validate after the formal validation.

2.3 Entity declaration

The entity declaration is a *macro* that is assigned a value that is replaced throughout the XML document. The syntax of the entity declaration is formed as:

```
<!ENTITY % a "v">
```

where a is a macro name and v is a replaced value. In XML documents, the defined macro `a` is used as `&a;` by enclosing "&" and ";" characters. The predefined macro `<` in the XML standard is a good example of entity.

3 PARSING EXPRESSION GRAMMARS

Parsing expression grammars (PEGs, Ford (2004)) are a new and pragmatic grammar foundation that has received much attention among the programming language communityFord (2014). This section is an brief introduction to PEGs.

3.1 Grammars, Expressions and Operators

A parsing expression grammar G is formalized with a 4-tuple $G = (N, \Sigma, P, p_s)$, where N is a finite set of nonterminals, Σ is a finite set of terminal characters, P is a finite set of expressions, and p_s is a start expression. We use the variables a, b, and c to represent terminals; A, B, and C for nonterminals, and p for parsing expressions. Each production, specified by an EBNF-like form $A = p$, is a mapping from a nonterminal A to a parsing expression p.

Table 3 shows a summary of PEG operators that comprise an expression. The string 'abc' exactly matches the same input, while [abc] matches one of these characters. The . operator matches any single character. The lexical match consumes the matched size of characters and moves forward to a matching position. The $p?$, $p*$, and $p+$ expressions behave as in common regular expressions, except that they are greedy and match until the longest position. The $p_1\ p_2$ attempts two expressions, p_1 and p_2, sequentially, backtracking to the starting position if either expression fails. The choice $p_1\ /\ p_2$ first attempts p_1 and then

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015

```
File       = PROLOG? _* DTD? _* Element _*
PROLOG     =  '<?xml' (!'?>' .)* '?>'
DTD        = '<!' (!'>' .)* '>'
Element    = '<' Name (_+ Attribute)* ('/>' / '>'

               Content '</' Name '>') _*
Name       = [A-Za-z:] ('-' / [A-Za-z0-9:._])*
Attribute  = Name _* '=' _* String
String     = '"' (!'"' .)*  '"'
Content    = (Element / CDataSec / CharData)*
CDataSec   =  '<![CDATA['  (!']]>' .)* ']]>' _*
COMMENT    = '<!--' (!'-->' .)* '-->' _*
CharData   =  (!'<' .)+
_          = [ \t\r\n]
```

Figure 4. Syntax Definition of XML

attempts p_2 if p_1 fails. The expression $\&p$ attempts p without any character consuming. The expression $!p$ fails if p succeeds, but succeeds if p fails.

Figure 4 shows an example of a PEG grammar for the XML syntax. Although the grammar is fairly simplified for readability, the readers will find that it is very simple and powerful, in such a way that at most 11 productions can specify the XML syntax. Note that PEGs are scanner-less, so we specify both lexical and syntax analysis in an integrated way. This means that no separated specification for lexical patterns is required.

3.2 DTDs vs. PEGs

Since DTDs are tree-based regular expressions, it is important to understand the difference between regular expressions and parsing expressions. Despite their syntactic similarity, parsing expressions and regular expressions do not usually accept the same language defined by the expression. For example, the regular expression $(a|ab)c$ accepts {ac, abc}, but the parsing expression $(a/ab)c$ does not accept the string abc. This difference comes from PEGs' deterministic behaviors in the ordered choice. Likewise, the regular expression $a*a$ accepts the string aaa, but the parsing expression $a*a$ does not accept anything, since the repetition $a*$ greedily consumes all subsequent a characters.

As shown below, regular expressions and parsing expressions differ in choice and repetition behaviors. Despite the differences, we can translate any regular expressions into an equivalent parsing expression using tricks described in Medeiros et al. (2014). More importantly, DTDs, as described in Section 2, disallow non-deterministic regular expressions such as $(a|ab)c$ and $a*a$, which behave differently in PEGs. As a result, the DTDs' restriction simply corresponds to the PEGs' determinism.

3.3 Parsing Algorithm

PEGs are usually implemented by a top-down recursive descent parsing algorithm with backtracking. The simplicity of recursive descent parsing is expected to yield good performance. Our preliminary investigation on an XML parser generated from Figure 1 shows very competitive performance compared to a standard XML parser, such as XercesXerces (2015). Likewise, we can expect that the PEG-based XML validator can achieve similar performance. This expectation is confirmed in Section 5.3.

Backtracking is a major concern, since it causes the potentially exponential time, in the worst case. However, DTDs, by nature, are designed to avoid backtracking at the specification level. This suggests that DTD-converted PEG involves very limited backtracking, in such a way that a lookahead of element names and attributes names enables the distinct determination of alternatives. As our experience reported in Kuramitsu (2015) indicates, this will not lead to the exponential behavior without the packrat parsingFord (2002) requiring a linear space consumption. We expect that the DTD-converted PEGs to achieve both linear time validation and the constant space consumption. This expectation is also confirmed in Section 5.3.

Lastly, the simplicity of the recursive descent parsing makes it easier for parsers to be implemented in many programming languages. Many implemented parser generators Ford (2014) are readily available and have supporting evidence for their implementability. In addition, there are several efficient dynamic

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015

$$
\begin{array}{lll}
P_{\texttt{CHAR}} & = & \texttt{.} \\
P_{\texttt{SPACING}} & = & \texttt{[\textbackslash t\textbackslash r\textbackslash n]} \\
P_{\texttt{PCDATA}} & = & \texttt{'\&'}\ P_{ENTITY}\ \texttt{/ (![<]}\ P_{CHAR}\ \texttt{)*} \\
P_{\texttt{ELEMENT}} & = & \texttt{'<'}\ P_{NAME}\ \texttt{(\_+}\ P_{ATTR}\texttt{)* \_* ('/>' / '>' \_*} \\
 & & P_{ANY}\ \texttt{'</'}\ P_{NAME}\ \texttt{'>') \_*} \\
P_{\texttt{NAME}} & = & \texttt{[A-Za-z:] ('-' / [A-Za-z0-9:.\_])*} \\
P_{\texttt{ATTR}} & = & P_{NAME}\ \texttt{'='}\ P_{CDATA} \\
P_{\texttt{CDATA}} & = & \texttt{'"' (!["<\&]}\ P_{CHAR}\ \texttt{)* '"'} \\
P_{\texttt{CDATASEC}} & = & \texttt{'<![CDATA[' (!']]>' .)* \#cdata ']]>' \_*} \\
P_{\texttt{COMMENT}} & = & \texttt{'<!--' (!'-->' .)* '-->'} \\
P_{\texttt{ANY}} & = & P_{ELEMENT}/P_{COMMENT}/P_{CDATASEC}/P_{PCDATA}
\end{array}
$$

Figure 5. Predefined Syntax for XML Values

parsers Medeiros and Ierusalimschy (2008); Kuramitsu (2016) that can load PEGs at runtime to parse. Since the dynamic parsing can pass through the code generation and its compilation process, we can use it as an existing XML validator.

4 CONVERTING DTDS TO PEGS

As shown in Section 2.1, the core part of a DTD is a regular language, although it is limited to the structural constraints on element/sub-elements. To obtain the full benefits of DTDs, attribute and entity declarations should be translated into PEGs. This section presents an algorithm for converting DTDs to PEGs.

4.1 Predefined Syntax Production

The DTD standard has several predefined lexical patterns, referred to as `#PCDATA` and `CDATA`. To begin with, we specify these patterns as predefined PEG productions. In this section, we use a large P prefix to denote the production name,in order to distinguish it from element names. Figure 5 illustrates the predefined PEG production rules that are commonly used in all conversions.

The XML standard Bray et al. (2006) uses an extended BNF specification to describe the full set of the XML syntax. Roughly, the predefined productions are translated from these BNF rules. For readability, this paper simplifies the definitions. P_{CHAR} is defined as any (.), instead of the defined set of available Unicode characters in the XML standard. $P_{SPACING}$ and $P_{COMMENT}$ are omitted for readability in this paper.

4.2 Converting Elements

We start by presenting a conversion of element-type declarations. For readability, this subsection proceeds under the assumption of the absence of attributes.

First, we consider whether the content model is `EMPTY` or otherwise, since the XML standard allows for a different syntax that is specialized for the empty element. Let P_X be a production name to specify the element X in PEGs. Without loss of the generality, we assume that all production names are distinct from the predefined productions in Section 4.1.

- $P_X = \texttt{'<X>'}\ \tau(e)\ \texttt{'<X>'}$, if `<!ELEMENT` $X(e)$ `>`
- $P_X = \texttt{'<X></X>'}\ \texttt{/ '<X/>'}$, if `<!ELEMENT` X `EMPTY>`

The function $\tau(e)$ is a inductive conversion function that takes a DTD sub-element (e) and then returns its corresponding parsing expression. For all elements defined in Figure 3, the function $\tau(e)$ is defined as:

- $\tau(\texttt{ANY}) = P_{\texttt{ANY}}$
- $\tau(\texttt{\#PCDATA}) = P_{\texttt{PCDATA}}$
- $\tau(X) = P_X$
- $\tau(e_1, e_2) = \tau(e_1)\tau(e_2)$

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015

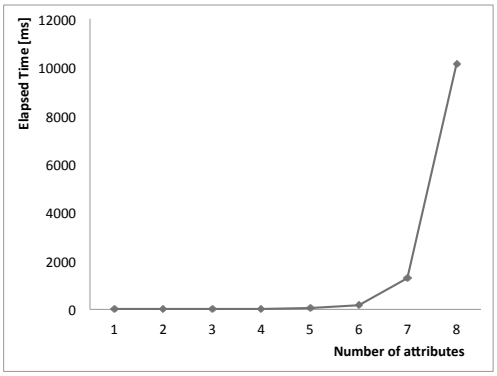

Figure 6. Performance cost of permutation

- $\tau(e_1|e_2) = \tau(e_1)\ /\ \tau(e_2)$
- $\tau(e?) = \tau(e)?$
- $\tau(e*) = \tau(e)*$

Notably, the interpretation of $e_1|e_2$ (in regular languages) is different from that of p_1/p_2 (in PEGs). Since e_1 and e_2 are disjoint in DTDs, we can treat the inductive one-to-one mapping. The repetition $e?$ is similar.

4.3 Converting Attribute Lists

We will turn to the presence of attributes. The element product must include the additional nonterminal to recognize the declared attribute. For simplicity, we first assume that a single attribute a is declared. Let P_{Xa} be a generated production specifying the attribute a of the element X. The included nonterminal differs depending on the attribute type, as follows:

- P_X = `'<X'` P_{Xa} `'>'` ..., if a is `#REQUIRED`
- P_X = `'<X'` P_{Xa}`?` `'>'` ..., if a is either `#IMPLIED` or `#FIXED`

The production P_{Xa} is defined depending on attribute types and attribute values, as specified in Tables **??** and **??**. Let n be a specified attribute name.

P_{Xa} =

- n `'='` P_{CDATA}, if the attribute type is `CDATA`
- n `'='` P_{NAME}, if the attribute type is `NMTOKEN`
- n `'='` P_{CDATA}, if the attribute type is either `ID` or `IDREF`
- n `'='` `'v1'/'v2'`, if the attribute value is $(v_1|v_2)$

Note that we simply treat the `ID`/`IDREF` attributes as `CDATA` and abandon specifying the uniqueness of `ID`/`IDREF` in PEGs. This is because the uniqueness specification is far beyond the expressive power of pure PEGs, and requires a class of context-dependent grammars. Many PEG-dialects, on the other hand, support ad hoc context-sensitive extensions, called *semantic actions*, which enable us to check the uniqueness and references with a small embedded code.

4.3.1 Encoding Unordered Sequence

We will turn to the multiple attribute a, b, c, which must be treated as an unordered sequence $\{a, b, c\}$. Since PEGs, as well as CFGs, cannot directly express such an unordered sequence, we need to encode it with the choice of all possible attribute sequence, or a permutation of attributes – say, $abc/acb/bac/bca/cab/cba$. However, the permutation method obviously fails if the number of attributes grows; n attributes needs $n!$ alternatives. Figure 6 shows our preliminary investigation on the time cost of n permutations with

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015

Validation	DTDs	PEGs
Missing elements	o	o
Undefined elements	o	o
Unique element (ID)	o	x
Element reference (IDREF)	o	x
Missing attributes	o	o
Undefined attributes	o	o
Duplicated attributes	o	# REQUIRED only
Attribute datatypes	o	o
Undefined entities	o	o

o: available, x: not available

Table 4. Comparison of DTDs and converted PEGs

1000-time iterations. Having more than 7 attributes shows an explosive behavior in time and is considered to be impractical.

To decrease the number of attributes in the permutation, we assume that the `#REQUIRED` attribute is strong in practice and not used as much in real DTDs. This assumption will be confirmed in Section 5.2. Based on the assumption, we make a permutation with only `#REQUIRED` attributes. Non-`#REQUIRED` attributes are optionally checked before and after `#REQUIRED` attributes. To summarize, the attribute list consisting of the following four attributes:

```
<!ATTLIST X
  a CDATA #REQUIRED
  b CDATA #REQUIRED
  c CDATA #IMPLIED
  d CDATA #IMPLIED   >
```

is transformed to the permutation of *a* and *b*, mixed within other optional attributes.

```
    (c/d)? a (c/d)? b (c/d)?
  / (c/d)? b (c/d)? a (c/d)?
```

In other words, the conversion of non-`#REQUIRED` multiple attributes is approximate, in that duplicated optional attributes are acceptable. We consider that this approximation to be reasonable enough because duplicated optional attributes will not cause significant errors, in practice.

4.4 Converting Entities

The entity declaration (`<!ENTITY % a "v">`) can be regarded as an additional lexical pattern for $P_{PCCDATA}$ and P_{CDATA}. For example, P_{PCDATA} that accepts an entity `&a;` is generated:

$$P_{PCCDATA} = \texttt{'\&' 'a' ';'} \;/\; \texttt{!'\&'}\; P_{CHAR}$$

The not-predicate `!'&'` checks undefined entities as unacceptable. If you do not want to check entities for several reasons, you may remove the not-predicate.

Finally, Table 4 is a summary of validation capabilities by comparing DTDs and converted PEGs.

5 EVALUATION

We have developed a DTD-to-PEG converter tool based on the conversion algorithm described in Section 4. In this section, we evaluate the converted DTDs and their performance.

5.1 PEG Tools

Nez is a PEG-based grammar tool that provides a parser development framework, including grammar converters, grammar optimizers, parser generators, and parser interpreters. Our DTD-to-PEG converter is

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015

```
RootElement = Element_article
Element_article = '<article' S* Attribute0 S* ('/>'
       / '>' S* Content0 S* '</article>') S*
Attribute0 = AttDef0_0 S* ENDTAG
AttDef0_0 = 'cite' S* '=' S* STRING S*
Content0 = Element_title Element_author*
Element_title = '<title' S* ('/>'
        / '>' S* Content1 S* '</title>') S*
Content1 = PCDATA*
Element_author = '<author' S* ('/>'
       / '>' S* Content2 S* '</author>') S*
Content2  = Element_name Element_affiliation
Element_name = '<name' S* ('/>'
       / '>' S* Content3 S* '</name>') S*
Content3 = PCDATA*
Element_affiliation = '<affiliation' S* ('/>'
       / '>' S* Content4 S* '</affiliation>') S*
Content4 = PCDATA*
```

Figure 7. A Nez version of converted DTDs

developed as a part of the Nez grammar converter. Figure 7 shows an Nez version of the converted DTD from Figure 2.

Nez can generate parsers for multiple platforms, including C, Java, and JavaScript. In this experiment, we tested three kinds of parsers, respectively referred to as CNez (for C), Nez (for Java), and MiniNez (for a small virtual machine on embedded systemsHonda and Kuramitsu (2016)). In addition to these parsers, we translate Nez grammars to major PEG-dialects supported by existing PEG tools. The translated PEG-dialects include:

- PEGTLHirsch and Frey (2014) – The PEG Template Library for C++0x expresses grammars via C++ templates
- LPegMedeiros and Ierusalimschy (2008) – LPeg pattern-matching library for Lua (written in C)
- Rats!Grimm (2006) – a part of the eXTensible C project that builds packrat parsers supporting modular, extensible syntax
- MouseRedziejowski (2007) – a backtracking recursive-descent parser generator
- PEGjsMajda (2015) – a parser generator that produces fast parsers for JavaScript

Many of the PEG parsers above can be integrated with the packrat parsing Ford (2002) to avoid exponential time cost, in the worst case. However, as we expected in Section 3.3, the backtracking activity is very limited in the converted DTDs, and no exponential behavior was observed throughout the experiment. As a result, all of the reported tests were run without any packrat parsing memoization.

5.2 Empirical Study

To begin with, we investigate the characteristics of real-world DTDs. Table 5 shows a summary of the investigated DTD files. The DTD files are randomly collected from the Web in a such way that we can investigate a variety of XML formats, ranging from documents to application data. The columns labeled "ELE" and "ATT", respectively, indicate the number of element declarations and attribute lists in DTDs. The column "#REQ" stands for the maximum occurrence of `#REQUIRED` attributes in a single element. The column "PEG" is the number of productions in the converted PEGs.

According to our empirical investigation, `#REQUIRED` attributes appear at most five times in a single element. This suggests that the use of `#REQUIRED` attributes is relatively rare, compared to `#IMPLIED` attributes. Even in the case of 5 `#REQUIRED` attributes, we need only $120 (= 5!)$ alternations that are still acceptable in size for the generated parsing expressions.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015

Table 5. Real-world DTDs

DTD files	ELE	ATT	#REQ	PEG	DTD files	ELE	ATT	#REQ	PEG
AddressBook.dtd	4	0	0	35	nitf-3-1.dtd	132	132	2	2114
ant141.dtd	214	214	2	2854	ops.dtd	12	1	1	60
ant151.dtd	323	323	1	4226	PCs.dtd	14	0	0	28
anzmeta-1.1.dtd	73	2	0	334	people.dtd	4	1	0	41
ASIX.dtd	18	7	1	142	platform.dtd	3	3	4	53
atributes.dtd	3	4	1	84	plugin.dtd	13	13	5	122
books.dtd	8	8	1	68	render.dtd	17	17	5	139
BrainData.dtd	61	0	0	151	replay.dtd	17	12	0	86
calstblx.dtd	10	10	3	116	retxml_doc.dtd	103	35	3	234
children.dtd	14	0	0	55	schematron.dtd	19	16	2	146
CIM_DTD.dtd	30	12	2	154	structures.dtd	30	23	2	227
client.dtd	18	2	1	72	supp2015.dtd	12	2	1	58
CommonMark.dtd	18	9	2	94	svg11-flat.dtd	80	80	2	1001
content.dtd	6	0	0	40	teiana2.dtd	10	10	2	96
dblp.dtd	37	16	1	234	tsung-1.0.dtd	65	54	4	555
faq.dtd	24	4	1	100	tvschedule.dtd	10	4	1	61
html5.dtd	109	109	0	8471	xhtml1-frameset.dtd	91	91	2	641
JATS.dtd	7	4	0	92	xhtml1-strict.dtd	77	77	2	596
OpenOffice.dtd	40	35	0	187	xmark.dtd				
LMPLCurriculo.dtd	284	242	5	2288	XMLSchema.dtd	26	26	2	220
log4j.dtd	27	17	2	144	xmlspec.dtd	220	192	3	2881
masterdb.html.dtd	7	7	1	57	xslt.dtd	35	34	3	261
metaadms.dtd	10	6	3	82	xv.dtd	16	16	5	131
mondial.dtd	41	21	2	198	Zeerex-2.0.dtd	44	25	2	217

5.3 Performance Study

We will turn to our performance study. We compare the following longstanding and highly-optimized XML validation tools, written in Java or C.

- Xerces-JXerces (2015) – an standard XML parser with DTD validation for Java. We run SAX-based parsing without any DOM generations
- Xerces-C – a C/C++ version of Xerces. We also run an SAX-based parsing without any DOM generations
- Libxml2Libxml2 (2015) – a well-established XML parser library for C/C++. We use the *xmllint* command for DTD validation.

We study synthetic and real-world data sets. For the synthetic data sets, we examine the XMark datasetsSchmidt et al. (2002), a standard benchmark of XML data. Figure 8 shows the performance comparison when we validate the generated 10MB XMark file with xmark.dtd. To compare the same condition, we modify xmark.dtd in a way that the ID/IDREF constraints are unchecked in Xerces and Libxml2. The performance is measured in milliseconds and displayed as their throughputs (KiB/s). Rats! and Mouse shows very poor performance, compared to the XML validators. This is perhaps because these PEG tools are highly optimized for programming language syntax, but perhaps for a large volume of simple data syntaxes. On the contrary, other PEG tools show competitive or even better performance, compared to Xerces and Libxml2.

We will now turn to the scalability. Figure 9 shows the throughput of the DTD validation (KiB/s) by scaling the XMark files from 1MB up to 1000MB. We confirm that PEG-based tools provide both linear time and constant memory consumption, as well as XML validation tools. In addition, this experiment implies that backtracking does not cause any super-linear costs. Indeed, all occurrences of backtracking are localized with a range of XML element names. This creates supporting evidence for the claim that DTD-converted PEGs are safe without any packrat parsing support.

Finally, we examine the validation performance of real-world XML files. The examined XML files are collected from open data repositories. If there is no given DTD, we use a DTD generatorKay (2001) to obtain the DTDs. In this experiment, we only focus on comparing C-implemented PEG tools to Xeares-C, the fastest XML validator in the XMark experiment. Figure 10 shows the performance comparisons in

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015

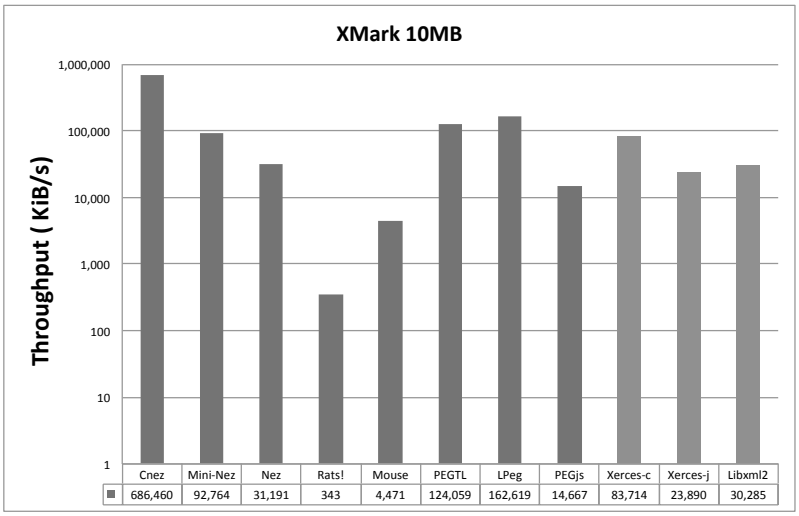

Figure 8. Throughputs in 10MB XMark File

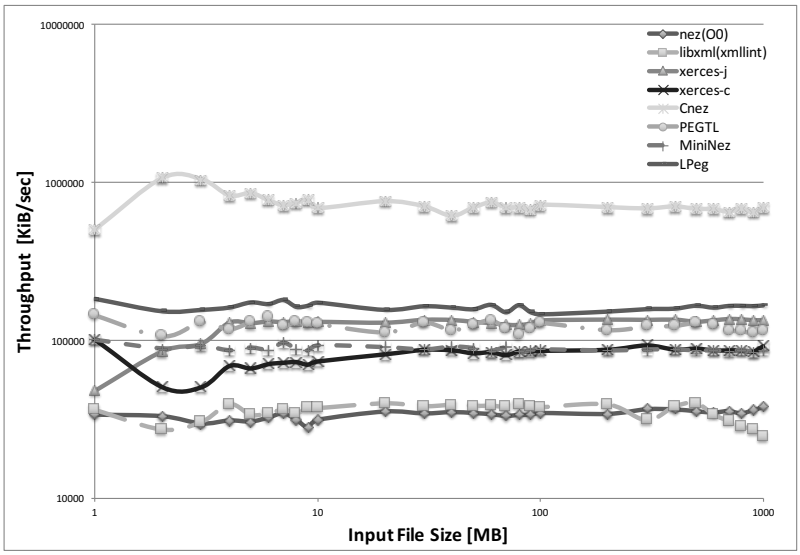

Figure 9. Throughputs in XMark (1MB to 1000MB)

proportion to the validation time of Xerces-C. As with in the XMark experiment, the PEG tools have competitive performance compared to Xerces-C and Libxml2. These results obtained from synthetic and real-world data suggest that DTD-to-PEG conversion is a practical method for implementing an efficient DTD validator.

6 RELATED WORK

Many intensive efforts on XML schema validation have intensively been made in the 2000s since the emergence of the XML standard. Enhanced DTDs and new XML schema languages such as XML Schema and Relax NG have been designed with a regular tree automataMurata et al. (2005). However, implementing tree automata usually requires tree constructions as their input, resulting in linear memory consumption in the size of the XML document. This is not desirable in many XML applications, especially in streaming XML processingMartens et al. (2005); Segoufin and Sirangelo (2006). Accordingly, tackling a constant memory algorithm has become a common research goal. For this purpose, pushdown automata, not tree automata, are generally used to argue about theoretical foundationsPapakonstantinou and Vianu (2000); Segoufin and Vianu (2002); Green et al. (2002); Schwentick (2007). Finally, visibly pushdown

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015

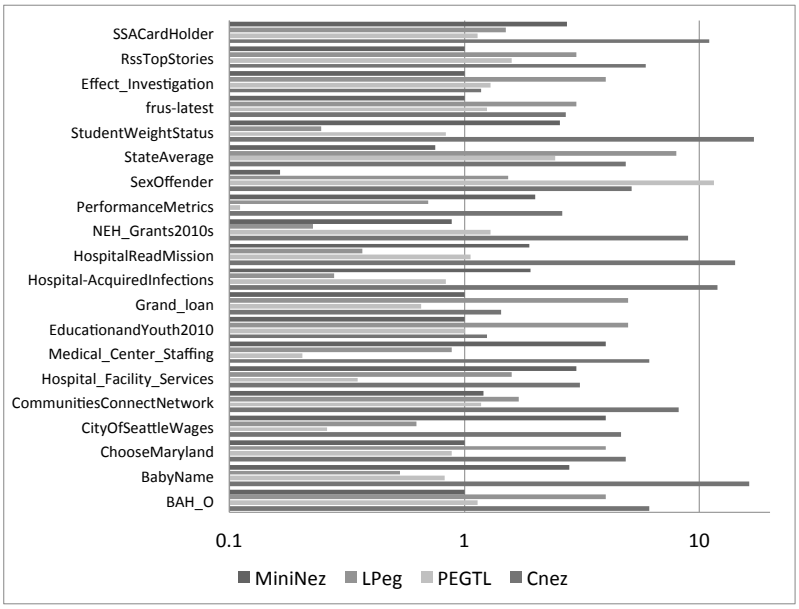

Figure 10. Performance comparison of C-implemented PEG tools to Xerces-C (=1.0)

automata Kumar et al. (2007); Thomo et al. (2008) are newly developed in this context as a robust, traceable class of languages. However, the lack of readily available tools makes it hard to apply these techniques to a validator implementation.

There are many parser generator tools based on a restricted class of context-free grammars. Among them, lex/yacc is longstanding and can produce very efficient parsers. Several researchers have attempted to make use of lex/yacc for XML parsing and validation. As with PEG tools, lex/yacc can also achieve high performance Kostoulas et al. (2006). Unlike PEG tools, however, lex/yacc requires both C code generation and compilation before XML validation. Some of our tested PEG tools, such as Nez and LPeg, run as dynamic parsing, in the same way that Xerces loads DTDs for validation.

To our knowledge, this paper is the first experimental report on the application of PEGs for XML schema validation. PEGs were originally developed to describe programming syntaxes Ford (2004), and are mostly used in the contexts of programming languages Grimm (2006). An interesting exception is LPeg, which was designed for pattern matching tasks, as an alternative to regular expressions Ierusalimschy (2009). This work is largely inspired by LPeg, but we address more nested and complicated patterns such as XML/DTDs.

In this paper, we only focus on the expressiveness of *pure* PEGs without any consideration on extended parsing expressions. Although the pure PEGs are broadly available, they are clearly insufficient for the set membership constraint that appears in XML attributes. We abandon the complete validation of optional attributes to express DTDs with pure PEGs. As with PEGs, the set membership problem appears in contexts of regular expressions, and the *shuffle operator*Ghelli et al. (2008) has successfully been extended. Likewise, the extended PEGs are necessary for complete XML validation but remain an open challenge.

7 CONCLUSION

This paper presents the conversion of document type declaration into parsing expression grammars. Since DTDs are fundamentally restricted to being deterministic regular expressions, PEGs has sufficient capability to recognize all syntactic and structural constraints of DTDs. A major limitation is the uniqueness constraint of ID/IDREF attributes, which is beyond the expressive power of PEGs. In addition, an unordered sequence of attributes requires a approximated conversion, at the cost of allowing duplicated optional attributes. We demonstrate that DTD-converted PEGs achieve competitive performances, compared to standard XML validators. In future work, we will investigate include an extension of PEGs to apply other schema validation tasks.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015

REFERENCES

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cowan, J. (2006). Extensible markup language (xml) 1.1 (second edition). http://www.w3.org/TR/xml11/.

Brüggemann-Klein, A. and Wood, D. (1998). One-unambiguous regular languages. *Inf. Comput.*, 142(2):182–206.

Ford, B. (2002). Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 36–47, New York, NY, USA. ACM.

Ford, B. (2004). Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA. ACM.

Ford, B. (2014). The packrat parsing and parsing expression grammars page. http://bford.info/packrat/.

Ghelli, G., Colazzo, D., and Sartiani, C. (2008). Linear time membership in a class of regular expressions with interleaving and counting. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM '08, pages 389–398, New York, NY, USA. ACM.

Green, T. J., Miklau, G., Onizuka, M., and Suciu, D. (2002). Processing xml streams with deterministic automata. In *Proceedings of the 9th International Conference on Database Theory*, ICDT '03, pages 173–189, London, UK, UK. Springer-Verlag.

Grimm, R. (2006). Better extensibility through modular syntax. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 38–51, New York, NY, USA. ACM.

Hirsch, C. and Frey, D. (2014). Parsing expression grammar template library. https://code.google.com/p/pegtl/.

Honda, S. and Kuramitsu, K. (2016). Implementing a small parsing virtual machine for embedded systems. Technical report, Yokohama National University.

Hosoya, H. and Murata, M. (2003). Boolean operations for attribute-element constraints. In *Proceedings of the 8th International Conference on Implementation and Application of Automata*, CIAA'03, pages 201–212, Berlin, Heidelberg. Springer-Verlag.

Ierusalimschy, R. (2009). A text pattern-matching tool based on parsing expression grammars. *Softw. Pract. Exper.*, 39(3):221–258.

Kay, M. H. (2001). Dtdgenerator - a tool to generate xml dtds - saxon. http://saxon.sourceforge.net/dtdgen.html.

Kostoulas, M. G., Matsa, M., Mendelsohn, N., Perkins, E., Heifets, A., and Mercaldi, M. (2006). Xml screamer: An integrated approach to high performance xml parsing, validation and deserialization. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 93–102, New York, NY, USA. ACM.

Kumar, V., Madhusudan, P., and Viswanathan, M. (2007). Visibly pushdown automata for streaming xml. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 1053–1062, New York, NY, USA. ACM.

Kuramitsu, K. (2015). Packrat parsing with elastic sliding window. *Journal of Infomration Processing*, 23(4):505–512.

Kuramitsu, K. (2016). Fast, flexible, and declarative consturction of abstract syntax trees with PEGs. *Journal of Infomration Processing*, 24(1):(to appear).

Libxml2 (2015). Validation and dtds. http://xmlsoft.org/xmldtd.html.

Majda, D. (2015). Peg.js - parser generator for javascript.

Martens, W., Neven, F., and Schwentick, T. (2005). Which xml schemas admit 1-pass preorder typing? In *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, pages 68–82, Berlin, Heidelberg. Springer-Verlag.

Medeiros, S. and Ierusalimschy, R. (2008). A parsing machine for pegs. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 2:1–2:12, New York, NY, USA. ACM.

Medeiros, S., Mascarenhas, F., and Ierusalimschy, R. (2014). From regexes to parsing expression grammars. *Sci. Comput. Program.*, 93:3–18.

Murata, M., Lee, D., Mani, M., and Kawaguchi, K. (2005). Taxonomy of xml schema languages using formal language theory. *ACM Trans. Internet Technol.*, 5(4):660–704.

Papakonstantinou, Y. and Vianu, V. (2000). Dtd inference for views of xml data. In *Proceedings of the*

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015

Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '00, pages 35–46, New York, NY, USA. ACM.

Redziejowski, R. R. (2007). Parsing expression grammar as a primitive recursive-descent parser with backtracking. *Fundam. Inf.*, 79(3-4):513–524.

Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manolescu, I., and Busse, R. (2002). Xmark: A benchmark for xml data management. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 974–985. VLDB Endowment.

Schwentick, T. (2007). Automata for xml—a survey. *J. Comput. Syst. Sci.*, 73(3):289–315.

Segoufin, L. and Sirangelo, C. (2006). Constant-memory validation of streaming xml documents against dtds. In *Proceedings of the 11th International Conference on Database Theory*, ICDT'07, pages 299–313, Berlin, Heidelberg. Springer-Verlag.

Segoufin, L. and Vianu, V. (2002). Validating streaming xml documents. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 53–64, New York, NY, USA. ACM.

Thomo, A., Venkatesh, S., and Ye, Y. Y. (2008). Visibly pushdown transducers for approximate validation of streaming xml. In *Proceedings of the 5th International Conference on Foundations of Information and Knowledge Systems*, FoIKS'08, pages 219–238, Berlin, Heidelberg. Springer-Verlag.

Thompson, H. S., Beech, D., Maloney, M., and Mendelsohn, N. (2004). Xml schema part 1: Structures second edition. http://www.w3.org/TR/xmlschema-1/.

Xerces (2015). Apache xerces project. http://xerces.apache.org/.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1503v1 | CC-BY 4.0 Open Access | rec: 12 Nov 2015, publ: 12 Nov 2015