Annals of Software Engineering 13, 357–379, 2002

Foundations of Fast Communication via XML

WELF M. LÖWE, MARKUS L. NOGA and THILO S. GAUL loewe@ipd.info.uni-karlsruhe.de
Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, Postfach 6980, 76128 Karlsruhe, Germany

Abstract. Communication with XML often involves pre-agreed document types. In this paper, we propose an offline parser generation approach to enhance online processing performance for documents conforming to a given DTD. Our examination of DTDs and the languages they define demonstrates the existence of ambiguities. We present an algorithm that maps DTDs to deterministic context-free grammars defining the same languages. We prove the grammars to be *LL*(1) and *LALR*(1), making them suitable for standard parser generators. Our experiments show the superior performance of generated optimized parsers. Our results generalize from DTDs to XML schema specifications with certain restrictions, most notably the absence of namespaces, which exceed the scope of context-free grammars.

1. Introduction

The Extensible Markup Language (XML) [W3C 1998] provides for logical markup: documents are tagged according to their content structure, not their visual appearance in specific presentation media. Sets of documents can be specified with Document Type Definitions (DTDs). A given document can be validated against a given DTD.

One area of application for XML is web serving. Content originating with diverse sources such as static documents or database queries is represented in XML. These documents are mapped to presentation formats like HTML through Extensible Stylesheet Language Transformation (XSLT) scripts or cascades thereof. Although certain generalizations are possible, a given script usually works only with documents conforming to a specific DTD. Parsing the document may account for half the processing time or more, so performance matters when serving dynamic documents.

Over the past years, XML also became popular for general-purpose platform-independent data exchange. When connecting software components, XML processing layers compete with middleware architectures such as Corba [OMG 2002] and (D)COM [Microsoft 2002]. In this arena, performance is crucial, and the DTDs are always available at compile time.

Most available XML parsers are generic: they read arbitrary well-formed XML, analyze the required DTD and validate the document against it. Along with many others, the parsers provided by the Apache project [Apache 2002], James Clark [Clark 2000] and IBM [IBM AlphaWorks 2001] fall into this category. As generic parsers cannot use DTDs to optimize parsing, faster solutions are possible.

We take the approach of specific parsing, where a parser applies to documents conforming to a given DTD only. DTDs may change rapidly in software evolution, so

manual implementation of specific parsers is a non-option. Compiler construction practice shows deterministic parsers to be manifestly faster than their ambiguous equivalents. Thus, we focus on the automatic generation of deterministic parsers.

At first glance, DTDs are ill-suited to this technique: while the XML specification [W3C 1998] states deterministic ones as desirable in appendix E, the entire appendix is non-normative. Many real-world DTDs, including some published in W3C documents, are, in fact, ambiguous.

For regular languages, the existence of deterministic grammars and corresponding automata is a well-known result from formal language theory. Due to its matching opening and closing tags, which correspond to nested parentheses in programming languages, XML is not regular. It belongs to the larger class of context-free languages. In general, the question if there is an equivalent deterministic grammar for a given context free grammar is not decidable.

This paper proves the existence of equivalent deterministic grammars for DTDs, i.e., for a subset of the context-free languages including XML. The proof is constructive and defines a mapping algorithm. The generated grammars are both $LL(1)$ and $LALR(1)$. Thus, standard parser generators for $LL(k)$ grammars or subsets thereof, such as ell [Grosch 1989; Vielsack 1988] and javacc [WebGain 2002] can be employed to generate efficient parsers, as well as tools for $LALR(1)$ grammars like yacc [Johnson 1975], bison [Donelly and Stallmann 1988] and lalr [Grosch 1989; Vielsack 1988]. Our experiments show them to outperform generic parsers by an order of magnitude.

The body of this paper is organized as follows: section 2 defines notions and cites results basic to our approach. Section 3 demonstrates DTD ambiguities and shows the transformation of a DTD into a context-free grammar. The generated grammars are proven to be $LL(1)$ and $LALR(1)$. Section 4 generalizes these results to some extensions of XML, that is, XML Schema specifications and namespaces. Section 5 compares the performance of parsers generated with our techniques to generic ones. We examine related work in section 6. Section 7 concludes the paper with directions for future work. The appendix contains additional definitions and a larger example.

2. Basic definitions and results

The following definitions and theorems can be found in every compiler construction or formal languages textbook. They are included here to introduce notational conventions. For details, we refer to [Apache 2002; W3C 2001]. We assume the reader is familiar with the notions of substitution and rewrite systems. Definitions of the notions of a formal language, a grammar, deterministic and ambiguous grammar, regular expressions and languages, and context-free grammars and languages can be found in the appendix A.

The next two theorems state that it is trivial to find a deterministic grammar for a regular language, but hard for a context-free language.

Theorem 1 (Deterministic regular grammars). There is an algorithm to find a deterministic grammar for every regular language.

Theorem 2 (Deterministic context-free grammars). It is not decidable whether a context-free language can be defined by a deterministic grammar.

Unfortunately, XML is not regular. To establish our results, we need the $SLL(1)$ and $SLR(1)$ grammar classes, which, in turn, require a basic understanding of deterministic *LL* and *LR* parsing strategies.

Both variants operate on an input stream of terminal symbols and an analysis stack containing terminal and non-terminal symbols. Deterministic *LL* parsers operate top-down. They derive words from the starting symbol Z, which is initially on the analysis stack. In each step, an *LL* parser distinguishes three cases:

(1) If the top of the analysis stack is a terminal symbol t and the current input symbol is t, the top of the analysis stack is removed and the next input symbol is read.

(2) If the top of the analysis stack is a terminal symbol t and the current input symbol is $t' \neq t$, parsing stops with an error (non-deterministic parsers back-track in this case).

(3) If the top of the analysis stack is a non-terminal symbol N, it is replaced by the right-hand side of a production with left-hand side N.

Analysis terminates successfully iff both input stream and analysis stack are empty.

Deterministic *LR* parsers work bottom-up. They start with an empty analysis stack and reduce the input word stepwise to the starting symbol Z. In each step, *LR* parsers either

(I) shift terminal symbols onto the stack; or

(II) reduce the topmost symbols on the stack to a terminal N if there is a production $N \to s$ whose right-hand side equals the topmost symbols reversed.

Analysis terminates successfully iff the input stream is empty and the analysis stack contains only Z.

For both types of parser, choosing the correct action is critical. *LL* parsers must load the correct right-hand side of a production in step (3) in the presence of multiple alternatives. *LR* parsers must decide whether to shift or to reduce if reduction is possible (shift-reduce conflict) and whether to reduce to N or N' if multiple alternatives are applicable (reduce-reduce conflict).

It is desirable to define a language with a grammar that allows the efficient resolution of these conflicts, to allow efficient parsing without backtracking. Two examples of such grammar classes are $SLL(k)$ and $LR(0)$, whose original definitions can be found in [Rosenkrantz and Stearns 1969] and [DeRemer 1971].

Definition 1 ($SLL(k)$ grammar). Let G be a context-free grammar. G is $SLL(k)$ iff the next k input symbols are always sufficient for an *LL* parser to choose the correct right-hand side to load in step (3).

Definition 2 (*LR*(0) grammar). Let G be a context-free grammar. G is *LR*(0) iff regular matches on the analysis stack state are always sufficient for an *LR* parser to correctly resolve all shift-reduce and reduce-reduce conflicts.

To decide whether a grammar is *SLL*(k), we define the k-head of a word w, denoted by $k : w$, the set of all k-heads of terminals of a word $FIRST_k(w)$ and the set of all k-heads of terminals following a word $FOLLOW_k(w)$. Given a grammar $G = (T, N, P, Z)$ with terminals T, non-terminals N, productions P and starting symbol Z.

Definition 3 (The k-head and first and follow sets). Let # be a symbol, $\# \notin T$, which marks the end of a word. Let $w \in (T \cup N)^*$, let $\Rightarrow^*$ denote a derivation of arbitrary length. The k-head of a word w is defined as

$$\begin{aligned} k : w = a, &\quad \text{iff} \quad w = ay \wedge |a| = k \quad \text{and} \\ k : w = w\#, &\quad \text{iff} \quad |w| < k. \end{aligned}$$

The first and follow sets, respectively, are defined by:

$$\begin{aligned} FIRST_k(w) &= \{r \mid \exists v \in T^* \text{ with } w \Rightarrow^* v \wedge r = k : v\}, \\ FOLLOW_k(w) &= \{r \mid \exists v \in (T \cup N)^* \text{ with } Z \Rightarrow^* uwv \wedge r \in FIRST_k(v)\}. \end{aligned}$$

Remark. We omit the subscript k for $k = 1$.

Definition 4 (*SLR*(k) grammar). Let G be a context-free grammar. G is *SLR*(k) iff

- for all pairs of productions $N_x \to x$ and $N_y \to x$ causing a reduce-reduce conflict in an *LR*(0) parser, $FOLLOW_k(N_x) \cap FOLLOW_k(N_y) = \emptyset$, i.e., the next k input symbols determine whether to reduce to N_x or to N_y.
- for all pairs of productions $N_x \to x$ and $N_y \to xy$ causing a shift-reduce conflict in an *LR*(0) parser, $FOLLOW_k(N_x) \cap FIRST_k(yFOLLOW_k(N_y)) = \emptyset$, i.e., the next k input symbols determine whether to reduce to N_x or to shift the symbols belonging to y.

Remark. *SLR*(k) grammars allow to resolve the remaining conflicts of an *LR*(0) parser using the next k input symbols.

Using the first and follow sets, the question whether a given grammar is *SLL*(k) can be decided with the following theorems.

Theorem 3 (*SLL*(k) property). A grammar is *SLL*(k) iff for all pairs of productions $N \to x$ and $N \to y$ with $x \neq y$:

$$FIRST_k\big(xFOLLOW_k(N)\big) \cap FIRST_k\big(yFOLLOW_k(N)\big) = \emptyset.$$

From a practical viewpoint, narrow definitions of language classes are less important than the potential to apply existing tools. Although $SLL(k)$ and $SLR(1)$ are stricter language classes, the applicability of $LL(k)$ and $LALR(1)$ parser generators is the dominant concern. The exact definition of look-ahead-$LR(1)$ – $LALR(1)$ – grammars and parser generation algorithms are not essential to this paper, but covered in [DeRemer 1971] or [Waite and Goos 1985]. The following theorems deliver the results of interest.

Theorem 4 ($SLL(k)$ and $LL(k)$ grammars).

$$SLL(k) \subset LL(k), \quad SLL(1) = LL(1).$$

Theorem 5 ($LR(k)$, $SLR(k)$ and $LALR(k)$ grammars).

$$LR(0) \subset SLR(k) \subset LALR(k) \subset LR(k).$$

3. DTDs and grammars

This section briefly examines well-formed XML and DTDs with their components. Examples for ambiguities in DTDs are given. Algorithms mapping individual DTD components and entire DTDs to equivalent grammars are described.

3.1. XML and DTDs

XML documents consist of elements, attributes and text. Text is a character sequence in the specified coding system, which must not contain the < character used in the representation of elements. An attribute is a triple of attribute name, = symbol and quoted attribute value (e.g., `a="foo"`), separated by arbitrary whitespace. We omit whitespace in our presentation. Elements are defined recursively. They consist of the following sequence:

(1) A start tag containing the element name (e.g., `<x`).

(2) A sequence of attributes, followed by the > symbol (e.g., `a="foo" b="bar">`).

(3) A sequence of elements or text.

(4) An end tag repeating the element name (e.g., `</x>`).

The sequences in (2) must not contain multiple occurrences of the same attribute name. In general, (3) may also contain comments, entities, processing instructions and unparsed data sections. We ignore them to simplify this presentation, as they are simple regular tokens. For empty elements the sequence in (3) has no entries. They can be represented using a short-cut that closes the element after the attribute definitions with '`/>`' (e.g., `<x a="foo" b="bar" />`).

DTDs define the structure of sets of documents. In this paper, we assume that all implicit and external components are inlined. Then, DTDs contain two kinds of

definition components, element definitions and attribute definitions. Element definitions have the form:

```
<!ELEMENT x (content_model_x)>
```

where `x` is the name of the element to be defined, and `content_model_x` is either a regular expression over element names and `#PCDATA` or `#EMPTY`, which denotes a regular expression accepting only the empty string. `#PCDATA` is a placeholder for possibly empty text. Regular expressions may employ the well-known operators iteration (`*` and `+`), alternative choice (`'|'`), option (`?`) and sequence (`,`), as well as grouping parentheses. Attribute definitions take the form:

```
<!ATTRIBUTE x a t o>
```

where `x` is the element name for which attribute `a` is defined as type `t`, with lower and upper bounds $l, u \in \{0, 1\}$ on its occurrence given in `o`. Types include character data, enumerations and global identifiers and references, i.e., `ID` and `IDREF`. As global references are non-context-free, our approach cannot accelerate their processing. Their implementation with standard techniques is orthogonal to our approach. Without loss of generality, we ignore that DTDs also allow multiple attributes of an element to be specified in a single definition. In XML documents conforming to the DTD, an element's attributes may appear in any order.

3.2. Ambiguous DTDs

Although a DTD defines the structure of documents concisely, the derivation of an XML document conforming to this DTD may be ambiguous. We give three examples of ambiguities in content models.

Example 1 (An ambiguous content model). According to the definition,

```
<!ELEMENT a (#PCDATA | b)*>
```

an `a` may contain an arbitrary sequence of `#PCDATA` and `b` children. `#PCDATA` is, in turn, a possibly empty string. Therefore, this XML fragment cannot be parsed deterministically:

```
<a></a>
```

It may be interpreted as an empty iteration of `(#PCDATA | b)*` or as an empty `#PCDATA` element.

Example 2 (An ambiguous content model for complex types). This is an excerpt of a DTD in an XML Schema recommendation [W3C 2001]. The ambiguity in the definition

```
<!ELEMENT %complexType; ((%annotation;)?,
                          ((%facet;)*|
                             ((%element;| %mgs; | %group; | %any;)*,
                               (%attribute;| %attributeGroup;)*,
                               (%anyAttribute;)?
                             )
                          )
                        )>
```

is due to possibly empty iterations composed by alternative and sequence operators. A simplification plainly shows the problem:

```
<!ELEMENT a (x?, (y* | z*))>
```

The empty element

```
<a></a>
```

may be interpreted as the missing x and an empty iteration of y or, alternatively, missing x and an empty iteration of z. This content model can be transformed into a deterministic definition:

```
<!ELEMENT a ((x,(y+|z+)?) | y+ | z+)?>
```

Actually, there are more compact regular expressions for the above language. We used the one directly derived from the deterministic minimum acceptor constructed from the ambiguous expression, cf. appendix B.

Apparently, deterministic definitions tend to become larger than ambiguous ones. As ambiguous DTDs are permitted and the human reader has no problem to understand the structures they define, ambiguous DTDs will occur in practice.

For practical document processing, a tree representation must offer highly selective access operations. Informally, selectivity is the fraction of nodes returned by an operation that we are actually interested in. For low selectivity, the burden of eliminating undesired results resides with the user. The more selective an interface is, the smaller is this extra effort. Standard access operations are based on position and name providing a quite low selectivity at content model level. This could be increased by giving explicit access to the parse tree. Then, an ambiguous content model not only affects the parsing speed but also its very result.

Example 3. The following DTD fragment is ambiguous:

```
<!ELEMENT a (x | y | (x, y))*>
```

Consider the document:

```
<a><x/><y/><x/><y/><x/><y/>...</a>
```

Each pair <x/><y/> can either be interpreted as an iteration over the first two alternatives or as the third alternative. A highly selective request like: “give me the first occurrence of the third alternative” is impossible.

In terms of efficiency, the situation looks bleaker still: even deterministic content models cannot guarantee that documents can be parsed without look-ahead or backtracking:

Example 4. The following DTD fragment is deterministic:

```
<!ELEMENT a ((x|y)*, x, (x|y))>
```

Try parsing this document fragment:

```
<a><x/><x/><x/><x/><x/><x/>...</a>
```

Although the parse tree for this fragment (and all other legal ones) is unique, the derivation cannot be computed just by stepping through the content model and checking for the right input. Without a look-ahead for the input `</a>`, it is not decidable whether an `<x/>` reduces to the term `(x|y)*` or to term `x` in the content model. This content model has no equivalent one without this negative property and all content models of the form

```
<!ELEMENT a ((x|y)*, x, (x|y)(x|y)(x|y)(x|y)...)>
```

have the same problem.

The example shows: without leaving the world of regular expressions, we cannot avoid a look-ahead or back-tracing and we cannot limit the look-ahead. Moreover, individual deterministic content models do not imply a deterministic overall grammar. We will show in the next subsections that for DTDs, they do.

3.3. Grammars for DTD components

Ambiguities in DTDs are not a property of the languages they define. There is a deterministic context-free language for each DTD and start element defining the same language, as the following two subsections will show. We start with

Lemma 1 (Content models and the *SLL*(1) property). For every content model, there is an *SLL*(1) grammar defining the same language.

Proof. We define an algorithm mapping a content model to a context-free grammar. We show the grammar to define the same language and prove membership in *SLL*(1).

Let S be the set of all element names in a DTD and $S' = S \cup \{\mathit{string}\}$, where *string* must not contain <. Replace each occurrence of `#PCDATA` by *string*? in the element definition expressions. Obviously, each of the modified expressions is a regular expression over S' per definition 8 in the appendix. We construct a minimal deterministic acceptor from the regular expression with standard formal language techniques:

(1) Construct an ambiguous acceptor from the regular expression.

(2) Make the ambiguous acceptor deterministic using the standard power set construction.

(3) Minimize the deterministic acceptor using the standard equivalence class construction.

Then, we generate the grammar directly from this acceptor:

(1) For each state t of the minimal deterministic acceptor, we generate a unique non-terminal $N(t)$.

(2) If t is an accepting state, we generate the production $N(t) \to \varepsilon$.

(3) If there is a transition from t to t' accepting symbol $s \in S'$, we generate the production $N(t) \to sN(t')$.

All above transformations are standard, see [Waite and Goos 1985], and the constructed grammar is proven to define the same language as the original regular expression. It remains to show the resulting grammar to be $SLL(1)$. We use theorem 3. Suppose there are two productions $N(t) \to x$ and $N(t) \to y$ with $x \neq y$. Let $N(t)$ corresponds to state t. According to our construction, we distinguish two cases:

(1) $x = aN(t')$ and $y = bN(t'')$, with $a, b \in S'$ and non-terminals $N(t')$ and $N(t'')$. As the acceptor is deterministic, it follows $a \neq b$ and

$$FIRST\big(xFOLLOW\big(N(t)\big)\big) \cap FIRST\big(yFOLLOW\big(N(t)\big)\big) = \{a\} \cap \{b\} = \emptyset;$$

(2) $x = aN(t')$ and $y = \varepsilon$. By construction, the follow set of each non-terminal is the virtual end-of-sentence marker '#'. Thus

$$FIRST\big(xFOLLOW\big(N(t)\big)\big) \cap FIRST\big(yFOLLOW\big(N(t)\big)\big) = \{a\} \cap \{\#\} = \emptyset. \quad \square$$

Using this result, a corresponding lemma for attribute definitions can be established.

Lemma 2 (Attribute definitions and the $SLL(1)$ property). For every set of attribute definitions of an element, there is an $SLL(1)$ grammar defining the same language.

Proof. Let R_e be the finite set of attribute definitions for element e in the DTD. In conforming XML documents, instances of e may contain at most one instance of every attribute in R_e, so every legal attribute instance sequence is finite. As there are only a finite number of permutations, the set of legal attribute instance sequences is also finite. Individual attribute instances are regular as they consist of three regular parts: the regular attribute name, the = character and the regular attribute value string.

Finite sets of finite sequences of regular components are regular, so we obtain an $SLL(1)$ grammar by applying the algorithm in lemma 1 to an appropriate regular expression. $\square$

Note that not all regular grammars are $SLL(1)$. The regular productions $S \to aA$ and $S \to aB$, e.g., have the same first set. The construction from the deterministic regular acceptor guarantees that such ambiguities do not occur.

The generated grammars capture all aspects of attribute occurrences, but due to their exponential size, they are mostly theoretical in nature. Practical implementations are advised to use the Kleene closure of R_e instead and limit occurrences through non-grammatical means. However, these results pave the way for the synthesis of grammars for DTD languages in the following section. According to the transformations defined above, we obtain deterministic grammars for content models and attributes. Their productions have the form $N \to \varepsilon$ and $N' \to sN''$, respectively, where $s \in S'$ are terminal symbols and N, N' and N'' are non-terminal symbols.

Lemma 3 (Content models, attribute definitions and the $SLR(1)$ property). Every content model as well as every set of attribute definitions of an element has a corresponding $SLR(1)$ grammar defining the same language.

Proof. Since the content model as well as the attribute definitions can be transformed into a grammar of the above form, we can prove the lemma by showing that deterministic grammars of this form are $SLR(1)$ in general. We sketch an $LR(0)$ parsing algorithm, which is an even stronger restriction.

First, we shift the entire word into the analysis stack. Additionally, we perform the state transitions of the deterministic acceptor the grammar is generated from (according to the proof of lemma 1). If this does not lead to a final state, the sentence does not belong to the language.

(1) Without loss of generality, let t be the final state of the acceptor. We reduce with production $N(t) \to \varepsilon$.

(2) Perform (3) until the start symbol is the only symbol of the analysis stack.

(3) Without loss of generality, let $wsN(t)$, $w \in T^*, s \in T$ and $N(t) \in N$, be the content of the analysis stack. Then, there must be a production $N(t') \to sN(t) \in P$. Furthermore, accepting the word w with the deterministic acceptor ends in the state t'. We reduce with this production.

As no look-ahead is performed, the parser is $LR(0)$ and thus $SLR(1)$. □

Remark. It is known that every regular language allows for $LR(0)$ parsing. We included a constructive proof to prepare our main results in the next section as a generalization of this algorithm.

3.4. Grammars for entire DTDs

A DTD D and a root element Z together define a language $L_{D,Z}$. This subsection defines algorithms mapping a DTD D and a root element Z to a grammar G with $L(G) = L_{D,Z}$.

Furthermore, we prove that G is both, $LL(1)$ and $LALR(1)$. Due to theorem 4 and theorem 5, it is sufficient to show that G is $SLL(1)$ and $SLR(1)$, respectively. Proceeding from the grammars for DTD components derived in the previous section, we are now ready to generate useful grammars for a language $L_{D,Z}$ defined by a DTD D and a starting element Z. In the following, we denote starting and closing tags of an element with name "`element`" by $element_[$ and $element_]$, respectively: $element_[$ = "`<element`" and $element_]$ = "`</element>`".

Lemma 4 (Context-free grammars and DTDs). There is a context-free grammar for every pair of DTD D and starting element Z defining the same language $L(G) = L_{D,Z}$.

Proof. We give an algorithm to construct a such a context-free grammar $G = (T, N, P, S)$ with $L(G) = L_{D,Z}$.

Let T be a set of terminal symbols, including *string*, *quotes*, = , > , and /> . Additionally, T contains pairs of symbols $element_[$ and $element_]$, for every element name "`element`" in the DTD, and a symbol for every attribute name "`attribute`" in the DTD. They are assumed to originate with a standard longest-match regular scanner.

Let $C = \{C_x, C_y, \ldots\}$ be the set of grammar productions for the content models of elements named "`x`", "`y`", ... resulting from the construction from lemma 1. Let $A = \{A_x, A_y, \ldots\}$ be the corresponding set of grammar productions for attributes of elements named "`x`", "`y`", ... from lemma 2. Without loss of generality, we assume all non-terminals of productions in C and A to be disjoint; the sets N_C and N_A denote the union of the element and the attribute grammars, respectively. Let $N_E = \{N_x, N_y, \ldots\}$ and $B_E = \{B_x, B_y, \ldots\}$ be sets of non-terminal symbols, pairwise disjoint and disjoint form those in N_C and N_A and corresponding to elements named "`x`", "`y`", ... We then define $N = N_E \cup B_E \cup N_C \cup N_A$.

Except for the symbol *string*, we replace all terminals x in productions of C by non-terminals N_x and denote the set of transformed productions by $K = \{K_x, K_y, \ldots\}$. (Remember: the terminals from productions in C are element names.) We introduce additional productions E_x for each element name `x`:

(1) $N_x \rightarrow x_[A_x B_x$

(2) $B_x \rightarrow >K_x x_]$

For all content model productions C_x with $\varepsilon \in L(C_x)$, we additionally include this production:

(3) $B_x \rightarrow />$

Let $E = \{E_x, E_y, \ldots\}$ be the set of grammar productions as defined above for elements named "`x`", "`y`", ... We set $P = K \cup A \cup E$. Finally, we set $S = N_z$ for the starting element Z, which completes the grammar construction.

By construction, $L(G) = L_{D,Z}$. As the left sides of all productions $p \in P$ consist of but one non-terminal, G is context free. □

In the appendix B, the above transformations are demonstrated on an example. Now, we are ready to prove our main results as additional properties of the constructed grammar.

Theorem 6 (*SLL*(1) grammars and DTDs). There is an *SLL*(1) grammar for every pair of DTD D and starting element Z defining the same language $L_{D,Z}$.

Proof. We prove that the theorem holds for the grammars G generated by the algorithm in lemma 4. We show the *SLL*(1)-property for all pairs of productions $N \to v$ and $N \to w, v \neq w$.

By construction, the non-terminals from the productions in C (generating the content model) and in A (generating the attribute definitions) are pairwise disjoint, disjoint from each other and from all other non-terminals. Hence, lemmas 1 and 2 continue to hold in the context of the grammar G. There is only one more case: productions $B_x \to >K_x x_]$ and $B_x \to />$ have the same left-hand sides. However, the right-hand sides start with different terminals (> vs. />), so the *SLL*(1) property holds for all productions. □

Theorem 6 shows how to generate deterministic *SLL*(1) grammars for XML documents conforming to a specific DTD. The same problem is not computable for context-free languages in general. The class of recursive descent parsers corresponding to *SLL*(1) grammars is one of the fastest known for context-free languages. As the requirement for deterministic element definitions in the XML 1.0 Specification is non-normative, this result is practically relevant – highly performant parsers may be generated even from ambiguous DTDs.

Top down parsing is quite intuitive and allows direct implementations as well as generation from an LL grammar using tools like javacc or ell. However, yacc, bison, lalr and many other parser generators only accept the more powerful *LR* grammars, actually *LALR*(1) grammars. We therefore establish

Theorem 7 (*LALR*(1) grammars and DTDs). There is an *LALR*(1) grammar for every pair of DTD D and starting element Z defining the same language $L_{D,Z}$.

Proof. We prove that the grammar generated by the construction algorithm given in the proof of lemma 4 is *LALR*(1). Actually, we prove the stronger *SLR*(1) property for these grammars. Let F_x be the deterministic acceptor for the content model of element x as generated in the proof of lemma 1.

Central to parsing is the following procedure analyzing one complete element. This procedure is initially called for the top element Z and, recursively, whenever an element start tag appears in the input:

(1) Shift symbol $x_[$. Set the current state to the initial state of F_x.

(2) Analyze attributes of element x using productions A_x according to the *SLR*(1) algorithm given in lemma 3. Instead of shifting the whole input string, shift only until

the next input symbol is either > or />. Reduce the attributes to A_x. Return with error if reduction is not possible or neither > nor /> is detected in the input.

(3) Shift the next symbol.

(a) If it is />, reduce it to B_x if such a production exists (otherwise return with error). The top of the stack is now $x_{[}A_xB_x$. Reduce this to N_x. Return N_x.

(b) If it is >, proceed with (4).

(4) Test the next symbol.

(a) If it is an opening tag, recursively call the corresponding procedure. Then proceed with (5).

(b) If it is *string*, shift it and proceed with (5).

(c) If it is $x_{]}$, proceed with (6).

(d) Otherwise, return with error.

(5) Perform a transition in the deterministic acceptor F_x from the current state. According to step (4), the top of the analysis stack must be one of:

(a) N_y, then perform the transition for y (step (4a) parsed element y before).

(b) *string*, then perform the transition for *string* (step (4b) parsed a *string* before).

Proceed with (4), or return with error if no transition is applicable.

(6) If the current state of F_x is

(a) an accepting state, the top of the stack is a sequence of non-terminal symbols $N_aN_bN_c\ldots$, where $abc\ldots$ is a word accepted by F_x. In other words, it is a word of the content model language $L(C_x)$ of element x. In analogy to step (3) in the algorithm from lemma 3, reduce the analysis stack top to K_x.

(b) Otherwise, an error is detected.

(7) Finally, shift $x_{]}$ and reduce $K_xx_{]}$ to B_x. The top of the stack is now $x_{[}A_xB_x$. Reduce this to N_x and return N_x.

The procedure is an *SLR*(1) parsing schema as only the next input symbol decides whether a shift or a reduce is executed, cf. in steps (2) and (4). Reduce-reduce conflicts do not occur. □

Remark. The above algorithm schema differs from the algorithms of *LR* parser generated by standard tools. It is not chosen for its efficiency. Instead, we designed the schema to simplify the proof of the *SLR*(1) property.

4. Extensions to XML 1.0

A number of extensions to XML 1.0 have been proposed by the W3C. One of them, XML Schema [W3C 2001], is a new language to specify document types. Another one, XML Namespaces [W3C 1999], introduces a mechanism to prevent name collisions independent of a specification language. Both require changes to the grammars developed in the previous sections.

4.1. XML Schema

In DTDs, attribute names are bound to types in an element context only, but element names are bound to a single type for the entire document. XML Schema weakens the latter link by introducing the notion of types, which are defined separately from elements.

There are two varieties of types, simple and complex ones. Simple types pertain to attribute values and text fragments. They supersede the attribute types and `#PCDATA` sections of DTDs. Still, attributes remain delimited by quotes and text sections by elements, so simple types do not threaten their regularity. As they do not pertain to parsing, we ignore them in the remainder of this paper.

Complex types apply to elements. They consist of a list of applicable attributes and a regular content model. Due to the latter, the ambiguities found in DTDs also apply to schemas. Element names remain unique within a complex type context, but the corresponding types are determined by pairs of name and context, as demonstrated in example 5.

Example 5. An XML Schema fragment:

```
<element name="x" type="A"/>

<complexType name="A">
  <sequence minOccurs="0" maxOccurs="unbounded">
    <element name="x" type="B"/>
  </sequence>
</complexType>

<complexType name="B">
  <choice>
    <element name="x" type="A"/>
    <element name="y" type="B"/>
  </choice>
</complexType>
```

The above definitions show that the same element name x is bound to different types depending on its context: a top-level element x is of type A, i.e., it consists of a sequence of x elements of arbitrary length. Those child elements conform to a type B

different from A. They, in turn, contain either an element x of type A, or an element y of type B.

XML Schema also provides for element and attribute wildcards, type overrides from instance documents and namespaces. Wildcards denote a choice from a given set, or even an arbitrary choice. The first kind can be transformed into a choice statement respectively a list of attribute statements, while the second truly extends the scope of the definable languages. Type overrides allow instance documents to supersede within certain bounds the type defined for an element in the Schema. The nature of the bounds makes overrides tractable in practice, but their theoretical treatment remains complicated. We chose not to discuss them in further detail. Namespaces are also a more complex issue, which we discuss below.

4.2. Namespaces

Namespaces were introduced in XML to prevent name collisions and increase the interoperability of document types from different sources. They extend the XML 1.0 naming mechanism, which provides for element and attribute names, to a pair-based mechanism: in XML Namespaces, a qualified name consists of a namespace and a local name.

Namespaces are identified by globally unique strings. To ensure international interoperability, URIs usually serve as namespace identifiers in practice. This is only a convention – processors do not treat URIs any different from arbitrary character sequences. Usage of an URI as a namespace does not imply the corresponding schema is in fact available under that address.

Because globally unique strings are unwieldy, XML Namespaces mandates the use of an abbreviation mechanism to conserve space. Under that convention, an instance document defines arbitrary identifiers called prefixes to represent a namespace. Prefixes precede local names, using a colon for separator. Their definitions take the form of virtual attributes with the reserved prefix `xmlns`.

The extent of prefix definitions resembles the extent of variable definitions in block-structured languages: a prefix represents the closest like-named definition in an ancestor element or the element itself. Thus, a prefix can be used before its definition, but forward references are limited to the scope of an opening tag. These rules are illustrated by example 6.

Example 6 (Namespaces and prefixes). In the following fragment

```
1  <a xmlns:p="http://www.noga.de/namespaces/XYZ">
2    <p:b/>
3    <p:b xmlns:p="http://www.noga.de/namespaces/ABC"/>
4    <p:b/>
5  </a>
```

the prefix `p` represents two different namespaces. The definition in line 1 is in scope for lines 1, 2, 4 and 5 and used in lines 2 and 4. The definition in line 3 is in scope in line 3

and used there in the element prefix – an instance of a forward reference. Substituting a different identifier for p, e.g., abracadabra, leaves the content invariant.

The concept of scope exceeds the modelling power of context-free grammars. In block-structured languages, this problem is commonly resolved by describing a superset of the language with a context-free grammar. Any additional constraints are realized in a separate semantic analysis phase operating on the parse tree.

As XML Namespaces limit forward references, prefix resolution only requires a single pass of a stack automaton. Using a suitable representation for qualified names, the grammars derived from DTDs or XML Schemas may be retained for namespace-aware processing. Single-pass treatment remains feasible if superset parsing, prefix resolution and qualified name matching are suitably interleaved.

4.3. Generalization

For now, we define the subset of restricted XML Schema to be the set of all schemas not employing or admitting wildcards, overrides and namespaces. In practice, the last restriction is severe, but the first two restrictions are mild ones. As stated, most wildcards can be transformed into equivalent constructs. Also, type overrides currently occur in very few documents. Using this notion of restricted XML Schema, we generalize our main result.

Theorem 8 (Restricted XML Schema and *SLL*(1) and *LALR*(1) grammars). There is an algorithm to find an *SLL*(1) grammar G defining the same language for every pair of restricted XML Schema and starting element. The same grammar G is also *LALR*(1).

Proof. We modify the algorithm in lemma 4 as follows: First, we generate grammar fragments for complex types. Since their content models are regular expressions differing from those in DTDs only in notation, we can use the construction in lemma 1. In the resulting grammar for the content of complex type t, we substitute all terminals s representing element names with new, unique non-terminals $N_{(t',s)}$, where t' is the type of s in this context. The grammars for complex types' attributes are generated as in lemma 2.

For every pair of complex type t and element name s in the schema, we then generate productions per lemma 4, using $N_{(t,s)}$ as the top-level non-terminal and the attribute and content model grammars of t.

The proof of theorem 6 applies by analogy, as element names remain unique in a complex type. □

Example 7. Applying theorem 8 to the fragment in example 5 leads to the following productions:

$$
\begin{array}{llll}
P = \{\ N_{(A,x)} \to x_{[}A_{(A,x)}B_{(A,x)} & N_{(B,x)} \to x_{[}A_{(B,x)}B_{(B,x)} & N_{(B,y)} \to y_{[}A_{(B,y)}B_{(B,y)} \\
\quad A_{(A,x)} \to \varepsilon & A_{(B,x)} \to \varepsilon & A_{(B,y)} \to \varepsilon \\
\quad B_{(A,x)} \to /> & & \\
\quad B_{(A,x)} \to >C_A x_{]} & B_{(B,x)} \to >C_B x_{]} & B_{(B,x)} \to >C_B y_{]} \\
\quad\quad C_A \to N_{(B,x)}C_A & \quad\quad C_B \to N_{(A,x)}C'_B & \\
\quad\quad C_A \to \varepsilon & \quad\quad C_B \to N_{(B,y)}C'_B & \\
 & \quad\quad C'_B \to \varepsilon & \\
\}
\end{array}
$$

Each of the above columns contains the productions of a pair (t, s), where t is the type assigned to element name s within a complex type: the first defines elements x of type A, the second elements x of type B and the last elements y of type B.

5. Performance

We implemented parser generators for Java and C. Due to just-in-time compilers (JITs) and garbage collection overheads, the Java platform cannot be fairly compared to native code. Thus, we only compare the faster implementations in the C/C++ language. Apache's Xerces-C and James Clark's expat enjoy widespread deployment and are generally considered highly performant.

To establish firm performance figures, we generated a series of test cases from the MOST specification [MOST 2002]. The file specifies real-world automotive components and spans two MB. Most functions reach a nesting depth of twelve or more. Using our aXMLerat toolkit, we generated a validating parser from the corresponding DTD. The generated parser builds a tree representation for the entire document in memory. It allows for all DOM accesses to this tree. Additionally, it explicitly captures the parse tree for the content models allowing a highly selective access to parts of an element's content. Xerces-C was tested in DOM and SAX mode – while both are validating, only the former actually builds a tree representation. expat neither validates nor builds a tree representation. Figure 1 shows their running times.

Measurements were taken on an Athlon 850 MHz with 256 MBytes RAM under Windows NT 4.0 SP 6. We tested Xerces-C version 1.3.0 and expat versing 1.2 using DOMCount and SAXCount respectively expat -t. The parser generated by aXMLerat is targeted towards the Linux platform. We tested it using the Cygwin environment under NT and separately under Linux on the identical machine.

Our generated Linux parser is up to 40% faster than the fastest Windows competitor, expat. Even more significant, our parser is validating and builds a full tree representation in memory, whereas expat does neither.

The Cygwin simulation environment clearly imposes a large performance hit compared to native compilers. Under Cygwin/Windows, our Windows parser runs some 15% slower than expat. Xerces-C, considered a high-performance implementation by its authors, is not remotely comparable. In DOM mode, it even aborts processing the largest input file due to memory restrictions.

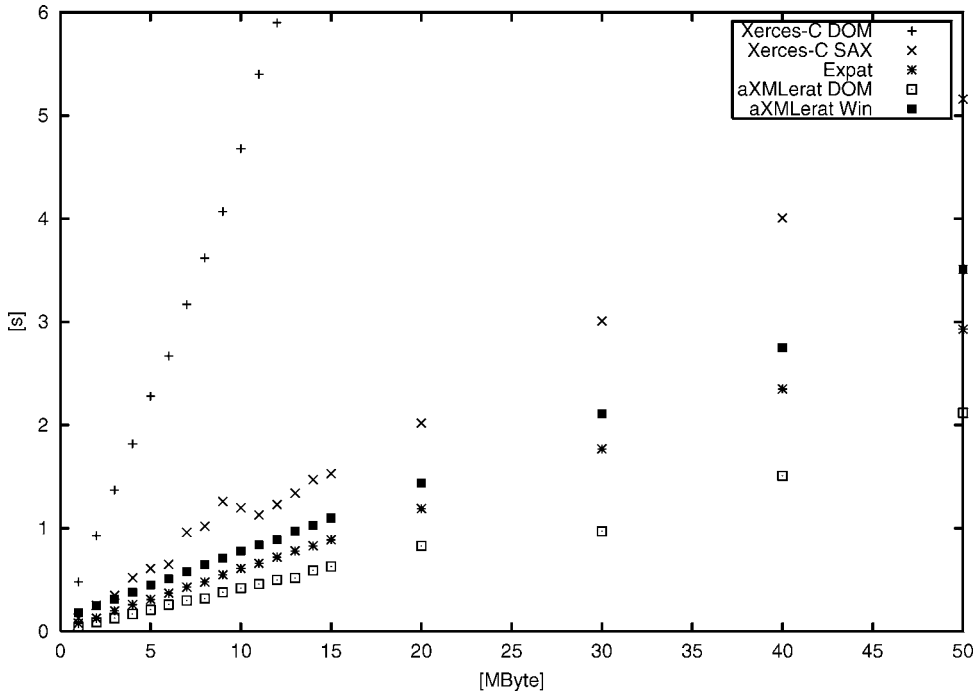

Figure 1. Parsing times.

6. Related work

The SGML [ISO 1986] designers solved the ambiguity problem by design restriction: SGML requires content models to be deterministic regular expressions. If this restriction continued to hold for XML, deterministic parsing would be no problem.

As shown in [Brüggemann-Klein 1993], there are deterministic regular expressions not allowing for directly for parsing without look-ahead or back-tracking. For some of them, no equivalent deterministic regular expressions without this negative property exist. This result justifies our approach of constructing context-free grammars for deterministic parsing.

A theoretical view on XML languages is given in [Berstel and Boasson 2000]. The authors ignore attributes, namespaces etc. In their paper, they take a language-theoretic point of view. They show that some properties which are not decidable for general context-free languages become decidable for XML grammars. The result most closely related to our work is that DTDs (called XML-grammars) have a unique normal form. However, this normal form can be ambiguous.

There are working parser generators for XML documents. Our own generators in the aXMLerat [B2B Group 2002] toolkit exploit the transformations described in the present paper. They generate lalr and javacc specifications from DTDs and directly generate an LL parser for XML Schema definitions. The XMLBooster [PhiDaNi 2001] uses its own specification language. Thereby, they avoid the problems of ambiguous DTDs

and schemas. Using the theoretical results above, it possible to transform their proprietary specification to DTDs. However, it remains unclear if transformations in the opposite directions are always possible.

7. Conclusion

This paper enables the generation of deterministic parsers for XML documents from arbitrary, even ambiguous DTDs. We investigated the context-free DTD languages. As our examples show, there exist ambiguous DTDs. Even deterministic DTDs may not be parsable without lookahead. Despite this, all DTDs can be transformed into deterministic grammars with the algorithms in this paper. The generated grammars are both $LL(1)$ and $LALR(1)$. Respective transformations are non-computable for context-free grammars in general. We also developed a generalization to a subset of the XML Schema languages. Possible solutions for the problems arising from namespaces were outlined.

Our generated grammars are suitable for common parser generators. With those tools, we automatically generated validating parsers that build full tree representations of documents in memory. Generation eliminates the need to analyze the DTD at runtime. Although they support more specific operations, our generated parsers are manifestly faster than generic plain DOM parsers. Due to our targeting the Linux environment, we are relegated to the Cygwin environment under Windows, so some no-op generic parsers still retain a small performance lead.

We aim to modify our generator to support native Windows execution of the generated parsers. We will continue experiments in static high-speed parsing in native Linux and Windows settings. Future test settings will vary document structures as well as sizes and include a wider array of competitors, e.g., the GDome2, Microsoft and Qt generic XML parsers and the XMLBooster generator. Finally, we expect to establish a correlation between memory consumption and running times.

In this paper, we only considered applications with pre-agreed document types. These settings admit for off-line parser generation. Future work will consider on-line settings where parser generation speed is crucial to the processing system.

Appendix A. Basic definitions

Definition 5 (Formal language). An alphabet T is a finite, non-empty set of symbols. The set of finite strings formed by concatenating symbols from T is denoted by T^+. T^* denotes T^+ augmented by the empty string ε. Each subset of T^* is a formal language over the alphabet T. Its elements are called words.

Formal languages are usually defined by grammars.

Definition 6 (Grammar and ambiguous grammar). A grammar is a quadruple $G = (T, N, P, Z)$ with an alphabet T, T and N disjoined, $Z \in N$, and $(T \cup N, P)$ a general

rewrite system. $L(G)$ denotes the formal language defined by G. N is called the set of non-terminal symbols of the grammar.

A grammar G is ambiguous if a sentence of $L(G)$ may be derived form Z using at least two different sequences of substitutions from P. Otherwise it is called deterministic.

Grammars may be classified according to the form of their productions:

Definition 7 (Regular and context-free grammars and languages). A grammar is *regular* if each production in P has the form $B \to b$ or the form $B \to bC$ with $B, C \in N, b \in T \cup \{\varepsilon\}$. A grammar is *context free* if each production in P has the form $B \to x$ with $B \in N$, $x \in (N \cup T)^*$. A language is regular (context free) iff it can be defined by a regular (context-free) grammar.

Regular languages may also be defined by regular expressions:

Definition 8 (Regular expressions and regular languages). Let T be an alphabet. Then $\{a | a \in T \cup \{\varepsilon\}\}$ are regular expressions r defining the regular language $L(r) = \{a\}$.

Let r_1 and r_2 be regular expressions. Then

- $r_1|r_2$ defining the regular language $L(r_1) \cup L(r_2)$,
- r_1, r_2 defining the regular language $L = \{ab \mid a \in L(r_1) \wedge b \in L(r_2)\}$,
- r_1^* defining the regular language $L = \{a \ldots a \mid a \in L(r_1)\} \cup \{\varepsilon\}$

are regular expressions.

Remark. Some definitions of regular expressions include:

- r_1^+ defining the regular language $L = \{a \ldots a \mid a \in L(r_1)\}$,
- $r_1?$ defining the regular language $L(r_1) \cup \{\varepsilon\}$.

These expressions can be derived, as $r_1^+ = r_1, r_1^*$ and $r_1? = r_1|\varepsilon$, respectively.

Appendix B. An example based on a DTD

We expand on example 2:

```
<!ELEMENT a ( x? , ( y* | z* ) )>
<!ELEMENT x ( #PCDATA )>
<!ELEMENT y ( #PCDATA )>
<!ELEMENT z ( #PCDATA )>
```

A corresponding ambiguous acceptor for the content model of `a` is given in figure 2. The right hand side shows the unambiguous acceptor resulting from our construction.

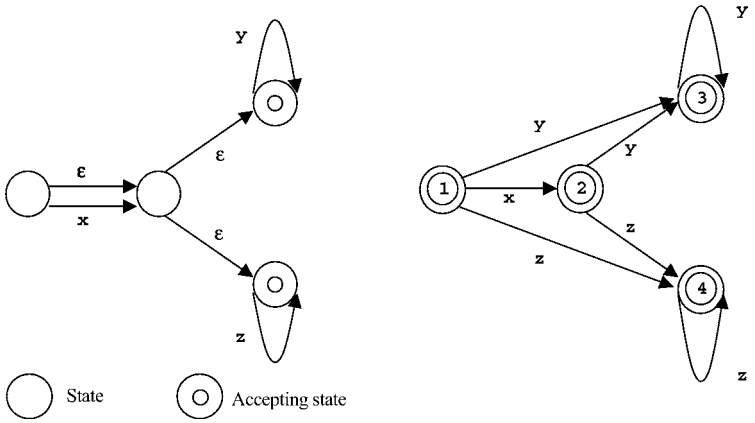

Figure 2. Deterministic finite acceptor.

According to the construction in lemma 1, this acceptor leads to the following *SLL*(1) and *SLR*(1) grammar for the content model of `a`:

1.	$N_a(1) \to \varepsilon$	
2.	$N_a(1) \to xN_a(2)$	$N_a(1) \to N_xN_a(2)$
3.	$N_a(1) \to yN_a(3)$	$N_a(1) \to N_yN_a(3)$
4.	$N_a(1) \to zN_a(4)$	$N_a(1) \to N_zN_a(4)$
5.	$N_a(2) \to \varepsilon$	
6.	$N_a(2) \to yN_a(3)$	$N_a(2) \to N_yN_a(3)$
7.	$N_a(2) \to zN_a(4)$	$N_a(2) \to N_zN_a(4)$
8.	$N_a(3) \to \varepsilon$	
9.	$N_a(3) \to yN_a(3)$	$N_a(3) \to N_yN_a(3)$
10.	$N_a(4) \to \varepsilon$	
11.	$N_a(4) \to zN_a(4)$	$N_a(4) \to N_zN_a(4)$

Where applicable, we printed the productions transformed by the construction in lemma 4 in the right column. The grammars for the content models of `x`, `y` and `z` are obtained accordingly. As they are identical up to renaming non-terminals, we display the one for `x`:

12. $N_x(1) \to \varepsilon$
13. $N_x(1) \to \mathit{string}$

As the example does not define attributes, all grammars for element attributes accept e only. The productions are identical up to renaming of non-terminals, so we display the ones for `a` (14) and `x` (15):

14. $A_a \to \varepsilon$
15. $A_x \to \varepsilon$

The final construction algorithm of lemma 4 adds the productions which are identical up to renaming of non-terminals and the opening and closing tag strings. We display those for a (16–18) and x (19–21):

$$
\begin{array}{ll}
16. & N_a \to \texttt{<a}A_aB_a \\
17. & B_a \to \texttt{>}N_a(1)\texttt{} \\
18. & B_a \to \texttt{/>} \\
19. & N_x \to \texttt{<x}A_xB_x \\
20. & B_x \to \texttt{>}N_x(1)\texttt{</x>} \\
21. & B_x \to \texttt{/>}
\end{array}
$$

We reconsider the example sentence that lead to two derivations for the original grammar.

```
<a><x> ... </x></a>
```

Its derivation is now unique:

$$
\begin{array}{lr}
N_a \to \texttt{<a}A_aB_a & (16) \\
\quad \to \texttt{<a}\varepsilon B_a & (14) \\
\quad \to \texttt{<a}\varepsilon\texttt{>}N_a(1)\texttt{} & (17) \\
\quad \to \texttt{<a}\varepsilon\texttt{>}N_xN_a(2)\texttt{} & (2) \\
\quad \to \texttt{<a}\varepsilon\texttt{><x}A_xB_xN_a(2)\texttt{} & (19) \\
\quad \to \texttt{<a}\varepsilon\texttt{><x}\varepsilon B_xN_a(2)\texttt{} & (15) \\
\quad \to \texttt{<a}\varepsilon\texttt{><x}\varepsilon\texttt{>}N_x(1)\texttt{</x>}N_a(2)\texttt{} & (20) \\
\quad \to \texttt{<a}\varepsilon\texttt{><x}\varepsilon\texttt{>}string\texttt{</x>}N_a(2)\texttt{} & (13) \\
\quad \to \texttt{<a}\varepsilon\texttt{><x}\varepsilon\texttt{>}string\texttt{</x>}\varepsilon\texttt{} & (5) \\
\\
\quad \equiv \texttt{<a><x> ... </x>}
\end{array}
$$

References

Apache (2002), *Xerces* C++ *Parser*, Apache XML Project, http://xml.apache.org/xerces-c/.

B2B Group (2002), *aXMLerate Project*, University of Karlsruhe, http://i44pc29.info.uni-karlsruhe.de/B2Bweb/.

Berstel, J. and L. Boasson (2000), "XML Grammars," In *Mathematical Foundations of Computer Science (MFCS'2000)*, N. Nielsen and B. Rovan, Eds., Lecture Notes in Computer Science, Vol. 1893, Springer, pp. 182–191. Long version as Technical Report IGM 2000-06, see www-igm.univ-mlv.fr/~berstel/Recherche.html.

Brüggemann-Klein, A. (1993), "Regular Expressions into Finite Automata," *Theoretical Computer Science 120*, 2, 197–213.

Clark, J. (2000), "Expat – XML Parser Toolkit Version 1.2," http://www.jclark.com/xml/expat.html.

DeRemer, F.L. (1971), "Simple *LR*(k) Grammars," *Communications of the ACM 14*, 7, 453–460.

Donelly and Stallmann (1988), "Bison Manual," The GNU Project, http://www.gnu.org/manual/bison/.

Grosch, J. (1989), "Generators for High-Speed Front-Ends," In *Proceedings of the 2nd Workshop on Compiler Compilers and High Speed Compilation*, D. Hammer, Ed., Lecture Notes in Computer Science, Vol. 371, Springer, Berlin, pp. 81–92.

IBM AlphaWorks (2001), "XML Parser for Java," IBM AlphaWorks, http://alphaworks.ibm.com/aw.nsf/techmain/xml4j.

ISO (1986), "Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)," ISO 8879.

Johnson, S. (1975), "Yacc – Yet Another Compiler-Compiler," Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ.

Microsoft (2002), "Component Object Model," Microsoft, http://www.microsoft.com/com/.

MOST (2002), "The MOST Cooperation," The MOST Cooperation, http://www.mostnet.org/.

OMG (2002), "Corba 2.4.2 Specification," Object Management Group, http://www.omg.org/technology/documents/formal/corbaiiop.htm.

PhiDaNi (2001), "The XML Booster," PhiDaNi Software, http://www.xmlbooster.com.

Rosenkrantz, D.J. and R.E. Stearns (1969), "Properties of Deterministic Top Down Grammars," In *Conference Record of ACM Symposium on Theory of Computing*, Marina del Rey, CA, pp. 165–180.

Vielsack, B. (1988), "The Parser Generators lalr and ell," Technical Report 93-3, Gesellschaft für Mathematik und Datenverarbeitung, Forschungsstelle Karlsruhe.

W3C (1998), "Extensible Markup Language (XML) 1.0," W3C Recommendation 10 February 1998, http://www.w3.org/TR/1998/REC-xml-19980210.

W3C (1999), "Namespaces in XML," W3C Recommendation 14 January 1999, http://www.w3.org/TR/1999/REC-xml-names-19990114.

W3C (2001), "XML Schema Part 1: Structures," W3C Recommendation 2 May 2001, http://www.w3.org/TR/2001/REC-xmlschema-1-20010502.

Waite, W. and G. Goos (1985), *Compiler Construction*, Texts and Monographs in Computer Science, Springer, Berlin.

WebGain (2002), "JavaCC – The Java Parser Generator," WebGain, http://www.webgain.com/products/metamata/java_doc.html.