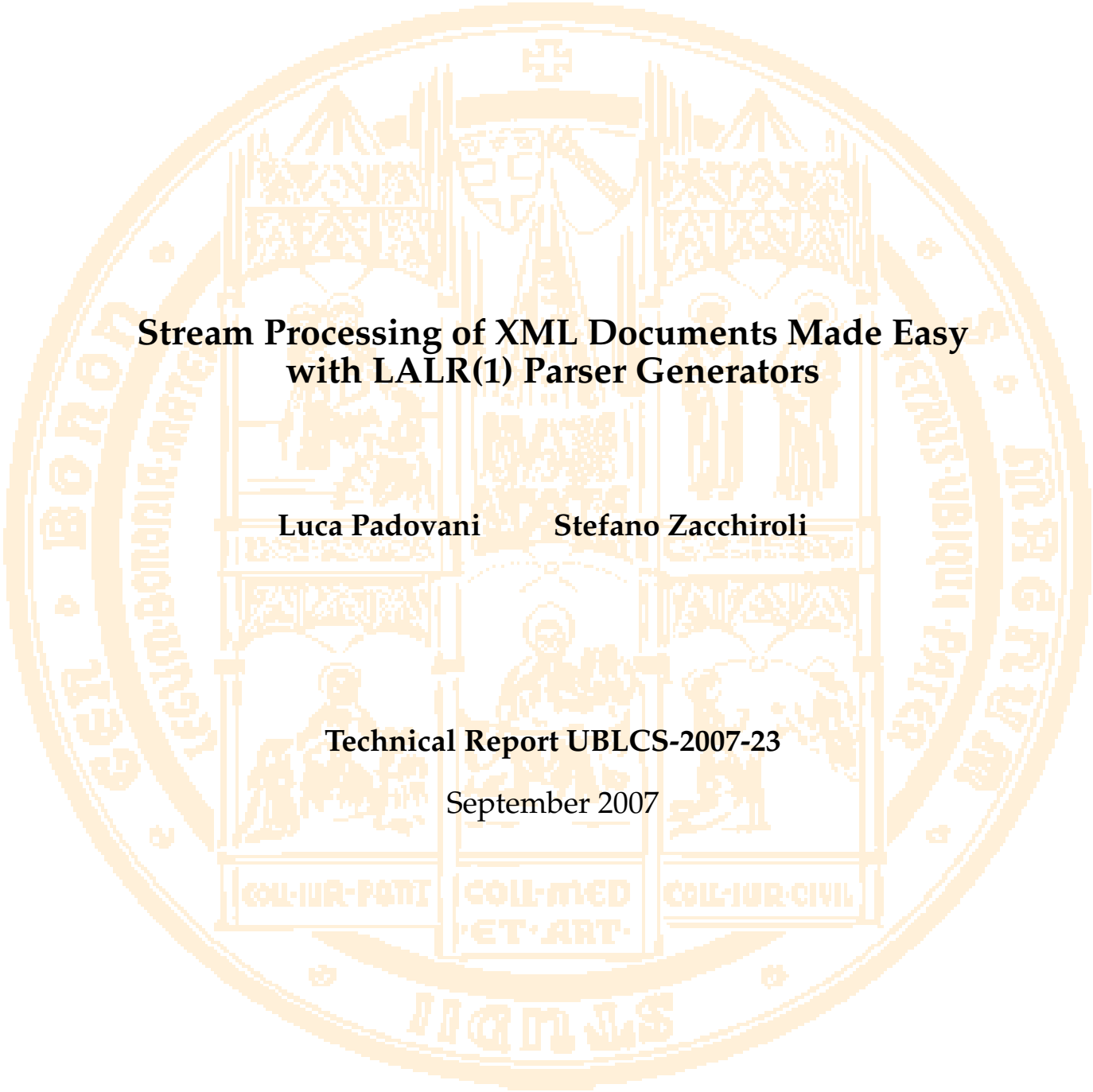

Stream Processing of XML Documents Made Easy with LALR(1) Parser Generators

Luca Padovani **Stefano Zacchiroli**

Technical Report UBLCS-2007-23

September 2007

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

The University of Bologna Department of Computer Science Research Technical Reports are available in PDF and gzipped PostScript formats via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` or via WWW at URL `http://www.cs.unibo.it/`. Plain-text abstracts organized by year are available in the directory `ABSTRACTS`.

Recent Titles from the UBLCS Technical Report Series

2007-01 *Extending the Choquet Integral*, Rossi, G., January 2007.

2007-02 *Towards Cooperative, Self-Organised Replica Management*, Hales, D., Marcozzi, A., Cortese, G., February 2007.

2007-03 *A Model and an Algebra for Semi-Structured and Full-Text Queries (PhD Thesis)*, Buratti, G., March 2007.

2007-04 *Data and Behavioral Contracts for Web Services (PhD Thesis)*, Carpineti, S., March 2007.

2007-05 *Pattern-Based Segmentation of Digital Documents: Model and Implementation (PhD Thesis)*, Di Iorio, A., March 2007.

2007-06 *A Communication Infrastructure to Support Knowledge Level Agents on the Web (PhD Thesis)*, Guidi, D., March 2007.

2007-07 *Formalizing Languages for Service Oriented Computing (PhD Thesis)*, Guidi, C., March 2007.

2007-08 *Secure Gossiping Techniques and Components (PhD Thesis)*, Jesi, G., March 2007.

2007-09 *Rich Media Content Adaptation in E-Learning Systems (PhD Thesis)*, Mirri, S., March 2007.

2007-10 *User Interaction Widgets for Interactive Theorem Proving (PhD Thesis)*, Zacchiroli, S., March 2007.

2007-11 *An Ontology-based Approach to Define and Manage B2B Interoperability (PhD Thesis)*, Gessa, N., March 2007.

2007-12 *Decidable and Computational Properties of Cellular Automata (PhD Thesis)*, Di Lena, P., March 2007.

2007-13 *Patterns for Descriptive Documents: a Formal Analysis*, Dattolo, A., Di Iorio, A., Duca, S., Feliziani, A. A., Vitali, F., April 2007.

2007-14 *BPM + DM = BPDM*, Magnani, M., Montesi, D., May 2007.

2007-15 *A Study on Company Name Matching for Database Integration*, Magnani, M., Montesi, D., May 2007.

2007-16 *Fault Tolerance for Large Scale Protein 3D Reconstruction from Contact Maps*, Vassura, M., Margara, L., di Lena, P., Medri, F., Fariselli, P., Casadio, R., May 2007.

2007-17 *Computing the Cost of BPMN Diagrams*, Magnani, M., Montesi, D., June 2007.

2007-18 *Expressing Priorities, External Probabilities and Time in Process Algebra via Mixed Open/Closed Systems*, Bravetti, M., June 2007.

2007-19 *Design and Evaluation of a Wide-Area Distributed Shared Memory Middleware*, Mazzucco, M., Morgan, G., Panzieri, F., July 2007.

2007-20 *An Object-based Fault-Tolerant Distributed Shared Memory Middleware*, Lodi, G., Ghini, V., Panzieri, F., Carloni, F., July 2007.

2007-21 *Templating Wiki Content for Fun and Profit*, Di Iorio, A., Zacchiroli, S., Vitali, F., August 2007.

2007-23 *Stream Processing of XML Documents Made Easy with LALR(1) Parser Generators*, Padovani, L., Zacchiroli, S., Vitali, F., September 2007.

Stream Processing of XML Documents Made Easy with LALR(1) Parser Generators

Luca Padovani[1] **Stefano Zacchiroli**[2]

Technical Report UBLCS-2007-23

September 2007

Abstract

Because of their fully annotated structure, XML documents are normally believed to require a straightforward parsing phase. However, the standard APIs for accessing their content (the Document Object Model and the Simple API for XML) provide a programming interface that is very low-level and is thus inadequate for the recognition of any structure that is not isomorphic to its XML encoding. Even when the document undergoes validation, its unmarshalling into application-specific data using these APIs requires poorly maintainable, tedious-to-write, and possibly inefficient code. We describe a technique for the simultaneous parsing, validation, and unmarshalling of XML documents that combines a stream-oriented XML parser with a LALR(1) parser in order to guarantee efficient stream processing, expressive validation capabilities, and the possibility to associate user-provided actions with specific patterns occurring in the source documents.

1. padovani@sti.uniurb.it, Information Science and Technology Institute, University of Urbino
2. zacchiro@cs.unibo.it, Department of Computer Science, University of Bologna

1 Introduction

When dealing with XML documents [9], *parsing* is just the first step of a more complex sequence of processing phases the applications go through. It is often the case that the XML documents must be *validated* against some *schema* which verifies structural and content invariants. Among the most popular *schema languages* for validating XML documents, with varying capabilities of capturing constraints about structure and content, we cite DTDs [9], XML Schema [10, 11], and Relax NG [6]. After validation, the document is *unmarshalled* into some internal, application-specific representation. In particular, the tree-structured representation imposed by XML does not necessarily correspond to an isomorphic or even only similar application-specific representation, unless the focus of the application is the XML document itself.

The Document Object Model (DOM, see [8]) has been designed to provide applications with a set of standard interfaces to access the structure and content of an XML document. The DOM API can be exploited to perform post-order recursive visits that unmarshal the document in a bottom-up fashion. Table 2 (A) shows a fragment of such unmarhsalling code for an hypothetical processor of XML messages conforming to the DTD shown in Table 1. The application specific objects corresponding to the various document fragments are created by otherwise undefined methods like *Header_MsgHeader*.

The DOM-based approach suffers from a number of serious drawbacks. First, it is expensive in terms of memory occupation because the whole document must be loaded into main memory, even though the document is going to be discarded soon after having been unmarshalled. Even if memory availability is not an issue, the XML encoding of a given data structure can be so verbose and redundant that its maintenance as a DOM tree may be considered impractical anyway.[3]

Second, if the XML representation of the document and the internal, application-specific representation have a significant difference in form and structure, the DOM introduces a computational overhead, as it forces the programmer to access the information in twisted ways. Since DOM interfaces are low-level and targeted at Java-like languages, they adapt poorly to different paradigms like functional languages [5].

Finally, explicit actions are required to ignore unnecessary nodes (these actions have been shaded in the code snippets of Table 2) unless the document is validated, and a lot of effort goes into recognizing patterns of elements inside the tree (think of elements with alternative content models). In general the validation phase results in a "yes" or "no" answer that tells very little about the actual *derivation steps* that produced the document from its specification, thus hindering an effective unmarshalling phase.[4]

As of today, there is another de-facto standard way for accessing the structure and content of XML documents: the *Simple API for XML* (SAX [7]) generates a sequence of events corresponding to the *syntactical entities* encountered in the source XML document. SAX does not require the whole document to be loaded into memory. Rather, there is a collection of callback functions (or methods) that are called as SAX events occur. Table 2 (B) shows a typical arrangement of such callback functions that keep track of the parsing context using a `ctxt` structure containing:

1. a stack of elements (`ctxt.elements`) whose start tags have been encountered up to a certain point;

2. a stack of values (`ctxt.values`) representing bits of unmarhalled document.

When the start tag of an element is encountered, we first make sure that the element is legal in its context and then we push its name on the element stack. When the end tag of an element is encountered, a "reduce" action updates the stacks and synthesizes the appropriate part of the unmarshalled document using one of the aforementioned functions. As the code sample demonstrates, the recognition of complex patterns in the source document must be explicitly

3. See [3] for a comparison of two popular free DOM implementations with respect to memory occupation.
4. XML Schema PSVI (Post Schema Validation Infoset) contains information about the type names the elements and attributes have been validated against. However, since type names may correspond to alternative or optional content models, this information does not uniquely determine the actual pattern matched by a subtree of a valid document.

Table 1. DTD for messages of a hypothetical XML-based protocol.

```
<!ELEMENT Envelope (Header?, Body)>
<!ELEMENT Header
    (MsgHeader | StatusRequest | StatusResponse)>
<!ELEMENT Body ANY>
<!ELEMENT MsgHeader (MessageId, ResponseTo?)>
<!ELEMENT MessageId #PCDATA>
<!ELEMENT ResponseTo #PCDATA>
<!ELEMENT MessageStatus #PCDATA>
<!ELEMENT TimeStamp #PCDATA>
<!ELEMENT StatusRequest (MessageId)>
<!ELEMENT StatusResponse
    (MessageId, MessageStatus, TimeStamp)>
```

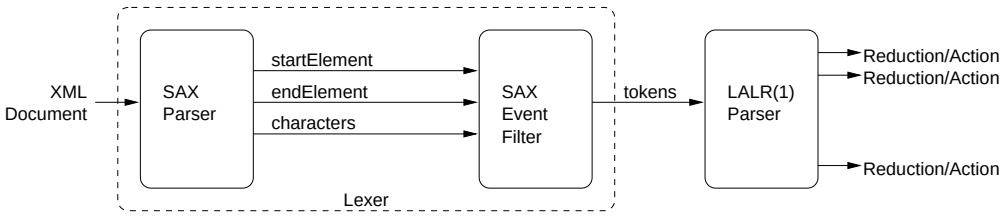

Figure 1. Architecture of FLEAGRAM.

coded, with the further difficulty with respect to the DOM-based approach that the document is not entirely loaded in memory, so neither lookahead nor backtracking are possible.

This paper presents a technique that combines the SAX-based parsing phase, the validation, and the unmarshalling of XML documents. The technique is based on a bottom-up parser which is fed with tokens coming from a SAX parser. The parser is automatically generated from a FLEAGRAM specification that simultaneously describes the syntactical validity constraints that the parsed documents must satisfy as well as semantic actions associated with document patterns that drive the unmarshalling into application-specific data structures. Table 2 (C) shows a sample FLEAGRAM specification that validates and unmarshals the same kind of documents handled by the code snippets (A) and (B).

The structure of the paper is as follows: in Section 2 we elaborate on the overall architecture of the tool. Section 3 describes the format and the meaning of the specification grammar used for generating the parser. Section 4 relates our approach with some of the most widespread schema languages for XML documents. We conclude in Section 5 where we also show some performance measurements. A prototype tool that implements the architecture described in this paper is publicly available at the address `http://www.cs.unibo.it/~lpadovan/flea/`.

2 Architecture

The basic idea is to consider the stream of events coming from a SAX-based parser as a stream of tokens, and to feed a LALR(1) parser with that stream (Figure 1). The subset of methods invoked by a SAX 2.0 parser that are relevant for our purposes is listed below (a complete and more extensive overview can be found in [7]):

`characters(text)`: notification of character data. `text` is a string representing the parsed characters;

`startElement(uri, localName, qName, attributes)`:

Table 2. Different approaches for parsing XML documents conforming to the DTD of Table 1: (A) implements a recursive traversal using the DOM API; (B) shows the callback functions for a stream parser using the SAX API; (C) shows a FLEAGRAM specification grammar.

(A)

```
function skipBlanks(NodeList nl, int i) {
  while (i < nl.length
         && (isBlank(nl[i]) || isComment(nl[i])))
    i++;
  return i;
}

function parseHeader(el) {
  NodeList children = el.childNodes;
  int i = 0;
  i = skipBlanks(children, i);
  if (i == children.length)
    error();
  if (children[i].nodeName == "MsgHeader") {
    return Header'MsgHeader(parseMsgHeader(children[i]));
  } else if (children[i].nodeName == "StatusRequest") {
    return Header'StatusRequest(
             parseStatusRequest(children[i]));
  } else if (children[i].nodeName == "StatusResponse") {
    return Header'StatusResponse(
             parseStatusResponse(children[i]));
  } else
    error();
}

function parseEnvelope(el) {
  NodeList children = el.childNodes;
  int i = 0;
  i = skipBlanks(children, i);
  if (i == children.length)
    error();
  if (children[i].nodeName == "Header") {
    Object headerV = parseHeader(children[i]);
    i++;
    i = skipBlanks(children, i);
    if (i == children.length)
      error();
    if (children[i].nodeName == "Body") {
      Object bodyV = parseBody(children[i]);
      if (skipBlanks(children, i + 1) < children.length)
        error();
      return Envelope'Header'Body(headerV, bodyV);
    } else
      error();
  } else if (children[i].nodeName == "Body") {
    Object bodyV = parseBody(children[i]);
    if (skipBlanks(children, i + 1) < children.length)
      error();
    return Envelope'Body(bodyV);
  } else
    error();
}

function parseBody(el) { /* ... */ }
```

(B)

```
function startElement(uri, localName, qName, attributes) {
  if (qName == "Envelope") {
    if (ctxt.parent != null)
      error();
  } else if (qName == "Header") {
    if (ctxt.parent != "Envelope"
        || ctxt.siblings.length > 0)
      error();
  } else if (qName == "Body") {
    if (ctxt.parent != "Envelope"
        || ctxt.siblings.length > 1
        || (ctxt.siblings.length == 1
            && ctxt.siblings.last != "Header"))
      error();
  } else {
    // ...
  }
  ctxt.push(qName);
}

function endElement(uri, localName, qName) {
  if (qName == "Envelope") {
    if (ctxt.siblings[0] == "Body")
      ctxt.reduce(Envelope'Body(ctxt.value[0]));
    else
      ctxt.reduce(Envelope'Header'Body(ctxt.value[0],
                                       ctxt.value[1]));
  } else if (qName == "Header") {
    if (ctxt.siblings[0] == "MsgHeader")
      ctxt.reduce(Header'MsgHeader(...));
    else if (ctxt.siblings[0] == "StatusRequest")
      ctxt.reduce(Header'StatusRequest(...));
    else
      ctxt.reduce(Header'StatusResponse(...));
  } else if (qName == "Body") {
    ctxt.reduce(Body(...));
  } else {
    // ...
  }
}

function characters(text) {
  // ...
}
```

(C)

```
%%
<Envelope>  : <Body> { $$ = Envelope'Body($1); }
            | <Header> <Body> { $$ = Envelope'Header'Body($1, $2); } ;
<Header>    : <MsgHeader> { $$ = Header'MsgHeader($1); }
            | <StatusRequest> { $$ = Header'StatusRequest($1); }
            | <StatusResponse> { $$ = Header'StatusResponse($1); } ;
<Body>      : ANY { $$ = Body(...); } ;
<MsgHeader> : ...
%%
```

notification of the beginning of an element in the namespace `uri` with local name `localName` and qualified name `qName`. `attributes` is the list of attributes set on the element;

`endElement(uri, localName, qName)`: notification of the end of an element in the namespace `uri`, with local name `localName` and qualified name `qName`.

Each one of these methods produces one or more tokens to be fed into the LALR(1) parser. In Figure 1, the component that is responsible for deciding which tokens must be generated and in which order is called *event filter*. Taken together, the SAX parser and the event filter play the role of a lexer. The event filter knows:

1. the set of XML element names that are explicitly referenced by the grammar (which will be described in Section 3);
2. for each element t, the set of attributes that are explicitly referenced by the grammar as attributes of t.

Explicitly referenced elements and attributes are said *relevant*.

We consider the following tokens:

$$\begin{array}{rcl} \langle token \rangle & ::= & \texttt{<}[ns\texttt{:}]tag \mid [ns\texttt{:}]tag\texttt{>} \mid \texttt{<?} \mid \texttt{?>} \\ & \mid & \texttt{<@}[ns\texttt{:}]attr \mid \texttt{@}[ns\texttt{:}]attr\texttt{>} \mid \texttt{@?} \\ & \mid & \texttt{S} \end{array}$$

For each relevant element with name tag in a namespace ns, the SAX methods `startElement` and `endElement` generate two tokens $\texttt{<}ns\texttt{:}tag$ and $ns\texttt{:}tag\texttt{>}$ when the start-tag and the end-tag of the element are encountered. For elements that are not relevant, the two tokens `<?` and `?>` are generated instead. After the start-tag of an element is encountered and the corresponding $\texttt{<}ns\texttt{:}tag$ token has been generated, for each relevant attribute $attr$ in a namespace ns, the SAX method `startElement` generates the $\texttt{<@}ns\texttt{:}attr$ and $\texttt{@}ns\texttt{:}attr\texttt{>}$ tokens. If the attribute is set on the element, a `S` token carrying the value of the attribute is generated in between the start and end attribute tokens. A canonical order of the relevant attributes is decided so that the attribute tokens for a particular element are always generated in the same order. If there are attributes other than the relevant ones that are set on the element, a single `@?` token is also generated. Note that there is a unique SAX method that carries information about both the start tag and the attributes set for an element, whereas in the token stream tags and attributes are treated separately. This will increase the expressiveness of the parser, as we will show in a later section. Finally, the `characters` method causes the generation of a `S` token whose associated value is the text string being parsed.

Each token has an associated semantic value which provides further information such as the position in the source document where the element or attribute was found, the complete namespace URI of the element or attribute, the actual string of characters matched in the case of the `S` token, the set of non-relevant attribute names and values for `@?` tokens.

As an example, the XML document

```
<doc version="2.0">
  <title>My title</title>
  <body>
    <p id="I12">a paragraph</p>
    <p id="I13" style="emph">another paragraph</p>
  </body>
</doc>
```

generates the following stream of tokens (we assume that whitespaces are ignored, that all elements are relevant and that all the attributes but `style` are relevant):

Table 3. Extended BNF for FLEAGRAM specification grammars.

⟨*spec*⟩	::=	⟨*preamble*⟩ `%%` { *rule* } `%%` ⟨*supporting code*⟩
⟨*preamble*⟩	::=	{ ⟨*source code*⟩ | ⟨*declaration*⟩ }
⟨*declaration*⟩	::=	`%union {{ ... }}`
	|	`%type` ... { ⟨*nt symbol right*⟩ }$^+$
	|	`%start` ⟨*nt symbol right*⟩
	|	`%namespace` *id URI*
⟨*nt symbol left*⟩	::=	*id* | <[*id* `:`]*id* { `@`[*id* `:`]*id* = *production* } [`@`$\star$]>
⟨*nt symbol right*⟩	::=	*id* | <[*id* `:`]*id*>
⟨*rule*⟩	::=	⟨*nt symbol left*⟩ `:` ⟨*production*⟩ { `|` ⟨*production*⟩ } `;`
⟨*production*⟩	::=	ε | `error` | { ⟨*atom*⟩ }$^+$
⟨*atom*⟩	::=	⟨*nt symbol right*⟩
	|	⟨*token*⟩
	|	`{{` ⟨*source code*⟩ `}}`

```
<doc <@version S @version>
  <title S title>
  <body <p <@id S @id> S p> <p <@id S @id> @? S p> body>
doc>
```

The strings `2.0`, `My title`, `I12`, `a paragraph`, `I13`, and `another paragraph` are associated with the various `S` tokens, in this order, and the singleton set containing the attribute name `style` and its pairing value `emph` is associated with the `@?` token.

3 Specification grammars

The token stream resulting from the event filter drives a LALR(1) parser. While this parser can be generated directly from a hand-written Yacc [1, 4] specification grammar, the context that we are considering requires particular care for a number of details. In particular, it is annoying and error-prone to distinguish between relevant and non-relevant attributes, to remember the order of relevant attributes, and the effort for producing a specification goes beyond that required for writing, say, a DTD for the same documents. It is thus compelling to provide a more specific and handy syntax for writing FLEAGRAM specification grammars, which we are going to describe.

3.1 Overall grammar structure

A FLEAGRAM specification grammar (Table 3) is made of three sections: the preamble contains declarations, source code, and directives used to fine-tune the behavior of the parser. Source code is enclosed inside `%{{...%}}`, and can contain declarations of symbols and variables used by rules or procedures in the second and third sections. The `union`, `type`, and `start` declarations have the same purpose as for regular Yacc specifications: they determine the structure of semantic values synthesized by the parser, the type associated with each non-terminal symbol, and the start symbol of the grammar. The `namespace` declaration associates a namespace with a prefix to be used in the rules. Note that there is no need to declare the tokens of the grammar, since either they are pre-defined (as is the case of the `<?`, `?>`, `@?`, and `S` tokens) or they can be automatically inferred by looking at the production rules for element tags and attribute names. The central section, consisting of a sequence of zero or more rules, specifies the structure of the XML documents to be parsed as well as the semantic actions to be executed. The last section of the specification is made of literal code to be included in the parser, and can provide further auxiliary functions, possibly invoked from semantic actions in the rules.

3.2 Grammar rules

In Yacc, a low-case identifier in the grammar rules usually indicates a non-terminal symbol. In our specifications, we would like to describe the content model of XML elements by means of grammar rules. To this aim we allow the following extended form for rules:

- when the left hand side of a rule is a id symbol, the symbol is to be intended as a non-terminal in the Yacc sense;
- when the left hand side of a rule is:

  $$<t\ \texttt{@}a_1 = x_1\ \cdots\ \texttt{@}a_n = x_n>$$

  the rule describes the content model of the t element whenever the attributes a_i respect the corresponding specifications x_i.

A production can be empty (that is, made of no atoms). In Yacc this is used to match so called ε-rules, but it is also useful in the context of XML to denote empty elements. As in Yacc, the `error` atom can be used to re-synchronize the parser if a syntax error occurs. Finally, a production may consist of a sequence of one or more atoms.

An atom can be a terminal symbol $token$, a non-terminal symbol id, a <t> (in this case an occurrence of the t element and its content is expected at that point of the input), or a semantic action, made of arbitrary code enclosed within `{{...}}`.

The addition of the <t> non-terminals in the head and in the body of rules does not change the expressive power of the specification. These rules can be syntactically expanded into equivalent forms that make use of plain tokens only. The basic idea is that a rule of the form

$$<t> : x_{11}\ x_{12}\ \cdots\ x_{1n_1}\ \texttt{|}\ \cdots\ \texttt{|}\ x_{k1}\ x_{k2}\ \cdots\ x_{kn_k}\ \texttt{;}$$

is just a shortcut for the rule

$$[\![t]\!] : <t\ x_{11}\ x_{12}\ \cdots\ x_{1n_1}\ t> \texttt{|}\ \cdots\ \texttt{|} <t\ x_{k1}\ x_{k2}\ \cdots\ x_{kn_k}\ t> \texttt{;}$$

where we use $[\![t]\!]$ to denote a fresh non-terminal symbol lexically derived from the tag t.

In general, let $\{\alpha_1, \ldots, \alpha_{r_t}\}$ be the set of relevant attributes for the element t and let us assume that $\texttt{@}a_1 = x_1, \ldots, \texttt{@}a_n = x_n$ are specified in the left hand side of the rule, as in

$$<t\ \texttt{@}a_1 = x_1\ \cdots\ \texttt{@}a_n = x_n> : x\ \texttt{;}$$

then the rule is expanded to

$$[\![t]\!] : <t\ \mathcal{A}(\alpha_1)\ \cdots\ \mathcal{A}(\alpha_{r_t})\ x\ t> \texttt{;}$$

where

$$\mathcal{A}(\alpha_i) = \begin{cases} <\texttt{@}\alpha_i\ x_j\ \texttt{@}\alpha_i> & \text{if } \alpha_i = a_j \\ <\texttt{@}\alpha_i\ any\_value\ \texttt{@}\alpha_i> & \text{if } \alpha_i \notin \{a_1, \ldots, a_n\} \end{cases}$$

and any_value is a non-terminal with the following productions:

$$any\_value : \varepsilon\ \texttt{|}\ \texttt{S}\ \texttt{;}$$

In other words, the start tag of an element t is *always* followed by the tokens for its relevant attributes. If the attributes are specified in the left hand side of the rule, the tokens coming from the lexer have to match the specification. Any relevant attribute that has not been specified in the left hand side of the rule must be taken into account, since the lexer will emit its tokens anyway. Thus we insert a non-terminal any_value that has the effect of ignoring them.

The special symbol `@*` matches an optional `@?` token, thus allowing an element to have zero or more attributes that are not relevant. If `@*` is missing, the element is not allowed to have attributes other than the relevant ones.

Any <t> occurring in the right hand side of a rule is simply replaced by $[\![t]\!]$, that is

$$id : x_1\ x_2\ \cdots\ \texttt{<}t\texttt{>}\ \cdots\ x_n$$

is a shortcut for

$$id : x_1\ x_2\ \cdots\ [\![t]\!]\ \cdots\ x_n\ .$$

After all the expansions have been performed, the grammar is ready to be converted into a fully Yacc-compatible one. All that has to be done is to collect the complete list of tokens used for tags and attributes, associate them with fresh Yacc tokens in the preamble of the Yacc grammar, and replace their occurrences in the rules.

3.3 Grammar well-formedness

The specification grammar shown in Table 3 must obey some conditions in order to be well-formed. The tool processing the grammar is responsible for checking these conditions:

1. namespace prefixes must be declared;
2. the right hand sides of productions associated with attribute values must not contain (either directly or as the result of the recursive expansion of non-terminal symbols) start-tag, end-tag, start-attribute, or end-attribute tokens;
3. the explicit start-tag token for an element t occurring in the right hand side of the rules must be followed (either directly or as the result of the recursive expansion of non-terminal symbols) by all the relevant attributes for t in the canonical order. Failure to do so will result in a parse error, since the lexer always emits tokens for an element to match such a sequence. This restriction might be relaxed in the future by means of a more expressive syntax for explicit start tags.

3.4 Semantic actions

It is possible to refer to semantic values associated with tokens and non-terminal symbols using the same syntax adopted by Yacc. Thus `$$` refers to the synthesized attributes associated with the non-terminal symbol on the left hand side of the rule, and `$`i refers to the attributes associated with the i-th atom (terminal or non-terminal) in the head of the rule. The index i is appropriately shifted during the expansion phase of the grammar, since atoms may be added at the beginning of the productions.

The notation `@`id can be used for referring to the semantic value associated with the XML attribute id, which must occur in the left hand side of the rule of the form

$$\texttt{<}t\ \cdots\ \texttt{@}id = x\ \cdots\ \texttt{>}$$

This notation avoids the grammar writer to worry about the set of relevant attributes for the XML element being described, and also about the canonical order which is chosen by the grammar processing tool. The tool will rewrite `@`id into the appropriate `$`i upon generation of the Yacc grammar.

3.5 A simple example

Let us consider a simple XML document representing arithmetic expressions made of constant values and the operators $+$ (n-ary addition) and $-$ (unary negation). The DTD for such documents could be

```
<!ELEMENT expr  (add | neg | const)>
<!ELEMENT add   (add | neg | const)+>
<!ELEMENT neg   (add | neg | const)>
<!ELEMENT const EMPTY>
<!ATTLIST const value CDATA #REQUIRED>
```

The following rules for a FLEAGRAM specification grammar can be used for evaluating such documents:

```
<expr>          : expr {{ printf("result: %d", $1); }} ;
expr            : <add> | <neg> | <const> ;
<add>           : add_expr {{ $$ = $1; }} ;
<neg>           : expr {{ $$ = -$1; }} ;
<const @val=S> : {{ $$ = atoi(@val); }} ;
add_expr        : expr {{ $$ = $1; }}
                | expr add_expr {{ $$ = $1 + $2; }} ;
```

This grammar would be expanded in the following form, where for clarity we have underlined references to semantic values affected by the expansion:

```
_expr_   : <expr expr {{ printf("result: %d", $2); }}
           expr> ;
expr     : _add_ | _neg_ | _const_ ;
_add_    : <add add_expr {{ $$ = $2; }} add> ;
_neg_    : <neg expr {{ $$ = -$2; }} neg> ;
_const_  : <const <@val S @val> {{ $$ = atoi($3); }}
           const> ;
add_expr : expr {{ $$ = $1; }}
         | expr add_expr {{ $$ = $1 + $2; }} ;
```

And finally the following Yacc grammar specification would be produced:

```
_expr_   : EXPR_START_TAG
           expr { printf("result: %d", $2); }
           EXPR_END_TAG ;
expr     : _add_ | _neg_ | _const_ ;
_add_    : ADD_START_TAG
           add_expr { $$ = $2; }
           ADD_END_TAG ;
_neg_    : NEG_START_TAG
           expr { $$ = -$2; }
           NEG_END_TAG ;
_const_  : CONST_START_TAG
           CONST_VALUE_START_ATTR S CONST_VALUE_END_ATTR
           { $$ = atoi($3); }
           CONST_END_TAG ;
add_expr : expr { $$ = $1; }
         | expr add_expr { $$ = $1 + $2; } ;
```

3.6 Lexing refinement

The architecture of FLEAGRAM as it has been described so far permits a form of document validation that is restricted to the document structure (the occurrence and location of element nodes, possibly annotated with attributes) but it does not permit finer forms of validation in which the textual content of attributes and `PCDATA` nodes is inspected. Conversely, other schema languages permit to restrict the textual content in more or less expressive ways. With respect to the unmarshalling problem, this limitation means that lexing and parsing of textual content must be done in the semantic actions without any sort of support from the grammar. This results in a significant burden put on the programmer since it is often the case that attributes have complicated types and consequently complex parsing rules (examples of attributes with complicated types can be found even in some standard XML applications such as MathML and SVG).

The main difficulty for overcoming this limitation derives from the fact that parsers generated by Yacc are supposed to work with one lexer only (more specifically, the lexer as seen from the parser has a single entry point). By allowing the content of attributes and of text nodes to be scanned, we need to be able to switch between different lexers, by inserting appropriate semantic

Table 4. Mapping DTD element declarations to FLEAGRAM.

DTD	FLEAGRAM specification
`<!ELEMENT x EMPTY>`	`<x> : ε ;`
`<!ELEMENT x (y | z)>`	`<x> : <y> | <z> ;`
`<!ELEMENT x (y+)>`	`<x> : y ; y : <y> | <y> y ;`
`<!ELEMENT x (y*)>`	`<x> : y ; y : ε | <y> y ;`
`<!ELEMENT x (y?)>`	`<x> : ε | <y> ;`
`<!ELEMENT x (#PCDATA | y)*>`	`<x> : y ; y : ε | S y | <y> y ;`
`<!ELEMENT x ANY>`	`<x>:` $\langle rel\_elt_1 \rangle \mid .. \mid \langle rel\_elt_n \rangle$ `;`

actions that imperatively select the lexer to be used from a certain point, and that restore it later on.

Switching lexers has potentially disruptive effects because of the way the Yacc-generated LALR(1) parsers work. Roughly speaking, in these parsers a finite state automaton keeps track in its states of *all* the productions in the grammar which are compatible with the sequence of tokens seen up to a certain point: productions in which this "point" is marked are called LR(1) items. If the lexer is switched so as to enable the scanning of the value for an attribute occurring in some LR(1) item, any other item in the same state of the automaton will be affected by the sequence of tokens coming from the lexer from there on. However, since attributes are associated with elements, since the token for the start tag of each element is emitted before any attribute, since tokens for relevant attributes are always emitted by the lexer in the same order, one can verify that states containing LR(1) items where the point is after a `<@`id token only contain productions regarding the same id attribute, and that all the items are marked just after the start tag for that attribute. In other words, as long as the lexing rules for an attribute do not depend on the value of other attributes, the lexer can be safely switched.

4 Comparison with other schema languages

Specification grammars described in Section 3 establish a set of syntactic rules that parsed document must obey. It is interesting to observe that the FLEAGRAM specification grammars implicitly define a schema language. In this section we compare the expressive power of FLEAGRAM specifications with those of some of the most widespread schema languages.

4.1 Document Type Definitions (DTDs)

DTDs [9] represent the most basic and widespread schema language for XML documents. DTDs are made of *element type declarations*, which constraint the content model of elements, and *attribute list declarations*, which constraint the set (and the content) of attributes that can be used for a given element.

As Table 4 shows, the constraints expressed by element type declarations can be expressed using FLEAGRAM productions. While DTDs require the content model of XML elements to be deterministic,[5] this restriction is dropped in FLEAGRAM as LALR parsers do not need it. Note that the rule corresponding to the `ANY` content model, where rel_elt_i is the i-th relevant element, can be automatically generated by FLEAGRAM.

With the lexer refinement described in Section 3.6, FLEAGRAM can express all the syntactical constraints about attributes that are possible in DTDs. What FLEAGRAM specification grammars are not able to enforce are referential integrity checks (in the DTDs these involve the `ID` and `IDREF` attribute types). Note however that these checks belong to the semantic side of the validation phase and are thus best implemented in the semantic actions.

5. Roughly speaking, this means that in a content model of the form $(x \mid y)$ it must be possible to determine whether the instance document must match the content model x or y, using a lookahead of just one element. For example, the content model $((b, c) \mid (b, d))$ is non-deterministic, while $(b, (c \mid d))$ is deterministic.

In DTDs, both attribute and element declarations are *closed specifications*: anything that is not explicitely allowed is forbidden. FLEAGRAM supports open specifications as well: `@*` can be used to allow the presence of non relevant attributes, and `<?, ?>` tokens can be used, possibly in conjunction with recursive productions, to allow non-relevant elements inside a given one.

4.2 XML Schema

XML Schema [10, 11] is meant to become the official schema language for XML documents. From the point of view of element content models, XML Schema augments DTDs expressiveness with *interleaving* and *local definitions*. The interleaving operator `<all>` states that an element must contain a set of elements regardless the order in which they appear. Efficient validation of this operator is still an open research topic and interleaving cannot be expressed conveniently in LALR(1) grammars. In principle we can augment FLEAGRAM with a production of the form

$$\langle production \rangle ::= \texttt{interleave}\ \{\ \langle atom \rangle\ \}^{+}$$

that would be expanded into a set of rules with all the permutations of the atoms. This would give more expressive power than XML Schema itself since no restriction on the interleaving content would be present, but it could likely cause an explosion in the number of states of the LALR parser.

Local definitions allow an element to have different content models depending on the context in which the element occurs. For example, it is possible to require an element `<x>` to have content model `<y>` when `<x>` occurs within a `<a>` element, and to have content model `<z>` when `<x>` occurs within a `<b>` element. In FLEAGRAM, contextual rules can be encoded in the right hand side of productions. The above example would be trivially encoded using the rules

```
<a> : <x <y> x> ;
<b> : <x <z> x> ;
```

Another major contribution of XML Schema is the definition of a hierarchy of simple types that enable fine-grained validation of attributes and text nodes; each simple type is characterized by a value space (the set of values of a datatype) and a lexical space (the set of textual representations of datatype values). A mechanism of inheritance is provided to derive new types from simpler ones. Three kinds of type derivation are provided: *by list* (creating types whose lexical spaces are blank separated lists of another type literals), *by union* (sum types) and *by restriction* (selecting a subset of the value and/or lexical space).

Assuming that lexers are available for all XML Schema primitive types, type derivations by *list* and *union* can be implemented by ordinary grammar productions, using the lexer refinement of Section 3.6. FLEAGRAM is able to express all constraints that restrict the lexical space of types, while the validation of types derived restricting the value space (for example the restriction "the value must be strictly less than a given constant") can be only performed in semantic actions.

4.3 Relax NG

In Relax NG [6], valid documents are described by a set of *patterns*, where each pattern is a regular expression over the content model of an element. The distinctive aspect of Relax NG is that patterns treat attributes and elements uniformly: they are both seen as having a common parent element. This feature permits to express some forms of *co-constraints*, that is constraints dependent on other constraints, which cannot be expressed using DTDs or XML Schemas. Examples of co-constraints that can be expressed in Relax NG are:

1. *the `<a>` element must have either a `b` attribute or a `<b>` child element*;
2. *the `<a>` element must have a `b` attribute; if `b` value is `foo`, `<a>` must have a `<c>` child element, otherwise it must have a `<d>` child element*.

FLEAGRAM specifications can express co-constraints except those depending on attribute or element values. For instance, the co-constraint (1) above is encoded in FLEAGRAM as follows:

```
<a @b = S> : ε ;
<a @b = ε> : <b> ;
```

Once more, FLEAGRAM recognition expressiveness stops at the syntactic level; semantic validation of values is only performed in semantic actions and does not interfere with parsing.

5 Concluding remarks

Tools that relate Yacc parsers and XML technologies are not infrequent. While most of them are compilers or conversion utilities for which XML is the target format (see [2]), FLEAGRAM goes into the opposite direction, despite the fact that XML documents are commonly thought to be "parsed". In this paper we claim that this is a misconception and that LALR(1) parsing of XML documents provides tangible advantages. Our aim is not to compete with complex schema languages like W3C Schemas or Relax NG. These techniques go far beyond the syntactic well-formedness of documents and allow one to specify semantic constraints on the content as well as the structure of documents. In the well-established frameworks that separate the lexing, parsing, and analysis phases, syntactic and semantic validity are verified by different means. While FLEAGRAM provides some flexibility in the kind of syntactic constraints that can be checked, for example by enabling some limited forms of context dependencies, it leaves any semantic check to a subsequent phase that makes use of the unmarshalled document, which might well provide more appropriate ways for ensuring its consistency.

In order to verify that our approach does not entail computational overhead, we have made some comparative tests between FLEAGRAM and other validation techniques. As test data we have chosen two classes of documents: the first class comprises deeply structured documents having a number of nodes ranging from nearly 2,000 up to more than 800,000, their depth ranging from a dozen up to nearly 600 nested nodes.[6] The second class of documents comprises artificially made documents with a flat structure (no more than 3 nested nodes) and with a total number of nodes ranging from 10,000 up to 900,000. The results can be seen in Figure 2 and Figure 3. As the implementation library we chose LIBXML2 for a number of reasons: first, LIBXML2 supports DTDs, XML Schema (at least partially) and Relax NG, hence it allows a quick comparison of some of the most significant schema languages within a single library; second, LIBXML2 is acknowledged as one of the most performant XML implementations available;[7] third, FLEAGRAM currently uses the LIBXML2 SAX parser, so it is fair to disregard the parsing time knowing that, in every case, the LIBXML2 parser does no more and no less in order to favor a subsequent validation phase. The tests show that FLEAGRAM validation time is smaller with respect to the other techniques.

FLEAGRAM permits the generation of self-contained XML parsers that allow for simultaneous validation and unmarshalling without requiring the creation of expensive XML document trees in main memory. The SAX parser takes care of the lexical analysis of the input document, relieving the programmer from all the dirtiness involved in XML processing (input/output, character encoding, namespace management, character entities, and so on). The syntactical correctness of the document is verified by a generated automaton that works on a stream of events. Semantic actions can be comfortably associated with patterns in a sole specification, the unmarhsalling phase goes along with the structural validation.

The advent of XML seemed to settle down issues with document parsing, but that was not entirely the case: connecting parsing and subsequent processing is a fundamental issue that has not been addressed in a satisfactory way so far. FLEAGRAM addresses it in a simple yet effective way.

---

6. These documents were not generated for the purposes of this test. They encode terms in a richly typed λ-calculus representing formal mathematical proofs, see `http://helm.cs.unibo.it/`.
7. See `http://xmlbench.sourceforge.net/` for an extensive set of comparisons

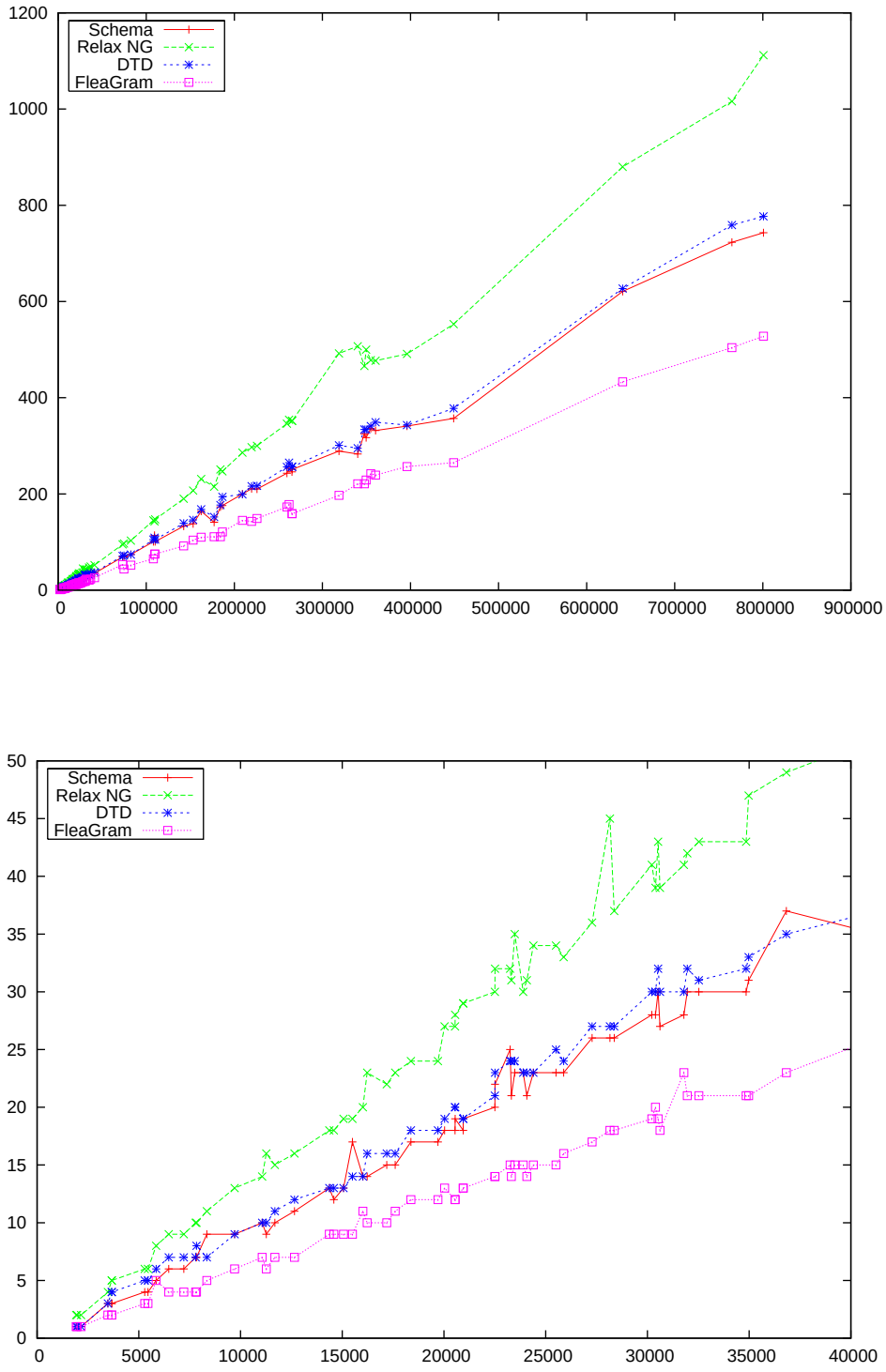

Figure 2. Validation times of *deeply structured* XML documents with an increasing number of nodes for different schema languages. The graph at the bottom is the zoomed version of the one on top, for documents up to 40,000 nodes.

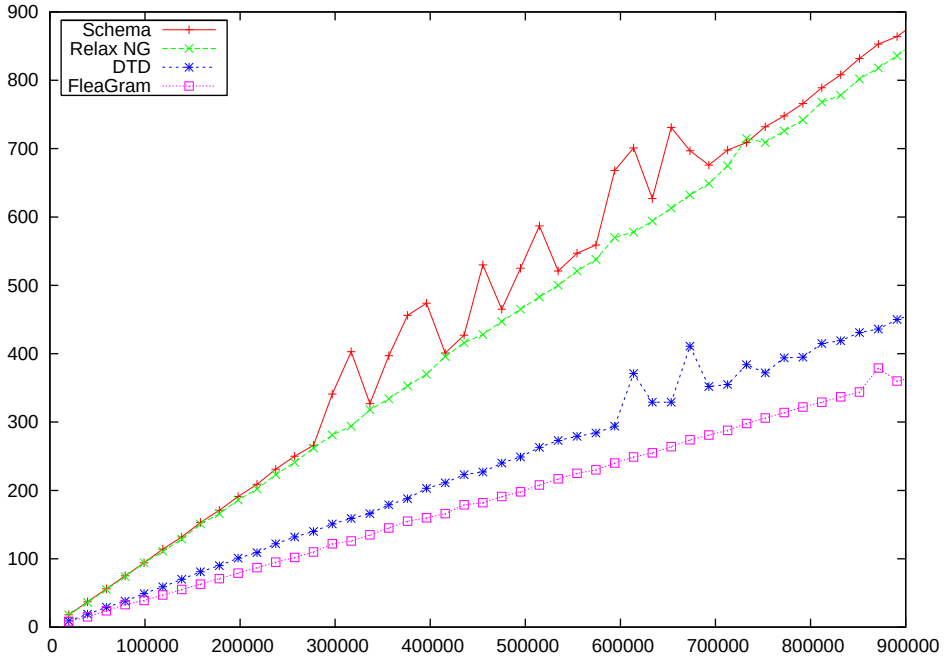

Figure 3. Validation times of *flat* XML documents with an increasing number of nodes for different schema languages.

References

[1] Aho A. V., Sethi R., and Ullman J. D. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] Alexander Nakhimovsky. Using parser-generators to convert legacy data formats to xml. In *Proceedings of the XML Europe 2001 Conference*, 2001. http://www.gca.org/papers/xmleurope2001/papers/html/s07-3.html.

[3] Casarini P. and Padovani L. "The Gnome DOM Engine". *Markup Languages: Theory & Practice*, 3(2):173–190, Apr. 2002.

[4] S. C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[5] Padovani L., Sacerdoti Coen C., and Zacchiroli S. A generative approach to the implementation of language bindings for the document object model. In *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*, volume 3286 of *LNCS*, pages 469–487, 2004.

[6] Relax ng specification. Committee Specification, Dec. 2001. http://www.relaxng.org/spec.html.

[7] The sax project. http://www.saxproject.org/.

[8] Document object model (dom) level 2 core specification. W3C Recommendation, Nov. 2000. http://www.w3.org/TR/DOM-Level-2-Core/.

[9] Extensible markup language (xml) 1.0 (2nd edition). W3C Recommendation, 2000. http://www.w3.org/tr/2000/rec-xml-20001006.

[10] Xml schema part 1: Structures. W3C Recommendation, Oct. 2004. http://www.w3.org/TR/xmlschema-1/.

[11] Xml schema part 2: Datatypes. W3C Recommendation, Oct. 2004. http://www.w3.org/TR/xmlschema-2/.