Adding Semantics to XML

Giuseppe Psaila[1] and Stefano Crespi-Reghizzi[1,2]

1: Politecnico di Milano, Dipartimento di Elettronica e Informazione
P.za Leonardo da Vinci 32 - 20133 Milano, Italy
psaila,crespi@elet.polimi.it
2: CNR-Cestia - P.za Leonardo da Vinci 32 - 20133 Milano, Italy

Abstract

Starting form the analogy between a document tagged by a mark-up language (XML, SGML) and a source string generated by a BNF grammar, we argue that XML parsers should benefit from the addition of semantic attributes and functions. Currently XML only includes initialized lexical attributes. By our approach a XML parser would be extended into a syntax-directed translator. Deep transformations of a document could be specified, sent over the network, and executed within the XML system. For the specification of the semantic attributes and functions we propose a XML Document Type Definition, that is conceptually similar to the metalanguage of a compiler-compiler. By this approach the additions to the XML standard are kept to a minimum. The differences between attribute grammars and attributed XML specifications are discussed, and the system architecture of a semantic evaluator generator is presented.

1. Introduction

In the production and dissemination of documents and hypertexts on the Web, the trend towards mark-up languages is undeniable. The hypertext mark-up language HTML is of course the best known example, but other important cases of marking-up occur for instance in computational linguistic, where efforts have been made to standardize the format of *language resources* (see TEI [20]). But what is the reason for discussing mark-up languages in a specialized workshop on attribute grammars? The straightforward, though perhaps not perspicuous, reason is that specific mark-up languages such as HTML are conceptually similar to a BNF grammar with very limited attribution. HTML is of course just an instance of a *document type*, designed at CERN for disseminating web pages. Many different document types have been and are being defined for other purposes, as the already mentioned TEI. The meta-notation for specifying document types and their grammars is provided by the *Standard Generalized Mark-Up Language* (SGML) (see [9]) and more recently by its simplified and upgraded version, *Extensible Mark-up Language* (XML) (see [4, 18]).

Then the analogy between compiler and mark-up approaches is resumed in the table:

Level	Compiler Technology	Mark-up Technology
Meta-notation	Meta languages for specifying AG's *Aladin* [15],*Olga* [10] and *GRAL* [5]	SGML or XML
Formal syntax	context-free, extended BNF	Document Type Definition

The syntactic meta-notation of XML is essentially equivalent to an extended BNF, i.e. a context-free syntax with regular expressions in the right parts. Incidentally SGML is more powerful than XML in that it allows to specify two or more syntax trees for the same document, a concept similar to using two or more grammars for the same source language.

On the other hand the semantic part of mark-up specifications is very poor. The attributes which can be associated to any "element" (such as title, section, etc.) are sort of initialized lexical attributes, belonging to a few predefined domains: string, finite enumeration, identification. The notion of semantic function is completely absent from mark-up specifications, hence there is no distinction between inherited and synthesized attributes.

As a consequence, XML document type specifications can only be used to check the syntactic validity of a document, but are of no help for specifying transformations, such as routinely done by syntax-directed translators.

This serious weakness of XML has the undesirable consequence that application writers have to implement document transforming software using ad hoc techniques, which hinder portability, reuse, and extensibility.

Coming to the point of this research, we propose to extend XML with semantic definitions including declaration of attributes and semantic function specification. This approach is intended to exploit the consolidated know-how of compiler writing systems in the area of network document processing.

But rather than adopting a traditional meta-notation used in compiler-compilers, we have prefered to comply with the mark-up approach: semantic specifications are cast into the mould of a new document type definition. Extensions to official XML standards are thus minimized.

Considering attribute evaluation, interesting perspectives are opened for matching multi-sweep algorithms with cascaded document types, representing partially decorated documents.

The paper proceeds with a running example, then in Section 2 it introduces XML and compares it with BNF metalanguage. Section 3 presents the proposed semantic specifications and outlines the system architecture of a mark-up based, syntax-directed compiler-compiler. Section 4 presents the semantic rules notation and discusses possible implementations using attribute evaluation techniques. Section 5 concludes and mentions future developments.

1.1. Running Example

In the rest of the paper, we illustrate the concepts by means of a simple running example.

Consider an XML language designed to specify structured textual documents. A document has a *name*, and is composed of a *title*, an arbitrary number of *sections*, an optional *appendix*, a *bibliography* at the end of the document. A section has a *title* and is composed of *paragraphs* which contain text. The appendix is in turn composed of sections. The bibliography is composed of *bibliographic items*, which are identified by a *label* A bibliographic item can be cited within paragraphs, by means of *citations*, which refer to the labels of the bibliographic items.

For example, consider the sample document reported in Figure 3, which is obtained nicely formatting the XML source document reported in Figure 2. We can imagine that an ad-hoc XML processor has formatted the source document performing the following tasks:

- numbering sections and bibliographic citations;
- resolving citations to bibliographic items and references to section labels;
- properly formatting the document based on a precise output style.

For example, observe that in the section number 2 the citation of Knuth's paper gives the number of the line which describes the desired paper in the bibliography.

We chose this example for its simplicity, but the proposed technique is really meant for deeper semantic operations, such as database processing ([17]).

2. Introducing XML

In this section, we introduce the basic XML features which are relevant for this paper by means of the running example.

In XML, each constituent corresponds to an *element*. An element is a marked up portion of document, delimited by a *start tag* and an *end tag*. For example, a paragraph element assumes the form

`<PAR>` *generic content* `</PAR>`,

where `<PAR>` is the start tag, and `</PAR>` is the end tag. There can exist *empty elements* as well, i.e. elements without content. An example can be a citation such as

`<CITE label="Knuth68"/>`

where the citation does not have any content, since it refers to a bibliographic item.

Elements can have *attributes*, i.e. named properties with a given value. For example, the attribute `label` appearing in the above citation indicates that the citation refers to a bibliographic item labeled `"Knuth68"`. Attributes are allowed to appear only in start and end tags. We call these attributes *Extensional Attributes*, since they are explicitely provided in the source document; they differ from the *Intensional Attributes*, which are object of the semantic evaluation and that will be discussed later (Section 4).

Non-empty elements can contain generic text or other elements. This is the case of paragraphs and citations: a paragraph can contain generic text or a citation pointing to a bibliographic item.

The type, structure and attributes of elements are defined in XML by the so called *Document Type Definition* (DTD): by means of suitable *meta-tags*, XML allows the specification of the syntactic structure of documents. The two main meta-tags of a DTD are the following:

- the meta-tag `!element` defines the content of elements; by means of a regular expression, it specifies the occurrences of other elements as children of the defined element.
- the meta-tag `!attlist` defines the attributes associated to an element. For each attribute, it is possible to specify if it is required or optional (`#REQUIRED` or `#IMPLIED`, respectively), the default value and the domain of the attribute, which can be a generic string (denoted as `CDATA`) or an enumeration of allowed values.

  In place of the enumeration or the `CDATA` keyword, it is possible to use the `#ID` and `#IDREF` keywords. The former specifies that the attribute is the unique identifier of the element; the latter refers to the identifier of another attribute. In XML, identifiers are global, i.e. there cannot be two distinct elements identified by the same value.

Figure 1 reports the DTD for the class of structured textual documents described in Section 1.1.

Let us discuss the DTD in details.

- The element `DOCUMENT` defines the overall document. It contains an element `TITLE`, zero or more elements `SECTION`, an optional element `APPENDIX`, an element `BIBLIOGRAPHY`. The sequence of elements allowed as content is defined by the regular expression

  `((TITLE,(SECTION)*,(APPENDIX)?,BIBLIOGRAPHY))`.

  The attribute `name` associated to a `DOCUMENT` denotes the name of the document.
- The element `TITLE` allows only text as content. This is specified by the keyword `#PCDATA`.

```
<!-- The DTD -->
<!element BIBITEM (#PCDATA)>
<attlist BIBITEM label ID #IMPLIED>

<!element BIBLIOGRAPHY (BIBITEM)+ >

<!element CITE EMPTY >
<!attlist CITE label #IDREF #REQUIRED >

<!element PAR (#PCDATA | CITE)+ >

<!element SECTION (PAR)* >
<!attlist SECTION title CDATA #REQUIRED
                  label #ID #REQUIRED >

<!element APPENDIX (SECTION)+ >

<!element TITLE (#PCDATA) >

<!ELEMENT DOCUMENT (TITLE,(SECTION)*,(APPENDIX)?,BIBLIOGRAPHY) >
<!attlist DOCUMENT name CDATA #REQUIRED>
```

Figure 1: The DTD for structured documents

- The element SECTION contains a possibly empty sequence of paragraphs, denoted by the element PAR. A section has a title, which is specified by the attribute title. A section has associated a label, denoted by the attribute label, which is the unique identifier (#ID) of the element.
- The element APPENDIX can contain a non-empty sequence of SECTIONs.
- A paragraph (element PAR) is a non-empty sequence of generic text content and citations (element CITE).
- The element BIBLIOGRAPHY can contain a non-empty sequence of bibliographic items (element BIBITEM.
- A bibliographic item (element BIBITEM) can contain only generic text. It has associated the attribute label, which is the unique identifier (#ID) of the element; this attribute is defined as #IMPLIED, meaning that its presence is optional. In a compliant document (see Figure 2) the value of each instance of label must be provided.

Example 1: Figure 2 reports a sample marked up source document, based on the DTD of Figure 1, while Figure 3 shows its appearance after formatting.

Observe the corrispondence between the mark-ups and the graphical features. In particular, in the formatted document the numbers of sections have been automatically generated, as well as citations which refer to the line number in which the referenced article appears in the bibliography. □

DTD and Extended BNF. Regular expressions which appear in element definitions suggest that the syntax of XML documents can be described by means of an Extended BNF grammar. In effect, it is easy to map a set of element definitions appearing in a DTD, into an equivalent, E-BNF syntax.

```
<?xml version="1.0"?>
<!DOCTYPE docs SYSTEM "docs.dtd">

<DOCUMENT name="sample doc">
   <TITLE>
      Using bibliography citations.
   </TITLE>

   <SECTION title="Introduction">
      <PAR>
         This document shows how to use a hypothetical XML language
         to describe structured documents which exploit
         the concept of citation
      </PAR>
   </SECTION>
   <SECTION title="A Citation">
      <PAR>
         Here we have an example of citation. We refer to
         Knuth's paper which introduced the concept of attribute
         grammar.
         This paper has the number <CITE label="Knuth68"/>
         in our bibliography.
      </PAR>
   </SECTION>

   <APPENDIX>
      <SECTION>
          This is an example of appendix added to the document.
      </SECTION>
   </APPENDIX>

   <BIBLIOGRAPHY>
      <BIBITEM label="ASU85">
         A.V. Aho, R. Sethi, J. D. Ullman,
         "Compilers: Principles, Techniques, Tools",
         Addison-Wesley, 1985.
      </BIBITEM>
      <BIBITEM label="Knuth68">
       <!-- Instance of the extensional attrribute "label" -->
         D. E. Knuth, "Semantics of Context Free Languages",
         Mathematical System Theory, Vol. 2, pp. 127-145, 1968.
      </BIBITEM>
      <BIBITEM label="Lorho84">
         B. Lorho, "Methods and Tools for Compiler Construction",
         Cambridge University Press, 1984.
      </BIBITEM>
   </BIBLIOGRAPHY>

</DOCUMENT>
```

Figure 2: The structured document of the running example.

Using Bibliography Citations.

1 Introduction

This document shows how to use a hypothetical XML language to describe structured documents which exploit the concept of citation

2 A Citation

Here we have an example of citation. We refer to Knuth's paper which introduced the concept of attribute grammar. This paper has the number [2] in our bibliography.

APPENDIX

A1 An Example of Appendix

This is an example of appendix added to the document.

BIBLIOGRAPHY

1. A.V. Aho, R. Sethi, J. D. Ullman, "Compilers: Principles, Techniques, Tools", Addison-Wesley, 1985.

2. D. E. Knuth, "Semantics of Context Free Languages", Mathematical System Theory, Vol. 2, pp. 127-145, 1968.

3. B. Lorho, "Methods and Tools for Compiler Construction", Cambridge University Press, 1984.

Figure 3: The document after processing

We now provide a set of rules to translate a DTD into an equivalent E-BNF specification. In order to avoid confusion, we use in the E-BNF the same symbols adopted in XML regular expressions; in particular, `( ... )*` denotes zero or more repetitions, `( ... )+` denotes one or more repetitions, `( ... )?` denotes optionality, `|` denotes alternative.

1. **Non-Empty Elements.** For each meta-tag defining a non-empty element E, generate an E-BNF production such as

$$
\begin{array}{lll}
E & \rightarrow & \texttt{"<"}\ \texttt{"}N\texttt{"}\ E\_Attrs\ \texttt{">"} \\
 & & RegularExpression \\
 & & \texttt{"</}E\texttt{>"}
\end{array}
$$

   E plays the role of nonterminal father of the production, where $RegularExpressions$ is the regular expression appearing in the meta-tag defining E, and E_Attrs is a non-terminal symbol (see below).

2. **Empty Elements.** For each meta-tag defining an empty element E, generate an E-BNF production such as

$$
E \rightarrow \texttt{"<"}\ \texttt{"}N\texttt{"}\ E\_Attrs\ \texttt{"/>"}
$$

   E plays the role of nonterminal father of the production and E_Attrs is a non-terminal symbol.

3. **Attribute Definitions.** For each definition of attributes $a_1 \ldots a_n$ associated to an element E, generate an E-BNF production such as

$$
E\_Attrs \rightarrow Def-of(a_1) \ldots Def-of(a_n)
$$

   where E_Attrs is the nonterminal appearing as child in the production defining E, $Def-of(a_i)$ denotes the actual regular expressions defining the syntax of attributes.

Document	→	"<" "DOCUMENT" DocumentAttrs ">"
		Title (*Section*)* [*Appendix*] *Bibliography*
		"</DOCUMENT>"
DocumentAttrs	→	"name" "=" *CData*
Title	→	"<" "TITLE" TitleAttrs ">" *PCData* "</TITLE>"
TitleAttrs	→	ϵ
Section	→	"<" "SECTION" SectionAttrs ">" (*Par*)* "</SECTION>"
SectionAttrs	→	"title" "=" *CData*
		"label" "=" *CData*
Appendix	→	"<" "APPENDIX" AppendixAttrs ">" (*Section*)+ ("</APPENDIX>")?
AppendixAttrs	→	ϵ
Par	→	"<" "PAR" ParAttrs ">" (*PCData* — *Cite*)+ "</PAR>"
ParAttrs	→	ϵ
Cite	→	"<" "CITE" CiteAttrs "/>"
CiteAttrs	→	"label" "=" *CData*
Bibliography	→	"<" "BIBLIOGRAPHY" BibliographyAttrs ">" (*Bibitem*)+
		"</BIBLIOGRAPHY>"
BibliographyAttrs	→	ϵ
Bibitem	→	"<" "BIBITEM" BibitemAttrs ">" *PCData* "</BIBITEM>"
BibitemAttrs	→	("label" "=" *CData*)?

Figure 4: E-BNF Syntax derived from the DTD of Figure 1.

$Def - of(a_i)$ are defined as follow:

If the attribute is, say, of domain CDATA and *required*, generate a regular expressions such as

"a_i" "=" *CData*

where a_i denotes the attribute name, while *CData* is a terminal symbol.

If the attribute is *implied*, generate a regular expressions such as

("a_i" "=" *CData*)?

where a_i denotes the attribute name, while *CData* is a terminal symbol which corresponds to string constants.

Observe that attributes defined as #ID or #IDREF are in effect generic string attribute, hence implicitely defined as CDATA. The properties of the attributes defined as #ID or #IDREF (i.e. uniqueness for the former and existence of the referenced element for the latter) cannot be specified by syntax.

4. **Elements without Attributes.** For each element definition without the corresponding definition of attributes, generate the E-BNF production

   $E\_Attrs \rightarrow \epsilon$

   where E is the element without attribute definition.

For example, performing such a mapping on the DTD of Figure 1, we obtain the E-BNF syntax in Figure 4.

Notice that the attributes are mapped onto terminal items of the E-BNF grammar. The values of the attributes can be viewed as initialized semantic attributes. We will call them *Extensional Attributes*.

Also note that the mapping from DTD to E-BNF syntax shows that XML languages are deterministic context-free and can be parsed by a LL(1) parser.

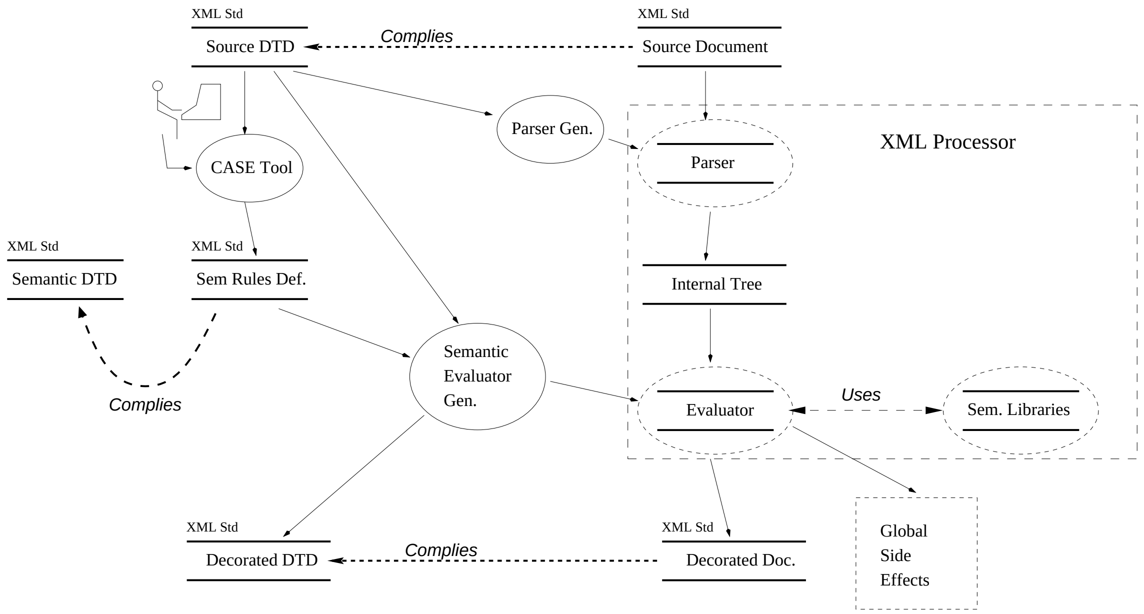

Figure 5: Data Flow Architecture of the Semantically Enriched XML Processing.

3. Semantic Processing Architecture

Before describing how semantic rules can be added to an XML DTD, we propose an architecture for semantic processing with XML. The choices made for defining the architecture are the result of our analysis of the applicative scenario for our proposal.

3.1. Applicative Scenario

The applicative scenario is characterized by two main factors:

1. XML will be mostly used as a document exchange format over internet. Thus, XML documents will be mostly processed by browsers.
2. XML can be used for any kind of information and document. The classical use is represented by textual documents, but XML can be successfully used for other issues, such as defining the schema of an object-oriented database and the objects stored in it (see *XML-Data* in [17]).

Consequently, this led us to define the following requirements:

- The automatically generated *XML processor* should recognize documents based on a standard DTD; this fact ensures portability of documents at least as far as the syntactic structure is concerned.
- The XML processor evaluates the semantics of the processed document.
- The XML processor should produce an XML document which is obtained by the original document by adding the semantic information obtained during the semantic evaluation. This requirement makes the XML processor interoperable with other XML processors: in fact, the output documents might be used for other process needs (e.g. visualization of the semantic properties).

- The XML processor might perform actions as side effects; this might be necessary when the content of the processed document must be used to drive additional devices or systems (e.g. an XML-Data specification is used to drive a DBMS).

3.2. The Processing Architecture

The processing architecture matches the above requirements. The architecture, reported in Figure 5, is based on data-flow diagrams, since we want to analize how documents and information are processed by tools and programs.

In particular, we use the following notation: tools are denoted by labelled ovals; documents and any kind of static information is represented by two parallel horizontal bars with a label in the middle; generated programs are represented as a dashed oval which contain two parallel bars with a label in the middle. Furthermore, solid arrows starting from a document and entering a tool (program), indicate that the document is taken as input by the tool (program); solid arrows starting from a tool (program) and entering a document (generated program) indicates that the document (program) is the output of the tool (program). Finally, dashed arrows labelled as *Complies* starting from a document and entering a DTD, indicate that the document complies with the syntactic rules specified by the DTD.

Let us discuss the architecture in detail.

- **Source DTD.** The source DTD (Document Type Defintion) contains the syntax specification of the XML language to be processed. It is a standard DTD, as the one in Figure 1.

- **Semantic Rules Definitions.** An additional document contains SRDs (Semantic Rules Definitions). This document integrates the DTD, specifying the semantic rules for each element defined in the DTD. A SRD is an XML document. An example is Appendix B.

  A SRD defines the set of *Intensional Attributes*, i.e. semantic attributes which are object of the semantic evaluation, and *Semantic Rules*, i.e. rules which derive the intensional attributes.

- **Semantic DTD.** We said that a SRD is itself an XML document; hence, the document containing a SRD complies with a basic DTD that is called *Semantic DTD* (reported in Appendix A). Observe that the Semantic DTD is fixed, i.e. it defines the structure of documents containing a SRD; furthermore, the semantic evaluator generator parses the Semantic DTD based on this definition; however, the semantic DTD is not in the processing flow.

- **CASE Tool.** A CASE tool assists the user in defining the Semantic Rules Definitions. Since a SRD is based on definitions appearing in the source DTD, the tool takes the source DTD as input, and, interacting with the user, generates the desired SRD. The tool reduces the clerical work caused by prolixity of XML specifications.

- **Parser Generator.** The tool *Parser Generator* analizes the source DTD and generates a parser for documents which comply with the source DTD. Observe that any of the developed XML parsers (see [21] for an extended listing) can be considered.

- **Semantic Evaluator Generator.** The tool *Semantic Evaluator Generator* processes the source DTD and the semantic rules definitions. As a result, it generates a semantic evaluator and a new DTD, named *Decorated DTD*.

- **Decorated DTD.** The decorated DTD contains the syntax specifications of the class of XML documents generated by the semantic evaluator. This new DTD is obtained by enriching the source DTD with elements that describe the intensional attributes computed by the semantic evaluator (more of that in Section 4). Observe that the decorated DTD is not used in the processing chain, but it is prepared for later process needs.

- **Semantic Libraries.** The definition of semantic rules can exploit the availability of basic operators for manipulating basic and complex data types supported by the semantic evaluator. However, the designer might require more complex or specific capabilities which are provided by external libraries, called by the evaluator. Such libraries are user provided, and are part of the final XML processor as well.

- **The XML Processor.** The union of Parser, Semantic Evaluator and Semantic Libraries constitutes the final *XML Processor* for the XML documents which comply with the source DTD.

- **Parser.** The generated parser performs the syntactic analysis of the source document. It produces an *Internal Tree*, i.e. a tree data structure which represents the source document. This could of course be represented as a marked-up text (this choice is open).

- **Semantic Evaluator.** The generated semantic evaluator applies semantic rules to the internal tree, evaluates intensional attributes and decorates the tree. It produces a new version of the document, which is called the *Decorated Document*.

- **Decorated Document.** The decorated document is obtained extending the source document with the intensional attributes evaluated by the semantic evaluator. This document complies with the Decorated DTD.

- **Side Effects.** The XML Processor can produce *side effects*. These can be intended as actions performed on other devices or systems, for example the generation of a formatted and printable document (as in our example), or the interaction with a Data Base Management System (as for XML-Data), etc..

Discussion. The interoperability requirement led us to define an XML processor that produces the Decorated Document. In the simplest case, all the extensional and intensional attributes appear in the decorated document, but this is not necessarily the case. For example, there might be attributes which are simply used to evaluate those attributes which describe really relevant properties of the document. Extensional attributes too might become not relevant after semantic processing. Hence, one might be interested in obtaining a decorated document with only the relevant attributes.

Consequently, we considered how to specify which are the (extensional and intensional) attributes to be kept, or (which is the same) the (extensional and intensional) attributes to be discarded. We identified three possible solutions.

- **Last Sweep.** The semantic evaluator generator analizes the semantic rules and determines the proper multi-sweep strategy (see [8]) for evaluating intensional attributes. Making the hypothesis that the relevant attributes are those evaluated in the last sweep, the generated semantic evaluator produces a decorated document with attributes of the last sweep only.

- **Directives in the Semantic DTD.** The designer of the SRD decides which attributes (both extensional and intensional) actually decribe relevant semantic features of the document and which attributes (both extensional and intensional) are only necessary to drive the evaluation of the relevant ones. By suitable directives, the designer specifies attributes (both intensional and extensional) to be discarded/kept in the decorated document.

- **Interaction between User and Evaluator Generator.** A third strategy that might be used by the semantic evaluator generator to identify the set of attribute relevant for the decorated document, is by interacting with the user. By means of a suitable interface, the tool interacts with the user, asking for the set of relevant attributes. This way, the user can perform several evaluator generation steps, based on the same SRD, which differ in the set of relevant attributes.

Observe that in all of the three cases, the decorated DTD is automatically generated in order to provide the XML features (elements and/or attributes) carrying the relevant intensional and extensional attributes.

4. Semantic Rules Definition

In this section we show the definition of semantic rules. The listing of the running example is in Appendix B.

The first question to answer is the following:

Meta-level or document-level for semantic rules definitions ?

The meta-level solution would have the advantage that the semantic rules are homogeneous with the syntactic rules, which are described in the DTD. However, this approach requires some extensions to the XML meta level.

The alternative course is to denote the semantic rules as an XML document, which complies with an ad hoc DTD. This approach was taken by DCD (Document Content Description, see [3]), a proposal to describe the structure and contents of documents using an XML document type. The advantage of this solution is that the XML meta-level does not have to be extended, thus the semantic rules definition still complies with the XML standard.

We have decided to describe the semantic rules in the style of an XML document.

4.1. Attributes

As introduced in Section 2, the concept of *attribute*, as present in XML, refers to properties of element instances. From the point of view of attribute grammars (AGs), such attributes can be viewed as lexical attributes, which are directly obtained from the parsing of the document.

We identified two categories of attributes.

- **Extensional Attributes.** They are defined in the DTD and specified in the document by the document writer. They constitute the bases for the semantic evaluation.
- **Intensional Attributes.** They are the targets of the semantic rules, which derive them from the extensional and possibly other intensional attributes.

Attribute Types. Extensional attributes in XML are always string typed attributes. This comes from their definition in the DTD: in fact, there can be attributes defined as `CDATA`, i.e. a generic string, or attributes whose domain is an enumeration of string values. Furthermore, attributes defined both as `#ID` and as `#IDREF` are string valued too (see for example the sample source document of Figure 2, where the attribute `label` of a `BIBITEM` element is its identifier).

In contrast, intensional attributes, should represent structured information, thus they must be based on a more complex type system. We identified the following list of types.

- `Integer` and `Real`: integer and real valued attributes are fundamental in attribute grammars, for instance to count repeated occurrences of a given element (e.g. section numbering).
- `String`: strings are the basic type for XML extensional attributes, then this type should be available for intensional attributes too.
- `Record`: in many cases, the information managed by an AG are complex. The basic construct is the record.

- `Set`: in many cases, semantic rules have to deal with lists or sets, such as the set of labels appearing in bibliographic items.
- `Token`: often, semantic rules compute side effects (e.g. translation into another language or interaction with external devices or systems), and dependencies between attributes determine the correct sequence of side actions performed by semantic rules. In order to clearly distinguish such *side effect rules* from rules which actually derive an attribute, the special type `token` is introduced (see [14]).

This is a minimal, yet necessary, set of intensional attribute types, to be reconsidered in future work.

Attribute Type Definition. The meta-language to be sketched for Semantic Rules Definition (SRD), provides constructs to define complex data types based on record and/or sets. For example, consider the following specification.

```
<types>
   <recordtype name="pair">
     <field name="label" type="string"/>
     <field name="number" type="int"/>
   </recordtype>

   <settype name="list" type="pair"/>
</types>
```

The element `types` defines new attribute types. In particular, by means of the `recordtype` element, SRD allows the definition of records (`pair` is defined as a record containing two fields: the first one is named `label` and is a string valued field; the second one is named `number` and is an integer valued field).

By means of the `settype` element, SRD allows the definition of sets (the type `list` is defined as a set of `pairs`).

Associating Intensional Attributes to Elements. After the definition of new attribute types, SRD allows the definition of intensional attributes.

Consider the following specification

```
<intensional-attributes>
  .  .  .
  <add-to-element name="CITE">
    <attribute name="number" type="int"/>
    <attribute name="citations" type="list"/>
    <attribute name="tr-i" type="token"/>
    <attribute name="tr-s" type="token"/>
  </add-to-element>
  .  .  .
</intensional-attributes>
```

The element `intensional-attributes` defines the intensional attributes. The block named `add-to-element` associates the intensional attributes to the element specified by `name`. Then, each intensional attribute associated to the given element is defined by a proper `attribute` tag, specifying the name and the type.

In the sample specification, we define four intentional attributes for element `CITE`: the first one is called `number` and is an integer; the second one is called `citations` and is defined on the complex type `list`, previously defined; finally, attributes `tr-i` and `tr-s` are defined of type `token`, since they are used to drive the formatting of the document.

4.2. Semantic Rules

Having defined the intensional attributes, SRD defines the semantic rules that derive their values.

The essential idea is to associate the rules to a pair father-child. In general, since an element has several children, the derivation of attributes depends on the relationship between father and children. In particular we can have

- rules deriving attributes of an element of type X from attributes of a children of type Y, that depend on the presence of an element X *father of* an element Y.
- rules deriving attributes of a children of type X from attributes of the father of type Y, that depend on the presence of an element Y *father of* an element X.
- rules deriving attributes of an element from attributes of the same element, that depend on the existence of given types of children (for example, different rules to execute whether the element is empty or has content).

Consequently, based on the parent-child relationship, it is possible to specialize semantic rules. However, there can be situations where different occurrences of elements of the same type which are children of the same father may require different semantic rules; this is due to the complex syntax structure determined by regular expressions defining element contents in the DTD [1].

Therefore, SRD tentatively proposes constructs to deal with such a complexity.

Syntactic Structure. SRD associates semantic rules to the syntactic structure using the parent-child relationships. In particular, a parent-child pair is used as a key to define the set of related semantic rules.

For example, if we consider the relationship `SECTION` *father of* `PAR`, we find in SRD a construct like the following:

```
<rules-for element="SECTION" father-of="PAR">
  . . .
</rules-for>
```

whose content is the set of semantic rules.

An element X can have generic text as content, denoted by `#PCDATA` in the DTD; an occurrence of `#PCDATA` can be viewed as the occurrence of a terminal symbol for generic strings. The relationship X *father of* `#PCDATA` is relevant and may have semantic rules.

This is the case of bibliographic items `BIBITEM`, that have only generic text as content. Rules are contained in the SRD block

```
<rules-for element="BIBITEM" father-of="#PCDATA"> . . . </rules-for>
```

[1] Similar problems are encountered when one tries to associate an attribute grammar to an Extended BNF syntax (see [12, 5]). Unfortunately, there is not a leading proposal for solving this problem.

Another case to consider is represented by *empty elements.* If the DTD allows an element X to be empty, the relationship X *father of* #EMPTY is relevant for semantic rules.

This is the case of citations CITE, where rules are contained in the SRD block

```
<rules-for element="CITE" father-of="#EMPTY"> .   .   . </rules-for>
```

The SRD specification must satisfy the following constraint:

for each parent-child pair (X, Y) *(where* Y *can be* #PCDATA *and* #EMPTY*) obtained from the DTD, there must be a* rules-for *mark-up in the SRD.*

This constraint is a necessary condition for the consistency of the semantic specification w.r.t. the syntax.

Static Rules. Consider now the semantic rules in a rules-for mark-up. The first type of rules we considered is represented by *static rules*, i.e. rules which do not depend on occurrences of children.

Consider the relationship SECTION *father of* PAR, for which we want to transfer the set of bibliographic items from the SECTION to the paragraphs PAR. We obtain the following specification:

```
<rules-for element="SECTION" father-of="PAR">
  <static> <derive attribute="PAR.citations" from="SECTION.citations"/>
  </static>
 .  .  .
</rules-for>
```

where the static block contains all the static rules. A rule is expressed by the derive tag: attribute is the left part of the rule and denotes the derived attribute by means of the dot notation *father.attribute*; from reports the expression which constitutes the right part of the rule. Hence, it corresponds to the classical representation

$$\texttt{PAR.citations} \leftarrow \texttt{SECTION.citations}$$

Observe that the concept of *inherited* and *synthesized* attributes, typical of AGs, are still valid: given a rules-for block, an attribute a is *inherited* if there is a derivation rule for a and a belongs to the child element; an attribute a is *synthesized* if there is a derivation rule for a and a belongs to the father element.

The right part of the rule can contain an attribute, both extensional and intensional, denoted by the dot notation; this attribute can belong to the father, to the child indicated in the rules-for tag, to another child.

A #PCDATA child has the predefined attribute Content, that contains the string matched by #PCDATA in the source text.

The right part can also contain constructors of complex types. For example, the rule

```
<derive attribute="BIBITEM.record" from="#RECORD(BIBITEM.label,BIBITEM.position"/>
```

derives, using the #RECORD constructor, the attribute record of BIBITEM from its extensional attribute label and the intensional attribute position.

Analogously, consider the rules for attributes of BIBITEM

```
<derive attribute="BIBITEM.cite-i" from="#EMPTYSET(list)"/>
<derive attribute="BIBITEM.cite-s" from="#ADDTOSET(BIBITEM.cite-i,BIBITEM.record)"/>
```

where the former initializes, by means of the `#EMPTYSET` constructor, the attribute `cite-i` to an empty set, whose type is `list`. The latter takes the attribute `cite-i` (defined as `list`, by means of the `#ADDTOSET` constructor adds the value of attribute `record` to the list, and the resulting set is the value of the derived attribute `cite-s`.

The right part of rules can also contain a call to an external function provided by a semantic library. For example, the rule

```
<derive attribute="BIBITEM.tr-s"
        from="#CALL.TransText(BIBITEM.tr-s,#PCDATA.Content)"/>
```

call the `TransText` function, that formats the text matched by `#PCDATA`. Observe the use of the attributes `tr-i` and `tr-s`, defined as `token`, to drive the side effects produced by the function.

Conditional Rules. In presence of repeated or optional occurrences of certain children, static rules become referentially inadequate. The way SRD proposes is constituted by *conditional rules*, derivation rules with associated conditions on the occurrences of children. Notice that this form of conditionals is different form the *conditional attribute grammars* proposed in [2], a technique introduced for overcoming circularities.

For instance, consider the repeated occurrences of a `BIBITEM` within the `BIBLIOGRAPHY`. The corresponding `rules-for` block is the following.

```
<rules-for element="BIBLIOGRAPHY" father-of="BIBITEM">
  <conditional>
    <if> <ISFIRST element="BIBITEM"/>
      <derive attribute="BIBITEM.position" from="1"/>
      <derive attribute="BIBITEM.cite-i" from="#EMPTYSET(list)"/> </if>
    <else/> <derive attribute="BIBITEM.position" from="#PRED(BIBITEM).position+1"/>
            <derive attribute="BIBITEM.cite-i" from="#PRED(BIBITEM).cite-s"/>
  </conditional>
</rules-for>
```

A `conditional` block contains the set of conditional rules. Within it, the `if` mark-up contains the condition and the semantic rules enabled when the condition is met. SRD allows several `if` blocks.

The `else` mark-up is followed by the semantic rules that holds when none of the conditions in the `if` blocks is met.

The example derives the attributes `position` and `cite-i` of the `BIBITEM` children: for the left-most occurrence, the attribute `position` must be set to `1` and the attribute `cite-i` must be the empty set; for the other occurrences, these attributes derive from the attributes `position` and `cite-s` of the preceding occurrence of `BIBITEM`.

The example gives an idea of what kind of conditions are necessary. In particular, we identified the following set of predicates.

- `ISFIRST`: this predicate is true if the specified element is the first (left-most) of the iteration.
- `ISLAST`: this predicate is true if the specified element is the last (right-most) of the repetition.
- `EX`: this predicate is true if there exists the specified element.

By means of the `<not> . . . </not>` mark-up, it is possible to specify conditions with negations.

Note that when the predicate refers to an element, it is here assumed the uniqueness of that element within the DTD rule. A more general notation for disambiguating the element reference remains to be established.

Consider a rule which derives an attribute of a repeated child. The right part of the rule may refer to attributes belonging to brothers, preceding or following. This can be done by means of suitable qualifiers. For example, using the `#PRED` qualifier, `#PRED(BIBITEM).cite-s` denotes the attribute `cite-s` belonging to the preceding occurrence of `BIBITEM`.

We identified the following qualifiers: `#PRED` (the predecessor), `#SUCC` (the successor), `#FIRST` (the first occurrence of the iteration), `#LAST` (the last occurrence of the iteration).

As far as derivation rules are concerned, the SRD specification must satisfy the following constraints.

- *An attribute associated to an element can be used either as inherited or as synthesized, not both.*
- *Rules in each* `if` *or* `else` *branch in a conditional block must derive the same set of attributes.*
- *There cannot be multiple derivation rules for the same attribute.*

Theses constraints are necessary to ensure the correctness of the SRD specification.

4.3. Attribute Evaluation

The problem of evaluating the attributes defined in a SRD is substantially similar to the problem of evaluating attributes defined by a classical AG. We believe that the available solutions should be adequate in the present frame too. In this section, we sketch the problem, that will be object of our future work.

The basic property that a SRD should satisfy is the *non-circularity* property, which ensures that the systems of semantic rules applied to an instance of tree has a solution.

It is known that based on this property, it is possible to obtain a general, although inefficient, semantic evaluator. Hence, more specialized evaluations methods are necessary to obtain fast semantic evaluation.

In the literature, several methods have been proposed for AGs. The best known are the *one-sweep* and *multi-sweep* grammars, described in [8], and the *ordered attribute grammars* proposed in [13].

We think that these methods can be adapted to the evaluation of SRDs; in particular, the adaptations should concern the way dependencies determined by different branches of conditional rules affect the evaluation strategy and the computation of the evaluation/visit sequences performed by the semantic evaluator.

Multi-sweep evaluation is potentially attractive to implement a cascade of document processing rules, where each tool performs a sweep on a decorated XML tree and computes the next one.

5. Conclusions

If mark-up specifications are viewed as meta-grammars, as we do here, the idea of enriching them with semantic attributes and functions comes forward as a logical step.

We have outlined a way for defining the semantics of a document by a XML-compliant specification, an approach that is conceptually homogeneous and allows to exploit existing XML parser generators for analysing semantics and for generating attribute evaluators. The system architecture of a XML-based syntax-directed compiler-compiler is analogous to a classical translator writing system. Open design issues concern the reification of decorated syntax-trees, preferably as automatically specified document types.

We intend to continue this work in the following directions:

- to complete the design of the meta-languages and document types for semantic specifications;
- to apply the technique to case studies requiring deeper semantic analysis;
- to propose the extensions to XML working groups.

We also intend to implement a compiler-compiler for processing semantically enriched XML. Two alternatives are under evaluation:

- to reuse an existing compiler-compiler, by converting XML metalanguages to the proprietary notation of the chosen meta-tool.
- to redesign a compiler-compiler using XML meta-language as specification of syntaxes and attribute grammars.

A bonus of the second approach is to create an XML compliant representation of grammars that could be proposed as a standard to the technical community, as done by the computational linguists who defined a common SGML representation of human lexica and grammars [20].

As a final remark, observe that we have avoided changes to XML, in order to gain acceptance from the XML community, but a bolder approach could be taken: to modify the XML meta syntax (DTD) to allow the direct specification of semantic attributes and functions.

A. A DTD for SRD

```
  <!element field EMPTY >
  <!attlist field name CDATA $REQUIRED        type CDATA #REQUIRED >
  <!element recordtype (field)+ >
  <!attlist recordtype name CDATA #REQUIRED>
<!element settype EMPTY >
<!attlist settype name CDATA #REQUIRED        type CDATA #REQUIRED >
  <!element attribute EMPTY >
  <!attlist attribute name CDATA #REQUIRED    type CDATA #REQUIRED >
  <!element add-to-element (attribute)+ >
  <!attlist add-to-element name CDATA #REQUIRED >
<!element derive EMPTY >
<!attlist derive attribute CDATA #REQUIRED    from CDATA #REQUIRED >
<!element static ((derive)+) >
  <!element ex EMPTY >
  <!attlist ex element CDATA #REQUIRED >
  <!element isfirst EMPTY >
  <!attlist isfirst element CDATA #REQUIRED >
  <!element islast EMPTY >
  <!attlist islast element CDATA #REQUIRED >
  <!element not (ex|isfirst|islast) >
  <!element if ((ex|isfirst|islast|not)+,(derive)+) >
<!element conditional ((if)+,else,(derive+) >
<!element rules-for ((static)?,(conditional)*) >
<!attlist rules-for element CDATA #REQUIRED   father-of CDATA #REQUIRED >
  <!element types (recordtype | settype)+ >
  <!element intensional-attributes (add-to-element)* >
<!element semantic-rules (rules-for)+ >
<!ELEMENT SRD ((types)*,intensional-attributes,semantic-rules) >
```

B. SRD for the Running Example

```
<?xml version="1.0"?>
<!DOCTYPE SRD SYSTEM "srd.dtd">
<!-- Types and Attributes -->
<types> <recordtype name="pair"> <field name="label" type="string"/>
         <field name="number" type="int"/>  </recordtype>
        <settype name="list" type="pair"/> </types>
<intensional-attributes>
 <add-to-element name="BIBITEM">
  <attribute name="position" type="int"/> <attribute name="record" type="pair"/>
  <attribute name="cite-i" type="list"/>  <attribute name="cite-s" type="list"/>
  <attribute name="tr-i" type="token"/>   <attribute name="tr-s" type="token"/>
</add-to-element>
<add-to-element name="BIBLIOGRAPHY">      <attribute name="citations" type="list"/>
  <attribute name="tr-i" type="token"/>   <attribute name="tr-s" type="token"/>
</add-to-element>
<add-to-element name="CITE">            <attribute name="number" type="int"/>
  <attribute name="tr-i" type="token"/> <attribute name="tr-s" type="token"/>
</add-to-element>
<add-to-element name="SECTION">           <attribute name="citations" type="list"/>
  <attribute name="position" type="int"/> <attribute name="prefix" type="string"/>
  <attribute name="tr-i" type="token"/>   <attribute name="tr-s" type="token"/>
</add-to-element>
<add-to-element name="PAR">              <attribute name="citations" type="list"/>
  <attribute name="tr-i" type="token"/>  <attribute name="tr-s" type="token"/>
</add-to-element>
<add-to-element name="DOCUMENT">         <attribute name="citations" type="list"/>
</add-to-element>
<add-to-element name="APPENDIX">         <attribute name="citations" type="list"/>
  <attribute name="tr-i" type="token"/>  <attribute name="tr-s" type="token"/>
</add-to-element> </intensional-attributes>

<!-- FUNCTIONAL DTD -->
<semantic-rules>
<rules-for element="BIBLIOGRAPHY" father-of="BIBITEM">
  <conditional> <if> <isfirst element="BIBITEM"/>
     <derive attribute="BIBITEM.position" from="1"/>
     <derive attribute="BIBITEM.cite-i" from="#EMPTYSET(list)"/> </if>
     <derive attribute="BIBITEM.tr-i" from="BIBLIOGRAPHY.tr-i"/> <else/>
      <derive attribute="BIBITEM.position" from="#PRED(BIBITEM).position+1"/>
      <derive attribute="BIBITEM.cite-i" from="#PRED(BIBITEM).cite-s"/>
      <derive attribute="BIBITEM.tr-i" from="#PRED(BIBITEM).tr-s"/>
  </conditional> </rules-for>
<rules-for element="BIBITEM" father-of="#PCDATA"> <static>
 <derive attribute="BIBITEM.record" from="#RECORD(BIBITEM.label,BIBITEM.position"/>
 <derive attribute="BIBITEM.cite-s" from="#ADDTOSET(BIBITEM.cite-i,BIBITEM.record)"/>
 <derive attribute="BIBITEM.tr-s" from="#TransText(BIBITEM.tr-i,#PCDATA.Content)"/>
 </static> </rules-for>
<rules-for element="DOCUMENT" father-of="TITLE"> <static>
  <derive attribute="TITLE.tr-i" from="#InitToken()"/> </static> </rules-for>
<rules-fro element="DOCUMENT" father-of="SECTION">
  <static> <derive attribute="SECTION.citations" from="DOCUMENT.citations"/>
   <derive attribute="SECTION.prefix" from="#CALL.NormalPrefix()"/> </static>
  <conditional> <if> <isfirst element="SECTION">
```

```
    <derive attribute="SECTION.position" from="1"/>
    <derive attribute="SECTION.tr-i" from="#PRED(TITLE).tr-s"/> </if> <else/>
     <derive attribute="SECTION.position" from="#PRED(SECTION).position+1"/>
     <derive attribute="SECTION.tr-i" from="#PRED(SECTION).tr-s"/>
    </conditional> </rules-for>
<rules-for element="DOCUMENT" father-of="APPENDIX"> <static>
  <derive attribute="APPENDIX.citations" from="DOCUMENT.citations"/> </static>
  <conditional> <if> <ex element="SECTION"/>
     <derive attribute="APPENDIX.tr-i" from="#LAST(SECTION).tr-s"/> </if>
   <else/> <derive attribute="APPENDIX.tr-i" from="TITLE.tr-s"/>
  </conditional> </rules-for>
<rules-for element="DOCUMENT" father-of="BIBLIOGRAPHY">
  <static> <derive attribute="DOCUMENT.tr-s" from="BIBLIOGRAPHY.tr-s"/>
   <derive attribute="DOCUMENT.citations" from="BIBLIOGRAPHY.citations"/> </static>
  <conditional> <if> <ex element="APPENDIX"/>
    <derive attribute="BIBLIOGRAPHY.tr-i" from="APPENDIX.tr-s"/> </if>
   <if> <not> <ex element="APPENDIX"> </not> <ex element="SECTION"/>,
    <derive attribute="BIBLIOGRAPHY.tr-i" from="#LAST(SECTION).tr-s"/> </if>
   <else/> <derive attribute="BIBLIOGRAPHY.tr-i" from"TITLE>tr-s"/>
  </conditional> </rules-for>
<rules-for element="APPENDIX" father-of="SECTION">
 <static> <derive attribute="SECTION.prefix" from="#CALL.AppendixPrefix()"/>
  <derive attribute="APPENDIX.tr-s" from="#LAST(SECTION).tr-s"/> </static>
 <conditional> <if> <isfirst element="SECTION">
    <derive attribute="SECTION.tr-i" from="#CALL.TransApp(APPENDIX.tr-i)"/>
    <derive attribute="SECTION.position" from="1"/> </if>
   <else/> <derive attribute="SECTION.tr-i" from="#PRED(SECTION).tr-s)"/>
     <derive attribute="SECTION.position" from="#PRED(SECTION).position+1"/>
  </conditional> </rules-for>
<rules-for element="SECTION" father-of="PAR">
  <static> <derive attribute="PAR.citations" from="SECTION.citations"/>
     <derive attribute="SECTION.tr-s" from="#LAST(PAR).tr-s"/> </static>
  <conditional> <if> <isfirst element="PAR"/>
   <derive attribute="PAR.tr-i" from="SECTION.tr-i"/> </if> <else/>
   <derive attribute="PAR.tr-i" from="#PRED(PAR).tr-s"/> </conditional> </rules-for>
<rules-for element="SECTION" father-of="#EMPTY"> <static>
  <derive attribute="SECTION.tr-s" from="SECTION.tr-i"/> </static> </rules-for>
<rules-for element="PAR" father-of="CITE">
  <static> <derive attribute="CITE.number"
           from="#Find(PAR.citations,#CALL.Check(CITE.label))"/> </static>
  <conditional> <if> <isfirst element="CITE"/>
    <derive attribute="CITE.tr-i"
            from="#CALL.TransText(PAR.tr-i,#PRED(#PCDATA).Content)"/> </if>
   <else/> <derive attribute="CITE.tr-i"
          from="#CALL.TransText(#PRED(PAR).tr-s,#PRED(#PCDATA).Content)"/>
    </conditional> </rules-for>
<rules-for element="PAR" father-of="#PCDATA">
  <conditional> <if> <ex element="CITE"/>
     <derive attribute="PAR.tr-s"
        from="#CALL.TransText(#LAST(CITE).tr-s,#LAST(#PCDATA).Content)"/> </if>
   <else/> <derive attribute="PAR.tr-s"
                  from="#CALL.TransText(PAR.tr-i,#PCDATA.Content)"/>
  </conditional> </rules-for>
<rules-for element="CITE" father-of="#EMPTY"> <static>
  <derive attribute="CITE.tr-s" from="#CALL.TransCite(CITE.tr-i,CITE.number)"/>
 </static> </rules-for> </SRD>
```

Bibliography

[1] H. Alblas and B. Melichar. Attribute grammars, applications and systems. *Lecture Notes in Computer Science*, 545, 1991.

[2] J. T. Boyland. Conditional attribute grammars. *ACM Transactions on Programming Languages and Systems*, 18(1), January 1996.

[3] T. Bray, C. Frankston, and A. Malhotra. Document content description for xml. Technical report, World Wide Web Consortium, July 1998.

[4] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml). Technical Report PR-xml-971208, World Wide Web Consortium, December 1997.

[5] S. Crespi-Reghizzi, G. Psaila, and M. Pagani. The coompactness of extended bnf semantic definitions. In *CC96 International Conference on Compiler Construction, Poster Session*, Linkoping, Sweden, April 1996.

[6] P. Deransart and M. Jourdan. Attribute grammars and their applications. *Lecture Notes in Computer Science*, 1990.

[7] P. Deransart, M. Jourdan, and B. Lorho. Attribute grammars. *Lecture Notes in Computer Science*, 1988.

[8] J. Engelfriet. Attribute grammars: Attribute evaluation methods. *In [19]*, 1984.

[9] C. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990.

[10] M. Jourdan, C. Le Bellec, and D. Parigot. The olga attribute grammar description language. *In [6]*, 1990.

[11] M. Jourdan, D. Parigot, C. Julie, O. Durin, and C. LeBellec. Design, implemenation and evaluation of the FNC-2 attribute grammar system. In *ACM SIGPLAN-90*, 1990.

[12] R. K. Jullig and F. DeRemer. Regular right-part attribute grammars. In *ACM SIGPLAN-84*, 1984.

[13] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1983.

[14] U. Kastens. Attribute grammars as a specification method. *In [1]*, pages 16–47, 1991.

[15] U. Kastens, B. Hutt, and E. Zimmermann. Gag: a practical compiler generator. *Lecture Notes in Computer Science*, 1982.

[16] D. Knuth. Semantics of context free languages. *Mathematical System Theory*, 2:127–145, 1968.

[17] A. Layman, E. Jung, E. Maler, N. H. Mikula, J. Paoli, J. Tigue, H. S. Thompson, and S. DeRose. Xml-data. Technical report, World Wide Web Consortium, December 1997.

[18] R. Light. *Presenting XML*. SAMS, 1997.

[19] B. Lorho. *Methods and Tools for Compiler Construction*. Cambridge University Press, 1984.

[20] TEI. Text encoding initiative. *http://www-tei.uic.edu/orgs/tei*.

[21] XML-Sofware. Xml parsers. *http://www.xmlsoftware.com/parsers*.