Design of a Platform based Approach for DTD Model Conversion based on Attributed Grammar

SOUHEIL TAWK, KABLAN BARBAR, JOSEPH CONSTANTIN *
LaRRIS, Faculty of Sciences, Lebanese University, Fanar,
LEBANON
*Corresponding Author

Abstract: This paper presents a novel platform based methodology for the transformation of Document Type Definition (DTD) models using an attributed grammar. The proposed approach consists of two main phases. In the first phase, we define a new representation of the DTD called DTD_{xml}, which is created using the attributed grammar theory from the compiler design. In the second phase, this intermediate model DTD_{xml} is used to facilitate the transformation of the DTD into arbitrary target models via XSLT stylesheets. To enable this transformation, we introduce an algebraic grammar for the DTD language that can syntactically recognize and parse any DTD specification. This grammar is then used to define an attribute system that can be used to semantically annotate DTD documents and transform them into an intermediate XML based representation. This structured representation, DTD_{xml}, serves as input for XSLT transformations that generate the corresponding target relational schemas or other model formats. The main contribution of this work is the decomposition of the model transformation process into two distinct phases: a structuring phase, in which the source DTD is rewritten into an XML syntax using attributed grammar techniques, and a target model generation phase, in which the normalized intermediate representation is translated into the desired model using Extensible Stylesheet Language Transformations (XSLT). This modular separation improves the reusability, scalability and extensibility of model transformation processes. It allows the compilation process to be carried out once, regardless of the number of transformations required. Each transformation is then defined via an independent stylesheet, so that a new lexical and syntactic analysis is no longer necessary.

Key-Words: Attributed Grammar; Model Conversion; DTD; Relational Schema; Theory of Computation; XSLT

Received: April 27, 2025. Revised: July 19, 2025. Accepted: September 2, 2025. Published: March 5, 2026.

1 Introduction

The modeling of data and processes is of fundamental importance for the representation, organization, implementation and management of data and processes within a system. Data modeling is about defining the entities within a system and the relationships between them. Process modeling, on the other hand, is about defining the activities and tasks within a system and their execution flow. With the advent of new technologies, the need to convert data between different types of data models has become crucial to ensure interoperability between systems, [1]. The task of converting between the XML data model and the relational data model is the subject of numerous articles and studies in the literature due to the importance, widespread use and numerous applications of these technologies in software applications and systems, [2].

XML is an emerging standard for the representation and exchange of data over the Internet, [3]. XML adds tags to describe the meaning of the data. Tags can be embedded, and an XML document is a tree of elements with a unique root that can be nested at many levels. The XML environment is based on very useful transformation languages such as XSLT, with which we can transform an XML document into another XML document or other documents of different types such as HTML, text and PDF. Relational databases are suitable for storing and displaying data in relationships. Each relationship consists of a series of records, with each record containing a list of fields. This architecture is implemented using a list of tables and has a very powerful manipulation language, the SQL language. In addition, a relational database management system stores data efficiently and without redundancy, as each piece of information is only stored in one place, [4]. It is important to know that we store data in relational databases when we want to manipulate it. If we want to exchange or transform data, we store it in XML documents. Therefore, the data from XML documents must be converted into relational databases in order to retrieve the data.

This paper proposes a new platform for model conversion that consists of two phases. In the first phase, the text model is rewritten into an intermediate model in XML format using the theory of attribute grammars in compiler theory. In the second phase, the intermediate XML model is converted into an arbitrary model using the XSLT language. The

contributions of our approach can be described as follows:

1. We propose a formalization of the set of Document Type Definitions DTDs using a context free grammar called G_DTD. In this formalism, each transformation of a DTD is represented by the definition of attributes over G_{DTD} and thus enables the construction of a compiler based on the principles of an attributed grammar.The introduction of an attribute system to semantically annotate the documents and convert them to an XML based intermediate representation DTD_xml is a key innovation.
2. We demonstrate the applicability of G_{DTD} in the field of DTD conversion. Specifically, G_{DTD} facilitates the definition of attributes that enable the transformation of a given DTD into a target DTD while preserving the declarations of elements and attributes and maintaining conformance with XML syntax. The resulting intermediate representation, referred to as DTD_{xml}, serves as an intermediate model. Any transformation algorithm that targets the original DTD can be specified as an XSLT stylesheet that is applied to DTD_{xml}. With this method, the compilation process can be executed once, regardless of the number of transformations required. Each transformation is then defined via an independent stylesheet, eliminating the need for a new lexical and syntactic analysis of the DTD source.

The paper is organized as follows: Section 2 deals with related work. In Section 3, we present our methodology for mapping DTDs to XML DTDs based on compiler design. In Section 4, we explain how we map DTDs to database schemas using XSLT. Section 5 shows the experimental results, and Section 6 concludes the paper.

2 Related Work

Mapping XML DTDs to relational schemas is not an easy task, as the structure of a DTD document differs fundamentally from that of a relational database. DTDs can contain arbitrary regular expressions such as recursion, disjunction and set values that require further analysis. Several mapping algorithms have been proposed in the literature to solve this problem, [5], [6]. However, semantic keys are not used efficiently because system generated IDs, such as ID, parentID and parentCODE, are widely used. Studies have shown, [7], that relying on these IDs leads to data and relationship redundancy given the limitations in XML documents. This approach does not take semantic dependencies into account and is therefore ineffective when it comes to reducing data redundancy.Other algorithms have completely ignored the semantics of DTDs and created relational schemas in which much of the original DTD information has been lost, [8], [9].

In the study, [10], the authors use relational table and object mappings. Both mappings model the data in XML documents and not the documents themselves. The table based mapping cannot handle mixed content, and the object based mapping of mixed content is inefficient. This makes it a poor choice for centralized documents. Other mapping algorithms preserve the cardinality constraints, [11], [12], [13], [14], but do not consider the functional dependencies over DTDs. In [15], the authors have developed a mapping algorithm that preserves both the structure of the DTDs and the semantics implied by the DTD constraints. However, the multivalued dependencies are not preserved by the mapping algorithm.

A mapping algorithm has been developed in the scientific literature to map a DTD to a relational schema that preserves not only the content and structure, but also the semantics of the original XML documents, [16]. To solve the problem of expressing constraints, the authors have introduced a method for defining functional dependencies and normalizing the DTD. In a normalized DTD, each constraint expressed by functional dependencies can be converted into keys. They used the key definitions for XML as the basis for creating relationships and kept the keys in the relationships.

Another technique, based on the hybrid inlining algorithm, [17], has been proposed to map the DTD to a relational schema by preserving the functional dependencies and creating relationships with less redundancy, [18]. Redundancies are reduced by moving attributes, creating relationships, introducing keys and retaining functional dependencies. In [19], the authors analyzed the sets in the data model of the XML document and defined the dependency relationships. Based on these relationships, they create a set of rules for mapping the XML document to a relational database. However, the proposed algorithms are complex and need to be reexecuted for multiple transformations, which slows down the online decision process.

In [20], the authors deal with the formal representation and transformation of DTDs into semantic SemODM schemas. While many projects have focused on the storage and exchange of XML data, this study deals with the formalization of these data models and their transformation and presents a concise framework for the definition of DTDs and SemODM Semantic Schemas. The formalization process begins with the definition

of DTDs, which serve as the basis for XML data structures. DTDs consist of elements and attributes with specific types and restrictions. At the other end of the transformation, SemODM semantic schemas are introduced as a high level data model. SemODM maps real world scenarios in a similar way to the entity relationship model and offers the advantages of the object oriented model. Categories and relationships are the central elements that reflect entities and relationships.

In [6], the authors address the problem of translating XML data into relational data by converting the DTD into a relational schema. Many previous works have proposed algorithms for converting a DTD into a relational schema, but have only focused on translating the structure of XML data and neglected the semantic constraints embedded in the DTD. In contrast, this paper proposes an algorithm called Constraints Preserving Inlining (CPI) to translate the DTD into a relational schema.

In [16], the authors investigated XML storage in relations. In contrast to traditional techniques, they took into account the semantics expressed by functional dependencies. They proposed an algorithm for mapping a DTD to a relational schema that preserves not only the content and structure, but also the semantics of the original XML documents. To solve the problem of expressing constraints, they introduced a way to define functional dependencies and normalize DTDs. The authors in [21], described a technique for mapping a given XML DTD to a relational schema. This technique consisted of converting the DTD to an Xschema, simplifying the Xschema constraints, inlining elements and attributes, handling key constraints, mapping collection types, mapping IDREF and IDREFS attributes, handling the union type and capturing the order specified in the XML model.

The authors in [22], hold a seminar to discuss how XML data can be represented in the relational model. The authors in [23], start with the introduction of XML query processing and then describe how RDBMS can be used to store and query XML data. Similarly, the authors in [24], focus on adaptive or flexible mapping methods to improve XML processing based on RDBMSs. In the study, [25], the authors present the advantages of hybrid storage combining structure mapping and XML data type. In [26], the study demonstrates only two mapping strategies: schema aware and schema less. In a recent review by [27], the study revisits the common methods for managing XML data through relational schemas using RDBMSs, but this review lacks a detailed discussion of these methods and open problems. The study, [28], provides a simple comparison between the Argo/3 approach, [29], and Single Table Data Mapping (STDM), [30]. The authors in [31], list several relevant works on RDF data storage and query processing. Previous studies have only focused on XML to relational or RDF to relational mappings.

Other approaches are model based. In these approaches, the mapping relies on path expressions within the XML tree without an associated schema type. In this method, the tree is systematically traversed and the path of each node visited is recorded in a relational table. Although the underlying concept remains unchanged, a number of strategies have been proposed to improve on previous implementations [32], [33], [34]. As these methods do not use schema types, relational tables are used to record the path information. This information is derived exclusively from the structural properties of the XML tree, without taking the semantic content into account. A major limitation of relational schemas created using these methods is that the data is split into numerous small components, which can lead to increased memory requirements in the database.

Unfortunately, all studies focus on conversion algorithms and overlook how the DTD source can be read and decomposed into smaller units for the conversion process. We believe that our approach is the first to focus on applying the syntactic and semantic analysis of the compilation process to transform the DTD source into an equivalent source in terms of XML syntax. Our goal was to implement each DTD conversion algorithm through an XSLT stylesheet.

3 Materials and Methods

In this study, we propose a new approach based on an attributed grammar and an XML environment to create platforms for conversion methods for textual models. In our view, textual models are strings, and a set of textual models is considered a context free language in the theory of formal languages. In the following, by ”model conversion” we mean the conversion of the textual model. Therefore, it is clear that all compilation techniques can be applied to a set of models. Finally, we assume that each model conversion method consists of two phases: phase 1, the structuring phase, and phase 2, the conversion phase, as follows (Fig. 1)

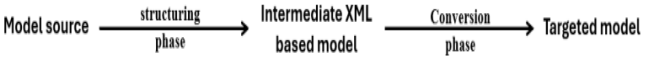


Figure 1: Conversion from source model to target model. Source: created by the authors.

We apply these concepts to the conversion methods of DTD to database and we have the following(Fig. 2):

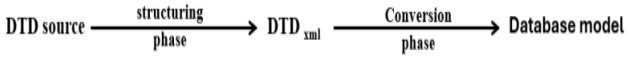


Figure 2: Conversion from DTD source model to database model. Source: created by the authors.

The structuring phase includes the specification of a context free grammar, called G_{dtd}, for the set of DTDs and the definition of an attributed system G_{dtd} to transform each DTD into an equivalent XML based one, called DTD_{xml}. This phase provides a formal description of the set of DTDs based on formal language theory. Thus, all compilation techniques that apply to context free grammars in formal theory can be imported into G_{dtd}. In particular, the use of attributes on the context free grammar for transformation requests. As an intermediate model, DTD_{xml} is XML based. The algorithm of each DTD transformation method can be expressed in the transformation phase by an XSLT stylesheet applied to the DTD_{xml}. In the following, we focus on the definition of the context free grammar G_{dtd} and the associated system used to transform each DTD source into a corresponding DTD_{xml}.

3.1 Phase 1: Conversion of DTDs to DTD_{xml}

3.1.1 Algebraic grammar for DTDs

We start the first phase by creating an algebraic grammar G_{DTD} for the set of all possible DTD_s, [35]. After eliminating ambiguity, left recursion and non left factorization, we obtain the grammar $G_{dtd} =< \Sigma, N, S, P >$, which is defined as follows:

- $\Sigma = \{<, >, (,), |, *, ?, +, ", !, [,], , , \#,$ identifier, DOCTYPE, ELEMENT, ATTLIST, EMPTY, ANY, PCDATA, id, CDATA, ID, IDREF, FIXED, REQUIRED, IMPLIED, string $\}$: denotes the alphabet.
- N = {Start, DTD, LineType, Attlist, AttSuite, ElementType, List, Occurs, AttType, Enumerate, EnumerateSuite, AttOption, InitialVal}: is the set of non terminals,
- S is the axiom.
- P is the following set of productions:

P_0: Start → DTD
P_1: DTD → <!DOCTYPE identifie [LineType]
P_2: LineType → <!ELEMENT identifier ElementType> AttList LineType
P_3: LineType → ϵ
P_4: AttList → <!ATTLIST identifier1 identifier2 AttType AttOption AttSuite> AttList
P_5: AttList → ϵ
P_6: AttSuite→ identifier AttType AttOption AttSuite
P_7: AttSuite → ϵ
P_8: ElementType → EMPTY
P_9: ElementType → ANY
P_{10}: ElementType→ (#PCDATA)
P_{11}: ElementType→ (List) Occurs
P_{12}: List → id Occurs , List
P_{13}: List → id Occurs | List
P_{14}: List → (List) Occurs, List
P_{15}: List → (List) Occurs | List
P_{16}: List → (List) Occurs
P_{17}: List → id Occurs
P_{18}: Occurs → *
P_{19}: Occurs → ?
P_{20}: Occurs → ϵ
P_{21}: Occurs → +
P_{22}: AttType → CDATA
P_{23}: AttType → ID
P_{24}: AttType → IDREF
P_{25}: AttType →(Enumerate)
P_{26}: Enumerate → string EnumerateSuite
P_{27}: EnumerateSuite → Enumerate
P_{28}: EnumerateSuite → ϵ
P_{29}: AttOption → #FIXED InitialVal
P_{30}: AttOption → #REQUIRED
P_{31}: AttOption → #IMPLIED
P_{32}: AttOption → InitialVal
P_{33}: InitialVal → ”Val”
P_{34}: Val → string
P_{35}: Val → ϵ

We define the productions P_0 and P_1, which specify the syntax of a DTD. We associate a non terminal to create an element and another non terminal to define the type of an element. These non terminals are LineType and ElementType respectively. We add the Attlist non terminal because the element can have attributes and the LineType can create

multiple elements (P_2 and P_3). The Attlist can create attribute declarations, the nonterminal AttType stands for the type of an attribute and the nonterminal AttOption creates the options for an attribute. We write the productions P_4 -P_7. The element type can be (#PCDATA), ANY, EMPTY and can have a list of subelements (a composed element) (P_8 -P_{11}). The structure of a composed element is therefore a compound expression with operators (+, *, …) and separators (, and |). We recommend using the productions P_{12} -P_{17} for List. The productions P_{18} -P_{21} generate terminal operators such as *, ?, +, and ϵ. The productions P_{22} -P_{25} generate the attribute type, which can be CDATA, ID, IDREF or enumerate with the initial value. The productions P_{26} -P_{28} are used to generate the character string enumerate in XML. The productions P_{29} -P_{33} are used to generate attribute options with an initial value. The attribute options can be #FIXED, #REQUIRED, #IMPLIED. The initial value is generated in the productions P_{32} and P_{33}. This value can be empty or defined as a character string. Using the algebraic grammar G_{dtd}, we run the syntactic analyzer, which takes a DTD as input and checks whether the DTD is syntactically correct or not. To build the derivation tree, a node should contain a ”label” field to store the non terminal character on the left side of the production, a ”production” field to store the production identifier, and several pointers corresponding to the number of symbols on the right side of the production. The goal is to help the source model understand an intermediate model that contains the information from the source model and encapsulates it in XML format. This correspondence preserves all the information of the element declaration. For this reason, it is necessary to create a new intermediate model with the designation DTD_{xml}.

3.2 Attributed grammars for DTDs

Next, we enriched the algebraic grammar with attribute equations associated with the productions of the G_{dtd} grammar to transform a DTD into an intermediate DTD in XML called DTD_{xml}. Attribute grammars are a formalism used to describe the semantic analysis of programming languages. The core idea is to assign attributes to each non terminal in a grammar and to define a system of equations for each production rule. These attributes are analyzed along the derivation tree to determine the semantics of the input. In the context of programming languages, each language has its own algebraic grammar. Attributes can be used to systematically compute the semantics of all statements in a program, which is considered as the input string of the grammar. An attribute grammar (AG) is defined as:

- $G =< \Sigma, N, S, P >$ where:

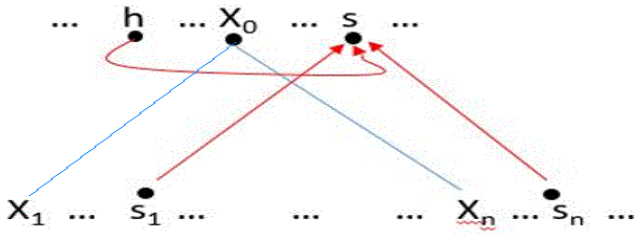


Figure 3: Definition of synthesized attributes. Source: [35].

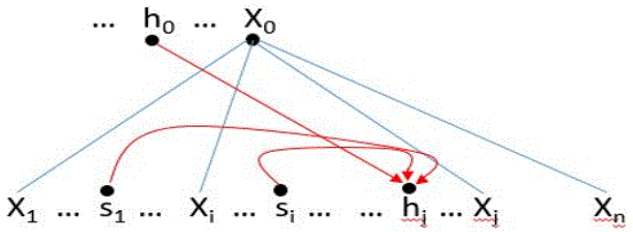


Figure 4: Definition of inherited attributes. Source: [35].

- Σ is the set of terminal symbols.
- N is a set of non terminal symbols.
- S is the start symbol (axiom) and $X \in N$.
- P is the set of production rules.
- A is a set of attributes divided into two subsets A_s and A_h. $A_s(X)$ is the set of synthesized attributes and $A_h(X)$ is the set of inherited attributes, [35] (Fig. 3).
- E is the set of equations in which for each synthesized attribute, [35] (Fig. 4).

In this study, AG are used to transform a DTD source into an intermediate DTD with respect to the XML language, denoted as DTD_{xml}. (DTD_{xml} is an XML document and preserves all the information of the DTD source). To summarize, we search for an AG with the name AG_{dtd} as follows (Fig. 5):

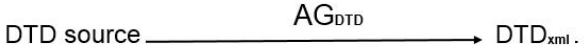


Figure 5: conversion from DTD source to DTD_{xml}. Source: created by the authors.

Since a DTD_{xml} is an XML document, the AG_{dtd} should contain an attribute named x that is associated

with the most important non terminals. The attribute x of the root of a derivation tree associated with a particular DTD source has a terminal value corresponding to the document DTD_{xml} of that DTD source. For example, if we take the following DTD source as input:

```
<!DOCTYPE Test [
    <!ELEMENT A (B,C, (D|E),F)>
    <!ELEMENT B (#PCDATA)>
    <!ELEMENT C (#PCDATA)>
    <!ELEMENT D (#PCDATA)>
    <!ELEMENT E (#PCDATA)>
    <!ELEMENT F (#PCDATA)>
]>
```

The corresponding DTD Format XML is:

```
<doctype name = ”Test”>
    <element name = ”A” type =”Composed”>
    <seq>
        <element name= ”B”/>
        <element name= ”C”/>
        <alt>
            <element name= ”D”/>
            <element name= ”E”/>
        </alt>
        <element name= ”F”/>
    </seq>
    </element>
    <element name=”B” type =”PCDATA”>
    </element>
    <element name=”C” type =”PCDATA”>
    </element>
    <element name=”D” type =”PCDATA”>
    </element>
    <element name=”E” type =”PCDATA”>
    </element>
    <element name=”F” type =”PCDATA”>
    </element>
</doctype>
```

Another key challenge in this AG transformation is the transformation of the list B, C, (D|E),F, which represents the structure of element A, into an XML sequence in the attribute equations. We apply the following strategy:

- The first operator in the list (the comma between elements B and C in the example above) determines the type of the list as a sequence (<seq> </seq>) or as an alternate (<alt> </alt>). For the example above, the list becomes a sequence <seq>.....</seq>. The value of the operator is transferred by the s attribute. The meaning of s(list) is the first operator generated by the list.
- The first operator of a list is transferred to all elements of this list by an inherited attribute s1.
- An inherited attribute s2 is connected to the list of non terminals to carry the operator that precedes this non terminal in the derivation tree (to the left of it). If the list of non terminals generates an identifier (or an element), s2 (list) is the operator that precedes this identifier.

The s1 and s2 attributes tell us how to add the identifiers generated by the non terminal list to the XML sequence corresponding to a list of subelements (a subelement is an identifier). The productions $P1-P33$ define the attributes for non terminals (Table 1, Table 2, Table 3, Table 4, Table 5, Table 6, Table 7, Table 8, Table 9 in Appendix).

3.3 Phase 2: DTD_{xml} conversion to Relational Database

Since the intermediate model DTD_{xml} is an XML document, it is obvious to use the XSLT language to perform all transformations on DTD_{xml} in the conversion phase (Fig. 6):

Figure 6: conversion from DTD_{xml} source model to target model. Source: created by the authors.

On the other hand, the DTD_{xml} has received all declarations of the elements and attributes from the DTD source. For example, the subelements of an element are retained in the source DTD and organized as a subtree in DTD_{xml}. The characteristics of the attribute are also retained and organized as nodes in the DTD_{xml} tree. Therefore, a conversion algorithm to convert a DTD into a database schema can be effectively implemented using an XSLT stylesheet applied to the DTD_{xml}.

This XSLT template defines the conversion behavior for <doctype> elements. In particular, it generates a SQL CREATE DATABASE statement using the value of the name attribute, inserts two line breaks for better readability and then applies templates to all subordinate element nodes whose type attribute is Composed.

```
<xsl:template match="doctype">
   CREATE DATABASE <xsl:value-of
       select="@name"/>;
  <xsl:text>&#xA;&#xA;</xsl:text>
    <xsl:apply-templates select=
  "element[@type='Composed']"/>
</xsl:template>
```

Once the elements and attributes are identified, our dynamic XSLT takes over the contextual generation

of scripts to create relational tables. Each script is created to accurately reflect the hierarchies and relationships defined in the XML source document. The following XSLT template corresponds to all element nodes with the Composed attribute type. It declares a tableName variable to store the value of the name attribute and then issues a CREATE TABLE SQL statement using that name. The template also inserts an opening parenthesis and a line break to prepare the definition of the table columns or constraints in subsequent transformations.

```
<xsl:template match="
   element[@type='Composed']">
     <xsl:variable name="tableName"
   select="@name"/>
     <xsl:text>CREATE TABLE
   </xsl:text>
     <xsl:value-of select="$tableName"/>
     <xsl:text> (&#xA;</xsl:text>
```

This part of the template defines a variable columns that summarizes all attributes and all child element nodes that are not of type Composed and thus identifies the fields that are to be mapped as table columns. A loop iterates through these fields to generate corresponding SQL column definitions.

```
<xsl:for-each select="$columns">
 <xsl:text> </xsl:text>
 <xsl:choose>
  <!-- Attributes -->
  <xsl:when test="self::attribute">
   <xsl:value-of select="@name"/>
   <xsl:text> </xsl:text>
   <xsl:choose>
    <xsl:when test="@type='ID'">
   INT PRIMARY KEY
   </xsl:when>
    <xsl:when test="@type='IDREF'">
     <xsl:variable name="refTable">
      <xsl:value-of select="
   key('elementByName', @name)/@name"/>
     </xsl:variable>
     <xsl:choose>
      <xsl:when test="$refTable != ''">
   INT REFERENCES
   <xsl:value-of select="$refTable"/>(id)
   </xsl:when>
      <xsl:otherwise>INT</xsl:otherwise>
     </xsl:choose>
    </xsl:when>
    <xsl:otherwise>VARCHAR(255)
  </xsl:otherwise>
   </xsl:choose>
  </xsl:when>
  <!-- Child elements -->
   <xsl:when test="self::element">
    <xsl:value-of select="@name"/>
    <xsl:text> VARCHAR(255)
 </xsl:text>
  </xsl:when>
 </xsl:choose>
</xsl:for-each>
```

The following section deals with child element nodes that are not composed by mapping each to an SQL column with the same name and the default type VARCHAR(255).

```
<xsl:when test="self::element">
      <xsl:value-of select="@name"/>
      <xsl:text> VARCHAR(255)</xsl:text>
      </xsl:when>
</xsl:choose>
```

The next fragment formats the SQL output by inserting a comma after every column definition except the last one. It inserts a line break after each column and concludes the table definition, followed by two line breaks for better readability.

```
 <xsl:if test="position() != last()">
        <xsl:text>,</xsl:text>
      </xsl:if>
      <xsl:text>&#xA;</xsl:text>
    </xsl:for-each>
    <xsl:text>);&#xA;&#xA;
  </xsl:text>
```

The following line applies the same transformation recursively to all nested element nodes of type Composed, so that XSLT can generate additional CREATE TABLE statements for hierarchical structures.

```
 <xsl:apply-templates
 select="element[@type='Composed']"/>
 </xsl:template>
```

The last line defines an XSLT key that indexes all element nodes with the type Composed based on their name attribute. It enables an efficient search for composed elements based on their name, which is used, for example, for the dynamic resolution of foreign key references.

```
<xsl:key name="elementByName"
 match="element[@type='Composed']"
 use="@name"/>
</xsl:stylesheet>
```

This completes the process of creating relational table scripts based on the XML structure provided. The generated XSLT is able to process different configurations of elements and attributes so that it can generate database scripts tailored to the structure specified in the XML source document.

4 Experimental results

At the implementation level, the syntactic analyzer consists of functions corresponding to all G_{dtd} productions, while the semantic analyzer consists of a single function for evaluating attributes in a derivation tree. This evaluation function is a switch statement where there is one case for each G_{dtd} production. We have implemented the G_{dtd} compiler in C++. The resulting model takes a DTD as input and returns the corresponding DTD_{xml} as output. Let's take the DTD as an example:

```
<!DOCTYPE Election [
    <!ELEMENT Election (ListeElectorales,
    Candidates, CandidatesList, Regions,
    AssociatedWiths, Cities, CitiesList,
    Votes)>
    <!ELEMENT ListeElectorales
    (ListeElectorale*)>
    <!ELEMENT ListeElectorale (nom)>
    <!ATTLIST ListeElectorale ListCode ID
      #REQUIRED >
    <!ELEMENT nom (#PCDATA)>
    <!ELEMENT Candidates (candidate*)>
    <!ELEMENT Candidate (nom )>
    <!ATTLIST Candidate numId ID
      #REQUIRED>
    <!ELEMENT CandidateList(CandidateList*)>
    <!ELEMENT CandidateList EMPTY>
    <!ATTLIST Candidatelist ListCode IDREF
      #REQUIRED numId IDREF #REQUIRED>
    <!ELEMENT Regions (Region*)>
    <!ELEMENT Region (nom)>
    <!ATTLIST Region RegionCode ID
      #REQUIRED >
    <!ELEMENT AssociatedWiths(AssociatedWith*)>
    <!ELEMENT AssociatedWith EMPTY>
    <!ATTLIST AssociatedWith ListCode ID
      #REQUIRED RegionCode ID
      #REQUIRED>
    <!ELEMENT Cities (City*)>
    <!ELEMENT City (nom)>
    <!ATTLIST City CityCode ID #REQUIRED>
    <!ELEMENT Citieslist (CityList*)>
    <!ELEMENT CityList EMPTY>
    <!ATTLIST CityList RegionCode ID
      #REQUIRED CityCode ID #REQUIRED>
    <!ELEMENT Votes (Vote*)>
    <!ELEMENT Vote EMPTY>
    <!ATTLIST Vote ListCode ID #REQUIRED
    CityCode ID #REQUIRED>
]>
```

After applying our conversion model, we obtain the following DTD_{xml} file:

```
<doctype name = "Election">
   <element name = "Election"
        type ="Composed">
  <seq>
    <element name= "ListeElectorales"/>
    <element name= "Candidates"/>
    <element name= "CandidatesList"/>
    <element name= "Regions"/>
    <element name= "AssociatedWiths"/>
    <element name= "Cities"/>
    <element name= "CitiesList"/>
    <element name= "Votes"/>
  </seq>
 </element>
 <element name = "ListeElectorales"
           type ="Composed">
  <alt>
    <element name= "ListeElectorale"
           occurs= "*"/>
  </alt>
 </element>
 <element name = "ListeElectorale"
           type ="Composed">
  <alt>
    <element name= "nom"/>
  </alt>
  <attributes name= "ListeElectorale">
  <attribute name= "ListCode"
          type ="ID"
          option = "REQUIRED"/>
  </attributes>
 </element>
 <element name = "nom"
                type ="PCDATA">
 </element>
 <element name = "Candidates"
           type ="Composed">
  <alt>
    <element name= "candidate"
          occurs= "*"/>
  </alt>
 </element>
 <element name = "Candidate"
           type ="Composed">
  <alt>
    <element name= "nom"/>
  </alt>
  <attributes name= "Candidate">
  <attribute name= "numId"
           type ="ID"
           option = "REQUIRED"/>
  </attributes>
 </element>
 <element name = "CandidateList"
           type ="Composed">
  <alt>
```

```
   <element name= "CandidateList"
           occurs= "*"/>
 </alt>
</element>
<element name = "CandidateList"
           type ="EMPTY">
 <attributes name= "Candidatelist">
   <attribute name= "ListCode"
               type ="IDREF"
               option = "REQUIRED"/>
   <attribute name= "numId"
               type ="IDREF"
               option = "REQUIRED"/>
 </attributes>
</element>
<element name = "Regions"
               type ="Composed">
 <alt>
   <element name= "Region"
                  occurs= "*"/>
 </alt>
</element>
<element name = "Region"
           type ="Composed">
 <alt>
   <element name= "nom"/>
 </alt>
 <attributes name= "Region">
 <attribute name= "RegionCode"
           type ="ID"
           option = "REQUIRED"/>
 </attributes>
</element>
<element name = "AssociatedWiths"
            type ="Composed">
 <alt>
   <element name= "AssociatedWith"
           occurs= "*"/>
 </alt>
</element>
<element name = "AssociatedWith"
        type ="EMPTY">
 <attributes name= "AssociatedWith">
 <attribute name= "ListCode"
           type ="ID"
           option = "REQUIRED"/>
 <attribute name= "RegionCode"
           type ="ID"
           option = "REQUIRED"/>
 </attributes>
</element>
<element name = "Cities"
               type ="Composed">
 <alt>
   <element name= "City"
                  occurs= "*"/>
 </alt>
</element>
<element name = "City"
               type ="Composed">
 <alt>
   <element name= "nom"/>
 </alt>
 <attributes name= "City">
 <attribute name= "CityCode"
           type ="ID"
           option = "REQUIRED"/>
 </attributes>
</element>
<element name = "Citieslist"
            type ="Composed">
 <alt>
   <element name= "CityList"
           occurs= "*"/>
 </alt>
</element>
<element name = "CityList"
               type ="EMPTY">
 <attributes name= "CityList">
 <attribute name= "RegionCode"
               type ="ID"
               option = "REQUIRED"/>
 <attribute name= "CityCode"
                  type ="ID"
               option = "REQUIRED"/>
 </attributes>
</element>
<element name = "Votes"
            type ="Composed">
 <alt>
   <element name= "Vote"
           occurs= "*"/>
 </alt>
</element>
<element name = "Vote"
               type ="EMPTY">
 <attributes name= "Vote">
   <attribute name= "ListCode"
               type ="ID"
               option = "REQUIRED"/>
           <attribute name= "CityCode"
               type ="ID"
               option = "REQUIRED"/>
 </attributes>
 </element>
</doctype>
```

The XSLT created is able to process different configurations of elements and attributes and thus generate database scripts tailored to the structure specified in the XML source document.

After applying XSLT, we have obtained the following SQL code:

```
CREATE DATABASE Election;
CREATE TABLE Election (
);
CREATE TABLE ListeElectorales (
);
CREATE TABLE ListeElectorale (
    ListCode INT PRIMARY KEY
);
CREATE TABLE Candidates (
);
CREATE TABLE Candidate (
    numId INT PRIMARY KEY
);
CREATE TABLE CandidateList (
);
CREATE TABLE Regions (
);
CREATE TABLE Region (
    RegionCode INT PRIMARY KEY
);
CREATE TABLE AssociatedWiths (
);
CREATE TABLE Cities (
);
CREATE TABLE City (
    CityCode INT PRIMARY KEY
);
CREATE TABLE Citieslist (
);
CREATE TABLE Votes (
);
```

Let us take another example of a DTD:

```
<!DOCTYPE company [
    <!ELEMENT company
        (employee+, department*)>
    <!ELEMENT employee (name, title, email?)>
    <!ATTLIST employee
        id ID #REQUIRED
        dept IDREF #IMPLIED
        status (active | inactive) ”active”>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT email (#PCDATA)>
    <!ELEMENT department (dname, location)>
    <!ATTLIST department
    id ID #REQUIRED>
    <!ELEMENT dname (#PCDATA)>
    <!ELEMENT location (#PCDATA)> ]>
```

We obtain the following DTD_{xml} file after applying our model conversion:

```
<doctype name="company">
 <element name="company"
     type="Composed">
  <element name="employee"
      occurs="+"/>
 </element>
 <element name="employee"
     type="Composed">
  <element name="name"/>
  <element name="title"/>
  <element name="email"/>
  <attributes name="employee">
   <attribute name="id" type="ID"
       option="REQUIRED"/>
   <attribute name="dept" type="IDREF"
       option="IMPLIED"/>
   <attribute name="status"
       type="Enumerate"
       valuesList="(active|inactive)"
       initial="active"/>
  </attributes>
 </element>
 <element name="name" type="PCDATA"/>
 <element name="title" type="PCDATA"/>
 <element name="email" type="PCDATA"/>
 <element name="department"
     type="Composed">
  <element name="dname"/>
  <element name="location"/>
  <attributes name="department">
   <attribute name="id" type="ID"
     option="REQUIRED"/>
  </attributes>
 </element>
 <element name="dname"
     type="PCDATA"/>
 <element name="location"
     type="PCDATA"/>
</doctype>
```

After applying XSLT, we obtained the following SQL code:

```
CREATE DATABASE company;
CREATE TABLE company (
    employee VARCHAR(255)
);
CREATE TABLE employee (
    name VARCHAR(255),
    title VARCHAR(255),
    id INT PRIMARY KEY,
    dept INT,
    status VARCHAR(255)
);
CREATE TABLE department (
    dname VARCHAR(255),
```

```
    id INT PRIMARY KEY
);
```

Table 10 in Appendix contains a comparative analysis of existing mapping approaches. The methods belonging to the semantic paradigm utilize the semantic constraints inherent in XML documents. While this paradigm can generate relational schemas with less redundancy, it ignores the broader concept of functional dependencies. Furthermore, none of the approaches examined provides a generalizable platform for transforming DTDs into alternative models by defining a simple XSLT transformation.

5 Discussion and Conclusion

In this paper we formalize the structuring phase in the conversion of Document Type Definitions (DTDs) into relational database schemas. Based on principles from compiler theory, we introduce an attribute grammar that is able to characterize the entire class of DTDs within the framework of an algebraic language. This formalism enables a systematic translation of a DTD into an intermediate XML based representation. The intermediate model faithfully preserves the semantic content of the original DTD, including the element and attribute declarations. The transformation algorithm specified for the original DTDs can be applied directly and without modification to this intermediate representation. Due to the XML centric nature of the model, the transformation process can be expressed declaratively using XSLT stylesheets. We analyzed the conversion time of several DTDs of varying sizes and complexity containing element and attribute definitions. Table 11 in Appendix presents the execution times required for the conversion. It is clear that our algorithm, implemented in C++ and XSLT, achieves a reduced execution time, sufficient to enable real time conversion.
As example let us consider the following DTD which contains all types of options:

```
<!DOCTYPE digitalLibrary [
    <!ELEMENT digitalLibrary (section+)>
    <!ELEMENT section (title, (book | magazine)+,
notes?)>
    <!ATTLIST section
        id ID #REQUIRED
        type(fiction|nonfiction|reference)
”fiction”>
    <!ELEMENT book (title, author+, publisher?,
year, genre*, description?)>
    <!ATTLIST book
        isbn CDATA #REQUIRED
        available (yes|no) ”yes”
        language CDATA #IMPLIED>
    <!ELEMENT author (#PCDATA)>
    <!ATTLIST author
        id ID #IMPLIED
        nationality CDATA #IMPLIED>
    <!ELEMENT publisher (#PCDATA)>
    <!ELEMENT year (#PCDATA)>
    <!ELEMENT genre (#PCDATA)>
    <!ELEMENT description
        (good | highlight | note)*>
    <!ELEMENT highlight (#PCDATA)>
    <!ATTLIST highlight color CDATA
        ”yellow”>
    <!ELEMENT note (#PCDATA)>
    <!ATTLIST note type
    (editorial|author|reader) ”editorial”>
    <!ELEMENT magazine
    (title, editor+, type, issue?, description?)>
    <!ATTLIST magazine
        issn CDATA #REQUIRED
        subscription (free|paid) ”paid”
        language CDATA #IMPLIED
    >
    <!ELEMENT editor (#PCDATA)>
    <!ATTLIST editor role CDATA ”chief”>

    <!ELEMENT title (#PCDATA)>
    <!ELEMENT issue (#PCDATA)>
    <!ATTLIST issue month CDATA #IMPLIED
            year CDATA #REQUIRED>
    <!ELEMENT notes (note*)>
    ]>
```

The time needed to convert this DTD to the following SQL code is equal to 10ms:

```
CREATE DATABASE digitalLibrary;
CREATE TABLE digitalLibrary (
);
CREATE TABLE section (
        id INT PRIMARY KEY,
        type VARCHAR(255)
);
CREATE TABLE book (
        isbn VARCHAR(255),
        available VARCHAR(255),
        language VARCHAR(255)
);
CREATE TABLE description (
);
CREATE TABLE magazine (
        issn VARCHAR(255),
        subscription VARCHAR(255),
        language VARCHAR(255)
```

```
);
CREATE TABLE notes (
);
```

For more complex DTDs that require substantial memory to represent the tree structure, each relevant component of the DTD can be converted separately. Our methodology shows that the use of attribute grammars to model the structural transformation of DTDs into relational schemas provides a generalizable framework that is applicable to other model to model transformations, especially when the original model is text based and subject to formal syntactic rules. To operationalize our approach, we have developed a dynamic XSLT stylesheet that transforms DTDs into a structured XML format suitable for creating relational schema scripts. The XSLT transformation contains built in validation mechanisms that enforce syntactic and semantic conformance of the generated output. The structural integrity of the resulting relational schema is verified by consistency checks performed throughout the transformation pipeline. In addition, a number of optimization strategies have been integrated to improve the performance characteristics of the generated SQL scripts and ensure their efficiency when used in a database execution environment.

Access conditions

We have developed a prototype based on a client server architecture to validate our proposed approach. The server exposes a web API that accepts a DTD file from the client application via an HTTP POST request. Upon receiving the request, the server initiates inter process communication to invoke a custom C++ compiler module responsible for processing the DTD. The compiler generates the corresponding XML output, which is then returned to the client as part of the HTTP response.

The client application comprises two main views. The first view provides a graphical interface for uploading and converting a DTD to XML. The second view enables the application of an XSLT transformation to the XML output produced in the first step. The compiled executable program used in this implementation is available at [36].

Acknowledgment

Special thanks to Miss Ferial Srour Nemr for her excellent proofreading.

Declaration of Generative AI and AI-assisted Technologies in the Writing Process

The authors wrote, reviewed and edited the content as needed and verifies that none utilized artificial intelligence (AI) tools were used. The authors take full responsibility for the content of the publication.

References:

[1] Humam Kourani, Alessandro Berti, Daniel Schuster, and Wil M. P. van der Aalst. *Process Modeling with Large Language Models*, page 229–244. Springer Nature Switzerland, 2024.

[2] M. Machkour, K. Afdel, and Y. I. Khamlichi. A reversible conversion methodology: Between xml and object relational models. In *International Conference on Information and Communication Systems (ICICS), Irbid, Jordan*, pages 270–275, 2016.

[3] Saba Amir Mohammad, Shahab Elham, Abdolrahimpour Hadi, Hakimi Mahsa, and Moazzam Akbar. A comparative analysis of xml documents, xml enabled databases and native xml databases. *arXiv*, 2017.

[4] Radoslava Stankova Kraleva, Velin Spasov Kralev, Nina Sinyagina, Petia Koprinkova Hristova, and Nadejda Bocheva. Design and analysis of a relational database for behavioral experiments data processing. *International Journal of Online and Biomedical Engineering (iJOE)*, 14(02):pp. 117–132, Feb. 2018.

[5] D. Florescu and D. Kossman. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in A Relational Database. In *Research Report, RR 3680, Inria 00072991*, pages 4–35, 1999.

[6] D. Lee and W. Chu. Cpi: Constraints preserving inlining algorithm for mapping xml dtd to relational schema. *Data and Knowledge Engineering*, 39:3–25, 2001.

[7] Q. Lee, S. Bressan, and W. Rahayu. XShreX:Maintaining Integrity Constraints in the Mapping of XML Schema to Relational. In *Proceedings of the 17th International Conference on Database and Expert Systems Application*, pages 492–496, 2006.

[8] S. Lu, Y. Sun, M. Atay, and F. Fotouhi. A new inlining algorithm for mapping XML DTDs to relational schemas. In *Proceedings of the First International Workshop on XML Schema and Data Management (XSDM2003) LNCS 2814, Spinger, New York*, pages 366–377, 2003.

[9] P. Bohannon, J. Freire, J.R. Haritsa, M. Ramanath, P. Roy, and J. Simeon. Bridging the XML relational divide with LegoDB: a demonstration. In *Proceedings of the 19th International Conference on Data Engineering (ICDE2003) IEEE Computer Society*, page 759–760, 2003.

[10] R. Bourret, C. Bornhovd, and A. Buchmann. A generic load/extract utility for data transfer between XML documents and relational databases. In *Proceedings Second International Workshop on Advanced Issues of Ecommerce and Web Based Information Systems*, pages 134–143, 2000.

[11] Y. Chen, S.B. Davidson, and Y. Zheng. Constraints preserving XML storage in relations. *Technical Report, MSCIS 02-04, University of Pennsylvania*, 2002.

[12] D. Lee, M. Mani, and W.W. Chu. Schema conversion methods between XML and relational models. *B. Omelayenko, M. Klein (Eds.), Knowledge Transformation for the Semantic Web, IOS Press*, pages 1–17, 2003.

[13] S. Davidson, W. Fan, C. Hara, and J.Qin. Propagating xml constraints to relations. In *Proceedings of the 19th International Conference on Data Engineering*, pages 543–554, 2003.

[14] C. Liu, M. Vincent, and J. Liu. Constraint preserving transformation from relational schema to xml schema. *Journal of World Wide Web*, 9(1):93–110, 2006.

[15] Teng Lv and Ping Yan. Mapping DTDs to relational schemas with semantic constraints. *Information and Software Technology*, 48:245–252, 2006.

[16] Z. Tan, J. Xu, W. Wang, , and B. Shi. Storing normalized xml documents in normalized relations. In *In Proceedings of The Fifth International Conference on Computer and Information Technology*, page 123–129, 2005.

[17] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *In Proceedings of the 25th VLDB Conference, Edinburgh, Scotland*, page 302–314, 1999.

[18] A. J. Rafsanjani and S. M. Hosseinabadi. RIAL: Redundancy Reducing Inlining Algorithm to Map XML DTD to Relations. In *In Proceedings of International Conferences on Computational Intelligence for Modelling Control and Automation*, 2008.

[19] Qing Zhu and Wangdong Yang. Mapping from the XML Schema to the Relational Database with Functional Dependency Preserved. In *In Proceedings of the International Conference on Machine Vision and Human Machine Interface (MVHI)*, 2020.

[20] Li Yang and Naphtali Rishe. Transforming semodm semantic schemas to dtds. In *Proceedings of the 43rd annual Southeast regional conference*, pages 237–242, 2005.

[21] A. A. Abd El Aziz and A. Kannan. Survey of mapping xml dtds (documents) to relational schemas. *International Journal of Computer Science and Management Research*, 2:1175–1193, 2013.

[22] S. Chaudhuri and Kyuseok Shimi. Storage and retrieval of xml data using relational databases. In *In Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405).*, 2003.

[23] Gang Gou and Rada Chirkova. Efficiently querying large xml data repositories: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1381–1403, 2007.

[24] Irena Mlynkova and Jaroslav Pokorny. Adaptability of methods for processing xml data using relational databases the state of the art and open problems. *Int. J. Comput. Sci. Appl*, 4(2):43–62, 2007.

[25] Yasser Abdel Kader, Barry Eaglestone, , and Siobhán North. An analysis of relational storage strategies for partially structured xml. . *In WEBIST (1)*, 1:165–170, 2008.

[26] Maya Mourya and Preeti Saxena. Survey of xml to relational database mapping techniques. *Adv. Comput. Sci. Inf. Technol. (ACSIT)*, 2:162–166, 2015.

[27] Amjad Qtaish and Mohammad T. Alshammari. A narrative review of storing and querying xml documents using relational database. *Journal of Information and Knowledge Management*, 18, 2019.

[28] Dusan Petkovic and Ali Piriyaie. Shredding json data into relational environment. *International Journal of Computer Applications*, 174:25–29, 2021.

[29] Craig Chasseur, Yinan Li, and Jignesh M. Patel. Enabling json document stores in relational systems. *In WebDB*, 13:14–15, 2013.

[30] Rahwa Bahta and Mustafa Atay. An experimental study on query processing efficiency of native xml and xml enabled database systems. In *In Proceedings of the 2019 ACM Southeast Conference*, pages 233–236, 2019.

[31] Marcin Wylot, Manfred Hauswirth, Philippe Cudré Mauroux, and Sherif Sakr. Rdf data storage and query processing schemes: A survey. *ACM Computing Surveys (CSUR)*, 51:1–36, 2018.

[32] P. Bohannon, J. Freire, P. Roy, and J. Simeon. Form XML Schemas to relations: a cost based approach to XML Storage. In *Proceedings of the 18th International Conference on Data Engineering (ICDE2002) IEEE Computer Society*, page 564–580, 2002.

[33] M. Yoshikawa, T. Amagasa, and T. Shimura. XRel: A Path based Approach To Storage and Retrieval of XML Documents Using Relational Database. *ACM Transactions on Internet Technology*, 1:110–141, 2001.

[34] J. Qin, SM. Zhao, SQ. Yang, and WH. Dou. XPEV: A Storage Model for Well Formed XML Documents. In *In: Wang, L., Jin, Y. (eds) Fuzzy Systems and Knowledge Discovery. FSKD 2005. Lecture Notes in Computer Science,Springer, Berlin, Heidelberg*, volume 3613, 2005.

[35] Souheil Tawk, Maroun Abi Assaf, Joseph Constantin, Kablan Barbar, and Jacques Bou Abdo. Design and analysis of a relational database for behavioral experiments data processing. *WSEAS Transactions on Information Science and Applications*, 21(02):pp. 547–557, 2024.

[36] DTD Conversion, [Online] http://dtdconversion.ul.edu.lb (Accessed Date: February 10, 2025).

Contribution of Individual Authors to the Creation of a Scientific Article (Ghostwriting Policy)

Souheil Tawk is a PhD student of the dissertation ”Systems based on attribute grammars for model conversion: Application to the conversion of DTDs to relational schemas”. He performed the experimental part and the optimization

Kablan Barbar is the supervisor of the dissertation. He was responsible for the logical part and the implementation of the attributed grammar equations.

Joseph Constantin is the cosupervisor of the thesis. He was responsible for finding the grammar productions, the assigned grammar equations, the logic and the algorithms part, and writing the paper.

Sources of Funding for Research Presented in a Scientific Article or Scientific Article Itself

No funding was received for conducting this study.

Conflicts of Interest

The authors have no conflicts of interest to declare that are relevant to the content of this article.

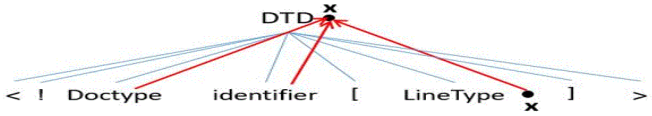
APPENDIX

Table 1: Semantic rules with graphic representation for the productions P_0-P_3 of the grammar G_{dtd}. Source: created by the authors.

P0: Start → DTD

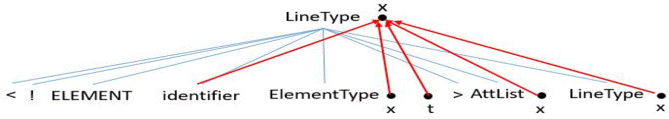
x(Start) = x(DTD)

P1: DTD → <!DOCTYPE identifier [LineType]>



x(DTD) = ”< Doctype name = ” & identifier & ” > ” & x(LineType) & ”</Doctype>”

P2: LineType → <!ELEMENT identifier ElemnetType > AttList LineType



x($LineType_0$) = ”<element name = ” & identifier & ” type = ” & t(ElementType) & ”>”

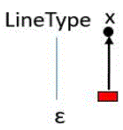
if ((t(ElementType) == ”EMPTY” ‖ t(ElementType) == ”ANY” ‖ t(ElementType) == ”PCDATA”) & (x(AttList) != ””)) then
x($LineType_0$) = x($LineType_0$) & x(AttList) & ” </element>” & x($LineType_8$)
endif

if ((t(ElementType) == ”EMPTY” ‖ t(ElementType) == ”ANY” ‖ t(ElementType) == ”PCDATA”) & (x(AttList) == ””)) then
x($LineType_0$) = x($LineType_0$) & ”</element>” & x($LineType_8$)
endif

if ((t(ElementType) == ”Composed”) & (x(AttList) != ””)) then
x($LineType_0$) = x($LineType_0$) & x(ElementType) & x(AttList) & ” </element>” + x($LineType_8$)
endif

if ((t(ElementType) == ”Composed”) & (x(AttList) == ””)) then
x($LineType_0$) = x($LineType_0$) & x(ElementType) & ” </element>” + x($LineType_8$)
endif

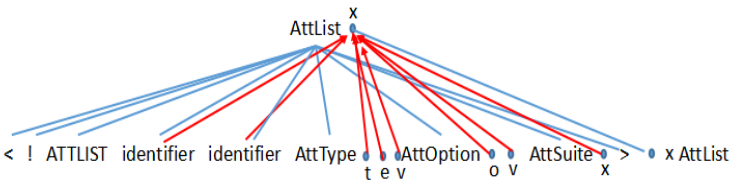
P3: LineType → ϵ



X(LineType) = ” ”

Table 2: Semantic rules with graphic representation for the productions P_4-P_7 of the grammar G_{DTD}.Source: created by the authors.

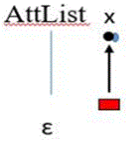
P4: AttList → <!ATTLIST identifier identifier AttType AttOption AttSuite> AttList



x(AttList) = ”<attributes name= ” & identifier & ”>”
x(AttList) = x(AttList) & ”<attribute name= ” & identifier + ” type = ” & t(AttType)
if (t(AttType) == ”Enumerate”) then
x(AttList) = x(AttList) & ” valuesList= ” & e(AttType) & ”ïnitial=”□ & o(AttOption)
endif
if (o(AttOption != ”” and t(AttType) != ”Enumerate”) then
x(AttList) = x(AttList) & ” option = ” & o(AttOption)
endif
if (o(AttOption == ”FIXED”) then
x(AttList) = x(AttList) & ” initial= ” & o(AttOption)
endif

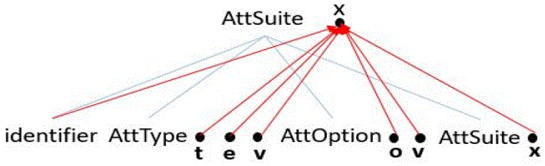
x(AttList) = x(AttList) & ”/>” & x(AttSuite)
x(AttList) = x(AttList) & ”</attributes>”

P5: AttList → ϵ



x(AttList) = ” ”

P6: AttSuite → identifier AttType AttOption AttSuite

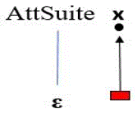


x($AttSuite_0$) = ”<attribute name= ” & identifier & ” type =”□ & t(AttType)
if (t(Atttype) == ”Enumerate”) then
x($AttSuite_0$) = x($AttSuite_0$) & ” valuesList= ” & e(AttType) & ”ïnitial= ” & o(AttOption)
endif

if (o(AttOption) != ”” and t(Atttype) != ”Enumerate”) then
x($AttSuite_0$) = x($AttSuite_0$) & ” option = ” & o(AttOption)
endif
if (o(AttOption) == ”FIXED”) then
x($AttSuite_0$) = x($AttSuite_0$) & ” initial= ” & o(AttOption)
endif
x($AttSuite_0$) = x($AttSuite_0$) & ”/>” & x(AttSuite4)

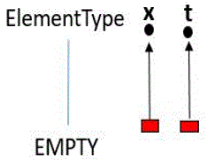
Table 3: Semantic rules with graphic representation for the productions P_8-P_{11} of the grammar G_{dtd}. Source: created by the authors.

P7: AttSuite $\rightarrow \epsilon$



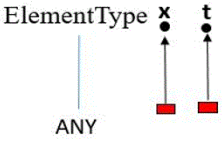
x(AttSuite) = ” ”

P8: ElementType $\rightarrow$ EMPTY



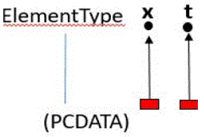
t(ElementType) = ” Empty ”
x(ElementType) = ””

P9:ElementType $\rightarrow$ ANY



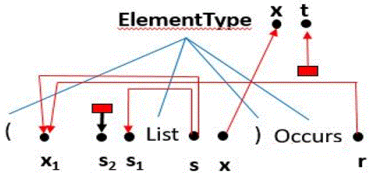
t(ElementType) = ”ANY”
x(ElementType) = ” ”

P10: ElementType $\rightarrow$ (PCDATA)



t(ElementType) = ”#PCDATA”
x(ElementType) = ” ”

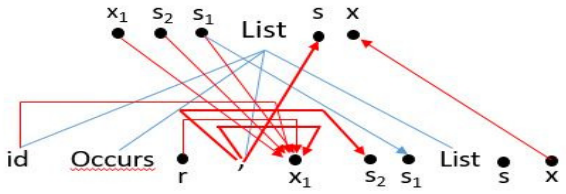
P11: ElementType $\rightarrow$ (List)Occurs



s2(list) = ” ”;
t(ElementType) = ”composed”
s1(List2) = s(List2)
if(r(Occurs) = ’ ’) then occ = ” ”
else occ = ” occurs = ” & r(Occurs) & ” ”
endif
If s(List) = ”,” then x1 (List) = ”<seq>” & occ & ”</seq>”
else x1(List) = ” <alt>” & occ & ” </alt> ”
endif
x(ElementType) = x(List)

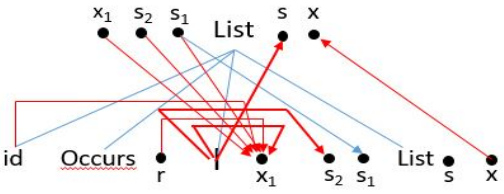
Table 4: Semantic rules with graphic representation for the productions P_{12}-P_{13} of the grammar G_{dtd}.Source: created by the authors.

P12: List → id Occurs , List



s1(List4)= s1(List0); s2(list4) = ’,’; s(List0) = ’,’
if(r(Occurs) = ’ ’) then occ = ””
else occ = ” Occurs = ” & r(Occurs) & ” ”
endif
CurrentElement = ”<element name = ” & id & ” ” & occ & ” />”
if(s2(List0)= ’ ’) then
x1(List4)= InsertBeforeTheLastTag (CurrentElement , x1(List0))
else if (s2(List0)= ’,’ and s1(List0)= ’,’) then
x1(List4)= InsertBeforeTheLastTag (CurrentElement , x1(List0))
else if (s2(List0)= ’,’ and s1(List0) != ’,’) then
x1(List4)= InsertBeforeTheLastTwoTags (CurrentElement , x1(List0))
else if (s2(List0) != ’,’ and s1(List0)= ’,’) then
x1(List4)= InsertBeforeTheLastTwoTags (CurrentElement , x1(List0))
else
x1(List4)= InsertBeforeTheLastTag (”<seq>” & CurrentElement & ”</seq>” , x1(List0))
endif
x(List0) = x(List4)

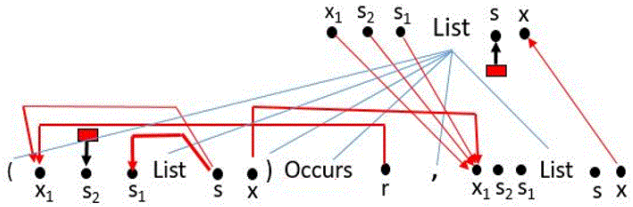
P13 : List → (List) Occurs , List



s1(List4) = s1(List0); s2(List4) = ’|’; s(List0) = ’|’
if r(Occurs = ’ ’) then occ = ””
else occ = ” Occurs =”” & r(Occurs) & ” ”
endif
CurrentElement = ”<element name =’ ” & id & ” ” & occ & ” />”
if(s2(List0)= ’ ’) then
x1(List4)= InsertBeforeTheLastTag (CurrentElement , x1(List0))
else if (s2(List0)= ’|’ and s1(List0)= ’|’) then
x1(List4)= InsertBeforeTheLastTag (CurrentElement , x1(List0))
else if (s2(List0)= ’|’ and s1(List0) != ’|’) then
x1(List4)= InsertBeforeTheLastTwoTags (CurrentElement , x1(List0))
else if (s2(List0) != ’|’ and s1(List0)= ’|’) then
x1(List4)= InsertBeforeTheLastTwoTags (CurrentElement , x1(List0))
else
x1(List4)= InsertBeforeTheLastTag (”<alt>” & CurrentElement & ”</alt>” , x1(List0))
endif
x(List0) = x(List4)

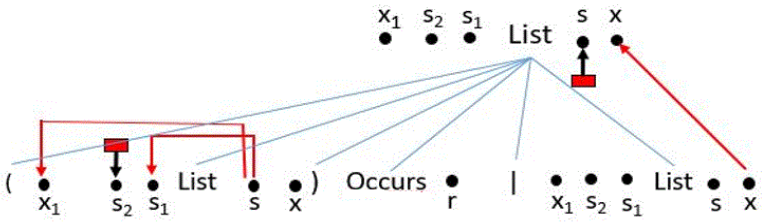
Table 5: Semantic rules with graphic representation for the productions P_{14}-P_{15} of the grammar G_{dtd}.Source: created by the authors.

P14: List → (List) Occurs , List



s2(List2) = ’ ’ ; s(List0) = ’,’
s1(List2) = s(List2)
if r(Occurs = ’ ’) then occ = ””
else occ = ” Occurs =” & r(Occurs) & ” ”
endif
if(s2(List0) = ’ ’) then
x1(List6) = InsertBeforeTheLastTag (x1(List2) , x1(List0))
else if (s2(List0)= ’,’ and s1(List0) = ’,’) then
x1(List6)= InsertBeforeTheLastTag (x(List2) , x1(List0))
else if (s2(List0) = ’,’ and s1(List0) != ’,’) then
x1(List6)= InsertBeforeTheLastTwoTags (x(List2), x1(List0))
else if (s2(List0) ! = ’,’ and s1(List0) = ’,’) then
x1(List6)= InsertBeforeTheLastTwoTags (x(List2), x1(List0))
else x1(List6)= InsertBeforeTheLastTag (”<seq>” & x(List2) & ”</seq>” , x1(List0))
endif
x(List0) = x(List6)

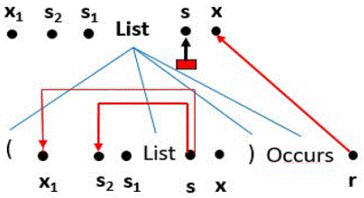
P15 : List → (List) Occurs | List



s1(List4) = s1(List0); s2(List4) = ’|’; s(List0) = ’|’ ;
if r(Occurs = ’ ’) then occ = ” ”
else occ = ” Occurs = ” & r(Occurs) & ” ”
endif
if(s(List1) = ’,’) then x1(List1) = ”<seq>” & occ & ”</seq>”
else x1(List1) = ”<alt>” & occ & ” </alt>”
endif
if(s2(List0)= ’ ’) then
x1(List4)= InsertBeforeTheLastTag (CurrentElement,x1(List0))
else if (s2(List0)= ’|’ and s1(List0)= ’|’) then
x1(List4)= InsertBeforeTheLastTag (CurrentElement,x1(List0))
else if (s2(List0)= ’|’ and s1(List0) != ’|’) then
x1(List4)= InsertBeforeTheLastTwoTags (CurrentElement,x1(List0))
else if (s2(List0) != ’|’ and s1(List0)= ’|’) then
x1(List4)= InsertBeforeTheLastTwoTags (CurrentElement,x1(List0))
else
x1(List4)= InsertBeforeTheLastTag (”<alt>” & CurrentElement & ”</alt>”,x1(List0))
endif
x(List0) = x(List4)

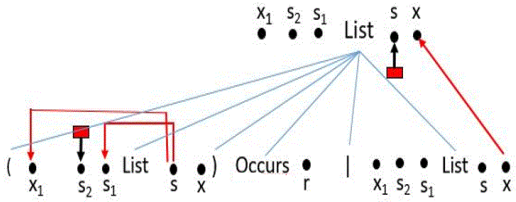
Table 6: Semantic rules with graphic representation for the productions P_{16}-P_{18} of the grammar G_{dtd}.Source: created by the authors.

P16: List → (List) Occurs



s2(List2) = ’ ’ ; s(List0) = ’ ’ ;
s2(List2) = s(List2) ;
if r(Occurs = ’ ’) then occ = ””
else
occ = ” Occurs =” & r(Occurs) & ” ”
endif
if(s(List2) = ’,’) then
x1(List2) = ”<seq>” & occ & ”</seq>”
else
x1(List2) = ”<alt>” & occ & ”</alt>”
endif
if(s2(List0) = s1(List0)) then
x(List0) = InsertBeforeTheLastLastTag (x(List2) , x1(List0))
else
x(List0) = InsertBeforeTheLastTwoLastTags (x(List2) , x1(List0))
endif

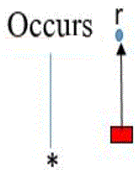
P17 : List → id Occurs



s(List0) = ’ ’ ;

if r(Occurs = ’ ’) then occ = ””
else occ = ”Occurs =” & r(Occurs) & ” ”
endif
CurrentElement = ”<element name =’ ” & id & ” ” & occ & ” />”
if(s2(List0) = s1(List0)) then
x(List0) = InsertBeforeTheLastLastTag (CurrentElement , x1(List0))
else
x(List0) = InsertBeforeTheLastTwoLastTags (CurrentElement , x1(List0))
endif

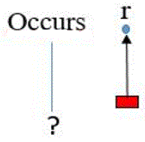
P18: Occurs → *



r(Occurs) = ”*”

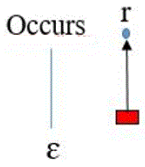
Table 7: Semantic rules with graphic representation for the productions P_{19}-P_{24} of the grammar G_{dtd}.Source: created by the authors.

P19 : Occurs → ?



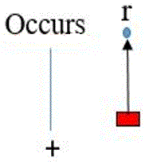
r(Occurs) = ”?”

P20 : Occurs → ϵ



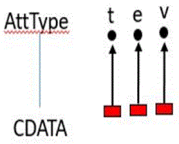
r(Occurs) = ””

P21 : Occurs → +



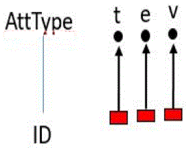
r(Occurs) = ”+”

P22 : AttType → CDATA



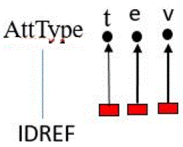
t(AttType) = ”CDATA”
e(AttType) = ””
v(AttType) = ””

P23 : AttType → ID



t(AttType) = ”ID”
e(AttType) = ””
v(AttType) = ””

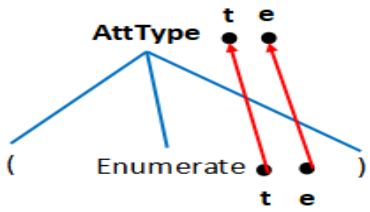
P24 : AttType → IDREF



t(AttType) = ”IDREF”
e(AttType) = ””
v(AttType) = ””

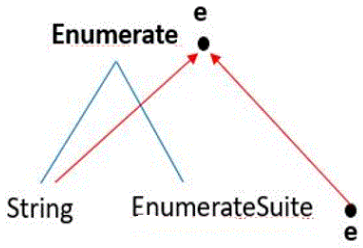
Table 8: Semantic rules with graphic representation for the productions P_{25}-P_{29} of the grammar G_{dtd}. Source: created by the authors.

P25 : AttType → (Enumerate)



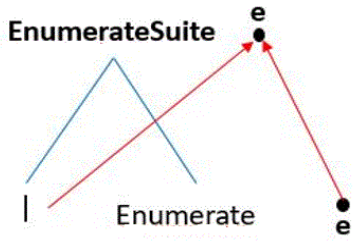
t(AttType) = ”Enumerate”
e(AttType) = ” (” + e(Enumerate2) + ”)”

P26 : Enumerate → String EnumerateSuite



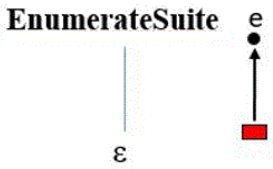
e(Enumerate) = ”string” & ”|” & e(EnumerateSuite)

P27 : Enumerate → | EnumerateSuite



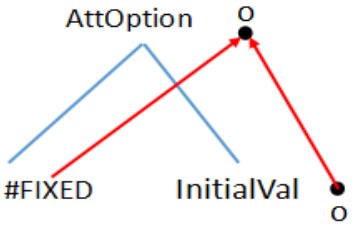
e(EnumerateSuite) = ’|’ & e(Enumerate)

P28 : EnumerateSuite → $epsilon$



e(EnumerateSuite) = ””

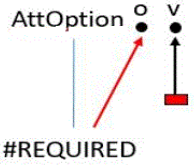
P29 : AttOption → #FIXED InitialVal



o(AttOption) = ” FIXED” & o(InitialVal)

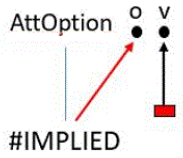
Table 9: Semantic rules with graphic representation for the productions P_{30}-P_{33} of the grammar G_{dtd}. Source: created by the authors.

P30 : AttOption → #REQUIRED



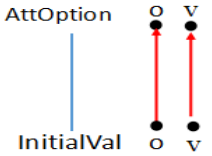
o(AttOption) = ”REQUIRED”, v(AttOption) = ””

P31 : AttOption → #IMPLIED



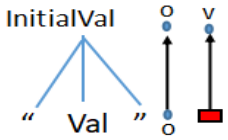
o(AttOption) = ”IMPLIED”, v(AttOption) = ””

P32 : AttOption → InitialVal



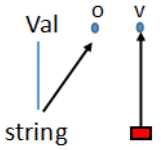
o(AttOption) = o(Initialval), v(AttOption) = ””

P33 : InitialVal → ”Val”



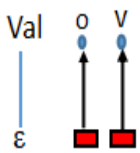
o(InitialVal) = o(Val), v(InitialVal) = ””

P34 : Val → string



o(Val) = string, v(Val) = ””

P35 : Val → ϵ



v(Val) = ” ”, o(Val) = ” ”

Table 10: Comparison of the existing mapping approaches. Sourche: created by the authors.

Approach	schema	Model Based	Structural Based	Semantic Based	DTD To Alternative models
[32]	No	Yes	No	No	No
[33]	No	Yes	No	No	No
[34]	No	Yes	No	No	No
[17]	Yes	No	Yes	No	No
[5]	Yes	No	Yes	No	No
[6]	Yes	No	Yes	No	No
[13]	No	No	No	Yes	No
[14]	Yes	No	No	Yes	No
[11]	Yes	No	Yes	Yes	No
Our Methodology	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Table 11: Conversion time of several DTDs to SQL of varying sizes and complexity. Source: created by the authors.

Data Type Definition (DTD)	Size in number of lines	Conversion time en milliseconds (ms)
Example 1 of a DTD	30 lines	10ms
Example 2 of a DTD	50 lines	20ms
Example 3 of a DTD	50 lines	26ms
Example 4 of a DTD	100 lines	34ms