

Applying Compiling Techniques in DTD Conversion

SOUHEIL TAWK¹, MAROUN ABI ASSAF ^{*2}, JOSEPH CONSTANTIN¹, KABLAN BARBAR¹,
JACQUES BOU ABDO³

¹ LaRRIS, Faculty of Sciences, Lebanese University, Fanar, LEBANON

² Department of Computer Science and Information Technology,
Holy Spirit University of Kaslik, Jounieh, LEBANON

³ School of Information Technology, University of Cincinnati, USA

* Corresponding Author

Abstract: In this paper, we propose to apply compilation techniques to Documents Type Definition (DTD). At the syntactic level, we create a context free grammar, denoted G_{DTD} , for the set of all DTDs in order to generate them. The semantic level is defined by using attributes. Each nonterminal of G_{DTD} is assigned a set of attributes and for each production of G_{DTD} a set of attribute equations is defined. This DTD compilation process can be used in different domains. In this article, we provide an attribute grammar that can be used to eliminate unnecessary parentheses in element declarations in a DTD. In addition, the DTD compilation process can be applied to the domain of model conversion. It makes it possible to split the model conversion into two steps: the syntactic transformation step, which consists of rewriting the source DTD in terms of XML syntax, and the second step of the effective conversion algorithm, which can be easily expressed by an Extensible Stylesheet Language Transformation (XSLT).

Key-Words: Attributed Grammar, Compiler Design, Computation Theory, DTD, Models Conversion

Received: March 14, 2024. Revised: September 13, 2024. Accepted: November 13, 2024. Published: December 20, 2024.

1 Introduction

DTDs are textual models for the structure of Extensible Markup Language (XML) documents which as well as used for data exchange over the Internet, [1]. They are integrated into XML documents as well as converted into relational models of databases. This is because the Structured Query Language (SQL) language offers a very powerful way of extracting data from databases, [2]. It is important to know that we store data in relational databases if we are interested in data manipulation. If we need to exchange or transform data over the Internet, we store it in XML documents.

Many mapping algorithms have been proposed in the literature for the problem of mapping DTDs to relational schemas, [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. Some of these algorithms did not consider the semantics of DTDs during the mapping process, so the resulting relational schema loses the original DTD information, [3], [4], [5]. In [6], the authors use a table and object relational mapping. The table based mapping cannot handle mixed content, and the object based mapping of mixed content is inefficient. Therefore, these two models are poorly suited for centered documents. Other mapping algorithms preserved the cardinality constraints, [7], [8]. However, they did not take functional dependencies via DTDs into account. In [9], the authors developed a mapping algorithm that preserves both the structure

of DTDs and the semantics implied by the constraints of DTDs, but the multivalued dependencies are not preserved by the mapping algorithm. The mapping algorithm in [10], was developed to map DTDs to a relational schema that preserves not only the content and structure, but also the semantics of the original XML documents. A hybrid inline algorithm was proposed to map DTDs to relational schemas while preserving functional dependencies, [11]. In [12], the authors defined the dependency relationships in XML documents and, based on these relationships, created a set of rules for mapping the XML document to a relational database. In [13], the authors described a technique for mapping a specific DTD to a relational schema. This technique consisted of converting the DTD to an Xschema, simplifying the Xschema constraints, and mapping the collection types.

Unfortunately, all studies focus on the conversion algorithms and ignore how to read the DTD source and divide it into small units for the conversion algorithms. We believe that our approach is the first to focus on applying of syntatic and semantic analysis of the compilation process to DTDs. In fact, the concept of attributed grammar has already been applied in various research area, [14], [15], [16]. In [14], the author introduces the concept of attributed grammar for compiler description as a declarative way by associating attributes with nonterminals and attribute equations with productions. In logic programming,

the authors of [15], consider an abstract substitution as an attribute attached to the nodes of a tree, and then the propagation process of abstract substitutions through the tree can be expressed in terms of attribute evaluation. In the field of web development, the authors of [16], give a system that is able to automatically generate web forms or editors to collect data related to predefined DTDs. Once the the data entry is complete, the web based form creates the corresponding XML documents. The system receives a DTD as input and relies on attributed grammars to assign semantics to the elements and generates an HTML based editor on the fly.

In this paper, we want to create a formal framework for analyzing DTDs viewed as text models or strings. For this purpose, we consider the set of DTDs as a context free language called L_{DTD} . We define a context free grammar G_{DTD} that recognizes the language L_{DTD} . Then, any DTD conversion can be realized by enriching the grammar G_{DTD} with an attributed system which consists of associating attributes with the nonterminals and attribute equations with the productions of the grammar G_{DTD} . We give an example of an attribute grammar that eliminates unnecessary parentheses in an element declaration in a DTD. An element declaration is a list or regular expression composed of the operator '?', the disjunction sequence '!', the opening and closing parentheses, and the names of the subelements. Another example is the model conversion from DTD to a relational model, which consists of converting the DTD to another intermediate model in terms of XML syntax. The contribution of this approach is that in the case of multiple transformations, a stylesheet can be written and applied to a suitable intermediate XML based model for each transformation without having to reanalyze the source DTD, [17], [18]. The main advantage of our approach:

1. The formalization of the set of DTDs by a context free grammar G_{DTD} . Then each operation over a DTD consists of defining attributes over G_{DTD} , which means developing an attributed grammar based compiler.
2. The application of G_{DTD} in the field of DTD conversion. We can define attributes via G_{DTD} that convert a given DTD into another DTD while preserving the declaration of elements and attributes and respecting the XML syntax. The intermediate model thus obtained is called DTD_{XML} and any conversion algorithm for the source DTD can be expressed by an XSLT stylesheet applicable to the intermediate model DTD_{XML} . This approach makes it possible to write the DTD conversion process in two separate modules: one module is an attribute gram-

mar application for DTDS or a DTD compiler and the other module is just an XSLT stylesheet for the conversion algorithm (Fig. 1).

The paper is structured as follows: Section 2 explains the principle of algebraic and attributed grammars in arithmetic theory. Section 3 defines an algebraic grammar for generating DTDs. Section 4 defines an attributed grammar for eliminating unnecessary parentheses in DTD declarations. Section 5 shows the results of the experiment, while section 6 summarizes the paper with a conclusion.

2 Algebraic and Attributed Grammars

This section is dedicated to explain the syntactic analysis of DTDs. These concepts include context free and attributed grammars, which form the basis of compilation theory in order to create compilers for programming languages. A context free grammar (CFG) is a system of rules that enables the generation of a set of words from an axiom. The alphabet of the language is divided into two groups: Terminals, i.e. the letters of the generated words, and nonterminals, i.e. intermediate symbols that can be replaced by terminals or other nonterminals. In formal language theory, a context free grammar is a formal grammar whose production rules can be applied to a nonterminal symbol independently of its context. In particular, in a context free grammar, each production rule has the form $A \rightarrow \alpha$, where A is a single nonterminal symbol and α is a string of terminals and/or nonterminals. α can be empty). Regardless of which symbols surround it, the single nonterminal A on the left can always be replaced by α on the right.

Attributed grammars are a formalism for describing the semantic analysis of programming languages (Fig. 2). The principle is to assign attributes to each nonterminal and to assign a system of attribute equations to each production of the grammar, [19], [20]. The attributes are evaluated along a derivation tree

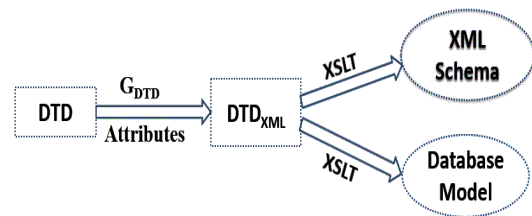


Figure 1: Conversion steps: formalisation of the set of DTDs by a context free grammar G_{DTD} . Transform of the DTD into intermediate model DTD_{XML} using G_{DTD} with attributes. Conversion algorithm can be expressed by an XSLT stylesheet

(for a specific input word) to determine the semantics of the input word. In the case of programming languages, there is a context free grammar for each programming language. The attributes are used to calculate the semantics of all statements of a program when they are considered as input words of the grammar of the corresponding programming language. An Attribute Grammar (AG) is defined as $AG = \langle G, A, E \rangle$ where:

- $G = \langle \Sigma, N, S, P \rangle$ where:
- Σ is the set of terminals symbols.
- N is a set of nonterminal symbols.
- S is the start symbol (axiom) and $X \in N$.
- P is the set of the productions rules.

$$P = p / p: X_0 \rightarrow \alpha_1 X_1 \alpha_2 \dots \alpha_n X_n \alpha_{n+1}$$

where $\forall i \in [0 \dots n], X_i \in N$ and $\forall j \in [1 \dots n + 1], \alpha_j \in \Sigma^*$

- A is a set of attributes divided into two subsets A_s and A_h , $A = A_s \cup A_h$, and for each $X \in N$ are associated: $A_s(X)$ is the set of synthesized attributes where $A_s(X) \subset A_s$, $A_h(X)$ is the set of inherited attributes where $A_h(X) \subset A_h$.
- E is the set of equations in which for each synthesized attribute $s \in A_s(X_0)$, there is an attribute equation defining $s(X_0)$ of the form : $s(X_0) = f_s(s_1(X_1), \dots, s_n(X_n), h(X_0))$ where $\forall i \in [1 \dots n], s_i(X_i) \in A_s(X_i)$ and $h \in A_h(X_0)$, as shown in Fig. 3, and for each inherited attribute $h_j \in A_h(X_j)$, there is an attribute equation defining $h_j(X_j)$ of the form: $h_j(X_j) = f_{h_j}(h_0(X_0), s_1(X_1), \dots, s_n(X_n))$ where $h_0 \in A_h(X_0)$ and $\forall i \in [1 \dots n], s_i \in A_s(X_i)$ as shown in Fig. 4.

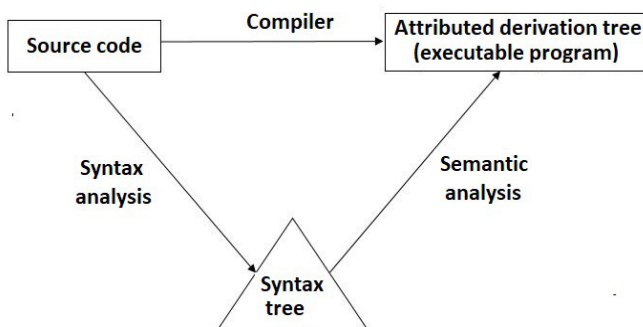


Figure 2: Structure of a compiler. It composed of syntax analysis and semantic analysis.

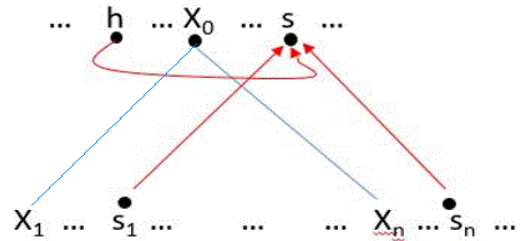


Figure 3: Order of computation of synthesized attributes.

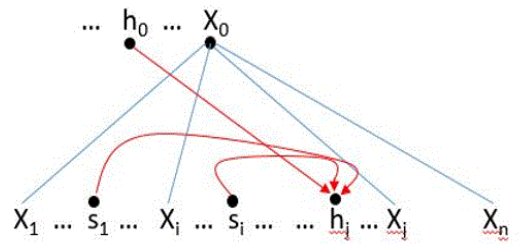


Figure 4: Order of computation of inherited attributes.

3 Definition of an Algebraic Grammar for DTDs

In our approach for the formalization of the syntactical analysis of the DTDs, we define the algebraic grammar $G_{DTD} = \langle \Sigma, N, S, P \rangle$ by :

- $\Sigma = \{ \langle, \rangle, (,), |, *, ?, +, ", !, [,], \#, , , \text{identifier, DOCTYPE, ELEMENT, ATTLIST, EMPTY, ANY, PCDATA, id, CDATA, ID, IDREF, FIXED, REQUIRED, IMPLIED, string} \}$: denotes the alphabet.
- $N = \{ \text{Start, DTD, LineType, Attlist, AttSuite, ElementType, List, Occurs, AttType, Enumerate, EnumerateSuite, AttOption, InitialVal} \}$: is the set of non terminals,
- Start is the axiom.
- P is the following set of productions:
 - $P_0: \text{Start} \rightarrow \text{DTD}$
 - $P_1: \text{DTD} \rightarrow \langle !\text{DOCTYPE identifier} [\text{LineType}] \rangle$
 - $P_2: \text{LineType} \rightarrow \langle !\text{ELEMENT identifier ElementType} \rangle \text{ AttList LineType}$
 - $P_3: \text{LineType} \rightarrow \epsilon$
 - $P_4: \text{AttList} \rightarrow \langle !\text{ATTLIST identifier1 identifier2 AttType AttOption AttSuite} \rangle$

- $P_5: \text{AttList} \rightarrow \epsilon$
- $P_6: \text{AttSuite} \rightarrow \text{identifier AttType AttOption AttSuite}$
- $P_7: \text{AttSuite} \rightarrow \epsilon$
- $P_8: \text{ElementType} \rightarrow \text{EMPTY}$
- $P_9: \text{ElementType} \rightarrow \text{ANY}$
- $P_{10}: \text{ElementType} \rightarrow (\#\text{PCDATA})$
- $P_{11}: \text{ElementType} \rightarrow (\text{List}) \text{Occurs}$
- $P_{12}: \text{List} \rightarrow \text{id Occurs , List}$
- $P_{13}: \text{List} \rightarrow \text{id Occurs | List}$
- $P_{14}: \text{List} \rightarrow (\text{List}) \text{Occurs, List}$
- $P_{15}: \text{List} \rightarrow (\text{List}) \text{Occurs | List}$
- $P_{16}: \text{List} \rightarrow (\text{List}) \text{Occurs}$
- $P_{17}: \text{List} \rightarrow \text{id Occurs}$
- $P_{18}: \text{Occurs} \rightarrow *$
- $P_{19}: \text{Occurs} \rightarrow ?$
- $P_{20}: \text{Occurs} \rightarrow \epsilon$
- $P_{21}: \text{Occurs} \rightarrow +$
- $P_{22}: \text{AttType} \rightarrow \text{CDATA}$
- $P_{23}: \text{AttType} \rightarrow \text{ID}$
- $P_{24}: \text{AttType} \rightarrow \text{IDREF}$
- $P_{25}: \text{AttType} \rightarrow (\text{Enumerate}) \text{InitialVal}$
- $P_{26}: \text{Enumerate} \rightarrow \text{string EnumerateSuite}$
- $P_{27}: \text{EnumerateSuite} \rightarrow \text{Enumerate}$
- $P_{28}: \text{EnumerateSuite} \rightarrow \epsilon$
- $P_{29}: \text{AttOption} \rightarrow \#\text{FIXED InitialVal}$
- $P_{30}: \text{AttOption} \rightarrow \#\text{REQUIRED}$
- $P_{31}: \text{AttOption} \rightarrow \#\text{IMPLIED}$
- $P_{32}: \text{InitialVal} \rightarrow \text{string}$
- $P_{33}: \text{InitialVal} \rightarrow \text{""}'$

We start in the grammar G_{DTD} to write the productions P_0 and P_1 that define the syntax of a DTD . We declare the elements with their productions. We associate a nonterminal to produce an element and another nonterminal to define the type of an element. These nonterminals are $ElementType$ and $LineType$ respectively. We add the $Attlist$ nonterminal because the element can have attributes and the $LineType$ can produce multiple elements (P_2 and P_3). The $Attlist$ can produce attribute declarations, the $AttType$ nonterminal stands for the type of an attribute and the $AttOption$ nonterminal produces the options for an attribute. We write the productions $P_4 - P_7$. The element type can be $(\#\text{PCDATA})$, ANY , EMPTY , composed or generate a list of operators ($P_8 - P_{11}$). To develop the nonterminal $List$, let's look at an example of the element declaration $\langle !\text{element A}(\text{B}, \text{C}^*, (\text{D}|\text{E}), \text{F}^*) \rangle$.

The structure of a type element is therefore a parenthesized expression with operators $(+, *, \dots)$ and separators $(, \text{ and } |)$. We therefore suggest 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 and enumerate with 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 $\#\text{FIXED}$, $\#\text{REQUIRED}$, $\#\text{IMPLIED}$. The initial value is generated in the P_{32} and P_{33} productions. This value can be empty or defined as a string.

Based on this grammar, the derivation tree of the first two lines of following DTD is shown in Fig. 5
 $DTD \rightarrow \langle !\text{DOCTYPE A} [\langle !\text{ELEMENT A}(\text{B}, (\text{C}, \text{D})) \rangle \langle !\text{ELEMENT B}(\#\text{PCDATA}) \rangle \langle !\text{ELEMENT C}(\#\text{PCDATA}) \rangle \langle !\text{ELEMENT D}(\#\text{PCDATA}) \rangle] \rangle$

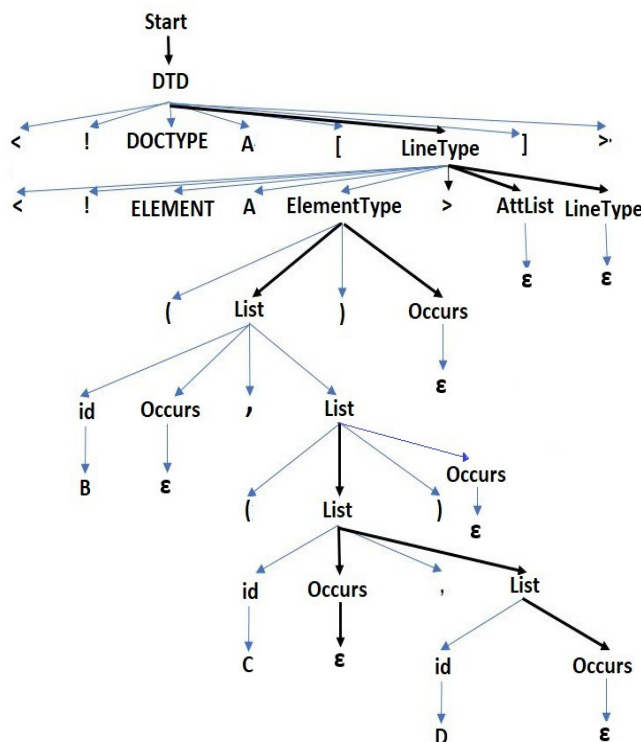


Figure 5: Example of a derivation tree for a given DTD: The productions P_0 and P_1 generate the first line of the DTD. The production P_2 defines the element A and calls P_{11} using the nonterminal $ElementType$. We use the nonterminal $List$ of productions P_{11} , P_{12} , P_{16} and P_{17} to define the elements B , C and D .

4 Attributed Grammar for Simplifying DTD Declarations.

In this section, we give an example of an attributed grammar for eliminating useless parenthesis element declarations in DTDs. The structure of an element declaration in a DTD is a rational expression on the alphabet composed of the operator (,), the or operator (|), the star operator (*), the exclamation mark (!), the parentheses, and the identifiers. We distinguish three categories of lists:

- The lists that contain only the comma operator, e.g. (B, C, D, E, F).
- The lists that contain only the or operator. For example, (B|C|D|E|F).
- The list with the commas and an or operator. As an example, (B, C|D, E).

Now, we will define the notion of useless parenthesis in the structure of the element declaration.

In general, parentheses are considered unnecessary if they do not change the meaning of the declaration and do not improve the readability of the code. In our case, we consider parentheses unnecessary if they only occur in inner lists. For example, the parentheses of the inner list (C, D) are useless in the list (B, (C, D), E) and then (B, (C, D), E) can be reduced to the corresponding list (B, C, D, E). In contrast, the parentheses of the inner list (C, D) should be retained in the list (B | (C, D), E) as the operators of the inner and outer lists are different.

The parentheses of the inner list (D, E, F) in the outer list (B, C, (D, E, F), G) are useless, as the only operator of the list D, E, F is a comma (,) and the preceding and following operators are also a comma (,). The list (B, C, (D, E, F), G) can be reduced to (B, C, D, E, F, G). In addition, the parenthesis of the inner list (C|D) should be retained in the list (A,B,(C|D),E). By definition, the inner brackets are useless iff :

- Its list has a unique operator (, or |).
- The two operators before and after the inner list in the outer list are the same and equal to the unique operator of the inner list.

Now, we consider the occurrence of the inner list in a derivation tree in the grammar G_{DTD} where we express:

- The unique operator of the inner list.
- The preceding and following operators in the form of attributes over the production of G_{DTD} .

First, we need an attribute to calculate the reduced list and the other parts of an element declaration; this attribute is the synthesized attribute which is called "d".

The unique operator of an inner list is represented by a synthesized attribute "o". To calculate the value of "o" for an inner list, we use another inherited attribute "o₁", which is initialized to null at the beginning of an inner list. Thus, the productions and their attributes are given in Table 1, Table 2, Table 3, Table 4, Table 5, Table 6, and Table 7 (Appendix). We can divide the productions of the grammar G_{DTD} into three groups:

- group 1 : from P_0 to P_{11} .
- group 2 : from P_{12} to P_{17} .
- group 3 : from P_{18} to P_{33} .

The productions of group 1 and group 3 reproduce in the attribute "d" all parts of the given DTD that are outside the inner parentheses. If we use the DTD :

```
<! Doctype A [
<! ELEMENT A ((B, (C))) >
<! ELEMENT B (#REQUIRED) >
<! ELEMENT C (#REQUIRED) >
]>
```

The evaluation of the attributes via the productions of group 1 and group 3 generates the main parts of the previous DTD, namely:

```
<! Doctype A [
<! ELEMENT A ( ) >
<! ELEMENT B (#REQUIRED) >
<! ELEMENT C (#REQUIRED) >
]>
```

The attribute evaluation on the productions of group 2 decides whether the parentheses of the inner lists (B, (C)) and (C) in the productions P_{14} , P_{15} , P_{16} should be eliminated or not. We note that the production P_{11} generates the outer parentheses in the declarations $<! ELEMENT A (...) >$ and initializes the preceding operator of the inner lists to the empty string. The production P_{12} and P_{13} do the same as P_{11} in terms of attributes. Note that if a nonterminal appears multiple times in a production, these occurrences are numbered in order. Let us define in production P_{14} , $o(List2)$ is the operator of the inner list "List2". The nonterminal "List" appears three times in the production. These occurrences are numbered from 0 to 2. $o1(List0)$ is the operator preceding the inner list "List2". The operator "," which appears after the nonterminal "Occurs" is the following operator of the list "List2". We check if the operator of the inner list "List2" and the preceding operator $o1(List0)$ are the same as the following operator ",". If it is true, we eliminate the parentheses, otherwise we regenerate them in the statement defining $d(List0)$. The production P_{15} does the same as the production P_{14} by using the operator "|". The production P_{16} verifies if the the operator preceding the inner list $o1(List0)$ is the same as the operator of the inner

list $o(List2)$, then the parenthesis are eliminated, otherwise they should be generated. The production P_{17} generates the last identifier.

5 Experimental Results

At the implementation level, the syntactic analyzer consists of functions that correspond to all G_{DTD} productions while the semantic analyzer consists of a single function for the evaluation of attributes in a derivation tree. This evaluation function is a switch statement, in which there is one case for each G_{DTD} production. Fig. 6 shows the structure of the DTD compiler. It is worth mentioning here that the syntactic analyzer is the same for all DTD conversion methods. However, we create a new attribute evaluation function for each DTD conversion method. The developed program takes a DTD as input and produces a DTD as output by calculating the structure of an element and removing unnecessary parentheses. An example of the DTD:

```
<!DOCTYPE A [
<!ELEMENT A ((B, (C)))>
<!ATTLIST A id ID #REQUIRED>
<!ELEMENT B (#PCDATA)>
<!ELEMENT C ((E | F))>
<!ELEMENT E (#PCDATA)>
<!ELEMENT F (#PCDATA)>
]>
```

We obtain the following output:

```
<!DOCTYPE A [
<!ELEMENT A (B, C)>
<!ATTLIST A id ID #REQUIRED>
<!ELEMENT B (#PCDATA)>
<!ELEMENT C (E | F)>
<!ELEMENT E (#PCDATA)>
<!ELEMENT F (#PCDATA)>
]>
```

6 Conclusion

In this paper, we have created a new formal framework for the set of DTDs in the form of a context free language and the corresponding context free grammar. The context free grammar generates a tree based implementation, called derivation tree, for each DTD in the computer memory. In addition to this, we can enrich the context free grammar G_{DTD} with attributes:

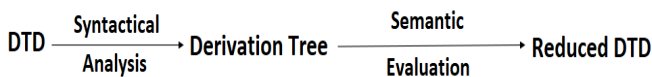


Figure 6: Algorithm Architecture using Compiling Techniques in DTD Conversion

Table 1: Semantic rules for the productions P_0 - P_4 of the grammar G_{DTD}

<p>P0:</p> <p>$d(Start) = d(DTD)$</p>
<p>P1:</p> <p>$d(DTD) = "<!DOCTYPE " \& identifier \& "[" \& d(LineType) \& "]" > "$</p>
<p>P2:</p> <p>$d(LineType0) = "<! Element" \& identifier \& d(ElementType) ">" \& d(AttList) \& d(LineType8)$</p>
<p>P3:</p> <p>$d(LineType) = ""$</p>
<p>P4:</p> <p>$d(AttList) = "<!ATTLIST" \& identifier1 \& identifier2 \& d(AttType) \& d(AttOption) \& d(AttSuite) \& ">"$</p>

- Perform syntactic transformation of the source DTD such as eliminating the unnecessary parentheses in the declaration in the DTD.
- Rewrite the source DTD into an XML document or an intermediate model while retaining the DTD declarations. Then each DTD transformation can be expressed by an XSLT stylesheet applicable to the intermediate model.

Our approach differs from existing approaches in that it provides a formal framework for analyzing DTDs based on compilation techniques. We define a context free grammar that recognizes the language of the DTD. Then any DTD conversion can be realized by enriching this grammar with an attributed system.

Our contribution is that in the case of multiple transformations, we can convert the DTD to another intermediate model such as XML and apply a stylesheet to the intermediate model for each transformation without having to analyze the source DTD.

In the future, the entire framework, grammar with attributes, can be used to split the model conversion into two steps: The first step is to convert the DTD into an intermediate model in terms of XML syntax, while the second step of effective conversion can be expressed by an XSLT stylesheet. We will also explore a way to map DTDs to relational schemas and use traditional relational database engines to process XML documents, since XML is emerging as the data format of the Internet and there is a need for storing and querying XML documents.

Acknowledgment:

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

References:

- [1] A. M. Saba, E. Shahab, H. Abdolrahimpour, M. Hakimi, and A. Moazzam, A comparative analysis of xml documents, xml enabled databases and native xml databases, *arXiv*, vol. abs/1707.08259, 2017. [Online]. Available: <https://arxiv.org/abs/1707.08259>.
- [2] M. Reed, *Python Programming and SQL*, Independently published, January 2023.
- [3] S. Lu, Y. Sun, M. Atay, and F. Fotouhi, A new inlining algorithm for mapping XML DTDs to relational schemas, in: *Jeusfeld, M.A., Pastor, O. (eds) Conceptual Modeling for Novel Application Domains. Lecture Notes in Computer Science*, Springer, Berlin, Heidelberg., vol. 2814, 2003, pp. 366–377.
- [4] 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*, 2002, p. 564–580.
- [5] P. Bohannon, J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Simeon, Bridging the XML relational divide with LegoDB: a demonstration, in *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, Bangalore, India, 2003, p. 759–761, doi: 10.1109/ICDE.2003.1260859.
- [6] 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, WECWIS 2000*, Milpitas, CA, USA, 2000, pp. 134–143, doi: 10.1109/WECWIS.2000.853868.
- [7] Y. Chen, S. Davidson, and Y. Zheng, Constraints preserving XML storage in relations, *International Workshop on the Web and Databases*, 2002. [Online]. Available: <https://api.semanticscholar.org/CorpusID:59848379>
- [8] D. Lee, M. Mani, and W. Chu, Schema conversion methods between XML and relational models, B. Omelayenko, M. Klein (Eds.), *Knowledge Transformation for the Semantic Web*, IOS Press, 2003, pp. 1–17,.
- [9] T. Lv and P. Yan, Mapping DTDs to relational schemas with semantic constraints, *Information and Software Technology*, vol. 48, no. 4, 2006, pp. 245–252.
- [10] Z. Tan, J. Xu, W. Wang, , and B. Shi, Storing normalized xml documents in normalized relations, in *The Fifth International Conference on Computer and Information Technology (CIT'05)*, Shanghai, 2005, pp. 123–129, doi: 10.1109/CIT.2005.175.
- [11] A. J. Rafsanjani and S. H. Mirian Hosseinabadi, RIAL: Redundancy Reducing Inlining Algorithm to Map XML DTD to Relations, In *Proceedings of International Conferences on Computational Intelligence for Modelling Control and Automation*, 2008, pp. 25–30, doi: 10.1109/CIMCA.2008.19.
- [12] Q. Zhu and W. Yang, Mapping from the XML Schema to the Relational Database with Functional Dependency Preserved, in *2010 International Conference on Machine Vision*

and *Human machine Interface*, Kaifeng, China, 2010, pp. 412–415.

- [13] A. A. A. El Aziz and A. Kannan, Survey of mapping xml dtds(documents) to relational schemas, *International Journal of Computer Science and Management Research*, vol. 2, 2013, pp. 1175–1193.
- [14] D. Knuth, Semantics of context free languages, *Math. Systems Theory*, vol. 2, 1968, pp. 127–145, <https://doi.org/10.1007/BF01692511>.
- [15] K. Barbar and K. Musumbu, Implementation of abstract interpretation algorithms by means of attribute grammar, in *Proceedings of 26th Southeastern Symposium on System Theory*, Athens, OH, USA, 1994, pp. 87–92, doi: 10.1109/SSST.1994.287905.
- [16] K. Barbar, Automatic generator of xml documents editors based on attributed grammars, in *Proceedings of the 5th international conference on Soft computing as transdisciplinary science and technology*, 2008, pp. 166–172, <https://doi.org/10.1145/1456223.1456260>.
- [17] D. Li, X. Li, V. Stolz, QVT based model transformation using XSLT, *ACM SIGSOFT Software Engineering Notes*, Vol.36, No.1, 2011, pp. 1-8, <https://doi.org/10.1145/1921532.1921563>.
- [18] H. Mizumoto, N. SUZUKI, An XSLT transformation method for distributed XML, *Journal of information processing*, Vol.23, No.3, 2015, pp. 353-365, doi:10.2197/ipsjip.23.353.
- [19] A. V. Aho, *Compilers: Principles, Techniques, and Tools*, ser. Always Learning. Pearson, 2014.
- [20] T. Mogensen, *Introduction to Compiler Design*, ser. Undergraduate Topics in Computer Science. Springer Cham, 2024. [Online]. Available: <https://doi.org/10.1007/978-3-031-46460-7>

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

SOUHEIL TAWK is a PHD student of the thesis "Systems based on attribute grammars for model conversion: Application to the conversion of DTD into relational schema". He carried out the experimental and the optimization parts. MAROUN ABI ASSAF was responsible of the Logic, algorithms, and the bibliography parts. JOSEPH CONSTANTIN is the cosupervisor of the thesis. He was responsible of finding the Grammar productions, the attributed Grammar equations, and writing of the paper. KABLAN BBRBAR is the supervisor of the thesis. He was responsible of the Logic part and the implementation the attributed Grammar equations. JACQUES BOU ABDO was an examiner of the thesis. He was responsible for the correction of the paper from logic errors.

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.

Creative Commons Attribution License 4.0 (Attribution 4.0 International , CC BY 4.0)

This article is published under the terms of the Creative Commons Attribution License 4.0 https://creativecommons.org/licenses/by/4.0/deed.en_US

APPENDIX

Table 2: Semantic rules for the productions P_5 - P_{10} of the grammar G_{DTD}

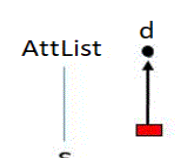
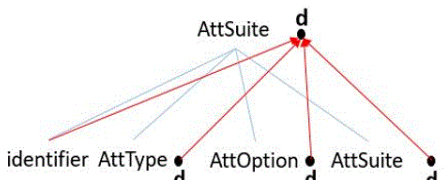
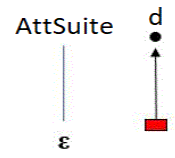
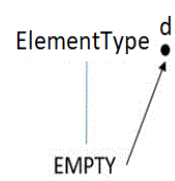
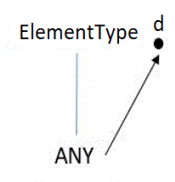
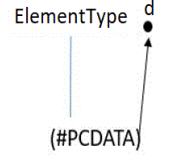
<p>P5:</p>  <p>$x(AttList) = ""$</p>
<p>P6:</p>  <p>$d(AttSuite0) = identifier \& d(AttType) \& d(AttOption) \& d(AttSuite4)$</p>
<p>P7:</p>  <p>$d(AttSuite) = ""$</p>
<p>P8:</p>  <p>$d(ElementType) = "Empty"$</p>
<p>P9:</p>  <p>$d(ElementType) = "ANY"$</p>
<p>P10:</p>  <p>$d(ElementType) = "#PCDATA"$</p>

Table 3: Semantic rules for the productions P_{11} - P_{13} of the grammar G_{DTD}

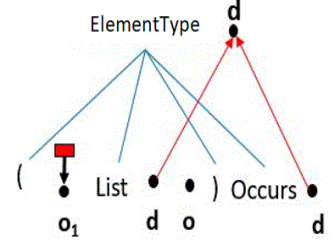
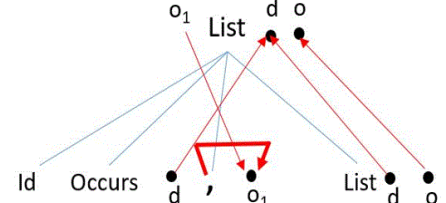
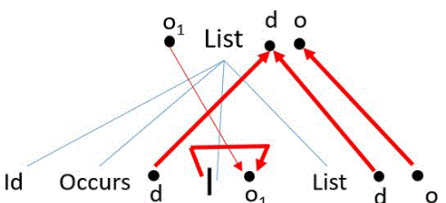
<p>P11:</p>  <p>$o1(List2) = ""$ $d(ElementType) = "(" \& d(List) \& ")" \& d(Occurs)$</p>
<p>P12:</p>  <p>$o1(List4) = \begin{cases} ', & \text{if } (o1(List0) = '' \text{ or } o1(List0) = ',) \\ 'm', & \text{otherwise} \end{cases}$</p> <p>$d(List0) = id \& d(Occurs) \& ",," \& d(List4)$ $o(List0) = o(List4)$</p>
<p>P13:</p>  <p>$o1(List4) = \begin{cases} ' ', & \text{if } (o1(List0) = '' \text{ or } o1(List0) = ' ') \\ 'm', & \text{otherwise} \end{cases}$</p> <p>$d(List0) = id \& d(Occurs) \& " ," \& d(List4)$ $o(List0) = o(List4)$</p>

Table 4: Semantic rules for the productions P_{14} - P_{15} of the grammar G_{DTD}

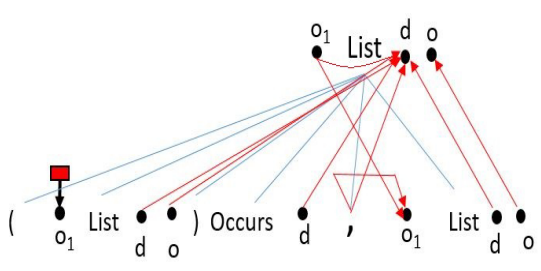
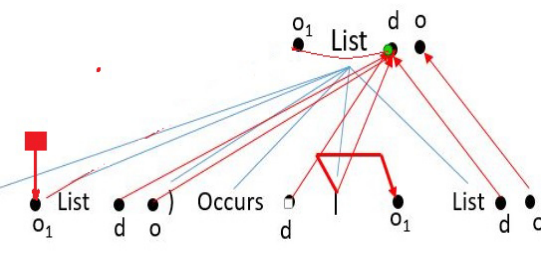
<p>P14:</p>  <p>$o1(List2) = ' '$</p> $o1(List6) = \begin{cases} ', ', if(o1(List0) = '' or \\ o1(List0) = ', ') \\ 'm', otherwise \end{cases}$ <p>if ($o1(List0) = o(List2) \ \& \ o(List2) = ', '$) then $d(List0) = d(List2) \ \& \ d(Occurs) \ \& ', ' \ \& d(List6)$ else $d(List0) = "' \ \& d(List2) \ \& '" \ \& d(Occurs) \ \& ', ' \ \& d(List6)$ endif $o(List0) = o(List6)$</p>
<p>P15:</p>  <p>$o1(List2) = ' '$</p> $o1(List6) = \begin{cases} ' ', if(o1(List0) = '' or \\ o1(List0) = ' ') \\ 'm', otherwise \end{cases}$ <p>if ($o1(List0) = o(List2) \ \& \ o1(List2) = ' '$) then $d(List0) = d(List2) \ \& \ d(Occurs) \ \& ' ' \ \& d(List6)$ else $d(List0) = "' \ \& d(List2) \ \& '" \ \& d(Occurs) \ \& ' ' \ \& d(List6)$ endif $o(List0) = o(List6)$</p>

Table 5: Semantic rules for the productions P_{16} - P_{20} of the grammar G_{DTD}

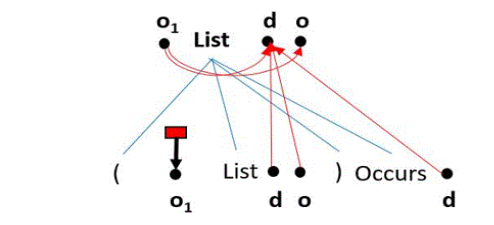
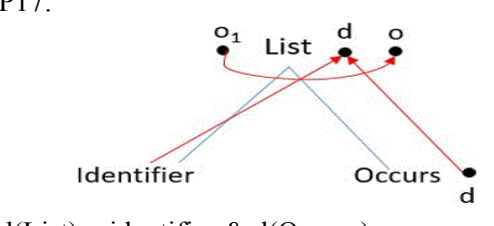
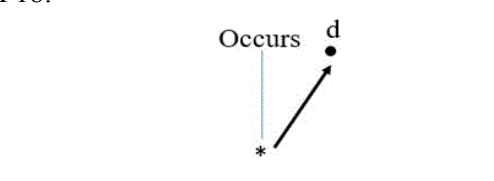
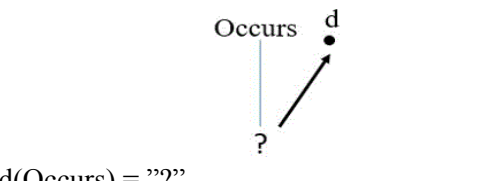
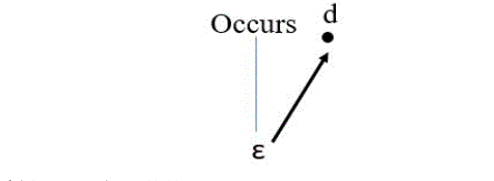
<p>P16:</p>  <p>$o1(List2) = ' '$ if ($o1(List0) = o(List2)$) then $d(List0) = d(List2) \ \& \ d(Occurs)$ else $d(List0) = "(\ \& d(List2) \ \&)" \ \& d(Occurs)$ endif $o(List0) = o1(List0)$</p>
<p>P17:</p>  <p>$d(List) = identifier \ \& \ d(Occurs)$ $o(List) = o1(List)$</p>
<p>P18:</p>  <p>$d(Occurs) = "*" "$</p>
<p>P19:</p>  <p>$d(Occurs) = "?" "$</p>
<p>P20:</p>  <p>$d(Occurs) = " " "$</p>

Table 6: Semantic rules for the productions P_{21} - P_{27} of the grammar G_{DTD}

<p>P21:</p> <p>$d(\text{Occurs}) = "+"$</p>
<p>P22:</p> <p>$d(\text{AttType}) = "CDATA"$</p>
<p>P23:</p> <p>$d(\text{AttType}) = "ID"$</p>
<p>P24:</p> <p>$d(\text{AttType}) = "IDREF"$</p>
<p>P25:</p> <p>$d(\text{AttType}) = "(" \& d(\text{Enumerate}) \& ")" \& d(\text{InitialVal})$</p>
<p>P26:</p> <p>$d(\text{Enumerate}) = \text{string} \& d(\text{EnumerateSuite})$</p>
<p>P27:</p> <p>$d(\text{EnumerateSuite}) = d(\text{Enumerate})$</p>

Table 7: Semantic rules for the productions P_{28} - P_{33} of the grammar G_{DTD}

<p>P28:</p> <p>$d(\text{Enumerate}) = ""$</p>
<p>P29:</p> <p>$d(\text{AttOption}) = "#" \& "FIXED" \& d(\text{InitialVal})$</p>
<p>P30:</p> <p>$d(\text{AttOption}) = "#REQUIRED"$</p>
<p>P31:</p> <p>$d(\text{AttOption}) = "#IMPLIED"$</p>
<p>P32:</p> <p>$d(\text{InitialVal}) = \text{string}$</p>
<p>P33:</p> <p>$d(\text{InitialVal}) = ""$</p>