DTD-Diff: A Change Detection Algorithm for DTDs

Erwin Leonardi [a], Tran T. Hoai [a], Sourav S Bhowmick [a] and Sanjay Madria [b]

[a] *School of Computer Engineering, Division of Information Systems, Nanyang Technological University, Singapore 639798*

[b] *Department of Computer Science, University of Missouri-Rolla, Rolla 65409*

Abstract

The DTD of a set of XML documents may change due to many reasons such as changes to the real world events, changes to the user's requirements, and mistakes in the initial design. In this paper, we present a novel algorithm called DTD-Diff to detect the changes to DTDs that defines the structure of a set of XML documents. Such change detection tool can be useful in several ways such as maintenance of XML documents, incremental maintenance of relational schema for storing XML data, and XML schema integration. We compare DTD-Diff with existing XML change detection approaches and show that converting DTD to XML Schema (XSD) (which is in XML document format) and detecting the changes using existing XML change detection algorithms is not a feasible option. Our experimental results show that DTD-Diff is 5–325 times faster than X-Diff when it detects the changes to the XSD files. Compared to XyDiff, DTD-Diff is up to 38 times faster. We also study the result quality of detected deltas.

Key words: Change detection, DTD, XML, algorithm, performance.

1 Introduction

XML has emerged as the leading textual language for representing and exchanging data over the Web. Since Web data changes frequently, a key feature of XML data is its dynamic property. That is, XML data may change at any time in any way. Hence, a tool is needed to detect such changes automatically. Consequently, there has been increasing research efforts in detecting

Email addresses: assourav@ntu.edu.sg, madrias@umr.edu (Sanjay Madria).

changes to XML data [7,13–15,22]. These approaches primarily focus on detecting changes to XML documents. However, in many applications a schema (i.e., *Document Type Definition* (DTD) or *XML schema* (XSD) [2]) is associated with a set of XML documents to define their legal structures. Schema of such XML documents may also need to be updated for various reasons [8,20,9]. Systems must be adapted to the real-world changes. Schemas may be initially defined as drafts and are subsequently refined due to changes to the real-world or due to the need of fixing errors in the previous versions. Commercial alliances change and expand. For example, consider the DTD D_1 in Figure 1(a) at time t_1. It may evolve to D_2 (Figure 1(b)) at time t_2 because the university may wish to restructure the information due to change in the university administrators' requirements. Hence, there is a strong need for a tool to detect changes to DTDs. Such change detection tool can be useful in at least following three ways.

- *Maintenance of XML documents.* A DTD change detection tool can be useful for *incremental* maintenance or revalidation of a set of XML documents when their DTD evolves. Note that many documents can be associated with the same DTD/schema and the brute-force approach of revalidating a complete document is known to be high [17]. For instance, let X be a set of XML documents where each document $x_i \in X$ conforms to DTD D. Assume that due to mistakes in the initial design, D is modified to D'. Consequently, $x_i \in X$ may not conform to D' anymore. Therefore, it is necessary to detect the differences between D and D' (denoted by $\triangle(D,D')$) *automatically* so that it can be used to transform $x_i \in X$ to x_i' such that x_i' conforms to D'. The basic idea is to keep track of the changes made to the DTD and to identify the portion of the DTD/schema that, because of these changes, requires revalidation. The document portions affected by those changes can then be identified and revalidated, thus avoiding a costly revalidation of the whole document.
- *Incremental maintenance of relational schema.* Recently, there has been a substantial research effort in storing and processing XML data using relational databases [12]. These approaches can be classified into two major categories. In the *schema-conscious* approach [19], a relational schema is created based on the DTD/schema of the XML documents. In the *schema-oblivious* approach [12], a fixed relational schema is used to store XML documents. The basic idea is to capture the tree structure of an XML document. This approach does not require existence of an XML schema/DTD. A DTD change detection tool can be particularly useful for incremental maintenance of the relational schema generated by a *schema-conscious approach*. This is because if the DTD is changed, then the respective relational schema may also need to be modified. The differences between the old and new versions of a DTD can help us in maintaining the relational schema incrementally.

```
1  <!ENTITY univName “Open University”>
2  <!ENTITY myScript SYSTEM “script.pl” NDATA pl>
3  <!ELEMENT university (information,school+)>
4  <!ELEMENT information
               (address,(telp|fax+|website))>
5  <!ELEMENT school (name,dean,department*)>
6  <!ELEMENT department (name,hod,courses)>
7  <!ELEMENT courses (course*)>
8  <!ELEMENT course (#PCDATA)>
9  <!ELEMENT name (#PCDATA)>
10 <!ELEMENT dean (#PCDATA)>
11 <!ELEMENT hod (#PCDATA)>
12 <!ELEMENT telp (#PCDATA)>
13 <!ELEMENT fax (#PCDATA)>
14 <!ELEMENT website (#PCDATA)>
15 <!ELEMENT address (#PCDATA)>
16 <!ATTLIST course code CDATA #REQUIRED
                  year CDATA #IMPLIED>
```

(a) D_1

```
1  <!ENTITY % info “name,head,website,telp,fax”>
2  <!ENTITY univName “Open University”>
3  <!ENTITY myScript SYSTEM “newScript.pl”
                  NDATA pl>
4  <!ELEMENT university (information,school+)>
5  <!ELEMENT information
           ((telp|website|fax?),address)>
6  <!ELEMENT school (sinfo,department*)>
7  <!ELEMENT sinfo (%info;)>
8  <!ELEMENT department (dinfo,courses)>
9  <!ELEMENT dinfo (%info;)>
10 <!ELEMENT courses (course+)>
11 <!ELEMENT course (#PCDATA)>
12 <!ELEMENT name (#PCDATA)>
13 <!ELEMENT head (#PCDATA)>
14 <!ELEMENT website (#PCDATA)>
15 <!ELEMENT telp (#PCDATA)>
16 <!ELEMENT fax (#PCDATA) >
17 <!ELEMENT address (#PCDATA) >
18 <!ATTLIST course code CDATA #REQUIRED
                  year CDATA #REQUIRED >
```

(b) D_2

```
1  <xs:schema xmlns:xs='… … … … '>
2    <xs:element name='course'>
3      <xs:complexType mixed='true'>
4        <xs:attribute name='code' use='required'/>
5        <xs:attribute name='year'/>
6      </xs:complexType>
7    </xs:element>
8    <xs:element name='courses'>
9      <xs:complexType>
10       <xs:sequence>
11         <xs:element ref='course' minOccurs='0'
                       maxOccurs='unbounded'/>
12       </xs:sequence>
13     </xs:complexType>
14   </xs:element>
15   <xs:element name='dean'>
16     <xs:complexType mixed='true'>
17     </xs:complexType>
18   </xs:element>
19   <xs:element name='information'>
20     <xs:complexType>
21       <xs:sequence>
22         <xs:element ref='address'/>
23         <xs:choice>
24           <xs:element ref='telp'/>
25           <xs:element ref='fax'
26                  maxOccurs='unbounded'/>
27           <xs:element ref='website'/>
28         </xs:choice>
29       </xs:sequence>
30    </xs:complexType>
31  </xs:element>
32  ... ... ...
```

(c) XSD of D_1

```
1  <xs:schema xmlns:xs='… … … … '>
2    <xs:element name='course'>
3      <xs:complexType mixed='true'>
4        <xs:attribute name='code' use='required'/>
5        <xs:attribute name='year' use='required'/>
6      </xs:complexType>
7    </xs:element>
8    <xs:element name='courses'>
9      <xs:complexType>
10       <xs:sequence>
11         <xs:element ref='course'
                       maxOccurs='unbounded'/>
12       </xs:sequence>
13     </xs:complexType>
14   </xs:element>
15   ... ... ...
16   <xs:element name='information'>
17     <xs:complexType>
18       <xs:sequence>
19         <xs:element ref='address'/>
20         <xs:choice>
21           <xs:element ref='website'/>
22           <xs:element ref='telp'/>
23           <xs:element ref='fax' minOccurs='0'/>
24         </xs:choice>
25       </xs:sequence>
26     </xs:complexType>
27   </xs:element>
28   ... ... ...
```

(d) XSD of D_2

Fig. 1. Two versions of a DTD and corresponding XSD files.

- *Maintenance of XML access control policies.* Changes to the DTD or XML schema can impact on the access control policies defined on the XML documents. Hence, it is necessary to incrementally maintain the access policies as the DTD changes.

In this paper, we propose a novel algorithm, called DTD-Diff, for detecting the changes to DTDs [1]. At this point, one would question the justification of this work. At first glance, it may seem that the DTD change detection problem can easily be addressed by existing change detection tools for XML documents [7,13–15,22]. Specifically, we can first transform two versions of a

[1] A shorter version of this paper is going to appear in [16].

DTD to XML Schemas (XSD), that are in XML format, using tools such as *Syntex dtd2xs* (www.syntext.com/downloads/index.htm) and LuMrix dtd2xs (http://www.lumrix.net/dtd2xs.php). For example, Figures 1(c) and 1(d) are the XSD representations of the DTDs in Figures 1(a) and 1(b), respectively. Then, the changes to the DTDs can be detected using existing XML change detection tools (such as X-Diff [22] and XyDiff [7]). Although this approach will clearly detect changes, we argue that it may often fail to detect *semantically correct* and *optimal* changes. For example, these algorithms may detect that the name of a `school` element in D_1 in Figure 1(a) has been updated to `sinfo` in D_2 in Figure 1(b) (Figure 2). However, this is semantically incorrect! Furthermore, these algorithms are not efficient as far as DTD change detection is concerned as they do not exploit the structure and semantics of the DTDs to improve response time. We shall elaborate on these issues further in Section 2.

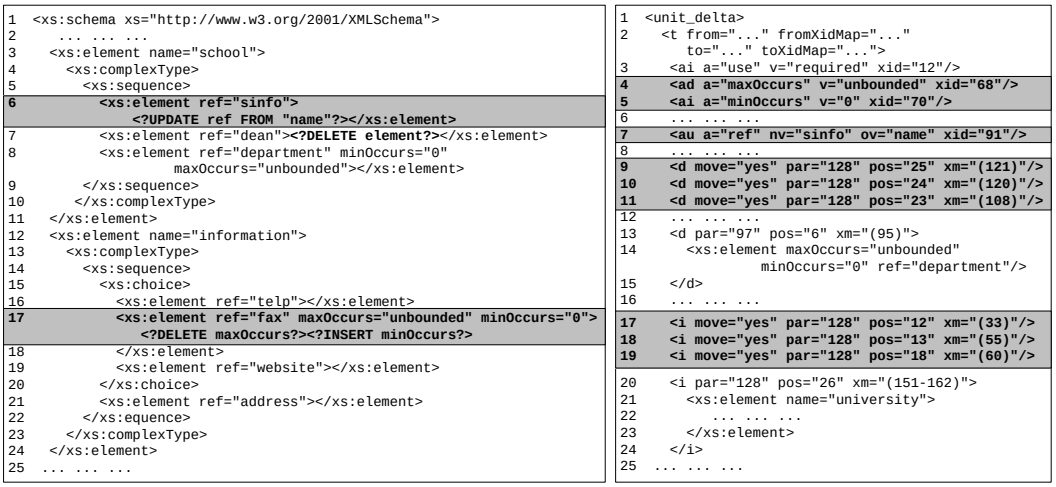
In summary, the main contributions of this paper are as follows. (1) In Section 3, we present the data model to represent the changes to DTDs. By using this data model we are able to detect the changes to DTDs correctly. (2) In Section 4, we propose a novel algorithm called DTD-Diff for detecting the changes to DTDs. *To the best of our knowledge, this is the first approach that addresses the DTD change detection problem.* The algorithm takes as input two versions of a DTD that are represented using our DTD data model and detects the changes *directly* without converting them to XSD format. (3) Through an extensive experimental study in Section 5, we show that our approach is significantly faster than state-of-the-art XML change detection algorithms such as X-Diff and XyDiff. Note that in our study, we convert DTDs to XSD files prior to employing X-Diff/XyDiff to detect the changes. We also show that DTD-Diff is also able to produce optimal or at least near-optimal deltas.

2 Related Work

To the best of our knowledge, there are not any published works on detecting changes to DTDs. In the preceding section, we mentioned that the two versions of a DTD can be first converted to XML schema using tools such as *Syntex dtd2xs* and then we can detect the changes using any existing XML change detection algorithm. Hence, we first compare our approach with existing XML change detection techniques.

2.1 XML Change Detection

Recently, a number of techniques for detecting the changes to XML data have been proposed. XyDiff [7] is a main-memory algorithm for detecting the changes in *ordered* XML documents. In an *ordered* XML, both the parent-child



```
1  <xs:schema xs="http://www.w3.org/2001/XMLSchema">
2     ... ... ...
3    <xs:element name="school">
4      <xs:complexType>
5        <xs:sequence>
6          <xs:element ref="sinfo">
             <?UPDATE ref FROM "name"?></xs:element>
7          <xs:element ref="dean"><?DELETE element?></xs:element>
8          <xs:element ref="department" minOccurs="0"
                    maxOccurs="unbounded"></xs:element>
9        </xs:sequence>
10      </xs:complexType>
11   </xs:element>
12   <xs:element name="information">
13     <xs:complexType>
14       <xs:sequence>
15         <xs:choice>
16           <xs:element ref="telp"></xs:element>
17           <xs:element ref="fax" maxOccurs="unbounded" minOccurs="0">
               <?DELETE maxOccurs?><?INSERT minOccurs?>
18           </xs:element>
19           <xs:element ref="website"></xs:element>
20         </xs:choice>
21         <xs:element ref="address"></xs:element>
22       </xs:equence>
23     </xs:complexType>
24   </xs:element>
25  ... ... ...
```

(a) X-Diff

```
1  <unit_delta>
2    <t from="..." fromXidMap="..."
       to="..." toXidMap="...">
3    <ai a="use" v="required" xid="12"/>
4    <ad a="maxOccurs" v="unbounded" xid="68"/>
5    <ai a="minOccurs" v="0" xid="70"/>
6    ... ... ...
7    <au a="ref" nv="sinfo" ov="name" xid="91"/>
8    ... ... ...
9    <d move="yes" par="128" pos="25" xm="(121)"/>
10   <d move="yes" par="128" pos="24" xm="(120)"/>
11   <d move="yes" par="128" pos="23" xm="(108)"/>
12   ... ... ...
13   <d par="97" pos="6" xm="(95)">
14     <xs:element maxOccurs="unbounded"
                minOccurs="0" ref="department"/>
15   </d>
16   ... ... ...
17   <i move="yes" par="128" pos="12" xm="(33)"/>
18   <i move="yes" par="128" pos="13" xm="(55)"/>
19   <i move="yes" par="128" pos="18" xm="(60)"/>
20   <i par="128" pos="26" xm="(151-162)">
21     <xs:element name="university">
22       ... ... ...
23     </xs:element>
24   </i>
25 ... ... ...
```

(b) XyDiff

Fig. 2. X-Diff and XyDiff results.

relationship and the left-to-right order among siblings are important. Wang et al. proposed X-Diff [22] for computing the changes to *unordered* XML documents. In *unordered* XML, the parent-child relationship is significant, while the left-to-right order among siblings is not important. All these algorithms suffer from scalability problem as they fail to detect changes to large XML documents due to lack of memory. Consequently, a number of approaches [13–15] have been proposed to address the scalability problem of XML change detection by using relational databases. In contrast to our approach, the above approaches are not designed to detect changes to DTDs. Consequently, even if we employ these algorithms to XSD representations of DTDs, they suffer from the following limitations.

- *Granularity of types of changes.* The above approaches support the following types of edit operations: *insert*, *delete*, *update*, and *move* operations of nodes in the tree representations of two versions of an XML document. A node in the XML tree represents an element, text or an attribute node. As the data format of a DTD is different from that of an XML document, the types of changes in old and new versions of a DTD are different from the ones that occur in XML documents. In fact, a DTD has richer variety of edit operations compared to XML documents. For instance, in an XML document an element does not have any cardinality associated with it. However, an element type in a DTD may have a cardinality which may be updated as the DTD evolves. We shall elaborate on different types of change operations in DTDs in Section 3.
- *Inability to detect changes to both unordered and ordered nodes.* Often DTDs contain *sequence* (denoted by ",") and *choice* (denoted by "|") groups. The order of elements in a sequence group is important, while the order of elements in a choice group is not significant. The current approaches for de-

tecting the changes to XML documents focus on either ordered XML [7,13] or unordered XML [14,15,22]. Hence, if we use existing techniques then it is indeed possible that certain types of changes are not accurately detected. For example, consider the element type declaration `information` in D_1 and D_2 in Figures 1(a) and 1(b), respectively. We observe that elements `telp`, `fax`, and `website` belongs to the choice group. Furthermore, `(telp|fax+|website)` and `address` are in the sequence group. The deltas generated by X-Diff and XyDiff, when the documents in Figures 1(c) and 1(d) are passed as input, are depicted in Figures 2(a) and 2(b), respectively. Interestingly, the output of XyDiff specifies that the element `fax` has undergone a move operation (Lines 9–11 and 17–19 in Figure 2(b)). However, this is incorrect as `fax` belongs to the choice group. On the other hand, X-Diff fails to detect the movement of `address` element (Figure 2(a)).

- *Detection of semantically incorrect changes.* Consider the element type declaration `school` in D_1 and D_2 in Figures 1(a) and 1(b). X-Diff and XyDiff both detect that element `name` in `school` is updated to `sinfo` (lines 6 and 7 in Figures 2(a) and 2(b), respectively). However, `sinfo` consists of `name`, `head`, `website`, `telp`, and `fax`. Hence, the change detected by these algorithms is semantically incorrect. The correct edit operations should be deletion of `name` element and insertion of `sinfo` element.
- *Generation of non-optimal edit scripts.* We notice that the changes to the cardinality of an element in the DTD can result in generation of *non-optimal* edit scripts [22] by X-Diff or XyDiff. For instance, consider the cardinality of element `fax` in the element type declaration `information` in D_1 and D_2. The cardinality of element `fax` is updated from `+` to `?`. X-Diff detects it as a result of two edit operations: a deletion followed by an insertion (line 17, Figure 2(a)). Similarly, XyDiff also detects it as two edit operations (lines 4–5, Figure 2(b)). However, the correct number of edit operations should be one (update of cardinality).
- *Performance bottleneck.* Lastly, existing XML change detection algorithms such as X-Diff are not efficient when they are used to detect changes to DTDs. This is because they do not exploit the structure and semantics of the DTDs to improve response time. As we shall see in Section 5, by exploiting such features of DTDs, our DTD change detection algorithm outperforms X-Diff by 5–325 times!

2.2 DTD and XML Schema Evolution

XEM [20] is an approach which provides XML-centric data and DTD evolution facilities. The authors proposed a set of evolution operators to achieve this. When DTDs are changed, XEM ensures that the existing XML documents still conform to the new DTD. Similarly, when the XML documents are

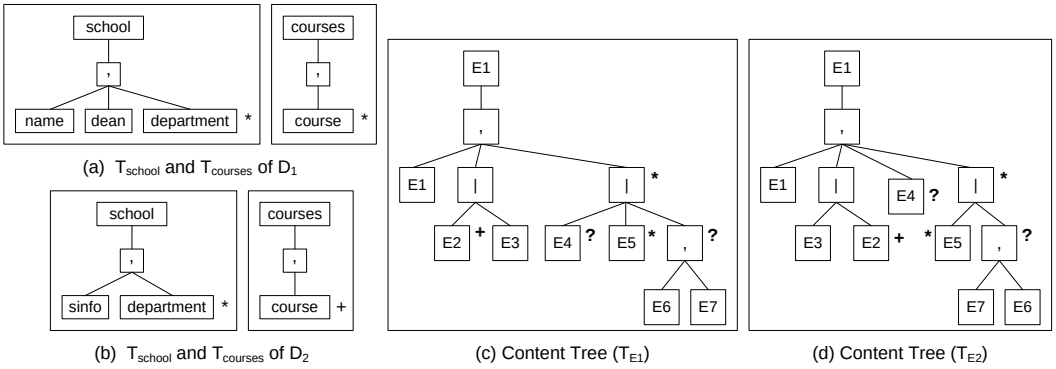


Fig. 3. Content Trees and Examples.

changed, XEM ensures that the changed XML documents still correspond to the specified DTD. In [8], the authors proposed an approach to manage DTD evolution. The authors also presented 25 DTD changes and defined their semantics by preconditions and post actions such that the new DTD is valid, existing documents conform to the new DTD, and data is not lost if possible. DTD evolution has also been investigated in [4] where the focus was on dynamically adapting the schema to the structure of most documents stored in an XML data source. Required modifications are deduced by means of structure mining techniques.

The above approaches are different from DTD-Diff as DTD-Diff focuses on detecting the changes to DTDs. XEM and the approach in [8] are used to manage DTD evolution and to ensure that when DTD is changed, the XML documents that conform to this DTD still conform to the new DTD. That is, XEM and the approach in [8] do not detect the changes.

Guerrini *et al.* addressed the problem of XML schema evolution in [9]. They proposed a set of *atomic evolution primitives* to be applied to the basic components of a schema. The authors showed that all required transformation in an XML schema can be expressed through a sequence of primitives in the set. Furthermore, a set of high level evolution primitives that allows complex changes to be expressed in a compact way is proposed. Finally, they analyze the impact of such primitives on the validity of XML documents known to be valid for the original schema. Our work differs from the above effort as follows. First, we focus on DTDs instead of XML schemas. Second, we address the problem of detecting the changes automatically whereas Guerrini *et al.* attempts to identify the parts of documents that need to be revalidated after a certain number of changes have occurred on the schema.

3 DTD Data Model

The *DTD Data Model* is a simple, flexible model for representing DTDs. In this section, we begin by briefly describing the DTD Data Model. Next, we present the basic change operations used to modify a DTD.

A DTD consists of *entity declaration* (`<!ENTITY ...>`), *element type declaration* (`<!ELEMENT ...>`), and *attribute declaration* (`<!ATTLIST ...>`) that describe entities, elements, and attributes, respectively. For example, consider the DTD D_2 in Figure 1(b). Lines 1–3, 4–17, and 18 are examples of entity declarations, element type declarations, and attribute declaration, respectively.

Element Type Declaration (ETD). In a DTD, XML elements are declared using element type declaration. Each element type declaration has a *name* and *element content*. For example, consider the DTD D_1 in Figure 1(a). The name and the content of the element type `school` in line 5 are `school` and `(name,dean,department*)`, respectively. Formally, ETD is defined as follows:

Definition 1 [Element Type Declaration (ETD)] *An element type declaration is a 2-tuple $E = (N_E, C_E)$, where N_E is the name of the element type E, and $C_E = (\Upsilon_1, \Upsilon_2, \ldots, \Upsilon_n)$ is a sequence called element content of E where Υ_i is a regular expression over the element types.*

Observe that an *element content* can be very complex with multiple levels of nesting. For example, `<!ELEMENT E1 (E1,(E2+|E3),(E4?|E5*|(E6,E7)?)*)>` where Υ_1 = `E1`, Υ_2 =`(E2+|E3)`, and Υ_3 = `(E4?|E5*|(E6,E7)?)*`. We represent the element content C_E as a *content tree* T_E. For example, the content trees of element types `school` and `courses` in D_1 and D_2 are depicted in Figures 3(a) and 3(b), respectively. Also, consider the element type declaration `<!ELEMENT E1 (E1,(E2+|E3), (E4?| E5*|(E6,E7)?)*)>`. The content tree T_{E1} is depicted in Figure 3(c). Formally, we define the content tree as follows:

Definition 2 [Content Tree] *Let $E = (N_E, C_E)$ be an ETD. Then, the content tree of C_E is a 3-tuple $T_E = (root, K, W)$, where*

- *root is the root node of T_E and label(root) = N_E*
- *K is a set of nodes in T_E. A node $k \in K$ is a 2-tuple $k = (V_k, \lambda_k)$, where if k is a leaf node, then V_k is the name of element in C_E. Otherwise, $V_k \in$ {“`,`”, “`|`”}. If $V_k =$ “`,`” then child nodes of k must be ordered. Otherwise, child nodes of k are unordered.*
- *λ_k is the cardinality of node k (optional) where $\lambda_k \in$ {“`?`”, “`+`”, “`*`”}.*
- *W is a set of edges in T_E.*

Attribute Declaration (AD). The attribute declaration in a DTD is used to define the attributes of an element. Each AD has a *name* of element type to which a set of attributes belongs. Each attribute in the attribute set has

a *name*, *type*, and an optional *default value*. For example, reconsider D_1 in Figure 1(a). The attribute declaration of element type `course` is in line 16. The type of data and default value of the attribute `code` are `CDATA` and `#REQUIRED`, respectively.

Definition 3 [**Attribute Declaration (AD)**] *An attribute declaration A is a 2-tuple $A = (N_A, S_A)$, where N_A is the name of element type and S_A is a set of attributes associated with N_A. An attribute $a \in S_A$ is a 3-tuple $a = (N_a, Y_a, D_a)$, where N_a is the name of the attribute a, $Y_a \in \{$"`CDATA`", "`(en1|en2|..)`", "`ID`", "`IDREF`", "`IDREFS`", "`NMTOKEN`", "`NMTOKENS`", "`ENTITY`", "`ENTITIES`", "`NOTATION`", "`xml:`"$\}$ is the type of data of the attribute a, $D_a \in \{$"*value*", "`#REQUIRED`", "`#IMPLIED`", "`#FIXED` *value*"$\}$ is the default value of the attribute a.*

Entity Declaration (ED). Entities are variables used to define shortcuts to common text. Entity references are references to entities. We have two kinds of entities: *general entity* and *parameter entity*. Consider DTD D_2 as depicted in Figure 1(b). Line 1 is an example of *parameter entity*. An example of *general entity* is in line 2. Note that we only consider the general entities. This is because the parameter entities automatically replace the entity references. Entities can be declared as *internal* or *external*. An *internal* ED has a *name* and a *replacement text*. On the other hand, an *external* ED has a *name*, *universal resource indicator (URI)*, and a *content notation*. For example, in D_2 line 2 is an example of an internal ED. The name and replacement text of this entity are `univName` and `"Open University"`, respectively. Line 3 (Figure 1(b)) is an example of an external ED. The name, URI, and content notation are `MyScript`, `"Script1.pl"`, and `"pl"`, respectively. Formally, the ED is defined as follows:

Definition 4 [**Entity Declaration (ED)**] *An **internal** entity declaration is a 2-tuple $I = (N_I, R_I)$, where N_I is the name of the entity, and R_I is the replacement text of the entity that replaces the entity reference. An **external** entity declaration is a 3-tuple $J = (N_J, U_J, P_J)$, where N_J is the name of the entity, U_J is a universal resource indicator* (URI) *of the entity, and P_J is the content notation.*

We are now ready to define how a DTD is represented in our DTD data model. Intuitively, one can think of a DTD as a set of ETDs, a set of ADs, and a set of EDs. We define the DTD formally as follows:

Definition 5 [**DTD**] *A DTD is a 3-tuple $D = (\mathcal{E}, \mathcal{A}, \mathcal{G})$ where $\mathcal{E}$ is a set of Element Type Declarations (ETD) in D, $\mathcal{A}$ is a set of Attribute Declarations (AD) in D, $\mathcal{G}$ is a set of internal and external Entity Declarations (ED). Also, if the numbers of ETDs, ADs, and EDs in a DTD are α, β, and γ, respectively, then $|\mathcal{E}| = \alpha$, $|\mathcal{A}| = \beta$, and $|\mathcal{G}| = \gamma$.*

3.1 Types of Changes

We now describe how a DTD is modified. We discuss the types of changes supported by *element type declaration*, *attribute declaration*, and *entity declaration*. We assume that $D_1 = (\mathcal{E}_1, \mathcal{A}_1, \mathcal{G}_1)$ and $D_2 = (\mathcal{E}_2, \mathcal{A}_2, \mathcal{G}_2)$ are the old and new versions of a DTD, respectively. Also, let $\mathcal{E}_i = \{E_{i1}, E_{i2}, \ldots, E_{in}\}$ be a set of ETDs and $\mathcal{N}_i = \{N_{i1}, N_{i2}, \ldots, N_{in}\}$ be a set of element names in the ETDs for $i \in \{1, 2\}$.

Types of Changes to Element Type Declaration (ETD). The types of changes to the ETDs are defined as following.

(1) ***Insertion of a new ETD***. $E_x = (N_x, C_x)$ is a new ETD iff $N_x \notin \mathcal{N}_1$ but $N_x \in \mathcal{N}_2$. For example, `sinfo` (line 7, Figure 1(b)) is inserted into D_2.

(2) ***Deletion of an ETD***. $E_x = (N_x, C_x)$ is a deleted ETD iff $N_x \in \mathcal{N}_1$ but $N_x \notin \mathcal{N}_2$. For example, `dean` (line 10, Figure 1(a)) is deleted from D_1.

We now describe the types of changes to the element content. As in our DTD model element contents are represented as content trees, we can define the types of changes to element contents in the context of such content trees. Suppose we have two ETDs $E_1 = (N_{E1}, C_{E1})$ and $E_2 = (N_{E2}, C_{E2})$, where $E_1 \in \mathcal{E}_1$, $E_2 \in \mathcal{E}_2$, and $N_{E1} = N_{E2}$. Let $T_{E1} = (root_1, K_1, W_1)$ and $T_{E2} = (root_2, K_2, W_2)$ be two content trees of C_{E1} and C_{E2}, respectively. The types of changes in the content trees are as follows.

(1) ***Insertion of a leaf node***. Let $parent(k)$ be the parent node of node k. Let $w(p, c)$ be an edge connecting node p to node c. Node $k = (V_k, \lambda_k)$ is a new leaf node iff $k \notin K_1$, $k \in K_2$, $w(parent(k), k) \notin W_1$, and $w(parent(k), k) \in W_2$. For example, consider T_{school} as depicted in Figures 3(a) and 3(b). Node *sinfo* is a new node with $V_k =$ "sinfo" and λ_k is empty.

(2) ***Deletion of a leaf node***. Node $k = (V_k, \lambda_k)$ is a deleted leaf node iff $k \in K_1$, $k \notin K_2$, $w(parent(k), k) \in W_1$, and $w(parent(k), k) \notin W_2$. Consider T_{school} as depicted in Figures 3(a) and 3(b). Nodes "*name*" and "*dean*" are deleted leaf nodes.

(3) ***Insertion of a subtree***. Let S_r be a subtree rooted at node r. Let $\mathcal{K}$ and $\mathcal{W}$ be two sets of nodes and edges in subtree S_r, respectively. Subtree S_r is a new subtree iff $r \notin K_1$, $r \in K_2$, $w(parent(r), r) \notin W_1$, $w(parent(r), r) \in W_2$, $\forall_{k \in \mathcal{K}}$ $(k_i \notin K_1)$, $\forall_{w \in \mathcal{W}}$ $(w_j \notin W_1)$, $\forall_{k \in \mathcal{K}}$ $(k_i \in K_2)$, and $\forall_{w \in \mathcal{W}}$ $(w_j \in W_2)$.

(4) ***Deletion of a subtree***. Subtree S_r is a deleted subtree iff $r \in K_1$, $r \notin K_2$, $w(parent(r), r) \in W_1$, $w(parent(r), r) \notin W_2$, $\forall_{k \in \mathcal{K}}$ $(k_i \in K_1)$,

$\forall_{w \in \mathcal{W}}$ $(w_j \in W_1)$, $\forall_{k \in \mathcal{K}}$ $(k_i \notin K_2)$, and $\forall_{w \in \mathcal{W}}$ $(w_j \notin W_2)$.

(5) ***Move a leaf node***. Node $k_1 = (V_{k1}, \lambda_{k1})$ is moved to be node $k_2 = (V_{k2}, \lambda_{k2})$ iff $k_1 \in K_1$, $k_2 \in K_2$, $w(parent(k_1), k_1) \in W_1$, $w(parent(k_1), k_1) \notin W_2$, $w(parent(k_2), k_2) \notin W_1$, $w(parent(k_2), k_2) \in W_2$, $V_{k1} = V_{k2}$, and $parent(k_1) \neq parent(k_2)$. Consider T_{E1} and T_{E2} as depicted in Figures 3(c) and 3(d), respectively. The node with $V_k =$ "`E4`" and $\lambda_k =$ "`?`" is a moved leaf node.

(6) ***Move a subtree***. Let $\mathcal{K}_x$ and $\mathcal{W}_x$ be two sets of nodes and edges in subtree S_{rx}, respectively. Subtree S_{r1} is moved to be subtree S_{r2} iff $r_1 \in K_1$, $r_2 \in K_2$, $w(parent(r_1), r_1) \in W_1$, $w(parent(r_1), r_1) \notin W_2$, $w(parent(r_2), r_2) \notin W_1$, $w(parent(r_2), r_2) \in W_2$, $\forall_{k_1 \in \mathcal{K}_1}$ $(k_{1i} \in K_1)$, $\forall_{w_1 \in \mathcal{W}_1}$ $(w_{1j} \in W_1)$, $\forall_{k_2 \in \mathcal{K}_2}$ $(k_{2i} \in K_2)$, $\forall_{w_2 \in \mathcal{W}_2}$ $(w_{2j} \in W_2)$, $V_{r1} = V_{r2}$, and $parent(r_1) \neq parent(r_2)$.

(7) ***Update of order***. Let k_x be a leaf/internal node in K_x. Let $order(k)$ be the left-to-right position of node k among its siblings. The order of node k_1 is updated to be the order of node k_2 iff $order(k_1) \neq order(k_2)$, $parent(k_1) = parent(k_2)$, $V_{parent(k_1)} =$ "`,`", and $V_{parent(k_2)} =$ "`,`". Consider T_{E1} and T_{E2} as depicted in Figures 3(c) and 3(d), respectively. The order of node with $V =$ "`E7`" is updated from "2" to "1". Similarly, the order of node with $V =$ "`E6`" is updated from "1" to "2". Note that this type of changes is different from the move a leaf node/subtree. In this type of changes, the parent nodes before and after the changes occur are the same. In the move a leaf node/subtree types of changes, the leaf nodes/subtrees have different parent nodes before and after the changes occur.

(8) ***Insertion of Cardinality***. Let $k_1 = (V_{k1}, \lambda_1)$ and $k_2 = (V_{k2}, \lambda_2)$ be two nodes where $k_1 \in K_1$, $k_2 \in K_2$, $V_{k1} = V_{k2}$, $\lambda_1 = \emptyset$, and $\lambda_2 \neq \emptyset$. Then λ is an inserted cardinality iff $\lambda_2 = \lambda$.

(9) ***Deletion of Cardinality***. Let $k_1 = (V_{k1}, \lambda_1)$ and $k_2 = (V_{k2}, \lambda_2)$ be two nodes where $k_1 \in K_1$, $k_2 \in K_2$, $V_{k1} = V_{k2}$, and $\lambda_1 \neq \emptyset$. Then, the cardinality of k_1 is deleted iff $\lambda_2 = \emptyset$.

(10) ***Update of Cardinality***. Let $k_1 = (V_{k1}, \lambda_1)$ and $k_2 = (V_{k2}, \lambda_2)$ be two nodes where $k_1 \in K_1$, $k_2 \in K_2$, $V_{k1} = V_{k2}$, $\lambda_1 \neq \emptyset$, and $\lambda_2 \neq \emptyset$. Then, the cardinality of k_1 is updated in k_2 if $\lambda_1 \neq \lambda_2$. Consider two $T_{courses}$ as depicted in Figures 3(a) and 3(b). The cardinality of node *course* is updated from "`*`" to "`+`".

Note that we do not consider *optimal* delta for representing update of a node name. That is, instead of representing the above change as a single "update" operation, we detect it as a sequence of "delete" and "insert" operations. This is primarily due to the following reason. Consider the ETDs `school` in D_1 and D_2. We cannot consider that the name of element `name` is updated to `sinfo` and element `dean` is deleted as it will lead us to have a delta that is semantically incorrect. On the other hand, suppose we have a "`lastname`" element

whose name is updated to "`surname`". In this case, the update operation is semantically correct. However, it is extremely difficult to automatically extract such semantic relationship between element names. Consequently, DTD-Diff detects this change as a deletion of element "`lastname`" and an insertion of element "`surname`" as we do not have information of semantic relationships between "`lastname`" and "`surname`". In other words, for this case we choose to detect sub-optimal deltas over optimal but semantically incorrect deltas. Note that our delta is still correct even though it is not optimal.

Types of Changes of Attribute Declaration (AD). Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be the sets of entity names in $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. The types of changes in ADs are defined as follows.

(1) ***Insertion of a new AD***. $A = (N_A, S_A)$ is a new AD iff $N_A \notin \mathcal{N}_1$ and $N_A \in \mathcal{N}_2$.

(2) ***Deletion of an ED***. $A = (N_A, S_A)$ is a deleted AD iff $N_A \notin \mathcal{N}_2$ and $N_A \in \mathcal{N}_1$.

Suppose we have two ADs $A_1 = (N_{A1}, S_{A1})$ and $A_2 = (N_{A2}, S_{A2})$ where $A_1 \in \mathcal{A}_1$, $A_2 \in \mathcal{A}_2$, and $N_{A1} = N_{A2}$. Let $\mathcal{N}_{s1}$ and $\mathcal{N}_{s2}$ be the sets of attribute names in S_{A1} and S_{A2}, respectively. The types of changes in ADs are defined as follows.

(1) ***Insertion of a new attribute***. Attribute $a = (N_a, Y_a, D_a)$ is a new attribute iff $N_a \notin \mathcal{N}_{s1}$ and $N_a \in \mathcal{N}_{s2}$.

(2) ***Deletion of an attribute***. Attribute $a = (N_a, Y_a, D_a)$ is a deleted attribute iff $N_a \in \mathcal{N}_{s1}$ and $N_a \notin \mathcal{N}_{s2}$.

(3) ***Update of attribute type***. The *attribute type* of attribute $a = (N_a, Y_a, D_a)$ is updated in the new version iff $N_a \in \mathcal{N}_{s1}$, $N_a \in \mathcal{N}_{s2}$, and $Y_a \neq Y_a{}'$, where $Y_a{}'$ is attribute type of a in S_{A2}.

(4) ***Update of default value***. The *default value* of attribute $a = (N_a, Y_a, D_a)$ is updated in the new version iff $N_a \in \mathcal{N}_{s1}$, $N_a \in \mathcal{N}_{s2}$, and $D_a \neq D_a{}'$, where $D_a{}'$ are default values of a in S_{A2}.

For example, consider D_1 and D_2 in Figures 1(a) and 1(b), respectively. The default value of the attribute `year` of element `course` is updated from `#IMPLIED` to `#REQUIRED`. Note that we do not consider the update of attribute name for the same reason as in the above discussion.

Types of Changes of Entity Declaration (ED). Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be the sets of entity names in $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively. The types of changes in EDs are defined as follows.

(1) ***Insertion of a new ED***. Entity G is a new ED iff $N_g \notin \mathcal{N}_1$ and

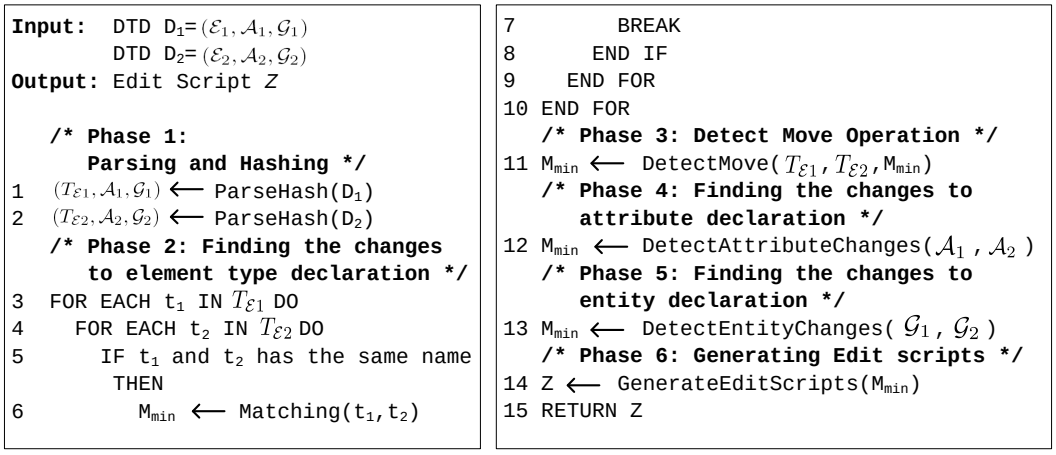


Fig. 4. The DTD-Diff Algorithm.

$N_g \in \mathcal{N}_2$ where N_g is the name of the entity in G.

(2) ***Deletion of an ED***. Entity G is a deleted ED iff $N_g \notin \mathcal{N}_2$ and $N_g \in \mathcal{N}_1$.

(3) ***Update of replacement text of internal ED***. The *replacement text* of internal entity $I_1 = (N_{I_1}, R_{I_1})$ is updated to the *replacement text* of entity $I_2 = (N_{I_2}, R_{I_2})$ iff $I_1 \in \mathcal{G}_1$, $I_2 \in \mathcal{G}_2$, $N_{I_1} = N_{I_2}$ and $R_{I_1} \neq R_{I_2}$.

(4) ***Update of location of external ED***. The *URI* of external entity $J_1 = (N_{J_1}, U_{J_1}, P_{J_1})$ is updated to the *URI* of the external entity $J_2 = (N_{J_2}, U_{J_2}, P_{J_2})$ iff $J_1 \in \mathcal{G}_1$, $J_2 \in \mathcal{G}_2$, $N_{J_1} = N_{J_2}$, and $U_{J_1} \neq U_{J_2}$.

(5) ***Update of content notation of external ED***. The *content notation* of external entity $J_1 = (N_{J_1}, U_{J_1}, P_{J_1})$ is updated to the *content notation* of the external entity $J_2 = (N_{J_2}, U_{J_2}, P_{J_2})$ iff $J_1 \in \mathcal{G}_1$, $J_2 \in \mathcal{G}_2$, $N_{J_1} = N_{J_2}$, and $P_{J_1} \neq P_{J_2}$.

Note that if an entity g is changed from being an *internal entity* to being an *external entity*, or vice versa, then we consider as a pair of a deletion of an entity and an insertion of an entity.

4 DTD-Diff Algorithm

In this section, we present the DTD-Diff algorithm. The outline of the algorithm is depicted in Figure 4(a). It takes as input two DTDs $D_1 = (\mathcal{E}_1, \mathcal{A}_1, \mathcal{G}_1)$ and $D_2 = (\mathcal{E}_2, \mathcal{A}_2, \mathcal{G}_2)$ representing old and new versions of a DTD, respectively, and returns an edit script Z containing the differences between D_1 and D_2. The algorithm consists of six phases: the *parsing and hashing phase*, the *matching phase*, the *move detection phase*, the *attribute declaration changes detection phase*, the *entity declaration changes detection phase*, and the *edit script generation phase*. We shall discuss each phase in turns.

```
Input: Node N
Output: The hash value of node N

1  IF N is leaf node THEN
2    RETURN MD5Value(label(N)• cardinality(N))
3  ELSE IF N is non-leaf node THEN
4    conentenated_text = empty text
5    FOR EACH child IN children OF N
6      CalculateHashValue ( child )
7    END FOR
8    IF N is choice group THEN
9      sort children of N by their hash values
10   END IF
11   FOR EACH child IN children OF N
12     conentenated_text •= HashValue(child)
13   END FOR
14   conentenated_text •= label(N)• cardinality(N)
15   RETURN MD5Value(conentenated_text)
16 END IF
```

Fig. 5. The *CalculateHashValue* Algorithm.

4.1 The Parsing and Hashing Phase

Given two DTDs, D_1 and D_2, DTD-DIFF parses D_1 and D_2 into $(\mathcal{T}_{\mathcal{E}1}, \mathcal{A}_1, \mathcal{G}_1)$ and $(\mathcal{T}_{\mathcal{E}2}, \mathcal{A}_2, \mathcal{G}_2)$, respectively, and computes their hash values. Note that $\mathcal{T}_1$ and $\mathcal{T}_2$ are two sets of content trees of $\mathcal{E}_1$ and $\mathcal{E}_2$, respectively. Since a content tree of an element type declaration has both *ordered* and *unordered* parts (the child nodes of the sequence and choice groups, respectively), the algorithm for computing the hash values must be able to address this issue. Given a node x in $T_E \in \mathcal{T}_{\mathcal{E}_i}$ where $i \in \{1, 2\}$, the hash value of node x is calculated as follows.

- If node x is a *leaf node*, then $Hash(x) = \mathbf{MD5Value}(label(x) \bullet cardinality(x))$.
- If node x is a *non-leaf node* and a *sequence group*, then $Hash(x) = \mathbf{MD5\text{-}Value}(Hash(c_1) \bullet Hash(c_2) \bullet ... \bullet Hash(c_n) \bullet label(x) \bullet cardinality(x))$, where $Hash(c_i)$ is the hash value of the child node of node x.
- If node x is a *non-leaf node* and a *choice group*, then $Hash(x) = \mathbf{MD5\text{-}Value}(Hash(c_1) \bullet Hash(c_2) \bullet ... \bullet Hash(c_n) \bullet label(x) \bullet cardinality(x))$, where $Hash(c_i)$ is the hash value of the child node of node x, and $Hash(c_1) < Hash(c_2) < ... < Hash(c_n)$.

Note that "•" denotes concatenation of strings. Function **MD5Value** is a hash function based on the MD5 Message-Digest algorithm [18]. We acknowledge that there is a very few probability of collisions of some hash functions (including MD5 hash algorithm) [5,10,11,21] that will influence the result quality of our approach. The hash function in DTD-DIFF can be replaced by other hash algorithms without any significant changes to the algorithm.

```
Input: Two root node r1 and r2
Output: a set of matching pairs M

1   M = empty set
2   push pair {r1,r2} into queue Q
3   WHILE (Q is not empty)
4     pop a pair {r1,r2} from queue Q
5     M = M ∪ {r1,r2}
6     IF HashValue(r1)<>HashValue(r2) AND
         N1, N2 are non-leaf nodes THEN
        /* compute the cost of matching every
           pair of child nodes of r1 and r2 */
7       FOR EACH child1 IN children of r1
8         FOR EACH child2 IN children of r2
9           IF label(child1)=label(child2) THEN
10            ComputeCost(child1, child2)
11          ELSE
12            Cost(child1,child2) = ∞
13          END IF
14        END FOR
15      END FOR
16      matched_pairs = set of pairs resulting from
            minimum-cost bipartite-matching among
            child nodes of r1 and r2
17      FOR EACH pair{x,y} IN matched_pairs
18        push pair{x,y} into queue Q
19      END FOR
20    END IF
21  END WHILE
22  RETURN M
```

Fig. 6. The *Matching* Algorithm.

The *CalculateHashValue* algorithm is depicted in Figure 4(b). We also calculate the hash values of AD in $\mathcal{A}$ and ED in $\mathcal{G}$. The hash value of AD $A \in \mathcal{A}$ is calculated as follows. $Hash(A) = \textbf{MD5-Value}(Hash(N_A) \bullet Hash(s_1) \bullet \ldots \bullet Hash(s_x)$, where $Hash(s_x) = \textbf{MD5-Value}(Hash(N_s) \bullet Hash(Y_s) \bullet Hash(D_s))$, $s_x \in S_A$, and $Hash(s_1) < Hash(s_2) < \ldots < Hash(s_x)$. The hash value of ED $E \in \mathcal{G}$ is calculated as follows. $Hash(E) = \textbf{MD5-Value}(Hash(N_E) \bullet \mathcal{H})$, where if E is an internal entity declaration, then $\mathcal{H} = Hash(R_E)$. Otherwise, E is an external entity declaration, and $\mathcal{H} = Hash(U_E) \bullet Hash(P_E)$. The overall complexity of calculating the hash values is $O(\sum_{i=1}^{|\mathcal{T}_1|}(|T_{Ei}| \times d_i) + \sum_{j=1}^{|\mathcal{T}_2|}(|T_{Ej}| \times d_j) + |\mathcal{A}_1| + |\mathcal{A}_2| + |\mathcal{G}_1| + |\mathcal{G}_2|)$ where $|\mathcal{T}_1|$ and $|\mathcal{T}_2|$ are the numbers of content trees in $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, $|T_{Ei}|$ is the number of nodes in T_{Ei}, and d_i is the average out-degree of T_{Ei}.

4.2 The Matching Phase

Given two content trees of ETDs E_1 and E_2, denoted as T_{E1} and T_{E2}, respectively, DTD-Diff invokes the *Matching* algorithm as depicted in Figure 6. The *Matching* algorithm returns a set of matching pairs M_{min}. The principle behind the *Matching* algorithm in DTD-Diff is based on the one in X-Diff [22]. That is, our matching technique finds the minimum-cost bipartite matchings of two content trees. However, there are critical differences between the *Matching* algorithm in DTD-Diff and the one in X-Diff as we exploit the unique structural and semantic features of a DTD.

First, the *Matching* algorithm in X-Diff is invoked once. DTD-Diff invokes the *Matching* algorithm as many as the number of ETDs. Observe that each ETD in a DTD has a unique name and hierarchy. Each root node in the content tree appears only once and mapping occurs only between nodes with the same signature. So each smaller content tree will be compared with another smaller tree from the second version having the root node with same name. For example, the content tree rooted at node labeled *school* in Figure 3(a) will be compared with the content tree in Figure 3(b) whose root has the same

```
Input: Two node r1 and r2
Output: C, Cost of matching r1 and r2

1   C = 0
2   IF HashValue(r1) = HashValue(r2) THEN RETURN 0
    /*Cost of update operation*/
3   IF cardinality(r1) <> cardinality(r2) THEN C=1
4   IF r1 and r2 are leaf node THEN RETURN C
     /* recursively compute the cost of matching every
        pair of child nodes of r1 and r2 */
5   FOR EACH child1 IN children of r1
6     FOR EACH child2 IN children of r2
7       IF label(child1)=label(child2) THEN
8         ComputeCost(child1, child2)
9       ELSE
10        Cost(child1,child2) = ∞
11      END IF
12    END FOR
13  END FOR
```

```
14  matched_pairs = set of pairs resulting from minimum-cost
       bipartite-matching among child nodes of r1 and r2
15  C = C + cost of minimum-cost bipartite-matching
       among child nodes of r1 and r2
16  FOR EACH child1 IN children of r1
17    IF child1  matched_pairs THEN
18      C = C + 1 /* cost of delete operation*/
19    END IF
20  END FOR
21  FOR EACH child2 IN children of r2
22    IF child2  matched_pairs THEN
23      C = C + size of child2 /* cost of insert operation*/
24    END IF
25  END FOR
26  IF r1 and r2 are sequence group THEN
27    C = C + number of local move operations required
28  END IF
29  RETURN C
```

Fig. 7. The *ComputeCost* Algorithm.

```
Input: A1 and A2
Output: Attribute Matching M

1  M = empty set
2  FOR EACH a1 IN A1 DO
3    FOR EACH a2 IN A2 DO
4      IF a2 is already matched THEN
5        CONTINUE
6      ELSE IF Hash(a1) = Hash(a2) THEN
7        M = M ∪(a1,a2)
8        Mark a1 and a2 that
            they are already matched
9        BREAK
10     ELSE IF a1.name = a2.name THEN
11       IF a1.type <> a2.type THEN
12         Mark a1 and a2 that
              their attribute types are updated
13       END IF
14       IF a1.defval <> a2.defval THEN
15         Mark a1 and a2 that
              their default values are updated
16       END IF
17       M = M ∪(a1,a2)
18       BREAK
19     END IF
20   END FOR
21 END FOR
22 RETURN M
```

Fig. 8. The *detectAttributeChanges* Algorithm.

label. Note that this computation is independent from the remaining content trees. Second, the *ComputeCost* algorithm in Figure 7 that is invoked by the *Matching* algorithm in DTD-Diff to compute the cost matching between r_1 and r_2 considers the cardinality changes (line 3, Figure 7). Note that the *Matching* algorithm in X-Diff does not consider the cardinality changes as it deals with XML documents, not DTDs. Third, unlike X-Diff which is based on unordered trees, a content tree can have ordered and unordered subtrees. Hence, in order to ensure our matching technique works on ordered subtrees as well, we adopt the technique used in XyDiff [7] to find the largest order preserving sequences among those matching pairs in sequence groups (lines 26–28, Figure 7).

We now analyze the complexity of the matching phase. Let $|T_{E1}|$ and $|T_{E2}|$ be the numbers of nodes in the content trees T_{E1} and T_{E2}, respectively. The complexity of finding the minimum-cost bipartite matchings is $O(|T_{E1}| \times |T_{E2}| \times max\{deg(T_{E1}), deg(T_{E2})\} \times log(max(\{deg(T_{E1}), deg(T_{E2})\})$, where $deg(T_{E1})$ and $deg(T_{E2})$ are the maximum out-degree in T_{E1} and T_{E2}, respectively [22]. Suppose the numbers of ETDs in D_1 and D_2 are α_1 and α_2, respectively. Then, the total complexity of finding the minimum-cost bipartite matching can be estimated as $O(min\{\alpha_1, \alpha_2\} \times \overline{|T_{\mathcal{E}1}|} \times \overline{|T_{\mathcal{E}2}|} \times max\{\overline{d_1}, \overline{d_2}\} \times log(max\{\overline{d_1}, \overline{d_2}\})$, where $\overline{|T_{\mathcal{E}1}|}$ and $\overline{|T_{\mathcal{E}2}|}$ are the average numbers of nodes of the content trees

in $T_{\mathcal{E}1}$ and $T_{\mathcal{E}2}$, respectively, and $\overline{d_1}$ and $\overline{d_2}$ are the average out-degree of the content trees in $T_{\mathcal{E}1}$ and $T_{\mathcal{E}2}$, respectively.

4.3 The Move Detection Phase

After we have a set of matching pairs M_{min}, DTD-Diff detects move operations. Formally, the move operation is defined as follows. Let n_1 and n_2 be two nodes in T_{E1} and T_{E2}, respectively. Let $parent(n)$ be the parent node of node n. Then, node n_1 is moved to be node n_2 iff ($parent(n_1)$,$parent(n_2)$) $\notin M_{min}$ and $Hash(n_1) = Hash(n_2)$. Note that we only consider a move operation if the hash values of n_1 and n_2 are the same. This is because if the hash values of n_1 and n_2 are different, then we need to check the differences in the subtrees rooted at n_1 and n_2. If the hash values of n_1 and n_2 are different, then the algorithm detects it as a deletion of n_1 and an insertion of n_2.

Now, we discuss how the move operations are detected. Let P and Q be two lists of the subtrees from the first and second versions, respectively, that have no matching subtrees in M_{min}. Subtrees in P and Q are sorted by their size in decreasing order. For each subtree in P, the algorithm checks whether there is a subtree in Q that have the same hash value. If $p_i \in P$ and $q_j \in Q$ have the same hash value, then the algorithm marks that subtree p_i in the first version is moved to be subtree q_j in the second version. The complexity of the algorithm for finding move operation is $O(n \times log(n))$, where n is the number of nodes in the content tree.

4.4 The Attribute Declaration Change Detection Phase

Recall that attribute list can be seen as a collection of attributes. Given two collections of attributes of an element in the first and second versions, the changes to the attribute list can be detected by using the algorithm in Figure 8. The complexity of the algorithm for finding the changes on the attribute lists is $O(n \times log(n))$, where n is the number of attributes defined in the DTD. Note that we do not consider the update of the attribute name for the reasons discussed in Section 3. We consider the update of the attribute name is represented as a pair of deletion and insertion of an attribute.

4.5 The Entity Declaration Change Detection Phase

The change detection mechanism of EDs is quite straightforward and similar to the approach for detecting changes to attribute declarations. Hence, we do not elaborate on this step further. The complexity of the algorithm for finding

Code	# Element Types	DTD		XSD	
		File size (Kb)	# Nodes	File size (Kb)	# Nodes
E005-B05-D02	5	2	105	7	390
E010-B05-D02	10	3	175	12	691
E015-B05-D02	15	4	275	17	1,031
E025-B05-D02	25	6	490	30	1,847
E050-B05-D02	50	12	900	56	3,460

Code	# Element Types	DTD		XSD	
		File size (Kb)	# Nodes	File size (Kb)	# Nodes
E075-B05-D02	75	18	1,430	87	5,360
E100-B05-D02	100	23	1,880	113	7,044
E150-B05-D02	150	36	2,785	170	10,564
E250-B05-D02	250	59	4,410	273	16,903
E500-B05-D02	500	122	9,280	570	35,076

(a) Different Number of Element Types

Code	Out-degree	DTD		XSD	
		Filesize (Kb)	# Nodes	Filesize (Kb)	# Nodes
E025-B05-D02	5	6	485	30	1,837
E025-B10-D02	10	12	1,585	82	5,022
E025-B15-D02	15	21	3,265	162	10,047
E025-B25-D02	25	45	7,975	385	24,021
E025-B40-D02	40	114	21,625	1,032	64,611
E025-B50-D02	50	167	31,325	1,500	94,014

(b) Different Number of Out-degree

Code	Depth	DTD		XSD	
		Filesize (Kb)	# Nodes	Filesize (Kb)	# Nodes
E025-B05-D01	1	5	150	13	868
E025-B05-D02	2	6	465	28	1,731
E025-B05-D03	3	10	1,215	68	3,896
E025-B05-D04	4	21	3,585	194	10,444
E025-B05-D05	5	46	9,045	500	25,720
E025-B05-D06	6	86	17,305	994	49,068
E025-B05-D07	7	209	43,465	2,853	122,182
E025-B05-D08	8	557	117,180	7,231	328,862

(c) Different Number of Depth

Fig. 9. Data Sets.

the changes on the entity declarations is $O(n \times log(n))$, where n is the number of entity declarations defined in the DTD.

4.6 The Edit Scripts Generation Phase

The edit script Z is generated as follows. (1) An edit script Z is initialized as a set of *move operations* detected in the preceding step. (2) Then, for all unmatching nodes in the first tree, *delete operations* are added into edit script Z. (3) Next, for all unmatching nodes in the second tree, *insert operations* are added into edit script Z. (4) For all pairs of matching nodes that have different cardinality, *cardinality update operations* are added into edit script Z. (5) For all pairs of matching nodes that belong to sequence groups and have incorrect local order, *local order move operations* are added into edit script Z. (6) The *changes to the attributes lists* are added into edit script Z. (7) Finally, the *changes to the entity declarations* are added into edit script Z. The overall complexity of this step is $O(\sum_{i=1}^{|\mathcal{T}_1|}(|T_{Ei}|)+\sum_{j=1}^{|\mathcal{T}_2|}(|T_{Ej}|)+|\mathcal{A}_1|+|\mathcal{A}_2|+|\mathcal{G}_1|+|\mathcal{G}_2|)$.

5 Experimental Results

We have implemented DTD-Diff entirely in Java. The experiments were conducted on a Microsoft Windows XP Professional machine having Pentium 4 1.7 GHz processor with 512 MB of memory. We use both real world DTDs and a set of synthetic DTDs generated by using our DTD generator. The second versions of DTDs are generated by using our DTD changes generator.

We vary the *numbers of element types*, the *percentage of changes*, the *out-degree* of each element types, and the *depth* of each element types.

Recall from Section 1, the results of state-of-the-art XML change detection algorithms (X-Diff and XyDiff) suffer from several limitations. However, as we are not aware of any publicly available change detection tools for DTDs, we compare the performance of DTD-Diff with the C version of XyDiff (downloaded from http://pauillac.inria.fr/cdrom/www/xydiff/index-eng.htm) [2] and the Java version of X-Diff[22] (downloaded from http://www.cs.wisc.edu/~yuanwang/xdiff.html). The C version of XyDiff was run in Pentium 4 1.7 GHz processor with 512 MB of memory and Red Hat 9 Linux Operating System. Note that as the Java version is in general slower than the C version, the execution times of XyDiff will differ by a constant factor in comparison with DTD-Diff and X-Diff.

Further, as X-Diff and XyDiff are not designed for detecting the changes on DTDs, we convert the DTDs into XML Schema (XSD) [2] using Syntex dtd2xs tool (downloaded from http://www.syntext.com/downloads/index.htm) before detecting the changes. For example, given a DTD depicted in Figure 1(a), the equivalent XSD generated by using Syntex dtd2xs is depicted in Figure 1(c) (partial view only). Note that we have also investigated several other DTD-to-XML Schema conversion tools that are freely available publicly. For instance, we used *LuMrix dtd2xs* (http://www.lumrix.net/dtd2xs.php) to convert DTDs into XSDs. To the best of our knowledge, these tools produced almost similar XSD files and the differences did not significantly influence the performance of X-Diff and XyDiff.

5.1 Execution Time vs Number of Element Types

We first study the performance of DTD-Diff by varying the number of element types. We set the *out-degree* and *depth* of each element type to "5" and "3", respectively. Note that the average of the maximum depth of real DTDs is "3" [6]. The number of attributes of each element is set to "3". We set the percentages of changes to "1%" and "9%". The characteristic of the data sets used in this set of experiments is depicted in Figure 9(a). Figure 10(a) shows the execution time of converting DTD into XSD file using Syntex dtd2xs.

Figure 10(b) depicts the performance of DTD-Diff, X-Diff, and XyDiff when the percentage of changes is set to "1%". We observed that DTD-Diff significantly outperforms X-Diff and XyDiff. DTD-Diff is 8.6–155 times faster than X-Diff and 7.63–26.42 times faster than XyDiff. Figure 10(c) presents

[2] Unfortunately, despite our best efforts (including contacting the authors), we could not get the Java version of XyDiff.

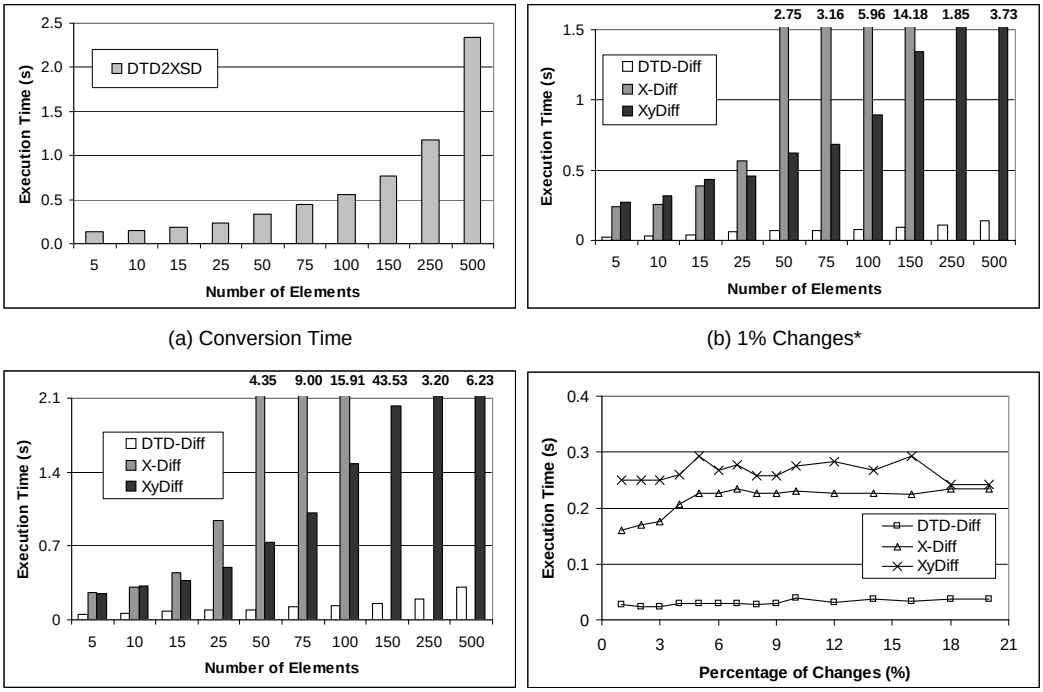


Fig. 10. Experimental Results (1).

the performances of DTD-Diff and X-Diff when the percentage of changes is set to "9%". In this case, DTD-Diff is 5–272 times faster than X-Diff and 4.68–20.10 times faster than XyDiff. X-Diff failed to detect the changes when the numbers of elements are more than or equal to 250 due to lack of main memory. The inability of X-Diff to process large number of nodes in XML data is also highlighted in [14,15].

We now elaborate on why our approach significantly outperforms X-Diff. First, the tree representations of XSD files (XSD tree) contain elements with same names. On the other hand, in DTD-Diff, each root node of the content trees in a DTD has a unique name. As a result, there exists a one-to-one mapping between a content tree in the old version to another content tree in the new version. Consequently, X-Diff does more number of bipartite matching compared to DTD-Diff. Second, the number of nodes in the content trees is lesser in most cases compared to an XSD tree. This further reduces the number and cost of bipartite matching in DTD-Diff. To elaborate further, let $|T_1|$ and $|T_2|$ be the numbers of nodes in the XSD trees of DTDs D_1 and D_2, respectively. The complexity of finding the minimum-cost bipartite matchings between $|T_1|$ and $|T_2|$ in X-Diff is $O(|T_1| \times |T_2| \times max\{deg(T_1), deg(T_2)\} \times log(max\{deg(T_1), deg(T_2)\})$ [22]. The improvement of DTD-Diff over X-Diff can be estimated as $O(|T_1| \times |T_2| \times max\{deg(T_1), deg(T_2)\} \times log(max\{deg(T_1), deg(T_2)\}))/O(min\{\alpha_1, \alpha_2\} \times \overline{|T_{\mathcal{E}1}|} \times \overline{|T_{\mathcal{E}2}|} \times max\{\overline{d_1}, \overline{d_2}\} \times$

DTD	Number of Element Types	Number of Attribute List
SigmodRecord	11	1
PSD	66	10
Policy7	56	26
DBLP	36	12
NewsML_1.1	117	114

(a) Real DTD Characteristics

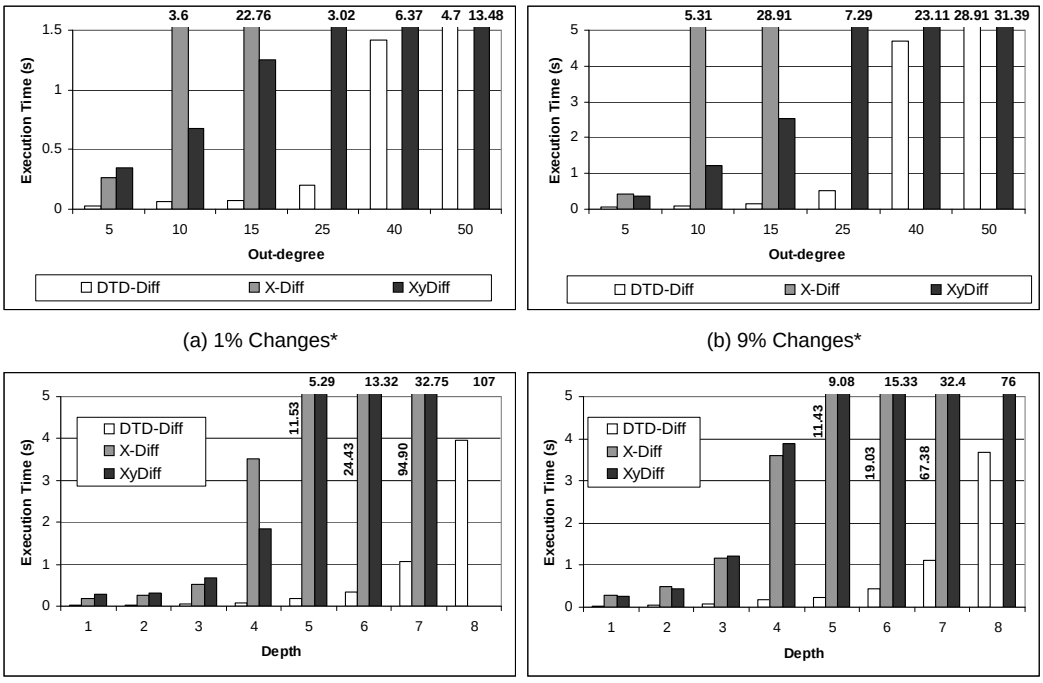
DTD	DTD-Diff	X-Diff			XyDiff		
		DTD2XSD	Detect	**Total**	DTD2XSD	Detect	**Total**
SigmodRecord	0.021	0.010	0.016	0.026	0.010	0.217	0.227
PSD	0.032	0.010	0.022	0.032	0.010	0.342	0.352
Policy7	0.031	0.010	0.021	0.031	0.010	0.333	0.343
DBLP	0.032	0.010	0.024	0.034	0.010	0.291	0.301
NewsML_1.1	0.041	0.010	0.027	0.037	0.010	0.517	0.527

(b) Execution Time (seconds)

Fig. 11. Experimental Results: Real Data Sets.

$log(max\{\overline{d_1}, \overline{d_2}\})))$. Assuming that $|\alpha_1| = |\alpha_2| = \alpha$, $\overline{|T_{\mathcal{E}1}|} = \overline{|T_{\mathcal{E}2}|} = n$, $|T_1| = |T_2| = t$, and $max(deg(T_1), deg(T_2)) \times log(max(deg(T_1), deg(T_2))) = x$, the complexity comparison becomes $O(xt^2/x\alpha n^2) = O(t^2/\alpha n^2)$. Note that $|T_1|$ and $|T_2|$ include the attribute and entity declarations of D_1 and D_2. However, in the case of DTD-Diff $\overline{|T_{\mathcal{E}1}|}$ and $\overline{|T_{\mathcal{E}2}|}$ do not include these declarations as their changes are detected separately. Based on our discussion in the preceding section, it is less expensive in DTD-Diff to compute changes to EDs and ADs. Furthermore, numbers of nodes in the XSD files are larger than the number of nodes in the content trees (from 2.8 up to 5.8 times larger, Figure 9). Therefore, $\alpha \times n^2 \leq t^2$. Hence, the performance of DTD-Diff is always faster or in the worst case comparable to X-Diff. DTD-Diff is faster than XyDiff due to the similar reasons as above.

We also study the performance of DTD-Diff and X-Diff for detecting the changes to the real world DTDs [1,3]. Figure 11(a) depicts the characteristics of the real world DTDs. We set the percentage of changes to 3%. Figure 11(b) depicts the performances of DTD-Diff and X-Diff. We notice that X-Diff has slightly better performance than DTD-Diff. This is primarily due to the characteristics of the data. For instance, although $NewsML\_1.1$ has 117 elements, the performance of DTD-Diff is comparable to X-Diff! Observe that for synthetic data set with similar size, DTD-Diff outperforms X-Diff significantly. This is because in $NewsML\_1.1$, only 6 out of 117 ETDs have nested content and the maximum depth of $NewsML\_1.1$ DTD is only 2. Hence, cost of bipartite matching is almost the same. In summary, X-Diff performs relatively better than DTD-Diff when the DTDs have simple and "flat" structure. When the DTD structure is complex, DTD-Diff outperforms X-Diff as shown using synthetic dataset. Also, note that DTD-Diff is still better than X-Diff because of the inaccuracies and incompleteness in the results generated by X-Diff due to the limitations highlighted in Section 1. Compared to XyDiff;



(a) 1% Changes* (b) 9% Changes*

(c) 1% Changes* (d) 9% Changes*

* Excluding the conversion time from DTD to XSD (for X-Diff)

Fig. 12. Experimental Results (2).

we observe that DTD-Diff is up to 12.61 times faster.

5.2 Execution Time vs Percentage of Changes

In this set of experiments, we study the effects of the percentages of changes on the execution time of DTD-Diff, X-Diff, and XyDiff. We use the `E005-B05-D02` data set, whose number of element types, out-degree, and depth are 5, 5, and 2, respectively, as the first version of the DTD. We vary the percentages of changes from "1%" to "20%". Figure 10(d) depicts the execution time of DTD-Diff, X-Diff, and XyDiff for different percentages of changes. We observe that the percentage of changes slightly affect the performances of DTD-Diff, X-Diff, and XyDiff.

5.3 Execution Time vs Out Degree

In this set of experiments, we study the effects of the number of out-degree of each element type on the execution time of DTD-Diff, X-Diff, and XyDiff. We set the number of element types and the depth to "25" and "2", respectively. We set the percentages of changes to "1%" and "9%". We vary the out-

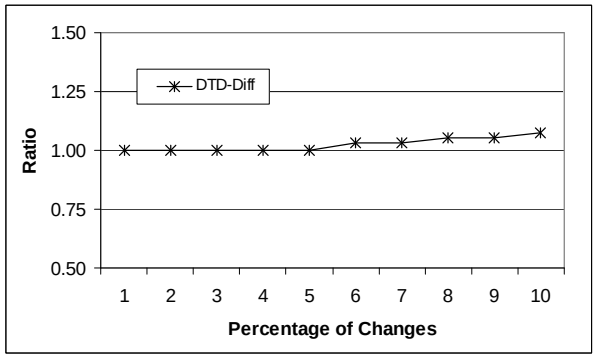


Fig. 13. Result quality.

degree of each element type from "5" to "50". The characteristic of the data sets used in this set of experiments is depicted in Figure 9(b). Figures 12(a) and 12(b) depict the performance of DTD-Diff, X-Diff, and XyDiff for different numbers of out-degree of each element type when the percentages of changes are set to "1%" and "3%", respectively. We observe that DTD-Diff is up to 325 times faster than X-Diff. DTD-Diff is 2.52–15.48 times faster than XyDiff. This is because of the reasons discussed above. We also notice that X-Diff cannot detect the changes to XSD files when the out-degree is more than or equal to 25 due to the lack of main memory.

5.4 Execution Time vs Depth

In this set of experiments, we study the effects of the depth of content tree on the execution time of DTD-Diff, X-Diff, and XyDiff. We set the number of element types and the out-degree to "25" and "5", respectively. We set the percentages of changes to "1%" and "9%". We vary the out-degree of each element type from "1" to "8". The characteristic of the data sets used in this set of experiments is depicted in Figure 9(c). Figures 12(c) and 12(d) depict the performance of DTD-Diff, X-Diff, and XyDiff for different depth of each content tree when the percentages of changes are set to "1%" and "9%", respectively. We observe that DTD-Diff is up to 89 times faster than X-Diff and 9.5–38 times faster than XyDiff. X-Diff failed to detect the changes when the depth is more than or equal to 8 due to the lack of main memory.

5.5 Result Quality

We also examine the quality of deltas detected by DTD-Diff. We use `E010-B05-D02` data set and the percentages of changes are varied between "1%" to "10%". The second versions are generated by using our DTD change

generator. Then, we calculate the result quality, that is, the ratio between the number of edit operations detected by DTD-Diff and the optimal one. Figure 13 depicts the ratios. We observe that DTD-Diff is able to detect the optimal deltas until percentage of changes is set to "5%". Afterwards, DTD-Diff detects almost optimal deltas. This is because, in some cases, a move operation is detected as a pair of deletion and insertion. Note that we do not compare the result quality of DTD-Diff to other approaches as, to the best of our knowledge, DTD-Diff is the first approach for detecting the changes to DTDs. We do not compare the result quality of DTD-Diff to the one of X-Diff (when we use XSD files) as the types of changes of DTD and XML are different.

6 Conclusions

A DTD change detection tool can be useful in several ways such as maintenance of XML documents, incremental maintenance of relational schema for storing XML data, and XML schema integration. In this paper, we present a novel technique for detecting the changes to DTDs. Our work is motivated by the problem that converting DTD to XML Schema (XSD) (which is in XML document format) and detecting the changes using existing XML change detection algorithms (X-Diff and XyDiff) is not a feasible option. Such effort is expensive and may generate semantically incorrect and non-optimal edit scripts. We propose an algorithm DTD-Diff that directly computes the changes between two versions of DTDs by taking into account the structural and semantic features of DTDs. We experimentally demonstrate that X-Diff performs relatively better than DTD-Diff when the DTDs have simple and "flat" structure. When the DTD structure is complex, DTD-Diff runs significantly faster (5–325 times) than X-Diff for the given data set. Further, even though DTD-Diff is implemented using Java, it is still up to 38 times faster than XyDiff (implemented in C). More importantly, DTD-Diff is also able to produce optimal or at least near-optimal deltas. As parts of future work, we will investigate on the problem of detecting the changes to XML Schema.

References

[1] UW XML Repository. *Database Research Group, University of Washington.* `http://www.cs.washington.edu/research/xmldatasets/`.

[2] XML Schema. *World Wide Web Consortium.* `http://www.w3.org/XML/Schema`.

[3] XML.ORG Registry and Repository for XML Schemas. `http://www.xml.org/xml/registry.jsp`.

[4] E. Bertino et al. Evolving a Set of DTDs According to a Dynamic Set of XML Documents. *In EDBT Workshops*, 2002.

[5] E. Biham, R. Chen, A. Joux, P. Carribault, W. Jalby, C. Lemuet. Collisions of SHA-0 and Reduced SHA-1. *In Eurocrypt 2005*, 2005.

[6] B. Choi. What are real DTDs like?. *In WebDB*, 2002.

[7] G. Cobena, S. Abiteboul, A. Marian. Detecting Changes in XML Documents. *In ICDE*, 2002.

[8] L. Al-Jadir, Fatmé El-Moukaddem. Once Upon a Time a DTD Evolved into Another DTD.... *Proc. of the 9th International Conference on Object-Oriented Information Systems*, 2003.

[9] G. Guerrini, M. Mesiti, D. Rossi. Impact of XML Schema Evolution on Valid Documents. *In ACM WIDM*, 2005.

[10] Vlastimil Klima. Finding MD5 Collisions a Toy For a Notebook. *Cryptology ePrint Archive, Report 2005/075*, 2005.

[11] Vlastimil Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute. *Cryptology ePrint Archive, Report 2006/105*, 2006.

[12] R. Krishnamurthy, R. Kaushik, J. F. Naughton. XML to SQL Query Translation Literature: The State of the Art and Open Problem.*In XSym*, 2003.

[13] E. Leonardi, S. S. Bhowmick, S. Madria. Detecting Content Changes on Ordered XML Documents Using Relational Databases. *In DEXA*, 2004.

[14] E. Leonardi, S. S. Bhowmick, S. Madria. Xandy: Detecting Changes on Large Unordered XML Documents Using Relational Databases. *In DASFAA*, 2005.

[15] E. Leonardi, S. S. Bhowmick. Detecting Changes on XML Documents Using Relational Databases: A Schema-Conscious Approach. *In ACM CIKM*, 2005.

[16] E. Leonardi, Tran T. Hoai, S. S. Bhowmick, S. Madria. DTD-Diff: A Change Detection Algorithm for DTDs. *In DASFAA*, 2006.

[17] M. Raghavachari, O. Shmueli. Efficient Schema-Based Revalidation of XML. *In EDBT*, 2004.

[18] Ronald L. Rivest. The MD5 Message Digest Algorithm. *Internet RFC 1321*, April 1992. `http://www.faqs.org/rfcs/rfc1321.html`.

[19] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton Relational Databases for Querying XML Documents: Limitations and Opportunities. *In VLDB*, 1999.

[20] H. Su, D. Kramer, L. Chen, K. Claypool, E. A. Rundensteiner. XEM: Managing the Evolution of XML Documents. *In RIDE*, 2001.

[21] X. Wang and H. Yu . How to Break MD5 and Other Hash Functions. *In Eurocrypt 2005*, 2005.

[22] Y. Wang, D. J. DeWitt, J. Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. *In ICDE*, 2003.