# DTD-Diff: A Change Detection Algorithm for DTDs

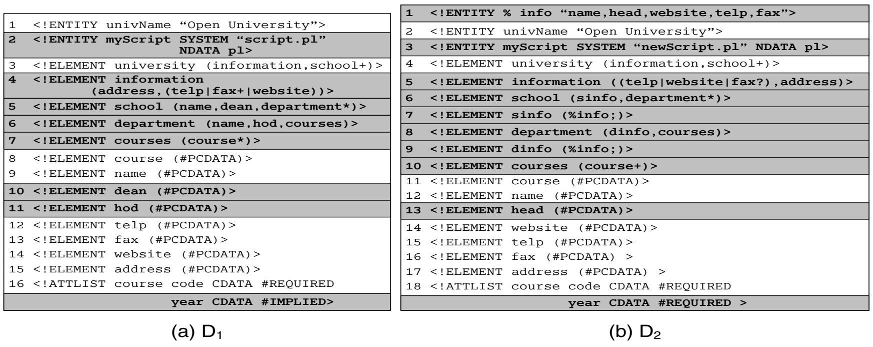
Erwin Leonardi[1], Tran T. Hoai[1], Sourav S. Bhowmick[1], and Sanjay Madria[2]

[1] School of Computer Engineering,
Nanyang Technological University, Singapore
[2] Department of Computer Science,
University of Missouri-Rolla, MO 65409, USA
{pk909134, assourav}@ntu.edu.sg, madrias@umr.edu

**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. We also study the result quality of detected deltas.

## 1 Introduction

XML has emerged as the leading textual language for representing and exchanging data over the Web. In many applications a schema (i.e., *Document Type Definition* (DTD) or *XML schema* (XSD) [3] is associated with a set of XML documents to define their legal structures. Schema of such XML documents may also need to be updated to reflect a change in the real world, a change in the user's requirements, mistakes in the initial design, etc. 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. Such DTD change detection tools can be useful in maintenance of XML documents when their DTD evolves, incremental maintenance of relational schema of the schema-conscious approach [9] for storing XML data, XML schema integration, etc. Let us elaborate further on the usage of DTD change detection tool in maintenance of XML documents. 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



```
<!ENTITY univName “Open University”>
<!ENTITY myScript SYSTEM “script.pl”
                     NDATA pl>
<!ELEMENT university (information,school+)>
<!ELEMENT information
           (address,(telp|fax+|website))>
<!ELEMENT school (name,dean,department*)>
<!ELEMENT department (name,hod,courses)>
<!ELEMENT courses (course*)>
<!ELEMENT course (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT dean (#PCDATA)>
<!ELEMENT hod (#PCDATA)>
<!ELEMENT telp (#PCDATA)>
<!ELEMENT fax (#PCDATA)>
<!ELEMENT website (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ATTLIST course code CDATA #REQUIRED
                 year CDATA #IMPLIED>
```

(a) $D_1$

```
<!ENTITY % info “name,head,website,telp,fax”>
<!ENTITY univName “Open University”>
<!ENTITY myScript SYSTEM “newScript.pl” NDATA pl>
<!ELEMENT university (information,school+)>
<!ELEMENT information ((telp|website|fax?),address)>
<!ELEMENT school (sinfo,department*)>
<!ELEMENT sinfo (%info;)>
<!ELEMENT department (dinfo,courses)>
<!ELEMENT dinfo (%info;)>
<!ELEMENT courses (course+)>
<!ELEMENT course (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT head (#PCDATA)>
<!ELEMENT website (#PCDATA)>
<!ELEMENT telp (#PCDATA)>
<!ELEMENT fax (#PCDATA) >
<!ELEMENT address (#PCDATA) >
<!ATTLIST course code CDATA #REQUIRED
                 year CDATA #REQUIRED >
```

(b) $D_2$

**Fig. 1.** Two versions of a DTD

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 $D'$.

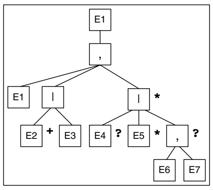
In this paper, we propose a novel algorithm, called DTD-Diff, for detecting the changes to DTDs. *To the best of our knowledge, this is the first approach that addresses the DTD change detection problem.* At first glance, it may seem that the DTD change detection problem can easily be addressed by existing change detection tools for XML documents [6, 7, 10]. Specifically, we can first transform DTDs to XSD files that are in XML format. Then, the changes to the DTDs can be detected using existing XML change detection tools (such as X-Diff [10] and XyDiff [6]). Although this approach will clearly detect changes, we argue that they suffer from these following limitations: *granularity of types of changes*, *inability to detect changes to both unordered and ordered nodes*, *detection of semantically incorrect changes*, *generation of non-optimal edit scripts*, and *performance bottleneck*. The details can be found in [8].

In summary, the main contributions of this paper are as follows. (1) In Section 2, we present data model to represent the changes to DTDs. By using this data model we are able to detect the changes to DTDs, that are discussed Section 3, correctly. (2) In Section 4, we propose a novel algorithm called DTD-Diff for detecting the changes to DTDs. 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 5–325 times faster than X-Diff [10]. Note that in our study, we convert DTDs to XSD files prior to employing X-Diff to detect the changes.

## 2 DTD Data Model

A DTD consists of *entity declaration* (`<!ENTITY ...>`), *element type declaration* (`<!ELEMENT ...>`), and *attribute declaration* (`<!ATTLIST ...>`) that describe entities, elements, and attributes, respectively. Formally,

**Definition 1 [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*



(a) Content Tree

```
1 Changes to Element Type Declaration (ETD)
  1.1 Insertion of new ETD
  1.2 Deletion of an ETD
  1.3 Changes to the Content Tree of an ETD
    - Changes to a leaf node
      1.3.1  Insertion of a leaf node
      1.3.2  Deletion of a leaf node
      1.3.3  Move of a leaf node
      1.3.4  Update of order of a leaf node
    - Changes to a subtree
      1.3.5  Insertion of a subtree
      1.3.6  Deletion of a subtree
      1.3.7  Move of a subtree
      1.3.8  Update of order of an internal node
             at which a subtree is rooted
    - Changes to Cardinality
      1.3.9  Insertion of a cardinality
      1.3.10 Deletion of a cardinality
      1.3.11 Update of a cardinality

2 Changes to Attribute Declaration (AD)
  2.1 Insertion of a new AD
  2.2 Deletion of an AD
  2.3 Insertion of a new attribute in AD
  2.4 Deletion of an attribute in AD
  2.5 Update of attribute type
  2.6 Update of default value

3 Changes to Entity Declaration (ED)
  3.1 Insertion of a new ED
  3.2 Deletion of an ED
  3.4 Update of replacement text of an internal ED
  3.5 Update of URI of an external ED
  3.6 Update of content notation of an external ED
```

(b) Types of changes

**Fig. 2.** Content Tree and Type of Changes

*(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 $\alpha$, $\beta$, and $\gamma$ then $|\mathcal{E}| = \alpha$, $|\mathcal{A}| = \beta$, and $|\mathcal{G}| = \gamma$.* □

For example, consider the DTD $D_2$ in Figure 1(b). Lines 1-3, 4-17, and 18 are examples of EDs, ETDs, and AD, respectively.

**Element Type Declaration (ETD):** In a DTD, XML elements are declared using element type declaration. Each element type declaration $E$ has a *name* $N_E$ and *element content* $C_E$. For example, consider the DTD $D_1$ in Figure 1(a). The name and the content of element type `school` (line 5) are `school` and `(name,dean,department*)`, respectively. Observe that *element content* can be very complex with multiple levels of nesting. For example, `<!ELEMENT E1 (E1,(E2+|E3),(E4?|E5*|(E6,E7)?)*)>`. We represent the element content $C_E$ as a *content tree* $T_E$. For example, consider the element type declaration `<!ELEMENT E1 (E1,(E2+|E3), (E4?|E5*| (E6,E7)?)*)>`. The content tree $T_{E1}$ is depicted in Figure 2(a). Note that in an element content $C_E$ we may have *sequence* (denoted by ",") and *choice* (denoted by "|") groups of elements. Observe that the elements in a *sequence* group must be ordered, and the order of elements in *choice* group is not significant. That is, a *content tree* $T_E$ may have *ordered* and *unordered* parts.

**Attribute Declaration (AD):** The attribute declaration in a DTD is used to define the attributes of an element. Each AD $A$ has a *name* $N_A$ of element type to which a set of attributes $S_A$ belongs. Each attribute $a$ in the attribute set $S_A$ has a *name* $N_a$, *type* $Y_a$, and an optional *default value* $D_a$. 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.

**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 $I$ has a *name* $N_I$ and a *replacement text* $R_I$. On the other hand, an *external* ED $J$ has a *name* $N_J$, *universal resource indicator (URI)* $U_J$, and a *content notation* $P_J$. 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. The details on the DTD data model can be found in [8].

## 3 Types of Changes

A set of DTD changes that can be detected by DTD-Diff is depicted in Figure 2(b). We notice that a DTD indeed has richer of types of changes compared to XML documents. In DTD, we have types of changes for cardinalities of elements, more meaningful types of changes for AD and ED, etc. The details of each type of changes depicted in Figure 2(b) can be found in [8]. In this section, we only briefly discuss two issues regarding the types of changes to DTDs.

**Update of Node/Attribute Name:** We do not consider update of node/ attribute name for 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`". DTD-Diff detects 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`". Note that the delta is still correct even though the result quality is reduced. Therefore, we consider the update of node/attribute name as a pair of deletion and insertion of a node in order to avoid semantically incorrect deltas in some cases.

**Changes to Entity Type:** 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 3(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 and returns an edit script $Z$ containing the differences between $D_1$ and $D_2$. The algorithm consists of six phases (Figure 3(a)). We shall discuss each phase in turns.

**The Parsing and Hashing Phase:** Given two DTDs, $D_1$ and $D_2$, DTD-Diff parses $D_1$ and $D_2$ into $(\mathcal{T}_1, \mathcal{A}_1, \mathcal{G}_1)$ and $(\mathcal{T}_2, \mathcal{A}_2, \mathcal{G}_2)$ 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 content tree of an element type declaration has both *ordered*

```
Input:  DTD D1=($\mathcal{E}_1, \mathcal{A}_1, \mathcal{G}_1$)
        DTD D2=($\mathcal{E}_2, \mathcal{A}_2, \mathcal{G}_2$)
Output: Edit Script Z

  /* Phase 1:
     Parsing and Hashing */
  ($T_{\mathcal{E}1}, \mathcal{A}_1, \mathcal{G}_1$) ⟵ ParseHash(D1)
  ($T_{\mathcal{E}2}, \mathcal{A}_2, \mathcal{G}_2$) ⟵ ParseHash(D2)
  /* Phase 2: Finding the changes
     to element type declaration */
  FOR EACH t1 IN $T_{\mathcal{E}1}$ DO
    FOR EACH t2 IN $T_{\mathcal{E}2}$ DO
      IF t1 and t2 has the same name
       THEN
        Mmin ⟵ Matching(t1,t2)
        BREAK
      END IF
    END FOR
  END FOR
  /* Phase 3: Detect Move Operation */
  Mmin ⟵ DetectMove($T_{\mathcal{E}1}$, $T_{\mathcal{E}2}$, Mmin)
  /* Phase 4: Finding the changes to
     attribute declaration */
  Mmin ⟵ DetectAttributeChanges($\mathcal{A}_1$, $\mathcal{A}_2$)
  /* Phase 5: Finding the changes to
     entity declaration */
  Mmin ⟵ DetectEntityChanges($\mathcal{G}_1$, $\mathcal{G}_2$)
  /* Phase 6: Generating Edit scripts */
  Z ⟵ GenerateEditScripts(Mmin)
  RETURN Z
```

(a) Outline of DTDDiff Algorithm

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

IF N is leaf node THEN
  RETURN MD5Value(label(N) • cardinality(N))
ELSE IF N is non-leaf node THEN
  conentenated_text = empty text
  FOR EACH child IN children OF N
    CalculateHashValue ( child )
  END FOR
  IF N is choice group THEN
    sort children of N by their hash values
  END IF
  FOR EACH child IN children OF N
    conentenated_text •= HashValue(child)
  END FOR
  conentenated_text •= label(N) • cardinality(N)
  RETURN MD5Value(conentenated_text)
END IF
```

(b) The *CalculateHashValue* Algorithm

**Fig. 3.** Outline of DTD-Diff Algorithm and The *CalculateHashValue* Algorithm

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

M = empty set
push pair {r1,r2} into queue Q
WHILE (Q is not empty)
  pop a pair {r1,r2} from queue Q
  M = M ∪ {r1,r2}
  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 */
    FOR EACH child1 IN children of r1
      FOR EACH child2 IN children of r2
        IF label(child1)=label(child2) THEN
          ComputeCost(child1, child2)
        ELSE
          Cost(child1,child2) = ∞
        END IF
      END FOR
    END FOR
    matched_pairs = set of pairs resulting from
        minimum-cost bipartite-matching among
        child nodes of r1 and r2
    FOR EACH pair{x,y} IN matched_pairs
      push pair{x,y} into queue Q
    END FOR
  END IF
END WHILE
RETURN M
```

**Fig. 4.** The *Matching* Algorithm

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. We use the *CalculateHashValue* algorithm as shown in Figure 3(b). Note that "•" in Figure 3(b) denotes concatenation of strings. Function **MD5Value** is a hash function based on the MD5 Message-Digest algorithm [1].

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. $Hash(A) =$ **MD5-Value**$(Hash(N_A) \bullet Hash(s_1) \bullet \ldots \bullet Hash(s_x)$, where $Hash(s_x) =$ **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. $Hash(E) =$ **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 $T_{Ei}$.

**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 4. The *Matching* algorithm returns a set of matching pairs $M_{min}$. The principle behind the *Matching* algorithm in DTD-Diff is based on

```
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. 5.** The *ComputeCost* Algorithm

the one in X-Diff [10]. 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. Note that this computation is independent from the remaining content trees. Second, the *ComputeCost* algorithm in Figure 5 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 5). 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 [6] to find the largest order preserving sequences among those matching pairs in sequence groups (line 26-28, Figure 5). The overall complexity of this phase is $O(min\{\alpha_1, \alpha_2\} \times \overline{|T_{\mathcal{E}1}|} \times \overline{|T_{\mathcal{E}2}|} \times max\{\overline{d_1}, \overline{d_2}\} \times log(max\{\overline{d_1}, \overline{d_2}\})$, 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, $\overline{d_1}$ and $\overline{d_2}$ are the average out-degree of the content trees in $T_{\mathcal{E}1}$ and $T_{\mathcal{E}2}$, respectively, and $\alpha_1$ and $\alpha_2$ are the numbers of ETDs in $D_1$ and $D_2$, respectively.

**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 this phase is $O(n \times log(n))$, where $n$ is the number of nodes in the content tree.

**The Attribute Declaration Change Detection Phase:** Recall that attribute list can be seen as a collection of attributes. The algorithm for detecting the changes to attribute declarations works as follows. Given two ADs, $A_1 \in \mathcal{A}_1$ and $A_2 \in \mathcal{A}_1$, we compare the hash values of these ADs. If $Hash(A_1) = Hash(A_2)$, then $A_1$ is the same as $A_2$ and we mark them to indicate that they have been matched and are not changed. Otherwise, we start to compare the attributes in the attribute list of $A_1$ to the ones in the attribute list of $A_2$. We use the hash values and the attribute name of these attributes. If the hash values of two attributes are the same, then they are not changed. Otherwise, we compare their attribute names. If their names are the same, then we check their attribute types and default values. Observe that if their attribute names are different, then we do not need to compare their attribute types and default values as we do not consider the update of the attribute name for the reasons discussed in Section 3. The cost of detecting the changes to attribute declarations is $O(n \times log(n))$, where $n$ is the number of attributes defined in the DTD.

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

**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. We compare the performance of DTD-Diff with the state-of-the-art approaches. Unfortunately, despite our best efforts (including contacting the authors), we could not get the Java version of XyDiff. Hence, we compared our approach to the Java version of X-Diff [10] (downloaded from http://www.cs.wisc.edu/∼yuanwang/xdiff.html) only. As X-Diff is not designed for detecting the changes on DTDs, we convert the DTDs into XSD [3] using *Syntex dtd2xs* (downloaded from http://www.syntext.com/downloads/) before detecting the changes. Note that the results of X-Diff suffer from the limitations discussed in Section 1. We also study the result quality of DTD-Diff.

**Execution Time vs 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" [5]. The number of attributes of each element is set to "3". We set the percentage of changes to "9%". The characteristic of the data sets used in this set of experiments is depicted in Figure 6(a).

| Code | # Element Types | DTD (DTD-Diff) | | XSD (X-Diff) | |
|---|---|---|---|---|---|
| | | 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 |
| 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

| DTD | # Element Type | # Attribute List |
|---|---|---|
| SigmodRecord | 11 | 1 |
| PSD | 66 | 10 |
| Policy7 | 56 | 26 |
| DBLP | 36 | 12 |
| NewsML_1.1 | 117 | 114 |

(d) Real DTD Characteristics

| Code | Out-degree | DTD (DTD-Diff) | | XSD (X-Diff) | |
|---|---|---|---|---|---|
| | | 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 |

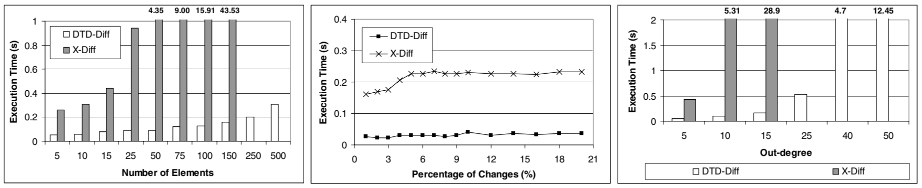
(b) Different Number of Out-degree

| Code | Depth | DTD (DTD-Diff) | | XSD (X-Diff) | |
|---|---|---|---|---|---|
| | | 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. 6.** Data Sets

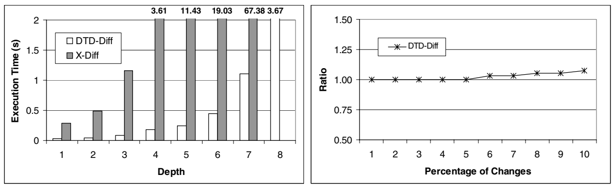
Figure 7(a) depicts the performance of DTD-Diff and X-Diff. We observed that DTD-Diff significantly outperforms X-Diff. DTD-Diff is 5–272 times faster than X-Diff. X-Diff failed to detect the changes when the numbers of elements are more than or equal to 250 due to lack of main memory. The inability of X-Diff to process large number of nodes in XML data is also highlighted in [7]. We now briefly discuss why our approach significantly outperforms X-Diff. First,



(a) Time vs Number of Elements (9%)*

(b) Time vs. Percentage of Changes*

(c) Time vs Out-degree (9%)*



| DTD | DTD-Diff | X-Diff | |
|---|---|---|---|
| | | DTD2 XSD | Detect |
| SigmodRecord | 0.021 | 0.010 | 0.016 |
| PSD | 0.032 | 0.010 | 0.022 |
| Policy7 | 0.031 | 0.010 | 0.021 |
| DBLP | 0.032 | 0.010 | 0.024 |
| NewsML_1.1 | 0.041 | 0.010 | 0.027 |

(f) Execution Time (seconds)

(d) Time vs Depth (9%)*

(e) Result Quality

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

**Fig. 7.** Experimental Results

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. Hence, 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. The details can be found in [8]. 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 6).

We also study the performance of DTD-Diff and X-Diff by using real world DTDs [2, 4]. Figure 6(d) depicts the characteristics of the real world DTDs. We set the percentage of changes to 3%. Figure 7(f) 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 [8].

**Execution Time vs Percentage of Changes:** We use the `E025-B05-D02` data set, whose number of element types, out-degree, and depth are 25, 5, and 2 respectively, as the first version of the DTD. We vary the percentages of changes from "1%" to "20%". Figure 7(b) depicts the execution time of DTD-Diff and

X-Diff for different percentages of changes. We observe that the percentage of changes slightly affect the performance of DTD-Diff and X-Diff.

**Execution Time vs Out Degree:** We set the number of element types and the depth to "25" and "2" respectively. We set the percentage of changes to "9%". We vary the out-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 6(b). Figure 7(c) depicts the performance of DTD-Diff and X-Diff for different numbers of out-degree of each element type. We observe that DTD-Diff is up to 325 times faster than X-Diff. 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.

**Execution Time vs Depth:** We set the number of element types and the out-degree to "25" and "5" respectively. We set the percentage of changes to "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 6(c). Figure 7(d) depicts the performance of DTD-Diff and X-Diff for different depth of each content tree. We observe that DTD-Diff is up to 89 times faster than X-Diff. X-Diff failed to detect the changes when the depth is more than or equal to 8 due to the lack of main memory.

**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%". 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 7(e) 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 and incremental maintenance of relational schema for storing XML data. 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 given data set. DTD-Diff is also able to produce optimal or at least near-optimal deltas.

## References

1. Ronald L. Rivest. The MD5 Message Digest Algorithm. *Internet RFC 1321*, April 1992. `http://www.faqs.org/rfcs/rfc1321.html`.
2. UW XML Repository. *Database Research Group, University of Washington.* `http://www.cs.washington.edu/research/xmldatasets/`.
3. XML Schema. *World Wide Web Consortium.* `http://www.w3.org/XML/Schema`.
4. XML.ORG Registry and Repository for XML Schemas. `http://www.xml.org/xml/registry.jsp`.
5. B. Choi. What are real DTDs like?. *In WebDB*, 2002.
6. G. Cobena, S. Abiteboul, A. Marian. Detecting Changes in XML Documents. *In ICDE*, 2002.
7. E. Leonardi, S. S. Bhowmick. Detecting Changes on XML Documents Using Relational Databases: A Schema-Conscious Approach. *In ACM CIKM*, 2005.
8. E. Leonardi, T. T. Hoai, S. S. Bhowmick, S. Madria. DTD-Diff: A Change Detection Algorithm for DTDs. *Technical Report, Center for Advanced Information System, Nanyang Technological University*, Singapore, 2005.
9. 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.
10. Y. Wang, D. J. DeWitt, J. Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. *In ICDE*, 2003.