A Classification Scheme for Semantic and Schematic Heterogeneities in XML Data Sources

Charnyote Pluempitiwiriyawej and Joachim Hammer
Department of Computer & Information Science & Engineering
University of Florida
Gainesville, FL 32611-6120
{cp0, jhammer}@cise.ufl.edu

The integration of distributed, heterogeneous information sources has been the topic of intense investigations for many years. Most of the research in this context has focused on integrating well-structured data stored in relational or object-oriented databases. In recent years, the World Wide Web (Web) has become the single largest, heterogeneous source for almost any kind of online information. The Extensible Markup Language (XML) is well suited to represent and share this Web data which is mostly semistructured in nature. Using XML, data is stored in an XML document whose structure is defined by one or more Document Type Definitions (DTD). Despite XML's support for the sharing and exchange of information, the integration of multiple, heterogeneous XML documents remains a challenge due to semantic and schematic discrepancies among the data.

In this report, we propose a directed, labeled graph model to describe the internal structure of XML documents and their associated DTDs. Based on our model, we introduce a classification of the possible conflicts that can occur between data that is represented in XML. We believe that such a classification is a useful and important guideline for resolving conflicts that occur when sharing data represented in XML.

1 Introduction

During the last decade, the World Wide Web (Web) has become the single, most widely used repository for online information of any kind. Since HTML is limited to describing how data should be formatted, (e.g. when displayed in a Web browser), the eXtensible Markup Language (XML) was introduced as a way to describe the data content in order to enhance the usability of Web data and to facilitate its sharing and exchange. Most of the data on the Web is semistructured or self-describing (Abiteboul et al. 2000), meaning that its structure may vary and that there is no fixed schema into which the data must be fitted.

However, interesting and useful information is not limited to the Web, but can also be found in conventional databases (i.e., relational and object-oriented databases) and other online stores (e.g., digital libraries). Integrating all these data sources is a challenging problem since data is heterogeneous and distributed. As a result, there is also a growing need for a data model that is both flexible and expressive enough to represent the integrated data that may originally be represented in a wide variety of data models, each with their own capabilities and constructs.

XML is well suited as a data model not only for representing semistructured data but also as a data model for representing the integrated data in sharing architectures (Chawathe et al. 1994; Hammer 1999). However, despite XML's support for data exchange and sharing (see Sec. 2), integrating data sources remains challenging due to the large number of semantic and syntactic conflicts that may exist between related data from different data sources. Although data integration problems have been investigated for several decades (Chawathe et al. 1994; Genesereth et al. 1997; Hammer et al. 1995; Levy 1998; Zhou et al. 1995), most work in this area has focused on integrating structured sources based on the relational or object-oriented data model. On the other hand, integrating semistructured data is a relatively new problem with its own challenges and problems.

Our main goal in this paper is to classify the most common conflicts that occur when integrating semistructured data from multiple sources containing related data. Although we assume that the data in the sources is represented in XML, our classification scheme can be adapted to other semistructured data models that use a graph-structure for representing data. We believe this classification can serve as a valuable resource for developing transformations that may be used by wrappers (Tork Roth et al. 1997) and mediators (Bressan et al. 1997; Cluet et al. 1998; Ludascher et al. 1999; Papakonstantinou et al. 1996).

The organization of this paper is as follows: In Section 2, we present an overview of XML and its support for information integration; we also propose a new, graph-like model for representing DTDs and XML documents (DOCs) internally. Related research is summarized in Section 3. Section 4 contains the details of the classification of conflicts that result when integrating XML-based information sources. Section 5 concludes the paper and describes how the classification is used in our integration system prototype.

2 Extensible Markup Language (XML)

XML was introduced as an exchange format for data that needs to be passed between data sources. For example, it can capture and represent more of the semantics than HTML. Therefore, XML is likely to become the *de facto* standard for Electronic Data Interchange (EDI) on the Web. XML is well suited not only for representing semistructured data but also for representing the integrated data in sharing architectures. XML documents can be integrated into a large (distributed) database by linking them through Uniform Resource Identifier (URI).

XML describes a class of data objects called XML documents (1998c). Data object instances are declared in an XML document (DOC) file. The structure of a document is defined by one or more optional document type definitions (DTD). A DTD can be either included in the DOC file or stored in a separate DTD file which is referenced by the corresponding DOC file. If both the data objects and the DTD information is stored in the same file, the DTD part must be declared in the beginning of the document (i.e., in the prolog portion).

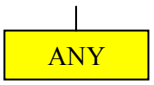
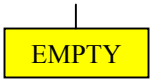
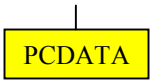
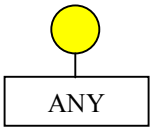
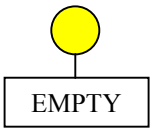
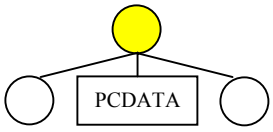
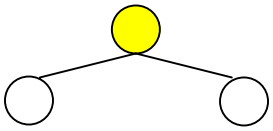
An XML document is *well-formed* if it meets the requirements outlined in (1998c; Harold 1998; Harold 1999). One way of verifying a well-formed XML document is by using a Web browser, for example, Internet Explorer 5.0. A document is *valid* if it is well-formed and associated with at least one DTD describing its structure. In this paper, we classify the conflicts that may occur in a well-formed or valid XML document. Each DOC contains one or more *elements* representing data object instances, each of which may have an associated set of *attributes*. An element in a DOC file is delimited by a start-tag and an end-tag or by an empty-element tag. Each tag has a label that specifies the *type* of the element (1998c).

An element may contain one or more child elements. Child elements form the element *content* of the parent element. In XML, the ordering of child elements is important. In general, each element has a *type* that is specified by its name (1998c), and has a *content type* that is specified by its content. An element is *primitive* if its content type is "primitive" (i.e., it has no child element in its structural definition). Otherwise, it is non-primitive. Each primitive element has one of the following three types: EMPTY, ANY, or PCDATA. Each non-primitive element has a type that is specified by its name, and has one of the following four content types: EMPTY, ANY, MIXED and PURE_ELEMENT_LIST.

A *pcdata* element is a primitive element representing real world data contents. Its type is #PCDATA and its content type is primitive. An element is *empty* if it is non-primitive and its content type is EMPTY (or it must be represented either by a start-tag followed immediately by an end-tag or by an empty-element tag). An *any* element is a non-primitive element whose content can consists of any element (i.e., its content type is is ANY). A *mixed* element is a non-primitive element whose content consists of at least one pcdata element (i.e., its content type is MIXED). A *complex* element is

a non-primitive element whose content consists of at least one element each of which is not a pcdata element (i.e., its content type is PURE_ELEMENT_LIST. Table 1 provides a summary of classifying an element based on its type and content type. In the last column, the block and circle shape indicate primitive and non-primitive elements, respectively. The elements that is being considered are shaded.

Table 1: Summary of types and content types of elements

Element Class	Name	Type	Content Type	Example
Primitive	-	ANY	primitive	
	-	EMPTY	primitive	
	pcdata	PCDATA	primitive	
Non-primitive	any	specified by tag label	ANY	
	empty	specified by tag label	EMPTY	
	mixed	specified by tag label	MIXED	
	complex	specified by tag label	PURE_ELEMENT_LIST	

A set of attributes illustrates properties that are associated with the element. Each attribute is declared in a DOC file as a <name, value> pair. Unlike elements, which are ordered, the order of attributes does not matter. Attribute name, type, constraint, and default value, as well as the type and content type of each element can be defined separately in the DTD. More details on declaring elements and attribute lists is given in (1998c).

We will use the notation `element1.element2.element3` to denote the fact that `element3` is a child of `element2` which is a child of `element1`. In addition, the notation `element1@attribute1` denotes the fact that `attribute1` is an attribute of `element1`. We will formalize the structural representations underlying the DTD and DOC respectively in the next sections.

2.1 *DTD Directed Labeled Graph*

A DTD is stored in text format and can be parsed by a DTD parser into an internal graph structure. Internally, this graph structure is viewed as a directed labeled graph (G_{DTD}) containing an element graph (G_E)[1] and a forest of attribute trees (F_A), one attribute tree for each element.

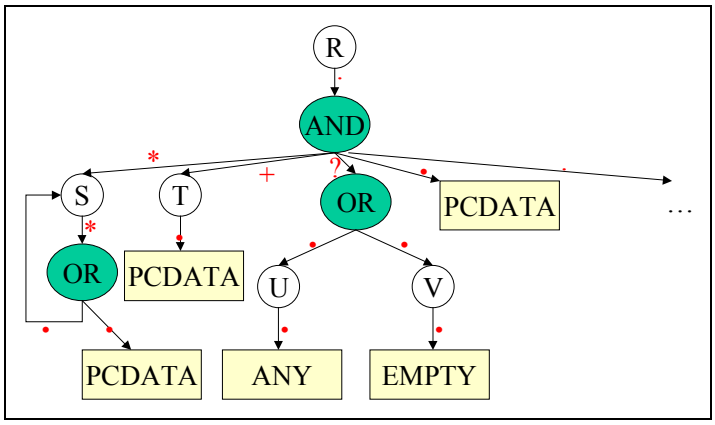


Figure 1: A sample element graph of a DTD graph.

Figure 1 shows a sample element graph (G_E). The element graph has the following properties:

- The graph consists of a set of nodes and a set of edges.
- The edge set consists of directed, labeled edges indicating the type of constraint which applies to the node pointed to by the edge ("*", "+", "?", or "."): "*" means zero-or-more, "+" means one-or-more, "?" means zero-or-one, and "." means exactly one node. By default, we assume that each node appears exactly one (i.e., the "." constraint).
- An edge can be either a forward-pointing edge (going from an ancestor to a successor of a subgraph) or a backward-pointing edge (going from a successor to an ancestor of a subgraph). In Figure 1, the edge from node "OR" to node "S" is a backward-pointing edge. All other edges are forward-pointing edges.
- The node set includes regular nodes, two types of special internal or *operator* nodes, (labeled "OR" and "AND", respectively), and three types of special leave nodes (labeled "EMPTY", "PCDATA", and "ANY", respectively). In Figure 1, the element graph contains five regular nodes (i.e., nodes labeled "R", "S", "T", "U" and "V"), three operator nodes (i.e., two "OR" nodes and one "AND" node), and five leave nodes (i.e., three "PCDATA" nodes, one "ANY" node, and one "EMPTY" node).
- Each regular node represents an element object class.
- Each regular node is assigned an identifier, *j*, and may have an associated attribute tree (a_j).
- Each regular node has a label describing the element type that corresponds to a tag label used in the DOC to delimit the scope of an element object instance.
- An operator node behaves as an articulation point between its parent and its children. A subgraph rooted at an OR (AND) node is called an OR (AND) subgraph.
- An AND subgraph indicates that the content of the parent object class is formed by *all* children of the AND node. If the graph has no articulation point, the AND node is assumed.

1 One may define an element graph as a set of finite state automata.

In other words, if two adjacent levels of regular nodes have no operator node, the AND relationship is implied.

- An OR subgraph indicates that the content of the parent object class is formed by *one or more* children of the OR node.
- The EMPTY, PCDATA and ANY leave nodes define the type of the object classes. Those nodes will be replaced in the DOC graph by an empty string, a non-empty string, or an object instance.

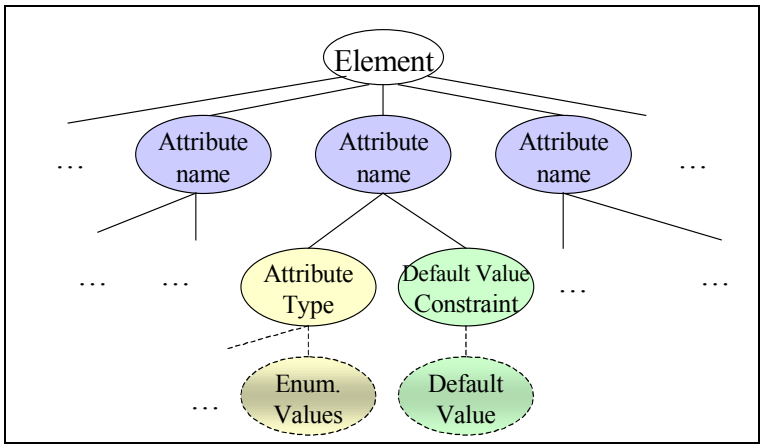


Figure 2: A 4-level attribute tree of a DTD graph.

Figure 2 depicts a 4-level attribute tree (as part of the DTD graph). Level 1 is called the root level. The root node is the element node to which the attribute tree is attached. The children of the root are the attribute-name nodes.

Level 2 is the name level. Each *attribute-name* node (or *name* node for short) has a label indicating the attribute name and exactly two children: a left child indicating the attribute type and a right child containing the default-value constraint (see below).

At level 3 each *attribute-type* node (or *type* node for short) describes the possible attribute types. The type can be one of ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS, CDATA, ENUMERATION, NOTATION and ENUM-NOTATION. The *default-value-constraint* node can be one of: #REQUIRED, #IMPLIED, #FIXED, or UNFIXED-DEFAULT-VALUE. The "#REQUIRED" keyword indicates that the attribute has no default value and that a value *must* be declared in the DOC (i.e., the declaration of attribute name and its value is required). The "#IMPLIED" keyword indicates that the attribute has no default value but *may* not be declared in the DOC (i.e., the declaration of attribute name and its value is optional). The "#FIXED" keyword indicates that the attribute always has a default value which can not be modified. Finally, the "UNFIXED-DEFAULT-VALUE" keyword indicates that the attribute has a default value which may be modified.

Level 4 contains the default values. If the type node in level 3 is labeled ENUMERATION or ENUM-NOTATION, its children must represent all possible attribute values. If the default-value-keyword node in level 3 is labeled #REQUIRED or #IMPLIED, it will have no child. Otherwise, it must have exactly one child storing the default value.

2.2 *DOC Directed Labeled Graph*

In this section, we present a formal definition for the XML-DOC structural model. DOCs are written in text format and can be parsed by a DOC parser to generate the underlying DOC graphs. After

parsing a DOC, the DOC structure is viewed as a directed labeled graph (G_{DOC}) containing an element tree (T_E) and a set of attribute graphs (G_A) each of which is associated with the element.

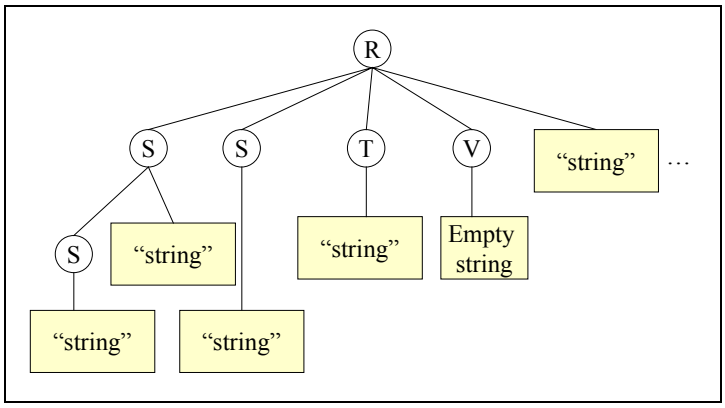


Figure 3: A sample element tree of a DOC graph.

Figure 3 shows a sample element tree as part of the DOC graph. The tree has the following characteristics:

- The root represents the *document* element delimited by the outer most open- and end-tag in the DOC.
- The tree contains a set of subtrees representing sub-elements.
- The label of the subtree root node indicates the type of the element.
- Each node may have an associated attribute (directed labeled) graph.
- Unlike the element graph in the DTD graph, the element tree does *not* contain special internal nodes (AND and OR nodes), or special leave nodes (PCDATA, EMPTY, and ANY). In addition, edges have no labels and backward edges are not allowed.
- Leave nodes contain either data values (by replacing the PCDATA or ANY node in the corresponding DTD graph) or the null string (by replacing the EMPTY or ANY node in the corresponding DTD graph).

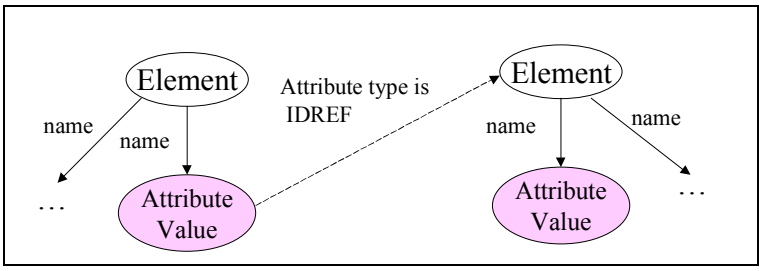


Figure 4: An attribute graph of a DOC graph.

Figure 4 depicts a sample attribute graph consisting of a set of nodes and a set of edges. There are two types of nodes: element and value nodes. Each element node represents the element that owns the attributes, and value nodes represent attribute values. In addition, each edge has a label indicating the attribute name. The edges going from the element nodes to the value nodes are called normal edges, while edges going from the value nodes to the element nodes are called crosslink edges. It is important to note that a crosslink edge can appear only if the attribute is of type IDREF.

2.3 *Relationship to Existing Representations for XML*

We have proposed a graph model for both XML DTDs and DOCs. Our model is very closely related to the Document Object Model (DOM) (1998b) which represents data in an XML DOC. Unlike our graph model, DOM does not represent DTDs. The World Wide Web Consortium (W3C) provides a set of APIs to manipulate the DOM tree. The implementation of DOM is the responsibility of an XML DOC parser.

In DOM, everything is a `Node`. In other words, a DOM tree contains a hierarchy of `Node` objects. A `Node` may have child nodes. If a `Node` has no children, it is a leave. Node types can be one of the following: `Document`, `DocumentFragment`, `DocumentType`, `EntityReference`, `Element`, `Attribute`, `ProcessingInstruction`, `Comment`, `Text`, `CDATASection`, `Entity`, or `Notation`. In our model, we only consider the `Element`, `Attribute` and `Text` types of a `Node`, since they form a minimal set for representing data items.

In addition to DOM, the W3C provides other working drafts of related XML components. For example, the eXtensible Stylesheet Language (XSL) (1998d) provides a framework for transforming XML and expressing stylesheets of a particular XML document. The XML Linking Language (XLink) (2000a) provides a framework for linking among related XML documents, among the XML documents and metadata, and among databases that reside in a location separate from the linked resources. The XML Pointer Language (XPointer) (1999b) provides a framework for addressing the internal structures of XML documents. The XML Namespace (1999a) supports a uniqueness of identifying the scope of elements and attributes in a particular XML documents. It is a collection of names that are used in XML documents as element types and attribute names. XML Schema (2000b; 2000c; 2000d) provides a richer and more specific for defining a database schema than the core XML DTD.

3 Related Research

Widom (Widom 1999) and Levy (Levy 1999) provide a list of research problems on data management for XML, in particular. Those problems are, for example, the understanding of the relationship between XML-DTD and conventional database schemas, and the need of resolving the various conflicts that arise when mixing the concepts of documents and databases. On the query processing point of view, the problem includes the need of query language for XML, and the need of techniques of query processing, planning and optimizing to efficiently handle a mixture of both structured, unstructured and semistructured data. Other problems includes the need of data model for XML information, the need of indexing mechanisms for large stores of XML data, the need of XML view maintenance mechanisms for both virtual and materialized views, the need of an algorithm for large-scale data integration (i.e., integrating a large number of relatively small files), and the need of theoretical results and practical techniques for translating a conceptual model of database into an XML encoding,

An important problem that must be overcome when integrating heterogeneous databases is the resolution of conflicts caused by differences in the underlying data model or representation of the data itself. Conflicts can be at the schema and/or semantic levels. Kashyap and Sheth (Kashyap and Sheth 1995) propose a formal model of similarity between objects in databases and classify conflicts based on such model. Batini and Lenzerini (Batini et al. 1986) analyze conflicts based on ER model and propose a framework for schema integration problems. Kim et al. (Kim et al. 1993) promote a Multidatabase language to resolve schematic conflicts among relational, object-oriented and Multidatabases. Kent (Kent 1991) applies an object-oriented database programming language to resolve domain and schema mismatch conflicts. Lakshmanan (Lakshmanan et al. 1996) and Miller (Miller 1998) introduce SQL-Schema, the language derived from SQL, to resolve schematic discrepancy. Siegel and Madnick (Siegel and Madnick 1991) and Sciore et al. (Sciore et al. 1994) use

metadata approach by attaching context information and resolve semantic conflicts based on the context.

It is generally accepted that a domain specific ontology is needed for data integration and interchange. In the Business-to-Business (B2B) application domain, several organizations (e.g., RosettaNet (RosettaNet 1999), CommerceOne (CommerceOne 1999), SchemaNet (SchemaNet 1999), XML/EDI (XML/EDI 1999), etc.) are working to specify the standard ontology. Farquhar et al. (Fankhauser and Neuhold 1992) introduce a web-based tool for ontology construction and provide an online ontology repository managed by the ontology server (1992). Maluf and Wiederhold (Maluf and Wiederhold 1997) provide the basic concepts of constructing ontology using knowledge representations and interoperation approach.

A formal XML specification can be found in (1998c). Harold (Harold 1998; Harold 1999) provides details of constructing XML documents and DTDs. Bosak (Bosak 1997) discusses the impact of XML on Java-based Web applications.

4 Semantic and Schematic Heterogeneities in XML-based Information Sources

Earlier research has attempted to resolve the conflicts that arise among data represented using the entity-relationship, relational, and object-oriented data models, which describe structured data and their schema. Unlike well-structured data, however, semistructured data has irregular, implicit and partial structure. The distinction between schema and the data is blurred (Abiteboul 1997). Therefore, conflicts due to integration of semistructured data are different from those caused by the integration of structured data. The goal of this paper is to describe a classification of conflicts that result when integrating semistructured data represented in XML.

4.1 *Integrating XML Data*

For the remainder of this paper, we assume the following set-up: The underlying, semistructured data sources which provide the heterogeneous data for our integration scenario are represented as a set of XML DOCs. Associated with each source DOC is a *source* DTD, which represents the schema information for the data source. There is also one so-called *target* DTD to describe the desired schema of the integrated data which will be stored in the integrated target document (i.e., target DOC). As described in the previous sections, we model DTDs and DOCs as graphs. Using such graphs, we classify conflicts into three main classes: *structural, domain,* and *data* conflicts.

Structural conflicts arise when the schema of the sources representing related or overlapping data exhibit discrepancies. Structural conflicts can be detected when comparing the underlying DTDs. The class of structural conflicts includes generalization conflicts, aggregation conflicts, internal path discrepancy, missing items, element ordering, constraint and type mismatch, and naming conflicts between the element types and attribute names.

Domain conflicts arise when the semantic of the data sources that will be integrated exhibit discrepancies. Domain conflicts can be detected by looking at the information contained in the DTDs and using knowledge about the underlying data domains. The class of domain conflicts includes schematic discrepancy, scale or unit, precision, and data representation conflicts.

Data conflicts refer to discrepancies among similar or related data values across multiple sources. Data conflicts can only be detected by comparing the underlying DOCs. The class of data conflicts includes ID-value, missing data, incorrect spelling, and naming conflicts between the element contents and the attribute values.

To integrate two related DOCs, we start by comparing their respective source DTDs with the target DTD in order to detect structural and domain conflicts. Note, for simplicity, we limit our discussions to integrating only two data sources. However, this approach can be extended to integrating three or more DOCs in a straightforward manner. Next, we try to resolve the detected

conflicts. A description of how to resolve each type of conflict is provided in the following sections. As a result of this conflict detection and resolution step, each source DOC will be transformed into an "aligned" source DOC. Finally, the aligned source DOCs will be merged into the target DOC. Although each aligned source DOC has the same structure as defined by the target DTD, data conflicts still remain. Merging source DOCs is necessary to detect and resolve such data conflicts.

Table 2: Overview of the different types of conflicts and their classification.

Conflicts Classes	Categories	Subcategories	XML context	Section
Structural	Naming	Case Sensitivity	EE, AA, EA, AE	4.3.1.1
		Synonyms	EE, AA, EA, AE	4.3.1.2
		Acronyms	EE, AA, EA, AE	4.3.1.3
		Homonyms	EE, AA, EA, AE	4.3.1.4
	Generalization/Specialization		EE, AA, EA, AE	4.3.2
	Aggregation	Intra-aggregation	EE, AA, EA, AE	4.3.3.1
		Inter-aggregation	EE, AA, EA, AE	4.3.3.2
	Internal Path Discrepancy		EE, AA, EA, AE	4.3.3.2
	Missing Item	Content Discrepancy	EE	4.3.5.1
		Attribute List Discrepancy	AA	4.3.5.2
		Missing Attribute	EA	4.3.5.3
		Missing Content	AE	4.3.5.4
	Element Ordering		EE	4.3.6
	Constraint Mismatch		EE, AA, EA, AE	4.3.7
	Type Mismatch		EE, AA, EA, AE	4.3.8
Domain	Schematic Discrepancy	Element-value-to-Element-label Mapping	EE	4.4.1.1
		Attribute-value-to-Element-label Mapping	AE	4.4.1.2
		Element-value-to-Attribute-label Mapping	EA	4.4.1.3
		Attribute-value-to-Attribute-label Mapping	AA	4.4.1.4
	Scale or Unit		EE, AA, EA, AE	4.4.2
	Precision		EE, AA, EA, AE	4.4.3
	Data Representation	Primitive Data Type	-	4.4.4.1
		Data Format	-	4.4.4.2
Data	Naming	Case Sensitivity	-	4.5.1.1
		Synonyms	-	4.5.1.2
		Acronyms	-	4.5.1.3
		Homonyms	-	4.5.1.4
	ID-value		-	4.5.2
	Missing Data		-	4.5.3
	Incorrect Spelling		-	4.5.4

4.2 *Conflict Classification Scheme*

Table 2 illustrates the classes and categories of different conflicts that can occur between the elements and attribute constructs in two different XML documents and their underlying DTDs (referred to as

source and target). Each category is further divided into subcategories depending on the XML context in which this type of conflict can occur. For example, the *Element-Element* (EE) subcategory includes conflicts that can occur between two elements from different DTDs/DOCs. The *Attribute-Attribute* (AA) subcategory contains conflicts between two attributes of a particular element declared different DTDs/DOCs. The *Element-Attribute* (EA) and A*ttribute-Element* (AE) subcategories comprise conflicts that occur when an element (attribute) in one DTD/DOC is mapped to an attribute (element) in another DTD/DOC. In most cases, EA and AE conflicts are analogous and hence we omit AE conflicts from now on to keep the discussions shorter.

4.3 *Structural Conflicts*

Structural conflicts arise when the same concept is represented differently in two DTDs. As discussed in Sec. 2, since a DTD is represented as a hierarchical graph, concepts are subgraphs in the DTD graph. Hence, structural conflicts occur when the same concept is represented (structurally) different subgraph in two DTDs. In the next subsections, we describe the different types of structural conflicts in details.

4.3.1 Naming

Naming conflicts are conflicts among names of attribute or element type defined in different DTDs. These conflicts arise when comparing both element and attribute *nodes* across DTDs. The relationship between two nodes is either equivalent (e.g., synonym and acronym) or incompatible (e.g., homonym) (Kashyap and Sheth 1995).

4.3.1.1 Case Sensitivity

In some languages (e.g., HTML, Pascal), text case is irrelevant. In other languages (e.g., XML, C, Java), text case is significant. Since terms used in databases may be defined differently, the case of the text can provide an important clue as to whether terms are related or not (e.g., when deciding whether `Windows` and `windows` are equivalent). In the next sections, case sensitivity is assumed. To resolve conflicts involving text from sources with different case sensitivity, the fact whether case sensitivity matters or not must be specified explicitly.

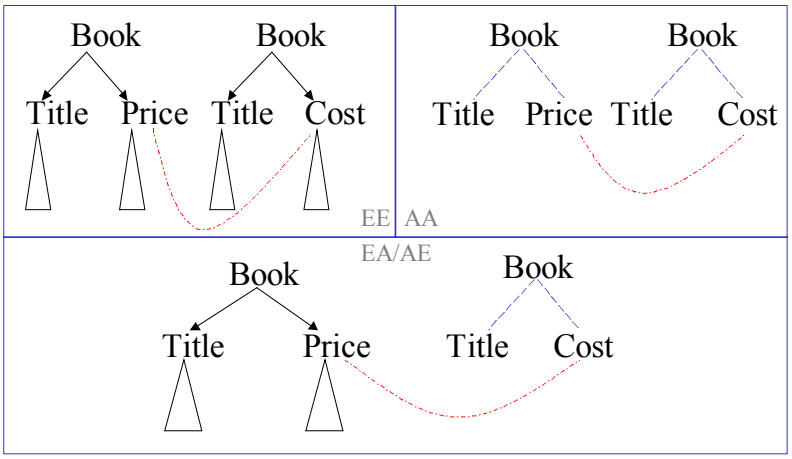


Figure 5: Structural conflicts of type synonyms.

4.3.1.2 Synonyms

When two different terms refer to the same real-world object or concept, they are also known as *synonyms*. A synonym relationship can occur between two elements (EE), two attributes (AA), and

between an element and an attribute (EA/AE) depending on how terms are defined in DTDs. This is displayed in Figure 5 in the top-left, top-right and bottom blocks, respectively. Each block in the figure contains two DTD graphs. The left (right) graph shows the structure of the source (target) DTD. In each DTD graph, an element connects to its contents (i.e., subelements) by solid lines, and to its attributes by long dashed (blue) lines. The dashed dotted (red) lines from nodes in the source DTD to nodes in the target DTD depict the mappings. In the rest of this paper, we will use those semantics to illustrate the graphs in every figure.

In Figure 5, `Price` and `Cost` of `Book` as shown in the three quadrants have the same meaning (`Price` and `Cost` are defined as an element type or attribute in different DTDs). One way of resolving the synonym conflicts is to look up alternative definitions in dictionary or request additional information from the user and generate mappings from unknown terms to the known terms. User input may also be needed to verify the mappings.

4.3.1.3 Acronyms

Parts of a series of words can form a new term. Such a new term is called *acronym* and the series of words is called the *full string* of the acronym. Like the synonym relationship, an acronym relationship can occur between two elements (EE), two attributes (AA), and between an element and an attribute (EA/AE) depending on how terms are defined in DTDs.

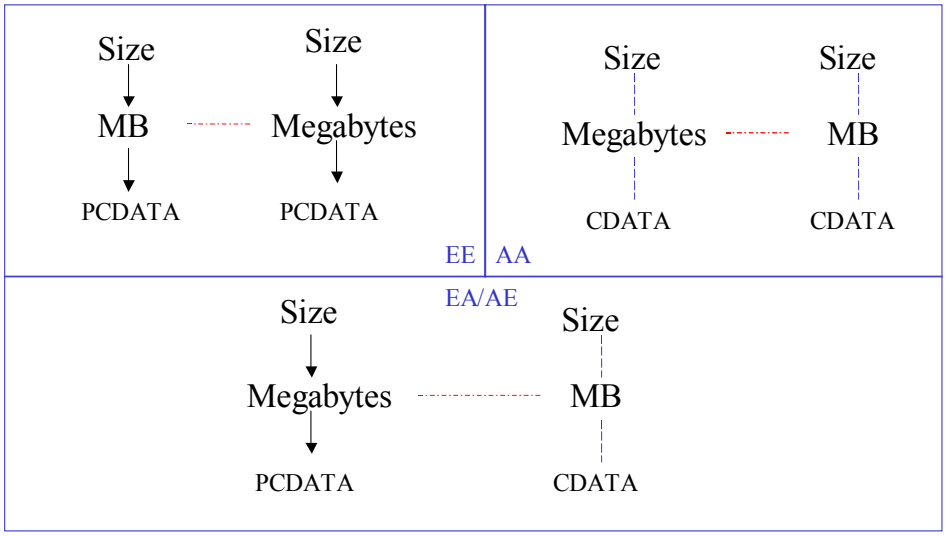


Figure 6: Structural conflicts of type acronyms.

Figure 6 gives examples of acronyms using different representations. For example, `MB` is an acronym for `Megabytes`. The acronyms can be seen as a special case of synonym (i.e., two different terms refer to the same real-world object or concept, and one is an acronym of the other). To resolve acronym conflicts, the same approach as for resolving synonym conflicts can be applied: using a dictionary to generate mappings from the acronym term to the full string. In addition, user input may be needed to verify the mappings.

4.3.1.4 Homonyms

When terms refer to different real-world objects or concepts, they are also known as *homonyms*. A homonym relationship can occur between two elements (EE), two attributes (AA), and between an element and an attribute (EA/AE) depending on how terms are defined in DTDs.

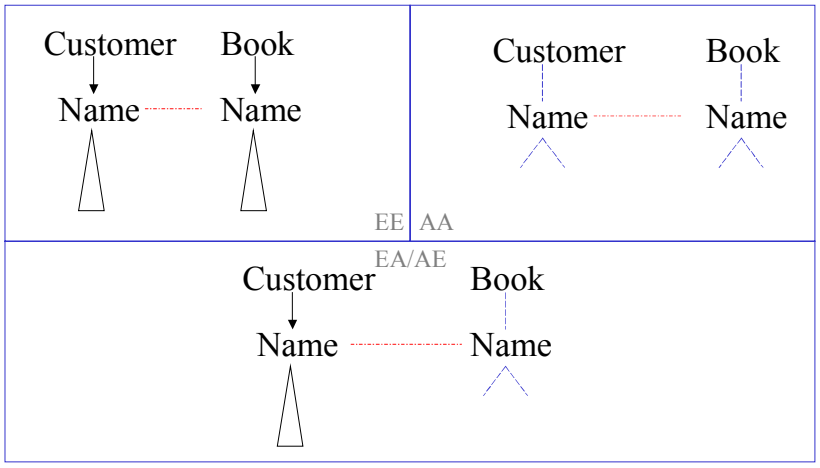


Figure 7: Structural conflicts of type homonyms.

Figure 7 shows homonym conflicts in categories EE, AA, EA and AE. `Name` in the context of `Customer` (i.e., `Customer.Name` or `Customer@Name`) and `Name` in the context of `Book` (i.e., `Book.Name` or `Book@Name`) are homonyms. Recall that we use the notation `element1.element2` to denote the fact that `element2` is a subtype (child) of `element1`, and the notation `element1@attribute1` to denote the fact that `attribute1` is an attribute of `element1`.

Homonym conflicts can be detected by considering the corresponding paths defined in the DTDs that are being matched. These paths lead from the root to the particular term. Homonym conflicts occur when there exists conflicting terms along the paths. To resolve homonym conflicts, for each pair of conflicting terms, we redefine the relationship between those two terms to be mismatched.

4.3.2 Generalization/Specialization

This type of conflict arises when the node in one DTD has a more general (special) meaning than the node in the other DTD. The `ISA` relationship is an example of a generalization/specialization. A generalization/specialization relationship can occur between two elements (EE), two attributes (AA), and between an element and an attribute (EA/AE) depending on how terms are defined in DTDs.

Figure 8 shows examples of generalization conflicts in different scenarios. The left (right) graph within each quadrant shows the structure of the source (target) DTD. In the top left quadrant, consider `CS-book` in the source DTD and `Book` in the target DTD. Both have a subelement `Title`. `Book` which also contains `Type` that specifies the type of the book (e.g., `Computer Science`, `History`, `Math`, etc.). `CS-book` is a kind of `Book` whose type is `Computer Science`. Since this conflict occurs between two elements, it is the EE conflict. Similar conflicts can arise in the AA, AE or EA scenario. For example, in the lower and in the top right quadrants, consider `HomePhone` of a `Customer` in the source DTD and `Phone` of a `Customer` in the target DTD. `HomePhone` is mapped to `Phone` with `ISA` relationship.

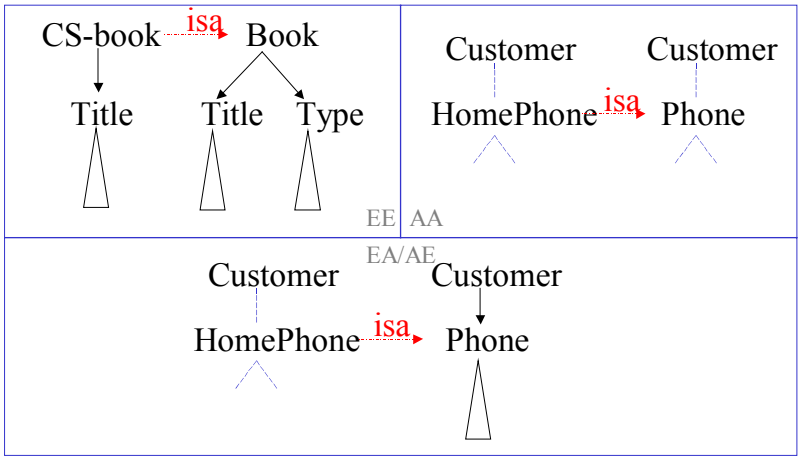


Figure 8: Generalization conflicts.

To resolve generalization concepts, the mappings from (to) the more specific to (from) the more general concepts must be defined (e.g., by looking them up in a dictionary or getting a user input). Note that the generalization is a relationship between two nodes. In other words, the relationship does not inherit to the children of those two nodes. For example, in the top left quadrant, the nodes `CS-book` and `Book` are mapped with `ISA` relationship, but the `title` of the `CS-book` and the `title` of the `Book` are mapped with equivalent relationship.

4.3.3 Aggregation

This type of conflict arises when aggregation is used to combine (or divide) source information to form target information. If the content of the source element or the attribute values of the source element are aggregated (or divided), we refer to this conflict as *intra*-aggregation. If a node in the source DTD is mapped to a node in the target DTD using some aggregation functions (e.g., sum, average, count), we refer to this conflict as *inter*-aggregation. An aggregation relationship can occur between two elements (EE), two attributes (AA), and between an element and an attribute (EA/AE) depending on how terms are defined in DTDs.

4.3.3.1 Intra-aggregation

In Figure 9, each category (e.g., EA, AA, EA and AE) contains two graphs. The left (right) graph shows the structure of the source (target) DTD. The figure shows several intra-aggregation conflicts involving the `Author` element that represents an author of a book. The element contains the name of the author. In the source DTD, the name of the author is partitioned into three parts: `First`, `Middle`, and `Last` referring to first name, middle name, and last name, respectively. In the target DTD, the name of the author is represented as one long string. Hence, the first name, middle name and last name in the source DTD are aggregated to form a long string. Note that the aggregation mapping occurs below the mapped element (e.g., `Author`).

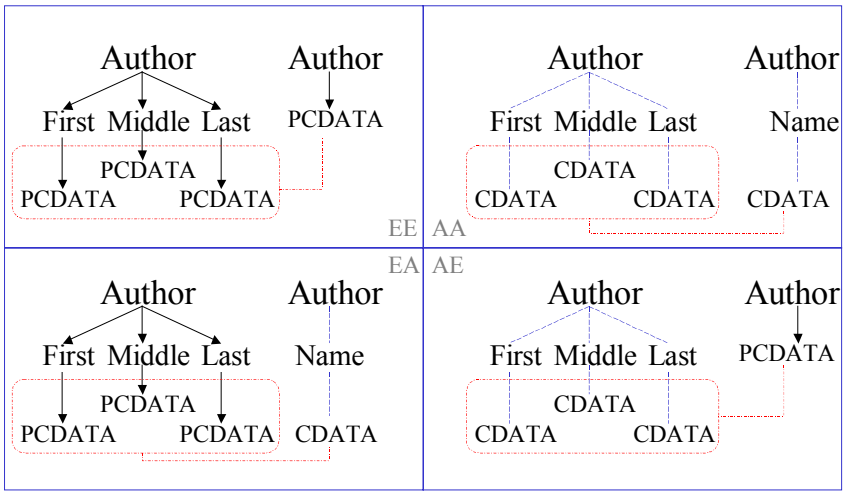


Figure 9: Intra-Aggregation conflicts.

To resolve intra-aggregation conflicts, we need to define a function that combines/divides the content of the mapped element (e.g., `Author`). For example, this `combine` function can simply be a string concatenation function taking three string values of parameters (e.g., `First`, `Middle`, and `Last`) and generating one output string whose value is the concatenation of the three input strings. However, in many cases, the needed combine/divide functions may be considerably more complicated than the simple string concatenation function shown here. Therefore, user-defined functions may be needed.

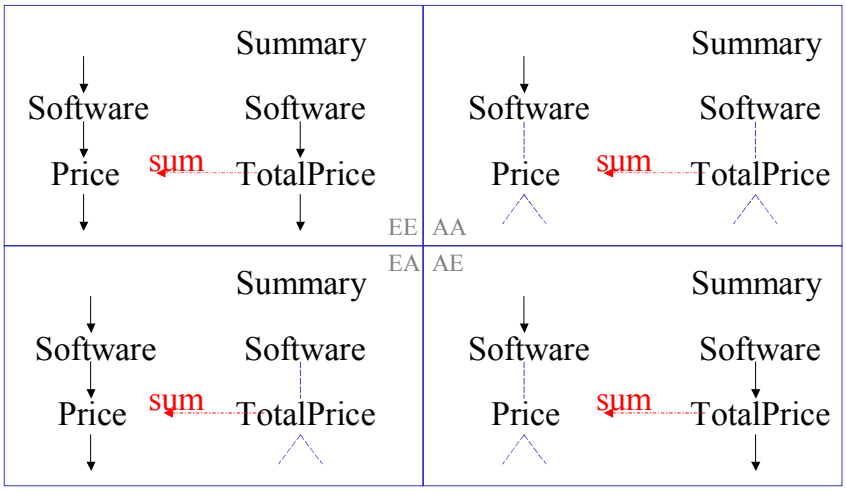


Figure 10: Inter-Aggregation conflicts.

4.3.3.2 Inter-aggregation

In Figure 10, each category (e.g., EA, AA, EA and AE) contains two graphs. The left (right) graph shows the structure of the source (target) DTD. The figure illustrates several inter-aggregation conflicts involving the `Price` (in the source DTD) and `TotalPrice` (in the target DTD) of `Software`. The `sum` function from `Price` to `TotalPrice` indicates that the value of `TotalPrice` in the target DOC is derived from all values of `Price` instances declared in the source

DOC. Note that inter-aggregation conflicts occur across several instances of the same element type (e.g., `Price`) in the source.

To resolve inter-aggregation conflicts, we need to define a function that aggregates the values of element instances declared in the source DOC. The aggregation function can be a simple arithmetic function (e.g., `sum`, `average`, `count`) or a user-defined function. Aggregation functions are used to specify the relationships between two elements, two attributes, or element and attribute.

4.3.4 Internal Path Discrepancy

So far we have described conflicts between related nodes in different structures. Internal path discrepancies arise when two paths to the same element only match partially or not at all. This includes paths that contain backward edges.

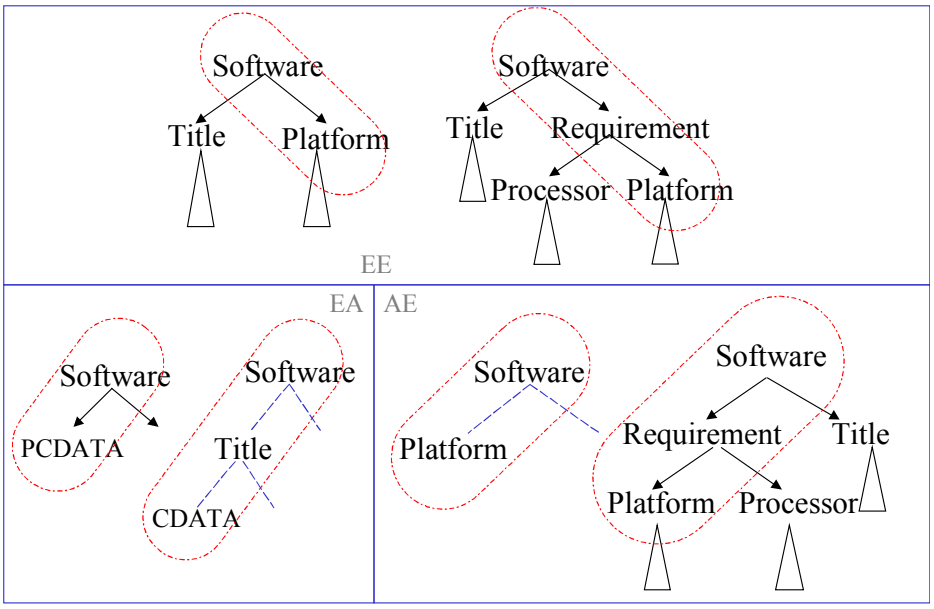


Figure 11: Internal Path Discrepancy conflicts.

Figure 11 depicts internal path discrepancies for the EE, EA and AE scenarios. Each scenario contains two graphs. The left (right) graph shows the structure of the source (target) DTD. Notice that there is no internal path discrepancy involving the AA scenario since the structure of attributes of a particular element is flat. In other words, XML does not allow complex attributes. Path conflicts involving EA and AE are possible since an element can be either simple or complex. Each attribute tree has a fixed height while the element graph has variable height.

Path conflicts involving EE arise when the source and target paths of elements have different lengths and are partially overlapping. As shown in the EE scenario in Figure 11, the target path, `Software.Requirement.Platform`, contains an extra node, `Requirement`, with respect to the source path, `Software.Platform`. Likewise, in the EA and AE scenarios, the source has a shorter path than the target. The opposite namely that the source has a longer path than the target is also possible. In addition, the difference in the path lengths can be more than one. Both paths may also contain backward edges.

Internal path discrepancies can be detected by comparing the source and the target paths from the root to the nodes that are being considered. If the paths contain a backward edge, we do not repeatedly count the subpath that forms the cycle. By comparing the source and the target paths, if there is an extra node in the target (source) path, the insertion (deletion) of the extra node need to be performed at the source path (because we are mapping from the source to the target paths.)

4.3.5 Missing Item

This type of conflict arises when the same element in two DTDs has a different definition and one or more items (subelements or attributes) of the element are missing in one document but not the other. We partition conflicts in this category into four subcategories based on how to represent the item: Content discrepancy (EE), attribute list discrepancy (AA), missing attribute (EA), and missing content (AE).

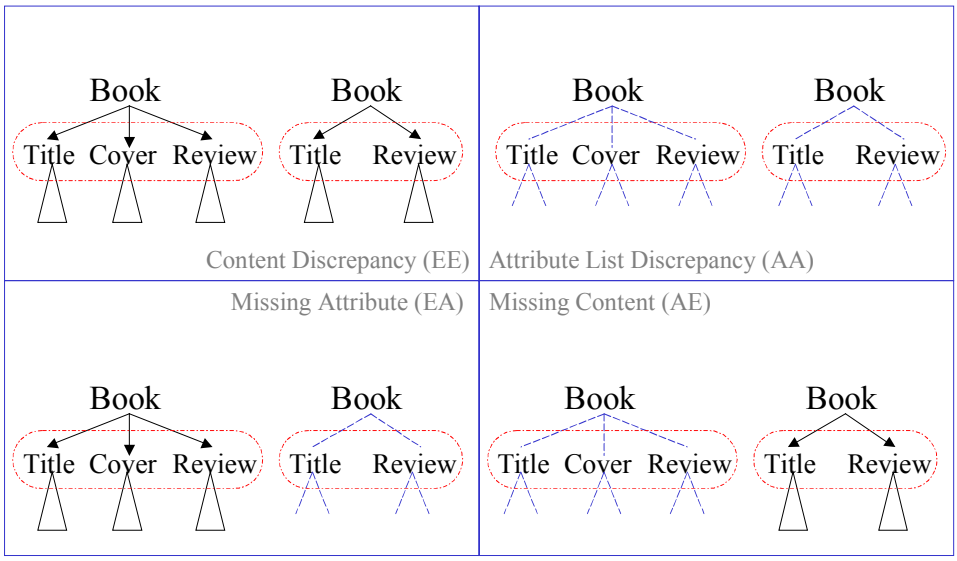


Figure 12: Missing Item conflicts.

Figure 12 illustrates the missing item conflict in the EA, AA, EA and AE categories. Each category contains two graphs. The left (right) graph shows the structure of the source (target) DTD.

4.3.5.1 Content Discrepancy (EE)

Different element definitions in two DTDs exhibit a discrepancy with respect to their content if the content of the element in the source or target DTD is missing. In Figure 12, `Book` contains `Title`, `Cover`, and `Review` in the source DTD and contains `Title` and `Review` in the target DTD. The element `Cover` in the target DTD is missing.

4.3.5.2 Attribute List Discrepancy (AA)

Different element definitions in two DTDs exhibit a discrepancy with respect to the underlying attributes if some of the attributes of the element in the source or target DTD are missing. In Figure 12, `Book` has attribute `Title`, `Cover`, and `Review` in the source DTD and has attributes `Title` and `Review` in the target DTD. The attribute `Cover` in the target DTD is missing.

4.3.5.3 Missing Attribute (EA)

This type of conflict arises when (1) the definition of a particular element in the target DTD contains a set of attributes, (2) the definition of the corresponding element in the source DTD contains a subelement structure, and (3) some of the attributes of the element in the target DTD is missing with respect to the subelement structure in the source DTD. In Figure 12, `Book` contains `Title`, `Cover`, and `Review` in the source DTD but only has attributes `Title` and `Review` in the target DTD. The attribute `Cover` in the target DTD is missing.

4.3.5.4 Missing Content (AE)

This type of conflict arises when (1) the definition of a particular element in the target DTD contains a subelement structure, (2) the definition of the corresponding element in the source DTD contains a set of attributes, and (3) some of the content of the element in the target DTD is missing with respect to the set of attributes of the corresponding element in the source DTD. In Figure 12, `Book` has attributes `Title`, `Cover`, and `Review` in the source DTD but only contains `Title` and `Review` in the target DTD. The element `Cover` in the target DTD is missing.

The missing item conflicts can be detected by comparing subelements and/or attribute sets of a particular element. Note that this type of conflict can occur together with other types of conflicts (e.g., synonyms, acronyms, etc.); hence the resolution of missing item conflicts may include resolutions of other types of conflicts. When a missing item is detected, the source structure can be transformed to the target structure by insertion and/or deletion of some of the content and attributes of the element involved.

4.3.6 Element Ordering

In XML, the order within a sequence of subelements is significant. Element ordering conflicts arise when the order of subelements for two elements in different DTDs is different. In Figure 13, the left (right) graph shows the structure of the source (target) DTD. In the source DTD, `Book` contains `Title` followed by `ISBN`. In the target DTD, `Book` consists of `ISBN` followed by `Title`. The sequence of subelements of `Book` is different in source and target DTDs.

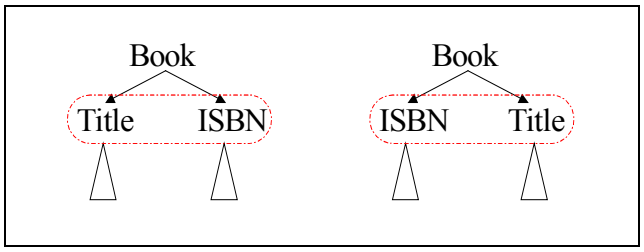


Figure 13: Element Ordering conflict.

Since the content of a particular element in the source is a sequence of subelements, such sequence of subelements must be reordered to form the content of the element in the target if the orders do not agree. Element ordering conflicts can be resolved by a `reorder` function that takes a sequence of subelements in the source and the target DTDs as inputs. The function must map index numbers of each element in the source sequence to those in the target sequence.

4.3.7 Constraint Mismatch

In XML, it is possible to express constraints on content and attributes in a DTD. The constraints for contents are zero-or-more (`*`), one-or-more (`+`), zero-or-one (`?`), and exactly-one (`.`). Possible constraints for attributes are `#REQUIRE`, `#IMPLIED`, `#FIXED` and `UNFIXED-DEFAULT-VALUE`. Constraint mismatch conflicts arise when the constraints for the same contents or attributes in different DOCs are different.

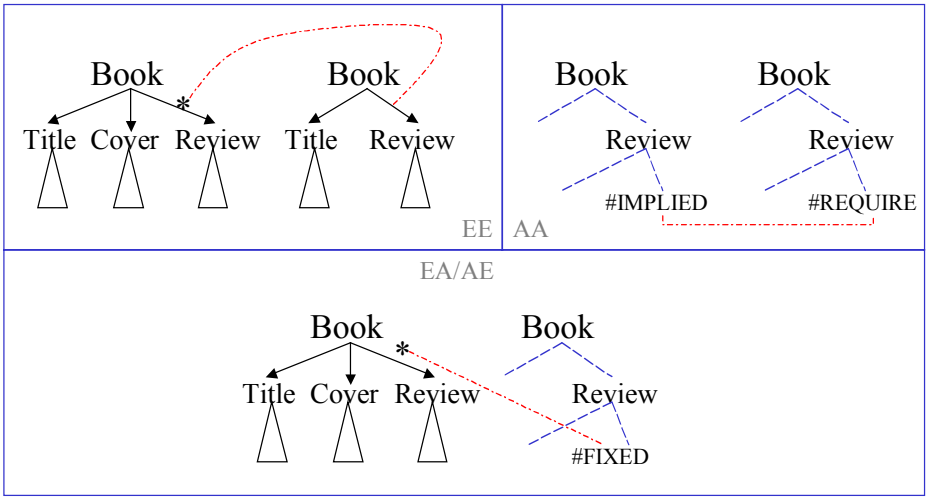


Figure 14: Constraint Mismatch conflicts.

Figure 14 illustrates the constraint mismatch conflicts in the EA, AA, EA and AE scenarios, as well as a set of mappings between content constraints and attribute constraints at the bottom right portion of the figure. Each scenario contains two graphs. The left (right) graph shows the structure of the source (target) DTD. In the EE scenario, `Book` is defined to have zero-or-more `Review` subelement in the source DTD, and exactly one `Review` subelement in the target DTD. In the AA scenario, in the source DTD, the constraint for attribute `Review` is `#IMPLIED` whereas in the target DTD it is `#REQUIRED`. In the EA/AE scenario, in the source, `Review` represents the content of `Book` with zero-or-more constraints, while it is represented as an attribute with constraint `#FIXED` in the target DTD. The corresponding pairs of constraints in each scenario are incompatible.

Notice that the most relaxed type of constraint for data content is "zero-or-more", followed by "one-or-more", "zero-or-one," and "exactly-one", respectively. For attribute constraints, the most relaxed constraint is `#IMPLIED` followed by `#REQUIRED`, `UNFIXED-DEFAULT-VALUE` and `#FIXED` respectively. In general, we can map a stronger constraint to a weaker constraint. Appendix A provides rule templates of mappings among attribute constraints and content constraints

4.3.8 Type Mismatch

These conflicts arise when the element or attribute types of related items in source and target DTD are different. Recall from Section 2 that (a) each element has a *type* that is specified by its name (1998c), and has a *content type* that is specified by its content (b) elements are categorized into two classes: primitive and non-primitive. For conflicts involving elements, in previous categories, we have considered the conflicts that occur between non-primitive elements (e.g., naming, internal path discrepancies, and element ordering conflicts) or a group of primitive elements (e.g., aggregation conflicts). In this category (i.e., type mismatch conflict), we are considering the conflict that occurs between types of two individual primitive elements whose parent elements are mapped under a relation. Those types are PCDATA, EMPTY and ANY. The primitive elements declared in a DTD will be replaced by instances declared in a corresponding DOC. In other words, a primitive element of type PCDATA declared in a DTD will be replaced by a non-null string in the corresponding DOC; a primitive element of type EMPTY declared in a DTD will be replaced by the null string in the corresponding DOC; and a primitive element of type ANY will be replaced by the non-null string, the null string, or a set of subelements in the corresponding DOC.

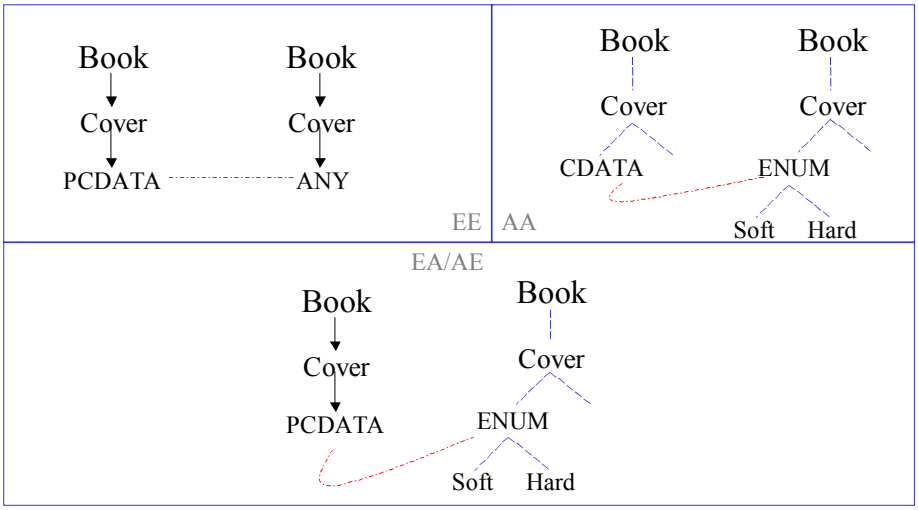


Figure 15: Type Mismatch conflicts.

For conflicts involving attribute, attribute types can be string (CDATA), enumeration (ENUMERATION), token (NMTOKEN), set of tokens (NMTOKENS), identifier (ID), reference identifier (IDREF), set of reference identifiers (IDREFS), entity (ENTITY), set of entities (ENTITIES), notation (NOTATION), and notation enumeration (ENUM-NOTATION).

Figure 15 illustrates the type mismatch conflicts in the EE, EA and AE scenarios. Each scenario contains two graphs. The left (right) graph shows the structure of the source (target) DTD. The figure shows the incompatible types for the node labeled `Cover`. In the EE scenario, the types for the node in the source and target are `PCDATA` and `ANY`, respectively. In AA scenario, the types for the node in the source and target are `CDATA` and `ENUMERATION`, respectively. Finally, in the EA/AE scenario, the types for the nodes in the source and target are `PCDATA` and `ENUMERATION`, respectively.

If an element is of type `ANY`, it can contain the non-null string, the null string, or a set of subelements, when the element is declared in a DOC. Therefore, if the source element is of type `PCDATA` or `EMPTY` and the target element is of type `ANY`, we can straightforwardly map the type of the source element to that of the target element. In addition, if the source element is of type `EMPTY` and the target element is of type `PCDATA`, we can also straightforwardly map the type of the source element to that of the target element, because the null-string can be considered a string of length zero. However, we cannot directly map a type `ANY` of the source element to a type `PCDATA` or type `EMPTY` of the target element; furthermore, we cannot directly map a type `PCDATA` of the source element to a type `EMPTY` of the target element. In other words, the most general element type is `ANY` followed by `PCDATA` and `EMPTY`, respectively.

The most general attribute type is `CDATA`. That is, any other attribute type can be obviously mapped to the `CDATA` type, but not vice versa.

If an attribute has type `NOTATION` (or `ENTITY`), the attribute *values* declared in a DOC must be one of the notation (or entity) *name* that is defined in an associated DTD using the tag `<!NOTATION` *name* `...>` (or `<!ENTITY` *name* `...>`). Basically, the name of the notation (entity) is a string. The notation is normally used to refer to either non-XML data (e.g., images) or other XML documents (e.g., XML files). The entity is normally used as alias name for an XML data element. It can also be used to refer to non-XML data, other XML-documents, and notations. For a particular document, the sets of all possible names for the notation and for entity must be disjoint. Therefore, there will be no mapping between the set of notation names and the set of entity names.

If an attribute has type ENUMERATION (or ENUM-NOTATION), the possible set of attribute values must be a set of strings (or notation names) and must be specified in a DTD. Since a notation name can be considered as a string, it is possible to map the enumeration set of the attribute of type ENUM-NOTATION to that of an attribute of type ENUMERATION.

If the type of the attribute is NMTOKEN, the value of the attribute is restricted to a valid XML name (i.e., a single string beginning with a letter or an underscore). If an attribute is of type NMTOKENS (ENTITIES), the value of the attribute can be composed of multiple XML names (entity names) separated by white space. Therefore, the attribute of type NMTOKEN (ENTITY) can be directly mapped to the attribute of type NMTOKENS (ENTITIES).

If the type of the attribute is ID, the value of the attribute must be a valid XML name and uniquely identify the element in the DOC. If the type of the attribute is IDREF, the value of the attribute must be an identifier for the referenced element in the DOC. If the type of the attribute is IDREFS, the value of the attribute can be composed of multiple identifiers for referenced elements in the DOC. Two elements containing attributes of type IDREFS can refer to the same element. In the mean time, each of those two elements can refer to multiple elements. Therefore, IDREFS support many-to-many relationships which are more general than many-to-one relationships. The many-to-one relationship can also be performed through the attributes of type IDREF. Two elements containing attributes of type IDREF can refer to the same element. However, each of those two elements can *not* reference multiple elements. Hence, attributes of type IDREF can be directly mapped to attributes of type IDREFS.

Mappings between the different types of an element to those of an attribute can be complicated. The CDATA type of an attribute indicates that the value of the attribute can be any text string that does not contain a less than sign (<), an ampersand (&), or quotation marks ("). If we want the text to contain those characters, we must use the pre-defined entity references (<, &, and &quote;). The PCDATA type of an element indicates that the content value of the element can be any text string that is not markup (i.e., that contains no children of its own). Therefore, an attribute of type CDATA can be mapped to an element of type PCDATA and vice versa with conversions of the special characters mentioned above.

4.4 *Domain Conflicts*

Like structural conflicts, domain conflicts arise when the same concept is represented differently in two DTDs. However, unlike structural conflicts, knowledge about the underlying data domain is needed for detecting domain conflicts due to the differences among the data values. The class of domain conflict is divided into four categories: schematic discrepancies, scale or unit conflicts, precision conflicts, and data representation conflicts.

4.4.1 Schematic Discrepancies

This category of conflicts was previously discussed in (Kashyap and Sheth 1995; Miller 1998). These conflicts arise when data in one schema corresponds to schema labels in the other. In a DTD, data can be represented as either a PCDATA element or a CDATA attribute type (i.e., the data values declared in a DOC are represented by the PCDATA element or the CDATA attribute defined in the associated DTD.) Also, in a DTD, a schema label corresponds to either an element label or an attribute name. In conventional data models, conflicts between data values and attributes, between attributes and entities, as well as between data values and entities fall into this category. In semistructured data models, the difference between entity and attribute (as used in the relational data model) is blurred (Abiteboul et al. 2000). In other words, we can represent an attribute value as an attribute value associated with a label or as an element. Based on the concepts of elements and attributes in XML, we classify this conflict into four subcategories known EE, AE, EA, and AA.

4.4.1.1 Element-value-to-Element-label Mapping (EE)

This type of conflict arises when a `PCDATA` element in the source DTD is mapped to an element label in the target DTD. Note that since a `PCDATA` element represents a data value declared in a DOC, the domain of the `PCDATA` element in the DTD is a set of all possible data values that can be declared in the DOC. Therefore, the mapping between the `PCDATA` element and the element label can be detected if and only if the domain of the `PCDATA` element is known.

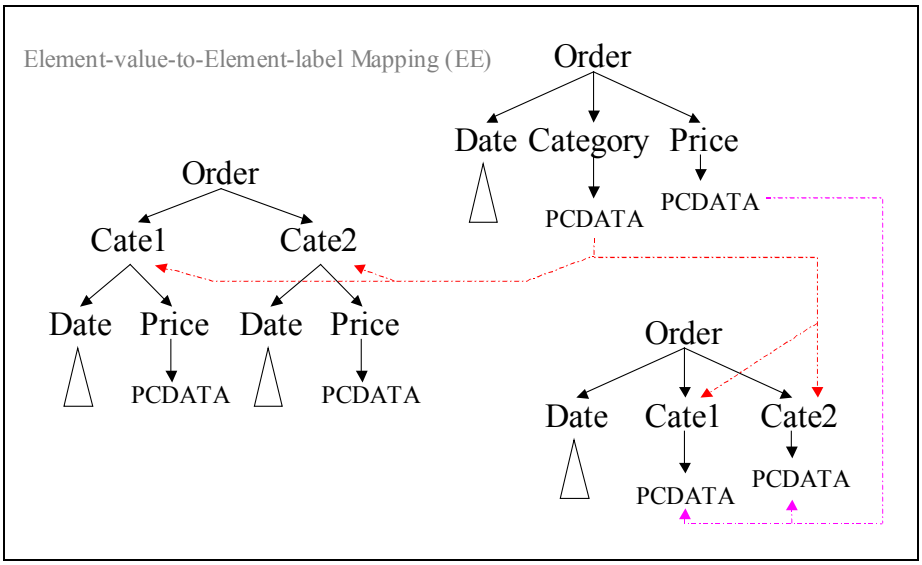


Figure 16: Schematic Discrepancy conflict in the EE scenario

In Figure 16, the `PCDATA` node of the element `Order.Category` (in the upper right hand graph) will be replaced by a data value when it is declared in a DOC. Its domain is limited to the set {"Cate1", "Cate2"}. The values in the domain set are equivalent to the types (e.g., Cate1 and Cate2) of the element (e.g., Order) defined in the target DTD (in either the left or lower right hand graph).

4.4.1.2 Attribute-value-to-Element-label Mapping (AE)

This type of conflict arises when a `CDATA` attribute in the source DTD is mapped to an element label in the target DTD. Note that since a `CDATA` attribute in the DTD represents an attribute value declared in a DOC, the domain of the `CDATA` attribute is a set of all possible attribute values that can be declared in the DOC. Therefore, the mapping between the `CDATA` attribute and the element label can be detected if and only if the domain of the `CDATA` attribute is known.

In Figure 17, the `CDATA` node representing the type of the attribute, `Order@Category` (in the upper right hand graph) will be replaced by a data value when it is declared in a DOC. Its domain is limited to the set {"Cate1", "Cate2"}. The values in the domain set are equivalent to the types (e.g., Cate1 and Cate2) of the element (e.g., Order) defined in the target DTD (in either the left or lower right hand graph).

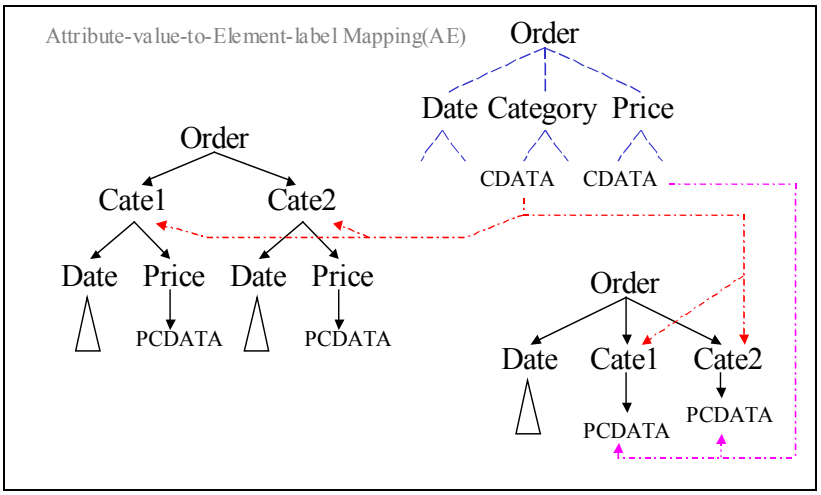


Figure 17: Schematic Discrepancy conflict in the AE scenario

4.4.1.3 Element-value-to-Attribute-label Mapping (EA)

This type of conflict arises when a `PCDATA` element in the source DTD is mapped to an attribute label in the target DTD. Note that since a `PCDATA` element represents a data value declared in a DOC, the domain of the `PCDATA` element in the DTD is a set of all possible data values that can be declared in the DOC. Therefore, the mapping between the `PCDATA` element and the attribute label can be detected if and only if the domain of the `PCDATA` element is known.

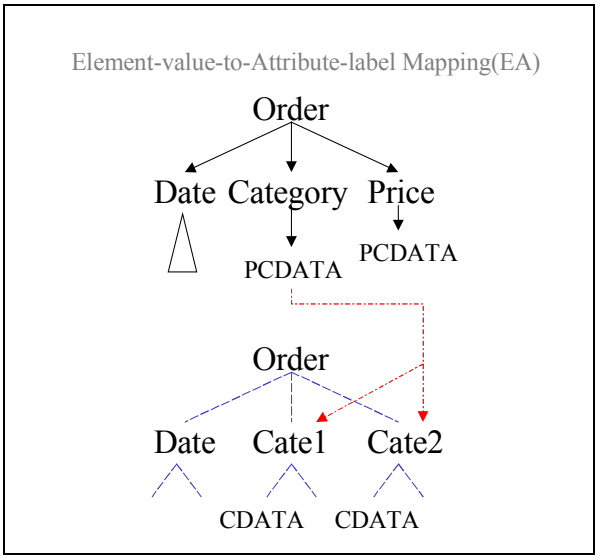


Figure 18: Schematic Discrepancy conflict in the EA scenario

In Figure 18, the `PCDATA` node of the element `Order.Category` (in the upper graph) will be replaced by a data value when it is declared in a DOC. Possible data values (i.e., the data domain) are limited to the set {"`Cate1`", "`Cate2`"}. The values in the domain set are equivalent to the attribute label (e.g., `Cate1` and `Cate2`) of the element (e.g., `Order`) defined in the target DTD (in the lower graph).

4.4.1.4 Attribute-value-to-Attribute-label Mapping (AA)

This type of conflict arises when a `CDATA` attribute in the source DTD is mapped to an attribute label in the target DTD. Note that since a `CDATA` attribute in the DTD represents an attribute value declared in a DOC, the domain of the `CDATA` attribute is a set of all possible attribute values that can be declared in the DOC. Therefore, the mapping between the `CDATA` attribute and the attribute label can be detected if and only if the domain of the `CDATA` attribute is known.

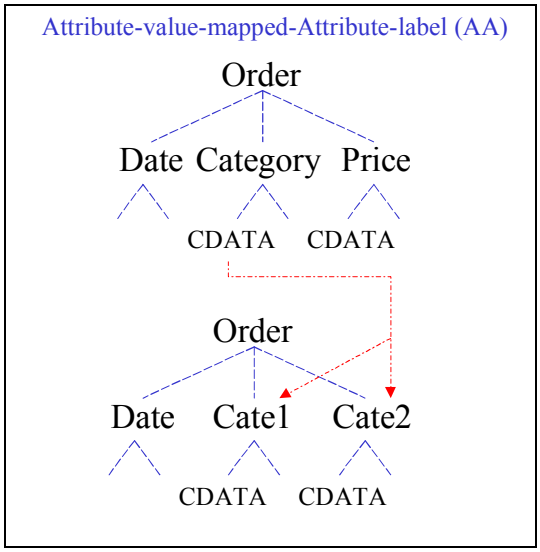


Figure 19: Schematic Discrepancy conflict in the AA scenario

In Figure 19, the `CDATA` node representing the type of the attribute, `Order@Category`, (in the upper graph) will be replaced by a data value when it is declared in a DOC. Possible data values (i.e., the data domain) are limited to in the set {"Cate1", "Cate2"}. The values in the domain set are equivalent to the attribute label (e.g., Cate1 and Cate2) of the element (e.g., Order) defined in the target DTD (in the lower graph).

4.4.2 Scale or Unit

This type of conflict arises when two simple elements (i.e., the element whose content is of type `PCDATA`) or attributes are mapped to each other and their values are represented using different scale or units.

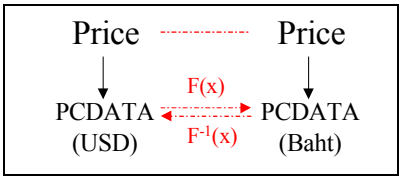


Figure 20: Scale or Unit conflict

Figure 20 shows the scale conflict between two simple elements. Knowledge about the source tells us that all values of element `Price` declared in the source DOC are represented in US dollars, whereas knowledge about the target reveals that all values of element `Price` declared in the target DOC are represented in Baht (i.e., the unit of Thai currency). To resolve this type of conflicts, we must define a two-way mapping function between the values or use a look up table, for example. The function and the table should be defined by a domain expert.

4.4.3 Precision

This type of conflict arises when two simple elements (i.e., the element whose content is type of `PCDATA`) or attributes are mapped and their values are represented using different precision. For example, as shown Figure 21, `Score` in the source DTD is mapped to `Rank` in the target DTD. Possible values of `Score` are real numbers between 1 and 10 while possible values of `Rank` are discrete values in the set of {A, B, C, D}. Similar to resolving Scale conflicts, a look up table and a two-way mapping function between the values are used. Both should be defined by a domain expert.

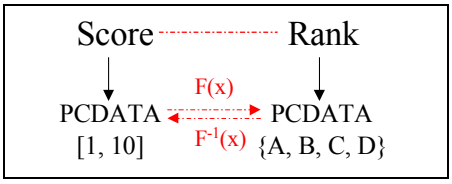


Figure 21: Precision conflict.

4.4.4 Data Representation

These conflicts arise when two simple elements or attributes are mapped and their values have different representation that include primitive data types (e.g., integer, floating point, string) and data format (e.g., 9-digit integer, 11-character string).

4.4.4.1 Primitive Data Type

In XML, every data value is of type string. However, data values in the conventional data source can have other primitive types. One way of resolving this conflict is to attach each simple element with an extra attribute indicating the original data type. Another way is to create an auxiliary document that contains information about element types and then attach the auxiliary document to the source document.

4.4.4.2 Data Format

Although the string type is the only primitive type in XML, there are still many ways one can represent the same information using strings. For example, one can represent a social security number as a string of numbers (e.g., 999999999) or as numbers with dashes (e.g., 999-99-9999). Note that data values are declared in a DOC and they are represented as a `PCDATA` element (or a `CDATA` attribute) defined in the associated DTD. If the general format of those data values is known, we can transform the `PCDATA` element (or the `CDATA` attribute) defined in the source DTD to that defined in the target DTD. Knowledge about the data format can be provided by user input can be attached as metadata to the `PCDATA` element (or the `CDATA` attribute), or can be enclosed as an auxiliary document associated with the source DTD. Even though if the information about data format is known, some additional user input may be needed to verify the transformation of the `PCDATA` element (or the `CDATA` attribute).

4.5 *Data Conflicts*

All of the conflicts we have mentioned in the previous sections can be detected by comparing source and target DTDs as long as we have knowledge about the domain of the data. However, data conflicts can only be detected when examining the DOCs themselves. Data conflicts arise when some properties of a real-world instance declared in two different DOCs represent a mismatch, or when two different real-world instances represented in different sources overlap. We divide data conflicts into four categories: Naming, ID-value, missing data and incorrect spelling.

4.5.1 Naming

Unlike naming conflicts in the class of structural conflicts, naming data conflicts are conflicts involving element or attribute values that are declared in DOCs. Recall that we modeled a DOC as a graph containing an element tree and a set of attribute graphs. These conflicts arise when *leave nodes* of each element subtree or *attribute value nodes* in the attribute graph are conflict. The two nodes are either equivalent (e.g., synonym and acronym) or incompatible (e.g., homonym) (Kashyap and Sheth 1995).

4.5.1.1 Case sensitivity

As mentioned before, each data value in XML is of primitive type string. In the case where the data values are text strings, the values can provide an important clue as to whether terms are related or not (e.g., when deciding whether or not `Windows` and `windows` are equivalent). In the next sections, case sensitivity is assumed. To resolve conflicts involving text strings from sources with different case sensitivity, the fact whether the case is sensitive or not must be specified explicitly.

4.5.1.2 Synonyms

This type of conflict arises when two different terms have the same meaning or when two elements declared in different DOCs refer to the same real-world object. For example, let us assume that we use a text string to represent the name of customers.

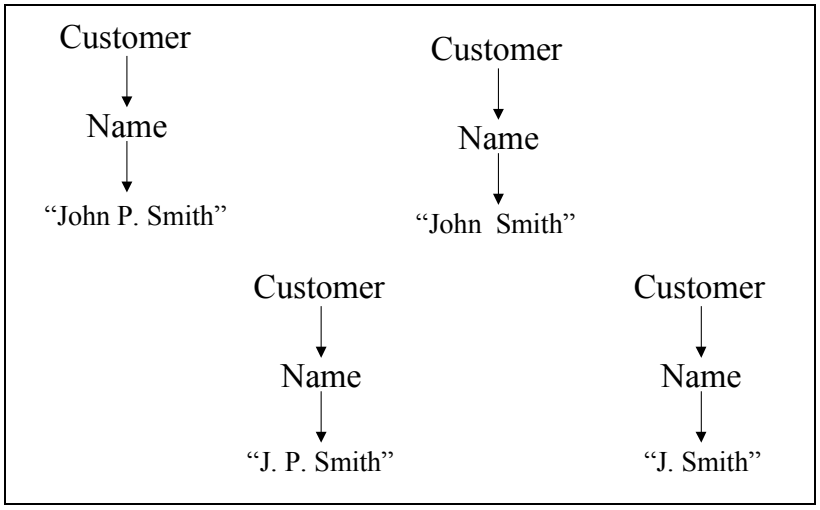


Figure 22: Data conflict of type synonyms.

Figure 22 gives a sample set of names, {"John P. Smith," "John Smith," "J. P. Smith," "J. Smith"} that represent the same person whose name is "John P. Smith." To resolve the data synonym conflicts, one way of doing so is by looking up alternative definitions in a dictionary to generate mappings from unknown terms to known terms. In addition, the approximation matching of text (i.e., edit distance-based matching) is another way of resolving the conflicts. As shown in Figure 22, if the approximation matching is applied to the `Name` element, we can figure out that all Customers are representing the same person.

4.5.1.3 Acronyms

Parts of a series of words can form a new term. Such a new term is called *acronym*, and such series of words is called the *full string* of the acronym. This type of conflict arises when one term is an acronym of the other. It can be considered as a special case of the synonym conflict since the acronym and its full name are semantically equivalent; hence, the same resolution procedure can be applied.

An acronym conflict is called *one-to-one* if an acronym is mapped to only one full string in a domain space. It is called *one-to-many* if an acronym is mapped to many full strings in the domain space. It is called *many-to-one* if there is more than one acronym that can be mapped to the same full string in the domain space. Finally, it is called *many-to-many* if there is more than one acronym that can be mapped to more than one full string in the domain space. Figure 23 only shows the one-to-many (in the upper block) and many-to-one (in the lower block) acronym conflicts.

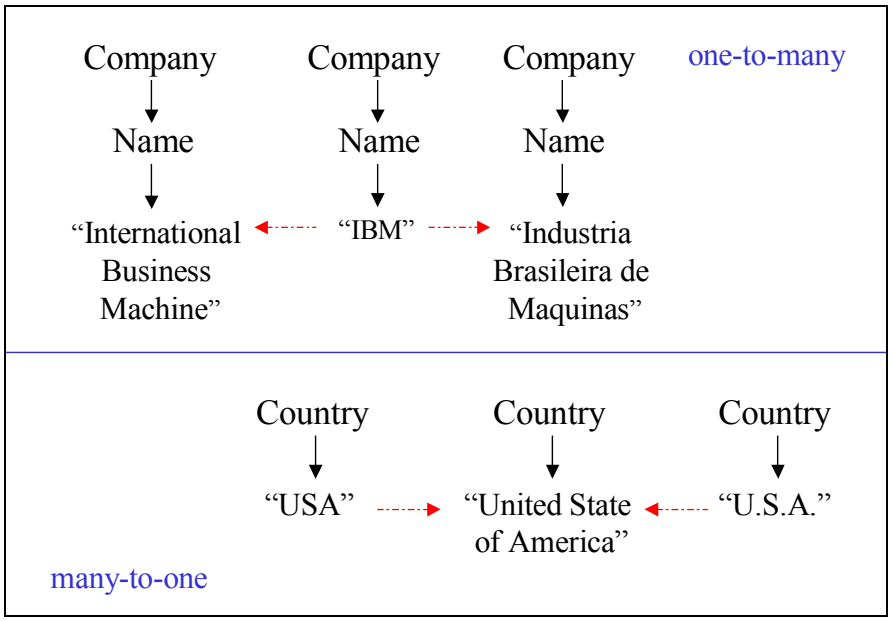


Figure 23: Data conflicts of type acronym.

4.5.1.4 Homonyms

This type of conflict arises when a term is used to refer to different real-world objects or concepts. As shown in Figure 24, "`Windows`" is the name of the element `OS` declared in the source DOC but belongs to the element `Software` in the target DOC. This raises the question whether or not the two elements "`Windows`" in the two different data sources are equivalent.

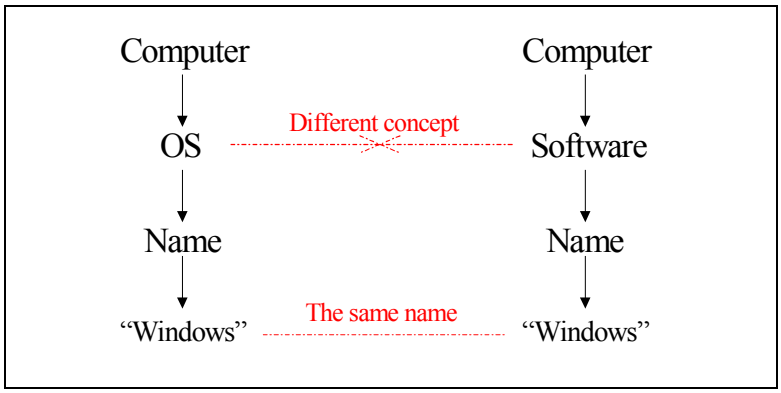


Figure 24: Data conflict of type homonym.

Homonym conflicts can be detected by considering the corresponding paths from the root to the nodes containing the same values. The paths and the data values are declared in DOCs that are being matched. Homonym conflicts occur when there exists conflicting terms along the paths. To resolve the

data homonym conflicts, for each pair of conflicting terms, we redefine the relationship between those two terms to be mismatched.

4.5.2 ID-value

In XML, each element type can have attributes of type ID associated with it. The attribute values of type ID must be unique for elements of the same type declared in the same DOC, but they are not guaranteed to be unique for those declared in different DOCs. Therefore, ID-value conflicts can occur. In other words, ID-value conflicts arise when two elements of the same type are declared in two different DOCs, and have different ID values associated with the elements. On the other hand, the ID-value conflicts can arise when two different elements declared in different DOCs have the same ID value.

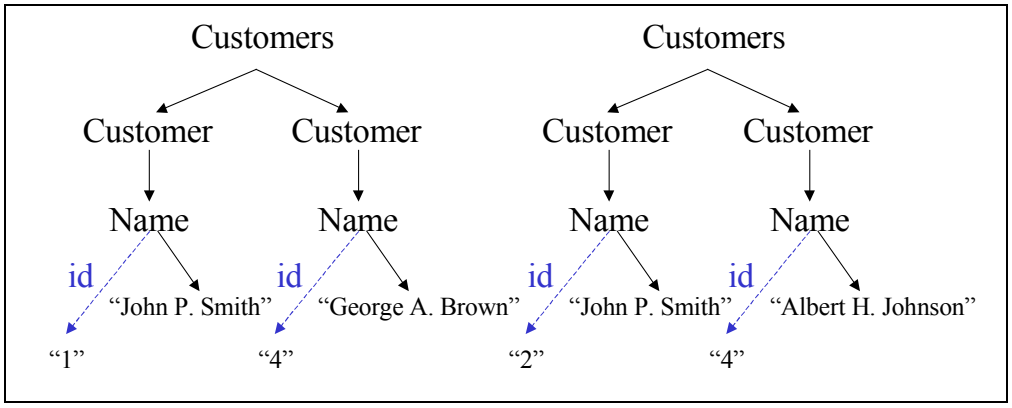


Figure 25: ID-value conflict.

Figure 25 illustrates the ID-value conflict. The figure contains two graphs representing source and target DOCs. Consider the two elements of type `Customer` whose `Name` values are "`John P. Smith`" in both DOCs. They are representing the same person but have two different `id` values (e.g., "`1`" and "`2`"). Also, consider two elements of type `Customer` whose `Name` values are "`George A. Brown`" and "`Albert H. Johnson`". We can see that those elements have the same `id` values (e.g., "`4`").

During the process of integrating XML DOCs (see Sec. 4.1), ID-value conflict is one of the conflicts that must be resolved. To resolve ID-value conflict, a global ID must be assigned to each element in the integrated document. Elements from several sources representing the same real world object must have the same global ID value. The global ID must be unique. The mapping between the global ID and the local one (i.e., the original ID associated with each element in the sources) can be arbitrarily chosen or can apply a function (e.g., Skolem function.)

4.5.3 Missing Data

Missing data conflicts arise when two element instances in two DOCs are mapped, but at least one subelement of one of the elements is missing. As shown in Figure 26, `Customer` with `Name` value "`John P. Smith`" has a `Phone` number in one DOC, but has no `Phone` number in the other. During the process of integrating XML DOCs (see Sec.4.1), missing data gives an impact on identifying whether or not two data elements from two sources represent the real world object. Closeness of two data elements is defined using a similarity function. The similarity function takes all parameters as inputs and generates a numeric value specifying distance between two data elements. Normally, those parameters refer to subelements of the two data elements. Therefore, if one of subelements of the two elements that are being mapped is missing, the distance between the two data elements may be large, meaning that they are representing different objects. Choosing an appropriated

similarity function will reduce the impact of the missing data conflict. It is still an open question how to define the appropriated similarity function.

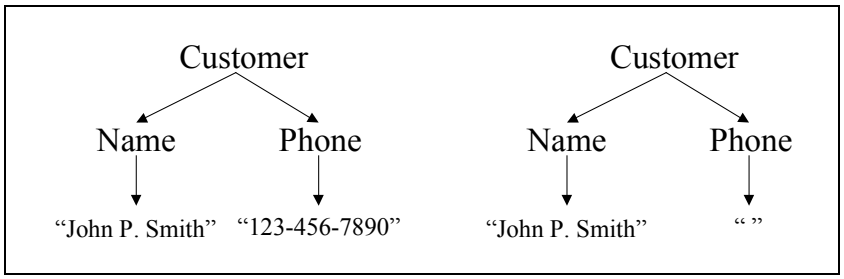


Figure 26: Missing Data conflict.

4.5.4 Incorrect Spelling

These conflicts arise when the same data values in the sources are misspelled. As mentioned previously, closeness of two data elements is defined using a similarity function that is evaluated based on the content values of those elements. If the incorrect spelling exists, it will have an impact on mapping the two data elements. Using a spell check can reduce the effect of this conflict.

5 Conclusion

We have proposed a set of directed-labeled graphs to model XML documents and DTDs. Based on such graph model, we introduce a classification of all the possible conflicts that can occur between two related XML documents containing similar data. We believe that such classification can be used as a guideline for resolving the existing conflicts during integrating heterogeneous semistructured data sources.

Note that we classify the conflicts based on the core XML specification. Although there are several proposed working drafts of alternatives to DTDs for richer schema definition, such as XML-Schema (2000b; 2000c; 2000d) and DCDs (1998a), we believe that our classification of conflicts remains useful, since the main concepts for representing data using XML do not change. For example, XML-Schema can only increase the richness of primitive data types that are explicitly defined for each data item. Unlike in XML-Schema, data types can be implicitly defined using the core XML.

We are developing an integration system prototype, called the Information Integration WIZard (IWIZ) (Hammer 1999). The prototype uses XML as a common, internal data representation. The sources are distributed and heterogeneous databases. The source data can be relational, object-oriented, unstructured or semi-structured data, and must be wrapped into a set of XML documents. Each document requires at least one DTD that defines its schema or structure. If the document does not have the associated DTD, the DTD generator will automatically create the associated DTD based on the information in the document. Given a target DTD, the source DTDs and documents are restructured and merged to form a target document (i.e., a integrated document) whose structure is defined in the target DTD. We are planning to use the directed labeled graph presented in this paper to model the internal structure of DTDs and DOCs. Furthermore, we plan to develop a system that can (semi-) automatically resolve all the conflicts presented in this paper.

References

(1992). "Ontology Server." Knowledge Systems Laboratory (KSL): http://www-ksl-svc.stanford.edu:5915.

(1998a). “Document Content Description for XML.” The World Wide Web Consortium (W3C), http://www.w3.org/TR/NOTE-dcd.

(1998b). “The Document Object Model (DOM) Level 1 Specification.” The World Wide Web Consortium (W3C), http://www.w3.org/TR/REC-DOM-Level-1/.

(1998c). “Extensible Markup Language (XML) 1.0.” The World Wide Web Consortium (W3C), http://www.w3c.org/TR/1998/REC-xml-19980210.html.

(1998d). “Extensible Stylesheet Language (XSL).” The World Wide Web Consortium (W3C), http://www.w3c.org/Style/XSL/.

(1999a). “Namespaces in XML.” The World Wide Web Consortium (W3C), http://www.w3.org/TR/REC-xml-names/.

(1999b). “XML Pointer Language (XPointer).” The World Wide Web Consortium (W3C), http://www.w3.org/TR/xptr.

(2000a). “XML Linking Language (XLink).” The World Wide Web Consortium (W3C), http://www.w3.org/TR/xlink/.

(2000b). “XML Schema Part 0: Premier.” The World Wide Web Consortium (W3C), http://www.w3.org/TR/xmlschema-0/.

(2000c). “XML Schema Part 1: Structures.” The World Wide Web Consortium (W3C), http://www.w3.org/TR/xmlschema-1/.

(2000d). “XML Schema Part 2: Data types.” The World Wide Web Consortium (W3C), http://www.w3.org/TR/xmlschema-2/.

Abiteboul, S. (1997). “Querying semistructured data.” *International. Conference on Database Theory*, 1-18.

Abiteboul, S., Buneman, P., and Suciu, D. (2000). *Data on the Web*, Morgan Kaufmann, San Francisco, CA.

Batini, C., Lenzerini, M., and Navathe, S. B. (1986). “A Comparative Analysis of Methodologies for Database Schema Integration.” *ACM Computing Surveys*, 18(4), 323-364.

Bosak, J. (1997). “XML, Java, and future of the Web.” *http://metalab.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm*, Sun Microsystems.

Bressan, S., Goh, C. H., Fynn, K., Jakobisiak, M., Hussein, K., Kon, H. B., Lee, T., Madnick, S. E., Pena, T., Qu, J., Shum, A. W., and Siegel, M. (1997). “The COntext INterchange Mediator Prototype.” *SIGMOD Record (ACM Special Interest Group on Management of Data)*, Tucson, Arizona, USA., 525-527.

Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J. (1994). “The TSIMMIS Project: Integration of Heterogeneous Information Sources.” *Tenth Anniversary Meeting of the Information Processing Society of Japan*, Tokyo, Japan, 7-18.

Cluet, S., Delobel, C., Simeon, J., and Smaga, K. (1998). “Your Mediators Need Data Conversion!” *SIGMOD*, 177-188.

CommerceOne. (1999). http://www.commerceone.com/.

Fankhauser, P., and Neuhold, E. (1992). “Knowledge Based Integration of Heterogeneous Databases.” Technical Report, Technische Hochschule Darmstadt.

Genesereth, M. R., Keller, A. M., and Duschka, O. M. (1997). “Infomaster: An Information Integration System.” *SIGMOD Conference*, 539-542.

Hammer, J. (1999). “The Information Integration Wizard (IWiz) Project.” Project Description *TR99-019*, University of Florida, Gainesville, FL.

Hammer, J., Garcia-Molina, H., Widom, J., Labio, W., and Zhuge, Y. (1995). “The Stanford Data Warehousing Project.” *Data Engineering Bulletin*, 18(2), 41-48.

Harold, E. R. (1998). *XML: Extensible Markup Language*, IDG Books Worldwide, Foster, California.

Harold, E. R. (1999). *XML Bible*, IDG Books Worldwide Inc.

Kashyap, V., and Sheth, A. (1995). "Semantic and Schematic Similarities between Objects in Databases: A Context-based approach." Technical Report, Department of Computer Science, University of Georgia, Atlanta, Georgia.

Kent, W. (1991). "Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language." *Seventeenth International Conference on Very Large Data Bases*, Barcelona, Spain, 147-160.

Kim, W., Choi, I., Gala, S., and Scheevel, M. (1993). "On Resolving Schematic Heterogeneity in Multidatabase Systems." *Distributed and Parallel Databases*, 251-279.

Lakshmanan, L. V. S., Sadri, F., and Subramanian, I. N. (1996). "SchemaSQL - A Language for Interoperability in Relational Multi-Database Systems." *Proceedings of the VLDB Conference (VLDB)*, 239-250.

Levy, A. (1998). "The Information Manifold Approach to Data Integration." *IEEE Intelligent Systems*, 13(??), 12 - 16.

Levy, A. (1999). "More on Data Management for XML." University of Washington.

Ludascher, B., Papakonstantinou, Y., and Velikhov, P. (1999). "A Framework for Navigation-Driven Lazy Mediators." *Proc. of the ACM Sigmod Workshop on the Web and Databases (WebDB'99)*, Philadelphia, Pennsylvania, 1-6.

Maluf, D. A., and Wiederhold, G. (1997). "Abstraction of Representation for Interoperation." *The 10th International Symposium on Methodologies for Intelligent Systems (ISMIS)*, 441-455.

Miller, R. J. (1998). "Using Schematically Heterogeneous Structures." *SIGMOD*, 189-200.

Papakonstantinou, Y., Abiteboul, S., and Garcia-Molina, H. (1996). "Object Fusion in Mediator Systems." *VLDB 1996*, 413-424.

RosettaNet. (1999). http://www.rosettanet.org.

SchemaNet. (1999). http://www.schema.net.

Sciore, E., Siegel, M., and Rosenthal, A. (1994). "Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems." *The ACM Transactions on Database Systems (TODS)*, 19(2), 254-290.

Siegel, M., and Madnick, S. E. (1991). "A Metadata Approach to Resolving Semantic Conflicts." *Seventeenth International Conference on Very Large Databases*, Barcelona, Spain, 133-145.

Tork Roth, M., Schwarz, P. M., and 266-275, V. (1997). "Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources." *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*, Athens, Greece, 266-275.

Widom, J. (1999). "Data Management for XML: Research Directions." Stanford University.

XML/EDI. (1999). http://www.xmledi.com.

Zhou, G., Hull, R., King, R., and Franchitti, J.-C. (1995). "Data Integration and Warehousing Using H2O." *Data Engineering Bulletin*, 18(2), 29-40.

Appendix A: Constraint Mappings

Given a source and a target DTDs, we are trying to semi-automatically generate mappings of elements and attributes defined in the two DTDs. We will use those mappings to transform the data in a source DOC.

I. Element-to-Element Constraint Mappings

Let L and M be elements whose structures are defined in the source DTD. M is a subelement of L, and associated with one of the following constraints: zero-or-more (*), one-or-more (+), zero-or-one (?) and exactly-one (.). Let P and Q be elements whose structures are defined in the target DTD. Q is a subelement of P, and associated with one of the following constraints: zero-or-more (*), one-or-more (+), zero-or-one (?) and exactly-one (.).

Table A1 : Element-to-Element Constrain Mappings

Source Element with Constraint	Target Element with Constraint	Document Transformation Rule Templates
*	*	If $j = 0$ then *DoNothing*( ) Otherwise, $E_k^t := E_k^s$ for all $k \in \mathrm{N}$ and $k \le j$
*	+	If $j = 0$ then Error or $E_1^t :=$ *GenerateAValue*( ) Otherwise, $E_k^t := E_k^s$ for all $k \in \mathrm{N}$ and $k \le j$
*	?	If $j = 0$ then *DoNothing*( ) If $j = 1$ then $E_1^t := E_1^s$ If $j > 1$ then $E_1^t :=$ *PickOne*($E_1^s, E_2^s, \ldots, E_j^s$)
*	.	If $j = 0$ then Error or $E_1^t :=$ *GenerateAValue*( ) If $j = 1$ then $E_1^t := E_1^s$ If $j > 1$ then $E_1^t :=$ *PickOne*($E_1^s, E_2^s, \ldots, E_j^s$)
+	*	$E_k^t := E_k^s$ for all $k \in \mathrm{N}$ and $k \le j$
+	+	$E_k^t := E_k^s$ for all $k \in \mathrm{N}$ and $k \le j$
+	?	If $j = 1$ then $E_1^t := E_1^s$ If $j > 1$ then $E_1^t :=$ *PickOne*($E_1^s, E_2^s, \ldots, E_j^s$)
+	.	If $j = 1$ then $E_1^t := E_1^s$ If $j > 1$ then $E_1^t :=$ *PickOne*($E_1^s, E_2^s, \ldots, E_j^s$)

Table A1 (continued): Element-to-Element Constrain Mappings

Source Element with Constraint	Target Element with Constraint	Document Transformation Rule Templates
?	*	If j = 0 then *DoNothing*( ) If j = 1 then $E_1^t := E_1^s$
?	+	If j = 0 then Error or $E_1^t :=$ *GenerateAValue*( ) If j = 1 then $E_1^t := E_1^s$
?	?	If j = 0 then *DoNothing*( ) If j = 1 then $E_1^t := E_1^s$
?	.	If j = 0 then Error or $E_1^t :=$ *GenerateAValue*( ) If j = 1 then $E_1^t := E_1^s$
.	*	$E_1^t := E_1^s$
.	+	$E_1^t := E_1^s$
.	?	$E_1^t := E_1^s$
.	.	$E_1^t := E_1^s$

Assume L is mapped to P and M is mapped to Q with some relations. Our goal is to map constraint of element M to constraint of element Q. Table A1 shows rule templates for the possible constraint mappings. Let N be a set of natural numbers. Symbols j, E_k^s and E_k^t, where $k \in \mathrm{N}$, represent variables whose values will be assigned during transformation of the source DOC. Symbol j represents the number of consecutive instances of M that are declared in the source DOC. E_k^s represents the k^{th} consecutive instance of M declared in the source DOC. E_k^t represents the k^{th} consecutive instance of Q declared in the target DOC.

DoNothing is a function that has no operation. *PickOne* is a template function that returns one of the input parameters. *GenerateAValue* is a template function that returns an arbitrary value.

II. Element-to-Attribute Constraint Mappings.

Let L and M be elements whose structure is defined in the source DTD. M is a subelement of L, and associated with one of the following constraints: zero-or-more (*), one-or-more (+), zero-or-one (?) and exactly-one (.).

Let P be an element defined in the target DTD. Let A be an attribute of P. Type and constraint of A is defined in the target DTD. Constraint of A is one of the following: #REQUIRED, #IMPLIED, UNFIXED-DEFAULT-VALUE (i.e., DEFAULT for short) and #FIXED. If constraint of A is DEFAULT, let D be the default value of A. If constraint of A is #FIXED, let F be the fixed value of A.

Assume L is mapped to P and M is mapped to A with some relations. Our goal is to map constraint of element M to constraint of attribute A. Table A2 shows rule templates for the possible

constraint mappings. V , j and E_k , where k is a natural number, are variables whose values will be assigned during transformation of the source DOC. Symbol j represents the number of consecutive instances of M that are declared in the source DOC. V represents the value to be assigned for the attribute A . E_k is the value of the k^{th} consecutive instance of M declared in the source DOC.

DoNothing is a function that has no operation. *PickOne* is a template function that returns one of the input parameters. *GenerateAValue* is a template function that returns an arbitrary value.

Table A2: Element-to-Attribute Constraint Mappings

Source Element with Constraint	Target Attribute with Constraint	Document Transformation Rule Templates
*	IMPLIED	If j = 0 then *DoNothing*( ); If j = 1 then $V := E_1$ If j > 1 then $V :=$ *PickOne*($E_1, E_2, \ldots, E_j$)
*	REQUIRED	If j = 0 then $V :=$ *GenerateAValue*( ) If j = 1 then $V := E_1$ If j > 1 then $V :=$ *PickOne*($E_1, E_2, \ldots, E_j$)
*	DEFAULT	If j = 0 then $V := D$ If j = 1 then $V := E_1$ If j > 1 then $V :=$ *PickOne*($E_1, E_2, \ldots, E_j$)
*	FIXED	If j = 0 then $V := F$ If j >= 1 and $E_k = F$, where $k \leq j$, then $V := F$ Otherwise Error
+	IMPLIED	If j = 1 then $V := E_1$ If j > 1 then $V :=$ *PickOne*($E_1, E_2, \ldots, E_j$)
+	REQUIRED	If j = 1 then $V := E_1$ If j > 1 then $V :=$ *PickOne*($E_1, E_2, \ldots, E_j$)
+	DEFAULT	If j = 1 then $V := E_1$ If j > 1 then $V :=$ *PickOne*($E_1, E_2, \ldots, E_j$)
+	FIXED	If j >= 1 and $E_k = F$, where $k \leq j$, then $V := F$ Otherwise Error

Table A2 (continued): Element-to-Attribute Constraint Mappings

Source Element with Constraint	Target Attribute with Constraint	Document Transformation Rule Templates
?	IMPLIED	If j = 0 then *DoNothing*( ) If j = 1 then $V := E_1$
?	REQUIRED	If j = 0 then V := *GenerateAValue*( ) If j = 1 then $V := E_1$
?	DEFAULT	If j = 0 then $V := D$ If j = 1 then $V := E_1$
?	FIXED	If j = 0 then $V := F$ If $E_1 = F$ then $V := F$; otherwise Error
.	IMPLIED	$V := E_1$
.	REQUIRED	$V := E_1$
.	DEFAULT	$V := E_1$
.	FIXED	If $E_1 = F$ then $V := F$; otherwise Error

III. Attribute-to-Element Constraint Mappings

Let P be an element defined in the source DTD. Let A be an attribute of P. Type and constraint of A is defined in the source DTD. Constraint of A is one of the following: #REQUIRED, #IMPLIED, UNFIXED-DEFAULT-VALUE (i.e., DEFAULT for short) and #FIXED. If constraint of A is DEFAULT, let D be the default value of A. If constraint of A is #FIXED, let F be the fixed value of A.

Let L and M be elements whose structure is defined in the source DTD. M is a subelement of L, and associated with one of the following constraints: zero-or-more (*), one-or-more (+), zero-or-one (?) and exactly-one (**.**).

Assume P is mapped to L and A is mapped to M with some relations. Our goal is to map constraint of attribute A to constraint of element *M*. Table A3 shows rule templates for the possible constraint mappings. V, j and E are variables whose values will be assigned during transformation of the source DOC. Symbol j represents the number of consecutive instances of M that are declared in the source DOC. V represents the value to be assigned for the attribute A. If A is not declared in the source DOC, V contains the null value. E is the instance of M that will be declared in the target DOC. *DoNothing* is a function that has no operation. *GenerateAValue* is a template function that returns an arbitrary value.

Table A3: Attribute-to-Element Constraint Mappings

Source Attribute with Constraint	Target Element with Constraint	Document Transformation Rule Templates
IMPLIED	*	If V = null then *DoNothing*( ) Otherwise, $E := V$
IMPLIED	+	If V = null then Error or E := *GenerateAValue*( ) Otherwise, $E := V$
IMPLIED	?	If V = null then *DoNothing*( ) Otherwise, $E := V$
IMPLIED	.	If V = null then Error or E := *GenerateAValue*( ) Otherwise, $E := V$
REQUIRED	*	$E := V$
REQUIRED	+	$E := V$
REQUIRED	?	$E := V$
REQUIRED	.	$E := V$
DEFAULT	*	$E := V$
DEFAULT	+	$E := V$
DEFAULT	?	$E := V$
DEFAULT	.	$E := V$
FIXED	*	$E := F$
FIXED	+	$E := F$
FIXED	?	$E := F$
FIXED	.	$E := F$

IV. Attribute-to-Attribute Constraint Mappings

Let L and P be elements whose structures are defined in the source and the target DTDs, respectively. Let A be an attribute of L and B be an attribute of P. Type and constraint of A (B) is defined in the source (target) DTD. Constraint of A and B is one of the following: #REQUIRED, #IMPLIED, UNFIXED-DEFAULT-VALUE (i.e., DEFAULT for short) and #FIXED. If constraint of A is DEFAULT, let D_A be the default value of A. If constraint of A is #FIXED, let F_A be the fixed value of A. Similarly, If constraint of B is DEFAULT, let D_B be the default value of B. If constraint of A is #FIXED, let F_B be the fixed value of B.

Assume L is mapped to P and A is mapped to B with some relations. Our goal is to map constraint of attribute A to constraint of attribute B. Table A5 shows rule templates for the possible constraint mappings. V_A and V_B are variables whose values will be assigned during transformation of the source DOC. V_A and V_B represent the values to be assigned for the attributes A and B,

respectively. Initially, V_A and V_B contain the null value. *GenerateAValue* is a template function that returns an arbitrary value.

Table A4: Attribute-to-Attribute Constraint Mappings

Source Attribute with Constraint	Target Attribute with Constraint	Document Transformation Rule Templates
IMPLIED	IMPLIED	$V_B := V_A$
IMPLIED	REQUIRED	If V_A = null then Error or V_B := *GenerateAValue*( ) Otherwise, $V_B := V_A$
IMPLIED	DEFAULT	If V_A = null then $V_B := D_A$ Otherwise, $V_B := V_A$
IMPLIED	FIXED	If V_A = null then $V_B := F_B$ Otherwise, Error or $V_B := F_B$
REQUIRED	IMPLIED	$V_B := V_A$
REQUIRED	REQUIRED	$V_B := V_A$
REQUIRED	DEFAULT	$V_B := V_A$
REQUIRED	FIXED	If V_A != F_B then Error or $V_B := F_B$ Otherwise, $V_B := V_A$
DEFAULT	IMPLIED	$V_B := V_A$
DEFAULT	REQUIRED	$V_B := V_A$
DEFAULT	DEFAULT	$V_B := V_A$
DEFAULT	FIXED	If V_A != F_B then Error or $V_B := F_B$ Otherwise, $V_B := V_A$
FIXED	IMPLIED	$V_B := F_A$
FIXED	REQUIRED	$V_B := F_A$
FIXED	DEFAULT	$V_B := F_A$
FIXED	FIXED	If F_A != F_B then Error Otherwise, $V_B := F_A$