Exploring Differences in Exchange Formats – Tool Support and Case Studies

Juanjuan Jiang
Institute of Software System, Tampere University of Technology, Finland
jiang@cs.tut.fi

Tarja Systä
Institute of Software System, Tampere University of Technology, Finland
tsysta@cs.tut.fi

Abstract

XML-based markup languages are widely used, e.g., for information exchange and as file formats in various software development and exploration tools. Still, using a metalanguage, such as XML, does not guarantee tool interoperability. The particular XML-based languages used by different tools often vary. They can, nonetheless, be processed by the same methods and tools. In most UML-based software development tools, support for tool interoperability is provided by using OMG's XML Metadata Interchange (XMI) as a file format. However, in many cases XMI has turned out to be insufficient for storing all information from the UML models. Thus the tool vendors typically extend and/or modify the language so introduce their own XMI dialect. This, in turn, means that the tool interoperability is sacrified.

In this paper we discuss a method and a tool called DTD-compaper for exploring differences in exchange formats. DTD-compaper can, in general, be used to identify differences in grammars of XML-based languages. Further, we discuss three different case studies in which we used DTD-comparer. We first compare few commonly used XMI dialects. We further use the tool for comparing different versions of the Graph eXchange Language (GXL).

1. Introduction

Extensible Markup Languages (XML) [10] are widely used in software forward and reverse engineering, e.g., for supporting tool interoperability. Being a metalanguage, XML provides a standard way to markup data with strucutral and meta information. This, in turn, means that various XML-based languages can be processed with standard tools. There is an obvious need, for instance, for tool interoperability among software development tools supporting the current de facto OMG standard Unified Modeling Language (UML) [3]. Therefore OMG has also developed an XML Metadata Interchange (XMI) language that intends to provide *a* standard way for users to exchange any kind of metadata that can be expressed using UML, or more generally, using the OMG's Meta-Object Facility (MOF) specification [3]. Most commonly used CASE-tools support XMI as a standard way to exchange UML models. Such tools include, e.g., Rational Rose [4] and Together [5]. Also, several implementations of UML, offered as libraries, are available for CASE-tool developers. Currently, a popular one is Novosoft UML (NSUML) [9] Java library, which also provides support for XMI. NSUML is used, e.g., in ArgoUML [2].

In software maintenance and reverse engineering, the need for tool interoperability is equally relevant. In addition to XMI, other exchange formats, e.g., the Graph eXchange Language (GXL) [8], have been used to support tool interoperability. GXL is supported by various reverse engineering tools.

The grammar of an XML-based language can be defined using a Document Type Definition (DTD). Tools used for processing XML-documents use DTDs to validate the particular documents in question. For instance, an XMI DTD can be used to check whether an example XMI-document uses proper elements in a proper way, i.e., if it meets the contraints specified in the XMI DTD. In practice, however, XMI DTD has many dialects. This is due to the fact that in many cases XMI has turned out to be insufficient for storing all information from the UML models. For instance, OMG's XMI does not support representational information (i.e., layouts). Thus the tool vendors typically extend and/or modify the language, providing their own dialect. This, in turn, means that the real tool interoperability is sacrified.

Analyzing and understanding the differences among DTDs is challenging due to the format used in DTDs and

Proceedings of the Seventh European Conference On Software Maintenance And Reengineering (CSMR'03)
0-7695-1902-4/03 $17.00 © 2003 IEEE

due to the lack of documentation. Only the information captured in parts of the XML-documents understood by both CASE tools can be exchanged. Therefore, the problem of data and metadata loss arises in the exchange of UML-models among different CASE-tools that employ different XMI DTDs.

The motivation for our research originates from a practical need to find differences between two XMI dialects. We noticed that identifying the differences in grammars defined in DTDs is not straightforward. Textual comparison is not a good solution in practise. For instance, the DTDs may seem textually different, yet beeing similar from the grammar point of view, i.e., allowing similar structures in XML documents. Moreover, the results of textual comparisons would be difficult to read and interprete.

In this paper we propose an approach and a tool called DTD-comparer to identify the differences between two DTDs. Such differences are difficult to noticed with a human eye. We use a data model that is revised from the one proposed by Su et al. [1] for representing DTDs. We have modified the original data model so that the new data model fits the DTD comparison as well. The differences may occur, for instance, in the names of elements used or in the attributes of the elements. Further, elements may be structured having subelements. The differences may also occur in the way grouping is done or in the multiplicity of the subelements (i.e., how many times they may occur). We separate these cases to make the results easy to read and understood.

The tool can be used, e.g., to find differences among various dialects of a specific XML-based language. Based on the differences, conclusions on the degree of interoperability among CASE-tools supporting that language can be made. We have applied the tool primarily to compare various commonly used XMI dialects. For that purpose, the tool provides an additional dialog that allows the user to focus on specific aspects of the XMI DTDs. For instance, the user can compare the behavioral elements only. The results of comparisons are presented in this paper. We also discuss another application area of the tool, namely, comparing different versions of the same file or data exchange format.

In Section 2, we give an overview of the revised DTD data model. In Section 3 we discuss the method used to compare two DTDs and in Section 4 we discuss the DTD-comparer tool itself. In Section 5, we walk through some examples of using DTD-comparer for comparing DTDs. Finally, concluding remarks are given and future work is discussed in Section 6.

2. Data Model

The formal grammar of an XML language can be given as a DTD. DTD defines legal building blocks (elements, attributes, entities, and build-in nodes) for a specific XML language. In a DTD, elements are the main building blocks that can contain subelements. In a DTD fashion, we categorize subelements into the following groups according to the type of information they include: empty subelements (EMPTY), subelements containing character data only (#PCDATA), and any other data (ANY). For every subelement, DTD uses specific characters (?,*,+) to represent multiplicity in an Extended Backus-Naur From (EBNF) [10] fashion. These characters specify the following cases: a subelement may or may not occur once (?), a subelement either does not occur or may occur multiple times (*), and a subelement always occurs once or multiple times (+). Further, symbols "," and "|" are used in DTDs to indicate how subelements are grouped inside a parent element. A comma (",") separates two sublements that occur subsequently, while a vertical bar ("|") separates two choices. Consider, for example, the following expression:

<!ELEMENT XMI (XMI.header?, XMI.content?, XMI.difference*, XMI.extensions*) >.

In the above example, element *XMI* includes four subelements in the following order: *XMI.header*, *XMI.content*, *XMI.difference*, and *XMI.extensions*. *XMI.header* and *XMI.content* elements may or may not occur once inside the XMI element itself. *XMI.difference* and *XMI.Extensions* elements are optional as well but may also occur multiple times. Elements may have attributes that provide additional information. Entities, in turn, define shortcuts for common text.

In the XEM DTD data model (XEMDM) [1], a DTD is modeled as a graph G=(N,p,l) where *N* is the set of nodes, *p* is called a parent function representing the edges in the graph, and *l* is the labeling function representing the properties of a node as tuple. The properties include the name of the node and other properties, if any. There are following three categories of nodes.

1. Tag nodes
   (a) An element node: Every element node in a DTD is a tag node.
   (b) An attribute node: Attributes belonging to one element in a DTD form one attribute node that is connected to the corresponding element node.
2. Constraint nodes

(a) A group node: Group nodes (“,” and “|”) represent how direct subelements of an element are grouped together.
(b) A quantifier node: A quantifier node represents how many times a subelement occurs inside the parent element. The types of quantifier nodes are “*”, “+”, and “?”. Each quantifier node in the graph has only one child, namely, the node representing the subelement in question.

3. Build-in nodes
   (a) Root node: the entry of the DTD graph.
   (b) Primitive data type node PCDATA. It is XML-specific rather than application-specific.

We propose a revised DTD data model (called REVIDM in the sequel) that depicts DTD as graph G’=(T, C, M, B, E), where T is the set of *tag nodes*, C is the set of *constraint nodes*, M stands for the set of *multiplicity nodes*, B is the set of *build-in nodes*, and E is the set of directed edges between the nodes describing the structure of the corresponding DTD in the obvious way.

In G’, a tag node corresponds to an element of N in XEMDM. The differences between T and N concern attributes. In REVIDM, an attribute is no longer represented as a node of its own. Instead, attributes are included in the element node as an appendant. We chose this approach, since presenting attributes as a separate node is not helpful in DTD comparison and because representing them as separate nodes increases the size of the graph radically.

The development of DTD-comparer has been motivated by practical need for comparing different XMI dialects. For allowing the user to focus on specific aspects of the XMI DTDs only, the data model used is extended with one additional attribute, attached to a tag node and indicating the aspect in question. For example, “UML:Foundation” attached to a tag node means that this tag node belongs to the “UML:Foundation” structure.

The constraint nodes in G’ (set C) correspond to group nodes in G. We have extended the types of group nodes in G (“,” and “|”) with a flag “£” that represents a case where there is only one subelement inside the parent tag node. The extra flag “£” is used for comparison purposes only.

The multiplicity nodes in G’ (set M) correspond to quantifier nodes in G. However, we separate a quantifier node from a constraint node and add one more type “$”, which means that there is exactly one occurrence of a subelement. Again, “$” is used for comparison purposes only.

The types of build-in nodes are also extended from the one in G. In addition to “#PCDATA”, we use types “ANY” and “EMPTY” to indicate any other data type and an empty subelement, respectively.

For the purpose of comparing the order of nodes, we add an index number called “Order-Index” upon each edge from a constraint node to a multiplicity node. In the actual implementation of DTD-comparer, Order-Index has not yet been taken into account. Supporting that belongs to our future work.

In REVIDM, we introduce a new concept of a *relationship*. In one element definition in a DTD, relationships describe how the element to be defined depends on other (sub)elements used in the definition. In G’, a relationship is a path between tag nodes defined with the help of possibly several constraint nodes but only one multiplicity node. We define a relationship $R_{t1,t2}$ in (TxT) as follows: if there exist constraint nodes cm_1, cm_2,..., $cm_{n-1} \in C$, a multiplicity node $cm_n \in M$, and two tag nodes t_1, $t_2 \in T$ so that a sequence $R_{t1,t2}= (t_1,cm_1), (cm_1,cm_2),..., (cm_n,t_2)$ of pairs of nodes forms a path from t_1 to t_2 , then $R_{t1,t2}$ is a *relationship* between tags t_1 and t_2. In the DTD, $t_1 \in T$ corresponds to the element to be defined and $t_2 \in T \cup B$ corresponds to one of the subelements of t_1.

Consider, for example, a fraction of OMG’s UML1.3 XMI1.1 DTD in Figure 1 and the corresponding G’ in the Figure 2. The different kinds of nodes in G’ are illustrated using different shapes. The relationships are shown as arcs. For instance, the relationship between *XMI* and *XMI.header* elements in Figure 2 can be represented as a collection consisting of “,” and “?”.

```
…
<!ELEMENT XMI (XMI.header?, XMI.content?, XMI.difference*,
        XMI.extensions*) >
<!ATTLIST XMI  xmi.version CDATA #FIXED "1.1"
        timestamp CDATA #IMPLIED
        verified (true | false) #IMPLIED
>
<!ELEMENT XMI.header (XMI.documentation?, XMI.model*, XMI.metamodel*, XMI.metametamodel*, XMI.import*) >
<!ELEMENT XMI.documentation (#PCDATA | XMI.owner | XMI.contact | XMI.longDescription | XMI.shortDescription |
XMI.exporter | XMI.exporterVersion |XMI.notice)* >
<!ELEMENT XMI.owner ANY >
```

…
<!ELEMENT XMI.notice ANY >
<!ELEMENT XMI.model ANY >
…

Figure 1. A part of the OMG UML1.3 XMI1.1 DTD

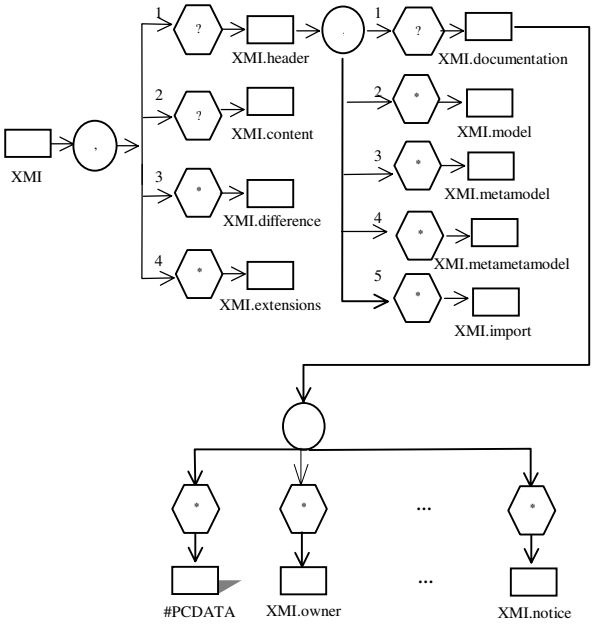

Figure 2. A graph G' representing the DTD in Figure 1. The tag nodes are shown as rectangles, constraint nodes as circles, multiplicity nodes as hexagons, build-in nodes as shadowed rectangles, and relationships as arcs. The Index-Order numbers are attached to the relationships from a constraint node to a multiplicity node.

Note that allowing multiple constraint nodes in a relationship is indeed needed because DTD specification allows subgrouping. Consider, for instance, the following simple element definition:

<!ELEMENT JJ (a*, (b?,c?))>

The corresponding graph representation is shown in Figure 3. A relationship between *JJ* and *b* elements, for instance, now consists of two constraint nodes and one multiplicity node: ",", ",", and "?".

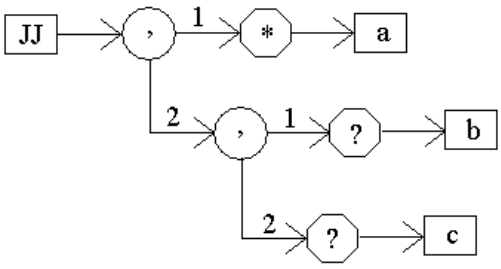

Figure 3. Graph presentation of a subgroup

In some cases there might be a need to be able to use many multiplicity nodes between tag nodes. For instance, subelements might be grouped so that in addition to their individual multiplicities, another shared multiplicity is applied to them. For instance, consider the following expression: <!ELEMENT JJ(a, (b?, c*)*)>. Since REVIDM permits only one multiplicity node to be associated with a tag node, we define operations to get the multiplicity of subelements in the cases described above. We first assume that a value range [0, +∞) represents "*", a value range [1, +∞) corresponds to "+", [0,1] is a value range representation for "?", and a value range [1,1] corresponds to "$". Then the final multiplicity of a tag node equals to the combination of the lowest lower bound and the highest higher bound. For instance, in order to get the multiplicity of b, we need to get the lowest lower bound and the highest higher bound of ranges [0, 1] and [0, +∞). Therefore, the value range [0, + ∞) represents the multiplicity of b. That is, the actual multiplicity of b is "*". Note that this simple way of combining multiplicities applies only when the order of the elements is ignored and only the actual number of occurrences is desired.

3. A method for comparing DTD specifications

The method for comparing two DTDs is based on the usage of the REVIDM data model. The differences are divided into three categories: tag node differences, relationship differences, and attribute differences. Assume T_A is the set of tag nodes of a XMI DTD graph A. Similarly, T_B is the set of tag nodes of a XMI DTD graph B. Then the tag node, relationship, and attribute differences are defined as follows.

(a) Tag node differences are obtained as symmetric differences of T_A and T_B. In other words, we calculate $(T_A - T_B) \cup (T_B - T_A)$. As a result, we get the set of elements belonging either to T_A or to T_B but not to both of them.

(b) To conclude the relationship differences we use bit sequences listed in Table 1. A relationship is composed from the bit sequences corresponding to constraint and multiplicity nodes associated with this relationship. Constraint nodes are represented using two bits and multiplicity nodes using three bits.

Table 1. Bit sequences corresponding to constraint and multiplicity nodes.

","	"|"	"£"	"*"	"+"	"?"	"$"
10	01	11	110	100	111	101

We define the bit sequence $B_{t1,t2}$ corresponding to a relationship $R_{t1,t2}$= (t_1,cm_1), (cm_1,cm_2),..., (cm_n,t_2) as follows: $B_{t1,t2}=Seq(cm_1)\cdot Seq(cm_2)\cdot\ldots\cdot Seq(cm_n)$, where *Seq* is the function that returns a bit sequence from constraint and multiplicity nodes according to the transformation rules presented in Table 1. The bit sequence of a relationship is unique because we always read the constraint nodes first and then the multiplicity node (2 bits, …, 2bits, 3 bits). For example, the relationship from *XMI* element to *XMI.header* element in Figure 2 is represented as 10111, which is a concatenation of 10 and 111.

The relationship difference is separated into two subcategories. Assume that t_1 and t_2 are tag nodes of $R_{t1,t2}$ in a DTD graph A, and t_3 and t_4 are tag nodes of $R_{t3,t4}$ in DTD graph B. Then the two subcategories of relationship differences are:

Category 1: $(R_A - R_B) \cup (R_B - R_A)$ and

Category 2: $B_{t1,t2} \wedge B_{t3,t4} \neq 0$.

The category 1 corresponds to tag node difference. It can be used to detect if two tag nodes are related in both DTDs. When counting relationship differences belonging to the category 2, a bitwise-exclusive-OR operation is applied. If the result of this operation is 0, namely, the two corresponding bits at each position are the same, then no relationship differences exist. Otherwise, the relationships differ. Category 2 can be used to detect if a relationship between two tag nodes (may exist in both DTDs) is similar. For instance, consider the following two element definitions in different DTDs:

```
<!ELEMENT JJ (a?,b?,c?)>
<!ELEMENT JJ ((a?|b?),d?)>
```

When applying the relationship difference operation of category 1, we would recognize, e.g., that a relationship between *JJ* and *c* in the former declaration does not exist in the latter declaration. When applying the relationship difference operation of category 2, we find out that the relationship between *JJ* and *a* is different in these two declarations. Note that this could not have been noticed when applying the category 1 operation, since there exists a relationship between *JJ* and *a* in both definitions.

(c) Attribute differences inspect the differences between attribute types and between their default values. Because attributes are included in tag nodes, we compare them one by one.

Since the DTDs to be compared are represented as graphs, the actual comparison of them is handled in a rather straightforward way by comparing the two corresponding graphs. In general, the similarity of context-free grammars is a more complex issue and is widely discussed in the literature [6,7,12].

4. Tool support

Based on the approach described in Section 3, we have developed a tool called DTD-comparer that can help the user to find out differences in two DTDs.

Figure 4 shows a corner of the main window of DTD-comparer. Buttons A and B in the tool bar are used for selecting the two DTD documents. A bull's eye opens another dialog, shown in Figure 5, for selecting difference categories. A button with label R is used to run the difference operations. Since DTD-comparer has been primarily used for comparing XMI DTDs, we have equipped the tool with an option that allows the user to make partial comparisons of two UML1.3 XMI1.1 DTDs. The user can therefore focus on selected parts of interest in the XMI DTDs. The rightmost button in the tool bar opens a dialog for selecting the focus points. This dialog is shown in Figure 6. The view in the main window in Figure 4 lists the differences according to the user's selections.

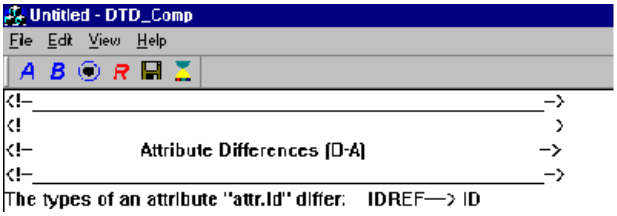

Figure 4. The main user interface of DTD-comparer

Figure 5. Difference category dialog

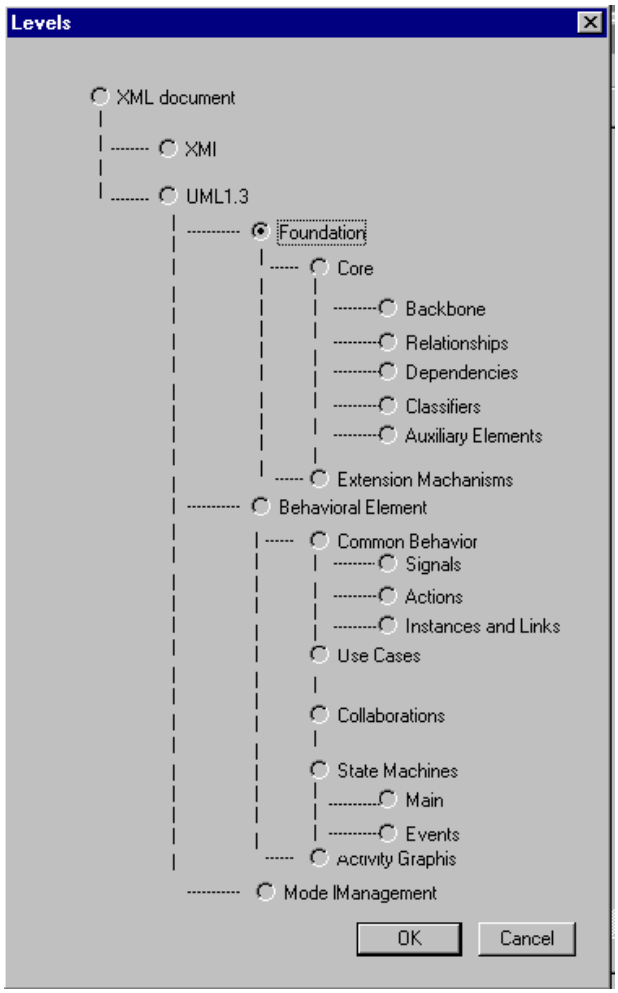

Figure 6. UML1.3 Hierarchy Dialog

5. Case studies

In the following examples, we use DTD-comparer to get DTD differences in three example cases. Element differences and relationship differences, as well as attribute differences, are detected.

5.1 Differences between the XMI1.1 DTDs of OMG and Rational Rose Extended UML1.3 XMI1.1 DTD

We first compare the Rational Rose Extended UML1.3 XMI1.1 DTD (called REDTD1.1 in the sequel) with the OMG UML1.3 XMI1.1 DTD (called OMGDTD1.1 in the sequel). Because of the length of XMI DTDs, we illustrate specific parts of them only. Figure 7 depicts a fraction of REDTD1.1 and Figure 8 shows a corresponding fraction of OMGDTD1.1. The fractions chosen illustrate the XMI specifications of an AssociationEnd concept. In UML, an association end is an endpoint of an association, which connects the association to a classifier[3].

```
…
<!-- ========= UML:AssociationEnd ========= -->
<!ELEMENT UML:AssociationEnd.isNavigable EMPTY>
<!ATTLIST UML:AssociationEnd.isNavigable xmi.value (true|false)
#REQUIRED>
<!ELEMENT UML:AssociationEnd.ordering EMPTY>
<!ATTLIST UML:AssociationEnd.ordering xmi.value
%UML:OrderingKind; #REQUIRED>
<!ELEMENT UML:AssociationEnd.aggregation EMPTY>
<!ATTLIST UML:AssociationEnd.aggregation xmi.value
%UML:AggregationKind; #REQUIRED>
<!ELEMENT UML:AssociationEnd.targetScope EMPTY>
<!ATTLIST UML:AssociationEnd.targetScope xmi.value
%UML:ScopeKind; #REQUIRED>
<!ELEMENT UML:AssociationEnd.multiplicity
(UML:Multiplicity)?>
<!ELEMENT UML:AssociationEnd.changeability EMPTY>
<!ATTLIST UML:AssociationEnd.changeability xmi.value
%UML:ChangeableKind; #REQUIRED>
<!ELEMENT UML:AssociationEnd.association (UML:Association)?>
<!ELEMENT UML:AssociationEnd.qualifier (UML:Attribute)*>
<!ELEMENT UML:AssociationEnd.type (UML:Classifier)?>
<!ELEMENT UML:AssociationEnd.specification (UML:Classifier)*>
<!ENTITY % UML:AssociationEndFeatures
'%UML:ModelElementFeatures; |
  UML:AssociationEnd.isNavigable |
  UML:AssociationEnd.ordering |
  UML:AssociationEnd.aggregation |
  UML:AssociationEnd.targetScope |
  UML:AssociationEnd.multiplicity |
  UML:AssociationEnd.changeability |
  …
  UML:AssociationEnd.specification'>
<!ENTITY % UML:AssociationEndAtts '%UML:ModelElementAtts;
  isNavigable (true|false) #IMPLIED
  ordering %UML:OrderingKind; #IMPLIED
  …
  changeability %UML:ChangeableKind; #IMPLIED
  …
  specification IDREFS #IMPLIED'>

…
```

Figure 7. A part of REDTD1.1 that describes the structure of an AssociateEnd

```
…
<!-- _____________________________________________ -->
<!--                                                - ->
<!-- CLASS: Foundation.Core.AssociationEnd          -->
<!--_____________________________________________-->
<!ELEMENT UML:AssociationEnd.isNavigable EMPTY >
<!ATTLIST UML:AssociationEnd.isNavigable
        xmi.value (false | true) #REQUIRED
>
<!ELEMENT UML:AssociationEnd.isOrdered EMPTY >
<!ATTLIST UML:AssociationEnd.isOrdered
        xmi.value (false | true) #REQUIRED>
<!ELEMENT UML:AssociationEnd.aggregation EMPTY >
<!ATTLIST UML:AssociationEnd.aggregation
        xmi.value (none | shared | composite) #REQUIRED
>
<!ELEMENT UML:AssociationEnd.multiplicity (#PCDATA |
XMI.reference)* >
<!ELEMENT UML:AssociationEnd.changeable EMPTY >
<!ATTLIST UML:AssociationEnd.changeable
        xmi.value (none | frozen | addOnly) #REQUIRED
>
<!ELEMENT UML:AssociationEnd.targetScope EMPTY >
<!ATTLIST UML:AssociationEnd.targetScope
        xmi.value (classifier | instance) #REQUIRED
>
<!ELEMENT UML:AssociationEnd.type (UML:Classifier)* >
<!ELEMENT UML:AssociationEnd.specification (UML:Classifier)* >
<!ELEMENT UML:AssociationEnd.association (UML:Association)*
>
<!ELEMENT UML:AssociationEnd.linkEnd (UML:LinkEnd)* >
<!ELEMENT UML:AssociationEnd.associationEndRole
                        (UML:AssociationEndRole)* >
<!ELEMENT UML:AssociationEnd (UML:ModelElement.name|
UML:ModelElement.visibility |
UML:AssociationEnd.isNavigable |
UML:AssociationEnd.isOrdered |
                UML:AssociationEnd.aggregation |
                UML:AssociationEnd.multiplicity |
                UML:AssociationEnd.changeable |
                UML:AssociationEnd.targetScope |
XMI.extension | UML:ModelElement.binding|
UML:ModelElement.template |
UML:ModelElement.templateParameter |
…
    )* >
…
```

Figure 8. A part of OMGDTD1.1 describing the structure of an AssociationEnd

After the selection of DTD documents, we first select the operation $(T_A - T_B) \cup (T_B - T_A)$ in the difference category dialog and "XML.document" in UML hierarchy dialog. As a result, we get tag node differences existing in the documents. Because of the length of the result, Figure 9 lists only some tag nodes that exist in REDTD1.1 but not in OMGDTD1.1, and vice versa. For instance, *UML:AssociationEnd.changeability* is a legal element in REDTD1.1 but not in OMGDTD1.1. The corresponding element in OMGDTD1.1 is named *UML:AssociationEnd.changeable*.

```
<!--_____________________________________________ -->
<!--            Tag nodes difference(A-B)          -->
<!--  A: Rational extended DTD, B: OMG DTD         -->
<!--_____________________________________________  -->
UML:Abstraction
…
UML:AssociationEnd.changeability
UML:AssociationEnd.ordering
…
UML:Message.message3
UML:Message.message4
UML:MessageAtts
…
<!--_____________________________________________ -->
<!--            Tag nodes difference(B-A)          -->
<!--  A: Rational extended DTD, B: OMG DTD         -->
<!--_____________________________________________  -->
UML:Action.message
…
UML:AssociationEnd.changeable
UML:AssociationEnd.isOrdered
…
UML:Message.message
UML:Message.message2
UML:MessageInstance
…
```

Figure 9. Element differences between the OMGDTD1.1 and REDTD1.1 generated by DTD-comparer.

By studying the results produced by DTD-compaper, we noticed several differences between REDTD1.1 and OMGDTD1.1. With domain knowledge on UML, we analysed the differences and categorized as follows:

- Some differences exist in the names of tag nodes. For instance: for an OMGDTD1.1 tag *UML:AssociationEnd.isOrdered* a correspoding tag in REDTD1.1 is named *UML:AssociationEnd.ordering* in REDTD1.1. Such slight differences in element namings are somewhat surprising.
- REDTD1.1 changes the format of some behavioral XMI elements, e.g., concerning sequence and collaboration diagrams. For example, REDTD has *UML:Message.message3* and *UML:Message.message4* elements, while OMGDTD1.1 has definitions for *UML:Message.message* and *UML:Message.message2*. In practise this means, for instance, that exchanging sequence diagrams between Rational Rose and a tool supporting OMGDTD1.1 is not possible without some extra modifications.

- The biggest differences between OMGDTD1.1 and REDTD1.1 originate from the lack of capabilities to express representational information in OMGDTD1.1. Therefore, REDTD1.1 includes extensions to allow Rational Rose users to store diagram layouts in XMI as well.
- In general, REDTD1.1 has more XMI element definitions than OMGDTD1.1. Also, REDTD1.1 is extended to include the definition of *UML:SubactivityState*, which does not exist in OMGDTD1.1.

Next we generated relationship differences of the type of category 2. Parts of the results are listed in Figure 10. Arcs "--->" in Figure 10 are indicators of relationship paths. Two consecutive lines grouped together compose one comparative pair where the first line shows one relationship in REDTD1.1 and the second line illustrates a corresponding relationship in OMGDTD1.1. Consider, for instance, the first pair in Figure 10. The first line in the pair means there is a relationship from *UML:AssociationEnd.association* element to *UML:Association* element through one constraint node "£" and one multiplicity node "?". The meaning of the second line is similar to the first line, except of the difference of the multiplicity node.

```
<!--_____________________________________________________-->
<!--              Relationship difference                -->
<!--_____________________________________________________-->

UML:AssociationEnd.association --->£ ---> ?--->UML:Association
UML:AssociationEnd.association --->£ ---> *--->UML:Association

UML:AssociationEnd.type --->£ ---> ?--->UML:Classifier
UML:AssociationEnd.type --->£ ---> *--->UML:Classifier

UML:Attribute.associationEnd --->£ ---> ?--->UML:AssociationEnd
UML:Attribute.associationEnd --->£ ---> *--->UML:AssociationEnd

UML:Attribute.initialValue --->| ---> ?--->UML:BooleanExpression
UML:Attribute.initialValue --->| ---> $--->UML:BooleanExpression

UML:Attribute.initialValue --->| ---> ?--->UML:Expression
UML:Attribute.initialValue --->| ---> $--->UML:Expression

UML:Attribute.initialValue --->| ---> ?--->UML:ObjectSetExpression
UML:Attribute.initialValue --->| ---> $--->UML:ObjectSetExpression

UML:Attribute.initialValue --->| ---> ?--->UML:ProcedureExpression
UML:Attribute.initialValue --->| ---> $--->UML:ProcedureExpression

UML:Attribute.initialValue --->| ---> ?--->UML:TimeExpression
UML:Attribute.initialValue --->| ---> $--->UML:TimeExpression

UML:Binding.argument --->£ ---> *--->UML:ModelElement
UML:Binding.argument --->| ---> *--->UML:ModelElement

UML:Constraint.body --->£ ---> ?--->UML:BooleanExpression
UML:Constraint.body --->£ ---> $--->UML:BooleanExpression

UML:Feature.owner --->£ ---> ?--->UML:Classifier
UML:Feature.owner --->£ ---> *--->UML:Classifier
```

Figure 10. Relationship differences (category 2) between the REDTD1.1 and OMGDTD1.1, generated by DTD-comparer.

From the results we can notice that some multiplicity nodes are changed. For example, compare "UML:Feature.owner--->£ ---> ?--->UML:Classifier" with "UML:Feature.owner --->£ ---> *--->UML:Classifier". The multiplicity associated with the tag node is changed from "?" to "*". Hence we need pay attention to *UML:Feature.owner* when storing UML models in XMI using different CASE-tools. The REDDTD1.1 permits UML:Classifier to occur at most once, but OMGDTD1.1 permits it occur more than one time.

5.2 Differences between the OMG UML1.3 XMI1.0 DTD and the IBM UML1.1 XMI1.0 DTD

Next we compare IBM UML1.1 XMI1.0 DTD (called IBMDTD1.0 in the sequel) used in the IBM's XMI Toolkit [11] with the corresponding (older) version of OMG's XMI specification, namely OMG UML1.3 XMI1.0 DTD (called OMGDTD1.0 in the sequel).

By analysing the results, we conclude the main differences to be the following:

- Lots of differences exist between elements belonging to a "state machines" package and a "Common Behavior" package. For instance, in OMGDTD1.0, more element definitions are included in "Action" and "Common Behavior" packages compared to IBMDTD1.0.
- OMGDTD1.0 makes UML use cases extensible by adding element definitions. E.g., an element definition "Behavioral_Element.Use_Cases.Extend.extension" is included in OMGDTD1.0 but not in IBMDTD1.0.
- Some differences exist in the "Data_Type" package. IBMDTD1.0 has the data type of "Enumeration" and "Geometry" that do not exist in OMGDTD1.0.
- In the "Model_Management" package, OMGDTD1.0 defines an "ElementImport". It is not included in IBMDTD1.0. Instead, IBMDTD has a definition of "ElementReference".

- OMGDTD1.0 contains a series of "Activity_Graphs" element definition, which does not exist in IBMDTD1.0.

5.3 Differences between GXL1.0.1 and GXL 1.0

In our last example, we compare different versions of the Graph Exchange Format (GXL) [8]. GXL is developed to offer an adaptable and flexible means to support interoperability between graph-based tools. Especially, GXL was developed to enable interoperability between software reengineering tools and components, such as code extractors (parsers), analyzers and visualizers. GXL is publicly available at http://www.gupro.de/GXL/. We compared the two version of GXL by detecting tag node differences, relationship differences, and attribute differences. As a result, we found no differences in tag nodes nor in relationships (as stated in the GXL documentation). The only differences were found in attributes. The only difference listed in Figure 11 means that element "attr" has an attribute "id", the type of which is "IDREF" in GXL1.0 but has been changed to a type "id" in GXL1.0.1.

```
<!—_____________________________________________-->
<!--              Attribute difference           -->
<!--_____________________________________________-->
The types of an attribute "attr.id" differ:    IDREF----> ID
```

Figure 11. Attribute Differences between GXL1.0 and GXL 1.0.1.

5.4 Analysis of the case studies

For estimating the amount of differences in case studies discussed in Sections 5.1, 5.2, and 5.3, we define four kinds of measures corresponding to each category of difference. These measures use a function *Count()* that returns the number of elements in a set. The measures are the following:

(a) Tag node difference = $\text{Count}((T_A - T_B) \cup (T_B - T_A)) / \text{Count}(T_A \cup T_B)$

(b) Relationship difference of category 1 = $\text{Count}((R_A - R_B) \cup (R_B - R_A)) / \text{Count}(R_A \cup R_B)$

(c) Relationship difference of category 2 = $\text{Count}(R_{D2}) / \text{Count}(R_A \cap R_B)$, where R_{D2} is the set of relationships that are different according to the relationship difference operation of category 2

(d) Attribute difference = $\text{Count}(T_C) / \text{Count}(T_A \cap T_B)$, where T_C is the set tag nodes that include attribute differences in the corresponding graphs A and B.

Table 2. The difference measures are used to estimate the degree of tag, relatioship, and attribute differences in three difference case studies. Cases 1, 2, and 3 correspond to case studies discussed in Sections 5.1, 5.2, and 5.3.

Cases	Tag node diff.	Relationship diff. of category 1	Relationship diff. of category 2	Attribute diff.
Case 1	51.09%	61.92%	5.77%	32.22%
Case 2	50%	52.12%	24.10%	0%
Case 3	0%	0%	0%	5.55%

The cases 1 and 2 in Table 2 both correspond to case studies in which XMI dialects were compared. In the former case dialects of XMI1.1 DTDs where compared, while in the latter case dialects of XMI1.0 DTDs where examined. From Table 2 we notice that when tag differences are examined, in case 2 the two dialects are more similar than in case 1. The same applies to relationship differences of category 1 (differences in subelement references). It is also worth noticing that amount of variation seems high. To large extent, this is explained by the extensions made, e.g., to allow presenting representation information (i.e., layouts). However, there are much less relationship differences of category 2 in case 1 than in case 2. This means that for relationships that occur between two elements in both documents compared, in case 2 there are more variations in the types of these relationships than in case 1. Finally, in case 2, there are no attribute differences.

The case 3 corresponds to the comparison of two versions of GXL. From the Table 2 we can conclude that no tag node nor relationship differences occur. The only change made concerns attributes.

Without domain knowledge, the severity of differences from the point of view of tool interoperability cannot be directly concluded from the outputs of DTD-comparer. For instance, when UML CASE-tools are considered, the differences in view information (e.g., layouts) can be considered less harmful than differences in model information since individual tools may have layout algorithms that can be applied after loading the model in the tool.

6. Discussion

In UML-based CASE-tools, tool interoperability is often supporting by allowing the user to store the models in XMI format. As shown in this paper, the used XMI dialects differ considerably in practise. Therefore, data and metadata loss is in many cases inevitable when exchanging models among different tools leaning on XMI only. The maintenance of the design thus becomes difficult in such heterogenous environments. The same problem occurs, for instance, with software exploration tools when different versions of XML-based exchange formats are used. Recognizing differences in DTDs, which define the grammar of the XML-based language used, help the CASE-tool users to be aware of the amount and the type of data and metadata loss encountered during the information exchange among the tools. In this paper, we present an approach and a prototype tool called DTD-comparer that can used get the differences between two DTDs.

DTD-comparer is able to recognize differences between names of elements defined, relationships between two elements, and attribute definitions. However, the comparison of subelement sequences has not yet been implemented in DTD-comparer. For instance, differences between "(a,b,c)" and "(a,c,b)" are not currently recognized. This problem can, however, be solved using Order-Index discussed in Section 2. Implementing this is part of our future work.

When developing and using DTD-comparer we have noticed that it would be very useful to find a convenient way of repairing DTDs according to the differences found, so that CASE-tool interoperability could be better supported. For example, slight name conflicts between corresponding elements cause unnecessary loss of data during the exchange of UML models. Yet, domain knowledge is required for ensuring that two differently named elements are indeed corresponding.

Acknowledgements

This research has been financially supported by Nokia. The authors would like to thank Kai Koskimies and Erkki Mäkinen for their valuable comments.

References

[1] S. Hong, D. Kramer, L. Chen, K. Claypool, and E.A. Rundensteiner, XEM: managing the evolution of XML documents, In *The Proc. of the Eleventh International Workshop on Research Issues in Data Engineering*, 2001, pp. 103-110.

[2] Tigris, Project Home Page, http://argouml.tigris.org/, 2002.

[3] Object Management Group, ttp://www.omg.org/uml/, 2002.

[4] Rational Software Corporation, Rose Enterprise Edition, 2002, http://www.rational.com.

[5] TogetherSoft Corporation, Together 5, 2201, `http://www.togethersoft.com`.

[6] H. B. Hunt III, D. J. Rosenkrantz, T.G. Szymanski, On the equivalence, containment, and covering problems for the regular and context-free language, *Journal of Computer and Systems Sciences* 12, 2, 1976, pp. 222-268.

[7] H. B. Hunt III, D. J. Rosenkrantz, On equivalence and containment problems for formal languages, *Journal of the ACM*, 24, 3, 1977, pp. 387-396.

[8] R. Holt, A. Winter, A. Schürr, GXL: Toward a Standard Exchange Format, In *Proc. of the 7th Working Conference on Reverse Engineering (WCRE)*, 2000, pp. 162-171.

[9] SourceForge.net, Novosoft UML library for Java, http://sourceforge.net/projects/nsuml, 2002.

[10] W3C, Extensible Markup Language (XML) 1.0 Specification, http://www.w3.org.

[11] IBM AlphaWorks, an XMI Toolkit, http://www.alphaworks.ibm.com/tech/xmitoolk.

[12] D. J. Rosenkrantz and H.B. Hunt III: The complexity of structural containment and equivalence, In Jeffrey D. Ullman (ed.): *Theoretical Studies in Computer Science*, Academic Press, Boston,1992, pp. 101-132.