Differences between Versions of UML Diagrams

Dirk Ohst
ohst@informatik.uni-siegen.de

Michael Welle
welle@informatik.uni-siegen.de

Udo Kelter
kelter@informatik.uni-siegen.de

Praktische Informatik
Fachbereich Elektrotechnik und Informatik
Universitaet Siegen, D-57076 Siegen

ABSTRACT

This paper addresses the problem of how to detect and visualise differences between versions of UML documents such as class or object diagrams. Our basic approach for showing the differences between two documents is to use a unified document which contains the common and specific parts of both base documents; the specific parts are highlighted. The main problems are (a) how to abstract from modifications done to the layout and other (document type-specific) details which are considered irrelevant; (b) how to deal with structural changes such as the shifting of an operation from one class to another; (c) how to reduce the amount of highlighted information. Our approach is based on the assumption that software documents are modelled in a fine-grained way, i.e. they are stored as syntax trees in XML files or in a repository system, and that the version management system supports fine-grained data. Our difference computation algorithm detects structural changes and enables their appropriate visualisation. Highlighting can be restricted on the basis of the types of the elements and on the basis of the revision history, e.g. only changes which occurred during a particular editing session are highlighted.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering (CASE); D.2.9 [**Management**]: Software configuration management; D.3.2 [**Language Classifications**]: Design languages

General Terms

Management, Documentation, Design

Keywords

fine-grained data model, versions, configuration, design transaction, software engineering environments, differences, UML diagrams

ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.

1. INTRODUCTION

Software configuration management (SCM) is an indispensable part of high-quality software development processes. SCM is a well established and common practice in the later phases of software development, notably during programming and integration. An advantage of using SCM systems is that one can create versions of a document and detect the differences between them. SCM is a less common practice during the early phases, i.e. analysis and design. Only a few SCM systems support versioning of analysis or design documents and can visualise the differences between such documents.

Existing SCM systems are not well suited for the detection and visualisation of differences between documents in the early phases. A large number of SCM systems and concepts are available [6], however most of them (incl. systems such as RCS, CVS and SCCS) only work with text files, i.e. files containing lines of text in pretty-printed format. In contrast to this, software documents in the early phases are not text, but diagrams (e.g. different types of UML diagrams). Diagrams are also mostly stored in files. In the case of binary formats, conventional SCM systems cannot detect differences at all. If the diagrams are stored in printable formats (e.g. XML formats), other problems arise. For example, in a file representing a class diagram, each class might be represented by a few lines of text. The order of these sections of text is irrelevant. The position where a class symbol appears in the diagram is explicitly stored in layout data. Therefore, diagram editors can store the sections representing the classes or other diagram elements in arbitrary order. After a short editing session, the editor can completely reshuffle the files content and cause a large number of significant textual differences. Different file contents can actually represent the "same" diagram. Conventional SCM systems cannot reasonably handle this situation. Tools are needed which take the logical structure of the document into account.

The visualisation of the differences is another point which needs to be addressed. Differences in textual data can easily be shown by arranging corresponding blocks side-by-side in two adjacent columns and highlighting the differences. Diagrams cannot be reasonable displayed in such manner. One has to distinguish between differences in the layout and differences in the (conceptual) diagram elements. Hence usual methods for displaying differences are not applicable.

Another issue is the number of differences between two documents. There are usually groups of small modifications

in a document which belong together. These modifications lead to a large number of differences between "distant" versions of a document. As a result of this, developers have problems identifying the context of single differences.

This paper presents concepts for detecting and visualising differences between versions of UML documents and methods to reduce the amount of highlighted differences. One basic assumption is that the documents are modelled in a fine-grained way, one feature of this approach is that it can distinguish between objects deleted and later created again and those shifted (e. g. an operation is shifted from one class to another).

The basic concepts presented here are applicable independently of the data management system (XML files, relational or object-oriented databases). Our implementation uses a repository system known as H-PCTE [8], a structurally object-oriented DBMS.

The rest of this paper is organised as follows. In section 2 we discuss the characteristics and kinds of differences between versions of UML diagrams. In section 3 we then presents our concept of how to visualise differences between diagrams. The computation of differences is described briefly in section 4. Section 5 addresses the problem of reducing the complexity of the highlighted differences. Section 6 gives a short overview about other concepts of detecting differences between documents. Concluding remarks are given in section 7.

2. DIFFERENCES BETWEEN UML DIAGRAMS

This section discusses and classifies the types of differences which occur in different types of UML diagrams.

2.1 Mental vs. Physical Models of UML Diagrams

What are differences between diagrams? We approach this from a developer's point of view: every detail which a developer might perceive as a difference between two versions of a diagram must be covered. In order to describe these details, we must assume a mental model in which diagrams are represented and which enables a developer to describe the differences.

In order to further structure this discussion, we distinguish between a diagrams layout and model data. Roughly speaking, *layout data* are the positions and the sizes of the nodes in a diagram or the positions of corner-points in edges, essentially everything which would be considered irrelevant in the textual representation of the diagrams content. The remaining data are *model data*, they express the semantics of the document. The UML specification [15] similarly distinguishes between semantics and notation in chapter 2 and chapter 3.

Chapter 2 of [15] defines the abstract syntax of complete and correct diagrams using class diagrams and other means; the model data of a diagram can thus be thought of as a set of instances of the types defined in these class diagrams. Such a model is called *fine-grained* because it represents all the details of the syntactical structure of the document.

UML tools, notably diagram editors, can use a similar data structure for representing a document at run-time in main memory, and a similar schema to store documents in an object oriented database. However, the structure of these "physical models" depends a lot on the programming language or the database model; moreover, tools must be able to handle incomplete and incorrect documents. In any case, the physical model of a document must be able to represent the more abstract mental model of a document.

Chapter 3 of [15] defines many details about how abstract documents should be represented on screen, and leaves many details open; it is at the discretion of tool developer as to how these details are designed. Consequently, the structure of layout data (both in a mental model and a physical model) may be specific for a tool. This leads to the (unpleasant) observation that the notion of a difference between two documents may be tool-dependent.

Differences in layout are mostly considered irrelevant for the developers. For example, the creation of another corner-point in an edge in a class diagram is normally not considered a "real change".

The rest of this section, analyses UML diagrams and differences between them at the mental model level; later, when the actual computation of differences is discussed, a physical model will be assumed.

2.2 Characteristics of UML Diagrams

This section discusses facets of UML diagrams which can be relevant for differences.

2.2.1 Semantics of the Layout

The diagram types of the UML can be divided into two categories, namely diagrams whose layouts are semantically relevant and those whose layouts are considered irrelevant. Sequence diagrams belong to the first category[1]. All other diagram types belong to the other category, which is the only one which we will discuss in detail in this paper.

2.2.2 Structure of Diagrams

All diagrams except sequence diagrams represent a graph. To be specific they contain nodes and relationships connecting them.

The node types depend on the diagram type, e.g. in class diagrams, there are classes, interfaces and comments, in activity diagrams there are actions, forks and joins, etc.

The structure of a node depends on its type. Most node types have only a few attributes (mostly only their name), while some node types have a complex structure. Examples of nodes with names only are forks and joins in activity diagrams. Examples of nodes with a name and further attributes are classes, operations, object attributes, states and transitions. Names also appear as attributes of relationships, e.g. the name of a role.

2.2.3 Kinds of Attributes

Most attributes are simple attributes, e.g. names, properties like {`abstract`} of a class, etc.

Some attributes, e.g. multiplicities, have an internal structure and could be regarded as multi-valued or even a complex component. However, for the purposes of showing differences, it is more sensible to consider them as strings.

The list of attributes or operations of a class, the set of attribute assignments on an object, the list of internal actions of a state and similar examples can be regarded as

[1]The order of operation calls is represented by the graphical arrangement of the corresponding arrows. Changes of the layout can thus change the order of operation calls.

multi-valued attributes or, if the order is relevant, as list-valued attributes. In most cases, each entry of the list has a fine-grained structure; however, it is normally more sensible to consider them as strings.

Another kind of attribute can (or should) be regarded as a reference to an attribute in another diagram. For example, the objects and operations in object diagrams or collaboration diagrams contain the names of the corresponding class or operation. These names should be regarded as mirrored values of attributes of the class or operation. The collaboration diagram contains only references to the original attributes. The following diagram types can also have references into other diagrams: state chart and activity diagrams (the events or operations), deployment diagrams and component diagrams (the interfaces).

Yet another kind of attributes are derived attributes. The only relevant example are sequence numbers in collaboration diagrams (and in sequence diagrams, if they are shown there). The mental model of these attributes depends a lot on how editors are assumed to work: If an editor for collaboration diagrams supports the insertion of a new operation call into an existing sequence of operation calls, with the automatic renumbering of all affected sequence numbers, then the mental model of the operation sequence is just a sequence, and the insertion or deletion of an element are the basic modifications of the sequence. The sequence numbers are only derived values. On the other hand, if sequence numbers can (or must) be edited directly by a developer, they are normal simple attributes.

2.2.4 Complex Nodes

Some node types have a complex structure. e.g. they can group other nodes or they can contain an entire subdiagram. Examples are package diagrams (which contain packages, classes and components), deployment diagrams (containing components) or states (may contain a state chart). In fact, the contents of such a node is often shown as a separate diagram.

2.3 Classification of Differences

Diagrams can be changed in several ways. Parts of a diagram can be created, deleted or shifted. The individual differences (or deltas) between two versions of one diagram can be classified as follows:

- intra-node differences: these are differences within a node of the graph, for example, the name of a state in a state chart might change, or an additional attribute could be added/deleted from a class
- differences in the structure of the graph, for example, nodes are shifted, created or removed.

The creation and the deletion of nodes result in differences which can be easily visualised, see section 3. The other changes will be discussed in more detail. As already mentioned above, we will not consider differences in sequence diagrams.

2.3.1 Intra-Node Differences

Intra-node differences are differences between attributes of two corresponding nodes in different versions of a diagram. Intra-node differences are not considered differences between the node as a whole; instead the node as such is considered a "common" part of both diagrams and the internal differences are shown inside the node symbol. The details depend on the kind of attribute:

- simple (single-valued) attributes: the only possible differences are changed values
- multi-valued or list-valued attributes: possible differences are created or deleted elements, e.g. an operation is added in a class
- reference attributes: the only difference considered relevant is a modification of the reference. A change of the value of the referenced attribute is not considered relevant.

2.3.2 Shifting Elements within a Diagram

Diagram elements cannot only be created or deleted, they can also be shifted within a diagram. There are four kinds of shifts:

- modifications to the layout: shifting classes around in the diagram
- structural shifts: one end of an edge representing a relationship between diagram elements is shifted from one element to another; e.g. a class `C` which is subclass of class `A` can be made a subclass of class `B` in this way
- inter-node shifts: e.g. shifting operations or attributes between classes
- position shifts: e.g. reordering the list of attributes or operations of a class (e.g. one attribute is shifted from the last to the first position in the list).

Modifications to the layout do not change the semantics of the diagram; they can be regarded as some kind of pretty-printing. Such shifts can be addressed by layout algorithms (see also section 3.3).

Structural shifts modify the structure of the graph represented by the diagram. This kind of shift means that an edge between two nodes `A` and `C` is deleted and that a new edge between nodes `B` and `C` is created. These shifts are visualised by colouring the edges between the nodes in an appropriate way; the involved nodes are not affected.

Inter-node shifts are semantical changes and therefore always relevant. The problem distinguishing between the shifting of a diagram element and the deletion and later re-creation of an element. Shifts can only be detected efficiently if this is supported by the editors. This can be achieved by associating surrogates with diagram elements, and is in fact implemented in our tools. The shifted element is coloured in a slightly brighter hue than other insertions and deletions.

Changing the position of an entry in an ordered list is relevant. The straightforward approach is to consider a shift as a deletion and an insertion and to show the shifted element at both positions in different colours. However, this approach does not work well if several elements are shifted and leads to ambiguities. Position shifts should rather be visualised by ordering the entries according to their position in one of the base documents. The index of the position of the entries in the other document is printed in front of each entry. The index is coloured appropriately. Stereotypes in the lists are handled in the same way.

3. PRESENTATION OF DIFFERENCES

The most widely applied paradigm for showing differences between textual documents is to use two columns. Each document is shown completely in one of the columns, identical parts in both documents are arranged side-by-side, differing parts are somehow highlighted.

The concept of using two columns works well with most textual documents, but fails if the lines of text are long (more than about 100 characters). It does not work well with graphical documents such as state charts, class diagrams, etc.: these documents (more specifically their graphical representation) are both "wide" and "tall" and cannot easily be split into corresponding sections which can be equal or differing (e.g. a class in two versions with different operations or attributes).

We have therefore adopted a different paradigm: the two documents are shown one *over* the other, they are so to say two transparencies arranged in two layers. The paradigm can also be used for text files. It is intuitively simple, but it leads to a number of non-trivial conceptual and technical problems, which we address in the following. The documents between which the differences should be shown will be called *base documents*.

3.1 The Unified Document

Obviously, the "common parts" of both documents should only be shown once. The complete picture thus consists of:

1. the common parts and
2. the specific parts of each base document.

Different colours can be used to distinguish these parts. If we disregard the colours, we see another document; it can be considered a representation of a "unified" (or "mixed") document of both base documents. We avoid the notion of a merged document here since merging is often associated with removing conflicts; in that sense, a unified document is not a merged document. However, it can be used as a basis to create a merged document.

Our unified document will usually have a new graph structure, which is similar, but not identical to the graph structure of either base document.

In the case of single-valued attributes, both versions are shown in different colours. In the case of multi-valued attributes, the two lists are compared, a common part is identified and parts appearing only in one version are coloured appropriately. List-valued attributes have already been discussed.

3.2 2- vs. 3-Way Differences

2-way differences are computed on the basis of two base documents. Three colours are sufficient here to show which parts appear in both, only the first or only second base document.

3-way differences are computed using a third document which is a common predecessor of both base documents (i.e. the base documents lie in parallel branches of a version tree, they are variants of each other). Five colours are necessary here: one for the common parts, two for insertions and deletions in the first branch and two for the changes in the second branch. This may be a bit too colourful, and it appears questionable whether a developer can maintain an overview of the bulk of information. Some experiments [22] have shown that too many colours confuse the developers. Because of this we do not consider 3-way differences here[2], and we do not use the words "insertion" or "deletion" in the rest of this text, we rather speak of the specific parts of the base documents.

3.3 Layout of Diagrams

Changes to the layout of a diagram are normally considered irrelevant[3].

The unified document generally has a new graph structure; therefore, it cannot have the layout of one of the base documents. Its layout should be similar to the layout of one of the base documents chosen by the developer, because normally a developer has to edit the parts of the document where differences occur, so he or she should be able to easily find these parts.

A layout of the unified document generated by a usual algorithm for automatic graph will hardly fulfil this requirement. Thus new algorithms are needed which produce a layout similar to the layout of one of the base documents. An ad-hoc approach employed in our prototype is to start with the layout of one base document (usually the one with the larger number of nodes) in order to keep the layout of the nodes and edges unchanged, and to place the remaining nodes of the unified document around them.

The development of more sophisticated layout algorithms is not subject of this paper. In any case, one should not expect a perfect solution. We conclude that one should be able to manually improve an initial, automatically generated layout.

3.4 An Example

Figure 2 and figure 3 show two class diagrams which could result from editing the class diagram shown in figure 1. In figure 2, a common super class `HTMLDocElem` has been added and the operation `add` has been deleted in the classes `HTMLList, HTMLCombo` and `HTMLForm`. In figure 3, three new classes `Export`, `HTMLExport` and `LaTeXExport` have been created and an operation `dumpCont` has been added to the class `HTMLDocElem` and to the subclasses. The operation `dump` has been shifted from `HTMLDoc` to `Export`. Note that the operation was not deleted at the old class and than re-created at the new class, but was shifted (see section 2.3.2).

Figure 4 shows the resulting unified diagram. Different line styles have been used in this diagram to represent different colours (colours would not be visible in black-and-white copies of this paper).

The classes `HtmlDoc, HtmlForm, HtmlList` and `HtmlCombo` are examples where some operations and attributes have been added, deleted or shifted.

When a class has been deleted or added the entire class is coloured (in this example the classes `HtmlDocElem, Export, HtmlExport` and `LaTeXExport`). The same also applies for relationships between classes.

[2]This argument is only valid for the visualisation of differences. A merge tool should use 3-way merging.

[3]If they are considered relevant, the paradigm of transparencies will not longer be applicable. If we arranged the transparencies in such a way that common parts lie over each other, the non-common parts could overlap and be unreadable. This would not be an acceptable representation, and one would be forced to abandon the idea of representing common parts only once.

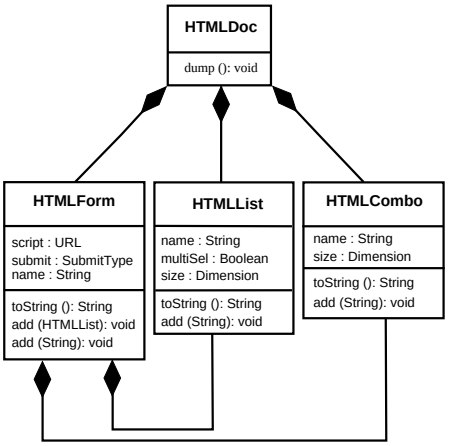

Figure 1: Example of a class diagram

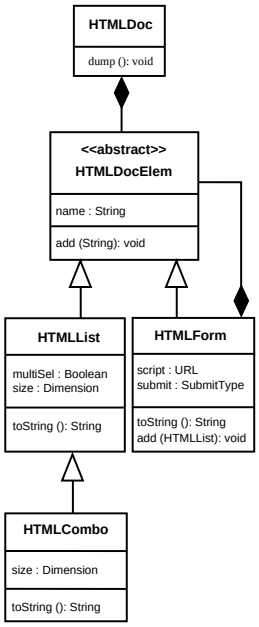

Figure 2: Class diagram of figure 1 extended with inheritance

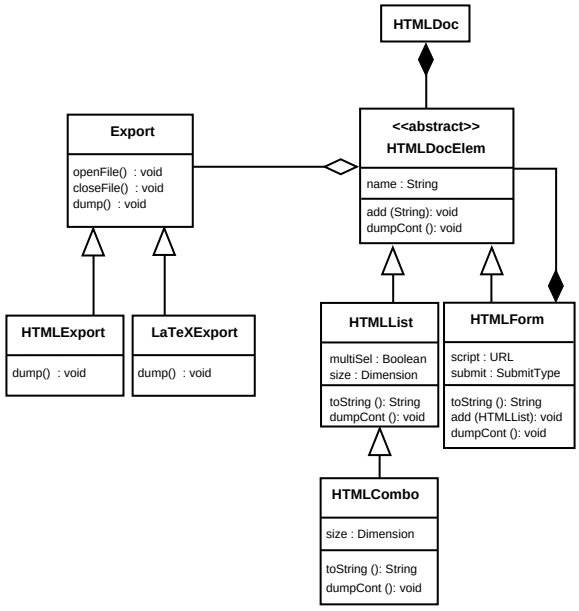

Figure 3: Class diagram of figure 2 extended with export functionality

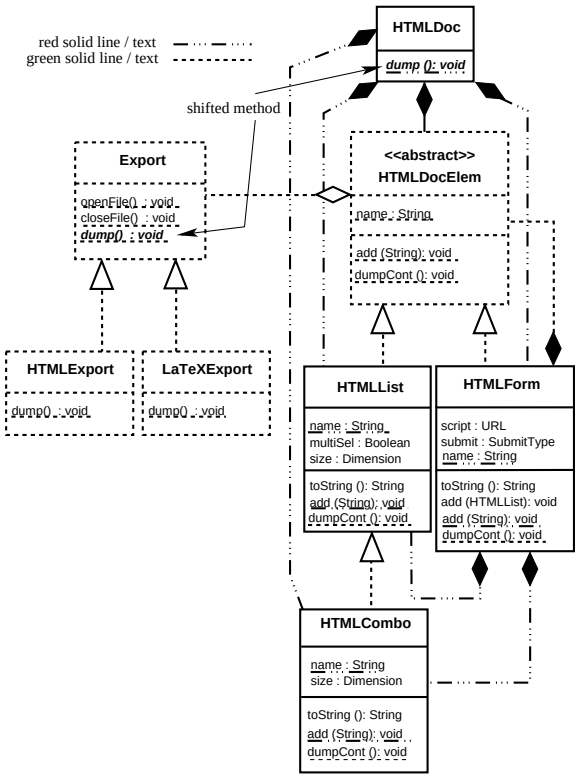

Figure 4: Unified document showing the differences between the base documents

4. COMPUTING THE DIFFERENCES

4.1 Data Model and Version Model

Our method of computing the differences between two base documents assumes that both documents are available in the form of a fine-grained physical data model (see section 2.1), which roughly resembles a syntax tree. All elements of a UML diagram are modelled as separate objects, for example the document, all classes, all operations, all attributes and all parameters of operations. Figure 5 shows a simplified meta model for fine-grained data models of UML class diagrams. There are component relationships between the object types, e. g. a document contains classes, a class contains operations and attributes, and so on. Further kinds of relationships between object types which represent classes can express inheritance or association between classes. The whole document is represented as a directed acyclic object graph with a root object. The composition relationships form a spanning tree of this graph.

The object attributes contain data describing the classes, operations, etc. Examples are the name or the type of an element, or layout information belonging to one diagram.

For example, the object graph shown in figure 6 partially represents the class diagram shown in figure 1; it uses the meta model from figure 5. Every component of a class is represented by an object of the appropriate type. Due to space considerations, the object graph shown in figure 6 does not contain all objects needed to represent the class diagram. The objects representing the class `HTMLCombo`, the operations and the attributes of the class `HTMLForm` and some attributes of the class `HTMLList` have been omitted.

In our prototype the documents are stored in the repository system H-PCTE [8] which is extended by versioning functionality [14]. Diagram editors based on H-PCTE work directly on the object graph, that is all editing commands are performed as operations upon the object graph.

Whenever a diagram editor is launched a new editing session is initialised, and all objects modified during that session are versioned automatically[4]. The versions created in an editing session form a so-called configuration. Internally, a configuration has a unique identifier and collects all relevant data.

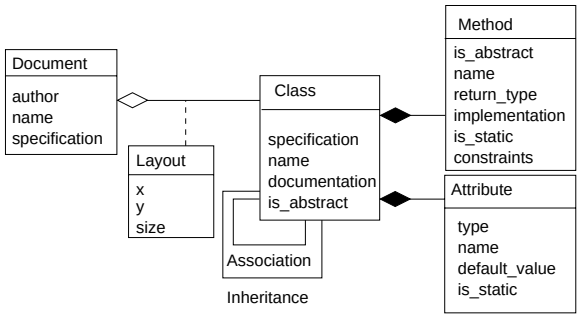

Figure 5: Meta model of a fine-grained data model

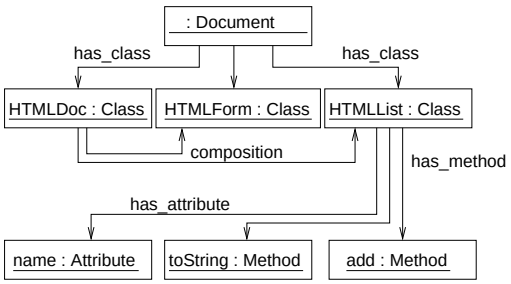

Figure 6: Example of an object structure

4.1.1 *Effects of Shifts*

The four kinds of shifts discussed in section 2.3.2 have different effects on the physical model of the diagram.

Shifts in the layout change only the object attributes storing the position and size of the nodes and edges. Structural shifts affect the relationships representing inheritance, aggregations or associations between classes. Creating an edge in the diagram results directly in creating a relationship of the appropriate type in the model[5].

Inter-node shifts (e. g. shifting an operation between classes) affect the component relationships in the model. The shifting of an operation between two classes results in the shifting of the object representing the operation from one parent object to another. That means one node of the spanning tree is shifted across the tree.

The effects of position shifts depend a lot on the chosen physical model. If explicit consecutive position numbers are used, then an insertion or deletion leads to the renumbering of all elements at the succeeding positions. Another approach would be the use of floating point values which express the positions by their relative values. A position shift can be achieved in most cases by simply modifying the position value. Details are beyond the scope of this paper.

4.2 The Difference Operation

The *difference operation* computes the differences between the model data of two base documents and creates a difference document containing all information necessary to visualise the unified document. We assume that the root object of the document is unchanged except for its attributes and components.

The difference operation traverses the spanning tree of the object graph of both base documents simultaneously. For each level of the spanning tree the corresponding sub-trees have to be found. The corresponding sub-trees are identified by the unique identifier of their root object[6]. Each pair of root objects of corresponding sub-trees is inserted in a queue and processed one after the other. Only the attributes and relationships (but not the component relationships which build the spanning tree) of the root objects are compared. After a pair of root nodes has been processed, a new object representing this pair is created in the unified document. The new object contains the unchanged attribute values, information about the changed attribute values and information about the relationships. Then the corresponding sub-trees of the current root objects are processed.

Sub-trees of one base document without a corresponding sub-tree in the other base document could have been:

1. created/deleted in one of the base documents or
2. shifted across the syntax tree.

One can distinguish between these two cases as follows: for each base document, we collect the sub-trees which are contained only in this base document. After processing the entire syntax trees these two sets are searched for corresponding sub-trees. Pairs of corresponding sub-trees are due to shifts in the diagram. All sub-trees without a corresponding sub-tree have been created or deleted in one of the base documents.

Our version management system provides additional information which allows us to simplify this search considerably and directly detect shifts across the syntax tree; this subject is beyond the scope of this paper.

As discussed in section 3.3 it does not make sense to compute the differences between the layout data of both docu-

[4]Unmodified objects are not versioned in an editing session.
[5]In our repository system H-PCTE, relationships have (a) a key (unique within the scope of the source object) and (b) attributes. Relationships are versioned independently of the source object.
[6]In storage systems where object identifier are not available one has to find corresponding objects by using heuristic methods based on the single- and multi-valued attributes of the nodes. This problem is not in the scope of this paper.

ments. Layout data must be handled in a special way. Because of the fine-grained modelling we can easily ignore layout data during the computation of the differences. The layout data of the unified document are computed in a second computation step. In the case of class diagrams we use an automatic layouter which starts with the layout data of one class diagram and places the classes belonging only to the other diagram at the border of the unified diagram. The user can layout the unified diagram manually as well.

An important observation at this point is that the computation of differences is document type-specific. Because layout data are usually stored in dedicated attributes, it is easy to ignore them during the first phase of the construction of the unified document; in fact, one can use a generic algorithm to which the set of layout attributes is passed as an argument. Things are more complicated in the second phase, the generation of the layout: in order to achieve good results, one has to use document type-specific heuristics.

5. RESTRICTING THE NUMBER OF COLOURED DIFFERENCES

The visualisation of differences using colours is only useful if the compared versions do not contain too many differences. If the document has evolved over a longer period of time or if the compared versions are variants of each other there can be a large number of coloured differences. This can lead to a unified diagram which is too colourful and therefore useless. Figure 4 shows an example of such a diagram.

More generally, developers often want to "see" only a subset of all differences. Mostly they are only interested in changes which meet certain criteria, e.g.:

- changes which affect elements of a particular type, e.g. changes affecting associations or subtype relationships in a class diagram, attribute assignments in an object diagram etc.; obviously, these selection criteria are document type-specific
- changes of "original" data; for example, if an operation is renamed, there is one change at the location where the operation is defined and maybe a large number of changes at locations where the operation is used and its name is referenced; the latter type of change can be considered irrelevant
- changes which occurred during a particular editing session, e.g. the changes caused by the fixing of a bug. The general problem is that local changes at the same diagram element (e.g. a class) which occurred in different contexts cannot be distinguished because they are all highlighted with the same colour. This problem occurs in the example in figure 4: this unified document shows the effects of two major changes which are basically independent from one another.

Our proposed solution to this problem is to restrict the coloured differences using information offered by the meta model and by the version management system. Only the differences of interest are coloured, all others are painted grey. The grey elements of the diagram are less eye-catching as the coloured ones; developers can thus concentrate on the differences of interest. The set of the differences to be highlighted can be restricted on the basis of:

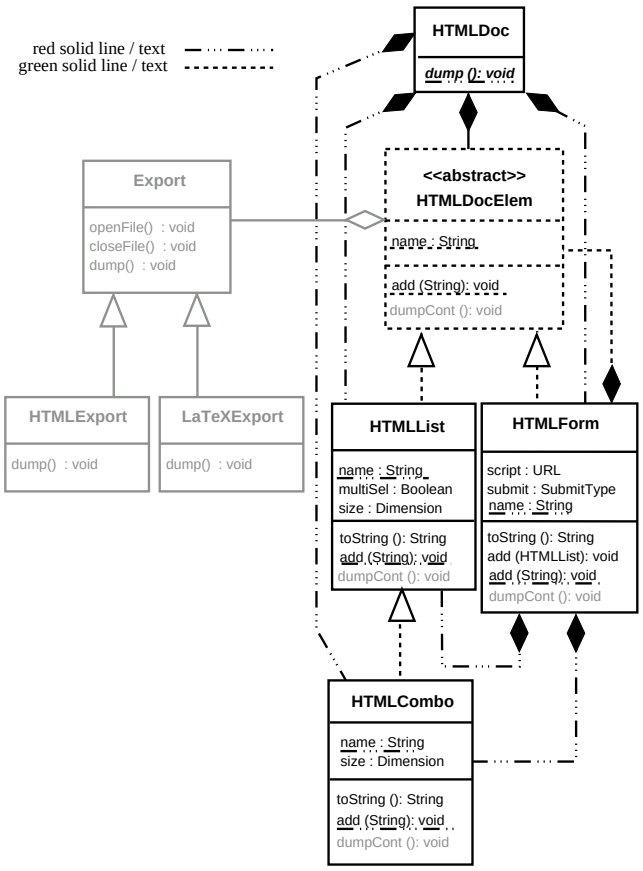

Figure 7: Unified document of figure 4 with grey parts

- the type of affected elements, e.g. only differences relating to classes, but no differences relating to attributes or operations are highlighted; here we exploit the fine-grained data model
- the history of revisions: only differences which emerged in a particular revision (which corresponds to a particular session) are highlighted
- inclusion or exclusion of differences relating to references.

The selections can be combined, e.g. one can highlight only the operations changed during a particular session.

5.1 Exploiting Revision Histories

Restricting the colouring of differences on the basis of the history of revisions is particularly useful if only few changes occur during a session.

An example of how revision histories can be exploited is shown in figure 7: only changes which correspond to the modifications shown in figure 2 are highlighted, while changes which correspond to the modifications shown in figure 3 are shown in grey.

In our own implementation, the history of revisions is automatically generated by the database kernel. Whenever a document is changed new versions of all modified objects are created. The newly created versions are combined to-

gether into a configuration. Each configuration has a unique identifier.

When the unified document is being created both base documents are read. Our database kernel delivers not only, the usual information about the objects and relationships and values of attributes, but also, for every object, the identifier of the configuration to which this version of the object belongs. These additional data are stored in the difference document and are later evaluated by the tools displaying the unified document.

5.2 Tool Support

The information about the involved configurations must be offered to the developer in an appropriate way (see figure 8). Our own tools[7] use a list of configuration identifiers. If the versions of the document belong to the same branch, the list contains all configurations between the configurations related to the base documents. If the versions belong to two branches with a common ancestor we use two lists, each one with the configurations between the common ancestor and the configurations relating to one of the base documents. These lists are displayed on the right side of the diagram. The developer can explicitly choose the differences to be highlighted by selecting the corresponding configurations.

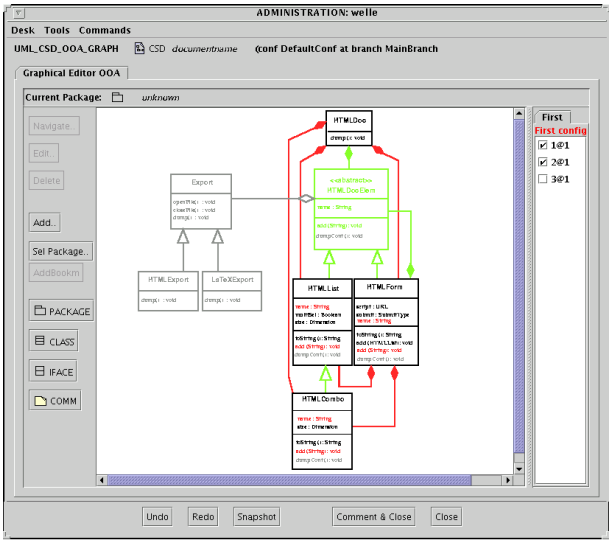

Figure 8: Difference view inside the class diagram editor (showing the diagram in figure 7)

It should again be obvious that difference tools are document type-specific since they must be able to display the specific forms of nodes, edges and other diagram elements. A difference tool for a specific document type is similar, but not identical to a graphical editor for documents of this type: normal graphical editors cannot evaluate the additional information contained in the difference document and cannot display differences (and, of course, do not offer a user interface to control the highlighting of differences). As a result, the implementation of difference tools can require considerable effort.

[7]We are using a generic framework for the construction of editors for a range of document types [12, 9].

6. RELATED WORK

There exist a number of algorithms for detecting differences between two documents, the chosen difference algorithm depends on the kind of data. The algorithm presented by Myers [13] is usable for textual files. This algorithm looks for the longest common sequence of characters inside the files. It has some disadvantages which have been partly solved in further work [7]. One example of such an improvement is Bdiff [19], which detects block-moves.

Algorithms for textual data are not suitable for structured data like UML class diagrams [16]. Structured data can often be represented as trees. Algorithms for finding differences between special kinds of structured documents like LaTeX-files, HTML-files or data in CAD databases [4, 3] are often based on algorithms solving the tree-to-tree correction problem [2, 23, 18, 17]. These algorithms interpret documents as ordered or as unordered trees. Ordered trees have an order between sibling nodes. The difference algorithms on trees try to find a sequence of atomic operations which transform one tree into the other. Such sequences are called edit scripts. The algorithms are based on different sets of atomic operations. All sets include the basic operations to create, delete or modify a node of the tree but only some of them can distinguish between the shift of a node and the combination of an insert and a delete.

Further algorithms [11, 5, 20] can detect differences between XML documents. These algorithms are based on the basic algorithms for text files. The algorithms for XML documents try to find similar sub-trees of the document and calculate the differences using matching sub-trees.

The algorithms mentioned above do not assume persistent node identifiers. In contrast to this objects managed by software configuration management systems can have persistent node identifiers [1, 10]. There are algorithms which exploit persistent node identifiers of versioned objects when searching for corresponding objects in two versions of a document. The amount of time required to find a corresponding node is then only linear to the number of nodes in the involved documents. But the calculation of differences between unordered trees without node identifiers is proven to be NP-hard [24].

There are also proposals which describe version management systems for software documents like class diagrams (e.g. [16]). These scientific prototypes offer a merge function or algorithms for detecting the differences between two UML class diagrams [25] which are stored in an object structure. ClearCase [21], a commercial software configuration management system, also supports the creation of versions of UML diagrams. ClearCase uses external tools to visualise differences or to merge versions of files in proprietary formats, e.g. files produced by Rational Rose. An example of such an external tool is the Model-Integrator, which is part of the Rose product. The Model-Integrator shows the differences in a tree; all meta data, e.g. unique identifiers, are included, positions are shown in the form of coordinates. This has the advantage that the tool must not address the layout. But this is also a big disadvantage because the developer has to map the internal tree representation mentally onto the external diagram representation. After merging two versions the resulting version of a diagram can be shown using Rose but without any information about differences.

Most of the known concepts and tools are generic, i.e. no special meta model is required to use them (textual difference tools are applicable on all kinds of texts, however the

quality of the results is poor). The Model-Integrator and our concept are specialised to a specific meta model, so that they are not applicable to arbitrary document types.

We are not aware of algorithms and tools which use colours for visualising differences between graphical software documents. Furthermore we are not aware of tools which enable a designer to control the amount of highlighted differences on the basis of several selection criteria, including one which is based on the version history.

7. CONCLUSION AND FUTURE WORK

We have discussed the UML diagrams and their characteristics. The diagrams represent directed graphs; the layout of the diagrams is irrelevant except for sequence diagrams. The diagrams consist of nodes and relationships. The nodes can contain attributes such as names or lists of components, groups of nodes or even sub-diagrams. On this basis we have discussed several kinds of differences and have presented concepts for computing and displaying differences between graphical documents. The differences are displayed in a diagram which can be seen as the union of the two base diagrams. The common parts of both base diagrams are painted black and the specific parts are coloured.

One particular feature of the proposed concept is its ability to detect shifts, e.g. the shifting of attributes or operations between classes in a class diagram. Another feature is the document type-specific filtering of highlighted information; filtering is based on the type of the elements and on the revision history.

The concepts have been implemented for UML class diagrams. The difference tool has been integrated into the tool environment PISET [9]. The meta-CASE architecture of this environment has significantly facilitated the implementation of the difference tool.

The most significant disadvantage of our approach is that the algorithms and tools are document type-specific. Difference tools are comparable to usual graphical editors for graphical documents; the effort required to implement such a tool can be significant; meta-CASE architectures and other technologies can reduce the effort required for implementing a family of difference tools. It is hard to imagine that one can find good algorithms and difference tools which are as "generic" as the algorithms and tools for textual files.

Further work will address the merging of document versions based on 3-way merging. Virtually all advantages of our approach are carried forward to the problem of merging, notably the number of merge conflicts will be reduced. Merge tools, however, must address a number of additional problems including document type-specific commands and the management of the revision history.

Acknowledgements

We would like to thank our colleagues with whom we discussed the ideas that led to the present paper, in particular M. Monecke.

8. REFERENCES

[1] U. Asklund. Identifying conflicts during structural merge. In B. Magnusson, editor, *Proceedings of NWPER'94, Nordic Workshop on Programming Environment Research*, June 1994.

[2] D. T. Barnard, G. Clarke, and N. Duncan. Tree-to-tree correction for document trees. Technical report, Departement of Computing andf Information Science Queen's University Kingston Ontario, Canada, January 1995.

[3] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 26–37, Tuscon, Arizona, May 1997.

[4] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.

[5] G. Cobéna, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *18. International Conference on Data Engineering (ICDE) San Jose, California, USA, February 26-March 1, 2002*, 2002.

[6] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[7] J. J. Hunt, K.-P. Vo, and W. F. Tichy. An empirical study of delta algorithms. In I. Sommerville, editor, *Software configuration management: ICSE 96 SCM-6 Workshop*, pages 49–66. Springer, 1996.

[8] U. Kelter. H-PCTE – a high-performance object management system for system development environments. In *Proceedings COMPSAC Illinois, September 23-25*, pages 45–50. IEEE Press, 1992.

[9] U. Kelter, M. Monecke, and D. Platz. Constructing distributed SDEs using an active repository. In *Proc. 1st Intl. Symposium on Constructing Software Engineering Tools (COSET '99); 17.-18.05.1999, Los Angeles, CA*, pages 149–158, 1999.

[10] B. Magnusson, U. Asklund, and S. Minör. Fine-Grained Revision Control for Collaborative Software Development. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering, Los Angeles, California*, pages 33–41, December 1993.

[11] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *Proceedings of the Twenty-seventh International Conference on Very Large Data Bases: Roma, Italy, 11–14th September, 2001*, pages 581–590, Los Altos, CA 94022, USA, 2001. Morgan Kaufmann Publishers.

[12] M. Monecke. *Adaptierbare CASE-Werkzeuge in prozessorientierten Software-Entwicklungsumgebungen.* PhD thesis, Universität Siegen, 2003.

[13] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–256, 1986.

[14] D. Ohst and U. Kelter. A fine-grained version and configuration model in analysis and design. In *Proc. of the IEEE International Conference on Software Maintenance 2002 (ICSM 2002), 3-6 October 2002, Montreal, Canada*, 2002.

[15] OMG. *Unified Modeling Language Specification.* OMG, March 2003. Version 1.5 formal/03-03-01.

[16] J. Rho and C. Wu. An efficient version model of software diagrams. In *Proc. 5th Asia-Pacific Software Engineering Conf., 2-4 December 1998 in Taipei, Taiwan, ROC*. IEEE Computer Society, Dec. 1998.

[17] S. M. Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 6(6):184–186, December 1977.

[18] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, July 1979.

[19] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, Nov. 1984.

[20] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-Diff: An effective change detection algorithm for XML documents. In *19th International Conference on Data Engineering, March 5 - March 8, 2003 - Bangalore, India*, 2003.

[21] B. A. White. *Software Configuration Management Strategies and Rational ClearCase*. Addison-Wesley, 2000.

[22] W. Yang. How to merge program texts. *The Journal of Systems and Software*, 27(2):129–141, November 1994.

[23] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18:1245–1262, 1989.

[24] K. Zhang, J. T.-L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs and related problems. In Z. Galil and E. Ukkonen, editors, *Combinatorial Pattern Matching, 6th Annual Symposium*, volume 937 of *Lecture Notes in Computer Science*, pages 395–407, Espoo, Finland, 5-7 July 1995. Springer.

[25] A. Zündorf, J. Wadsack, and I. Rockel. Merging graph-like object structures. In *Tenth International Workshop on Software Configuration Management (SCM-10) New Practices, New Challenges, and New Boundaries May 14-15, 2001 Toronto, Canada (a workshop of 23th ICSE 2001)*. http://www.ics.uci.edu/∼andre/scm10/, 2001.