A Generic Difference Algorithm for UML Models

Udo Kelte, Jürgen Wehren, Jörg Niere
Software Engineering Group
Department of Electrical Engineering and Computer Science
Siegen University
{kelter|wehren|niere}@informatik.uni-siegen.de

Abstract: It is state-of-the-art to use the Unified Modelling Language (UML) to describe software system models. In order to support cooperative team work a version management system which supports UML models is absolutely necessary. The essential part of such systems is the ability to calculate differences and present them to the developer. This paper presents an approach for computing differences between UML models encoded as XMI files. In contrast to our previous work, we present a generic approach, with which we are able to cover a broad range of UML diagram types. It also does not require persistent identifiers of diagram elements. Our prototype implementation shows, that our difference algorithm used leads to good runtimes in the case of small documents and acceptable runtimes in the case of large documents. Overall, it has a very low error rate, i.e. the quality of the differences is almost optimal.

1 Introduction

The Object Management Group (OMG) proclaims Model Driven Architecture (MDA) as the new philosophy of software development in the 21^{st} century. The central point of MDA is the model of a software system, which becomes the central part of the whole development. In MDA models can be transformed to other models. Especially the transformation of a Platform Independent Model (PIM) to a Platform Dependent Model (PDM) allows for generating source code and for getting an executable ‘model‘, which is better called application or program.

It is state-of-the-art to use the Unified Modelling Language (UML) to describe models for software systems. For example Figure 1 shows a part of an initial class diagram of a generic graph editor. Following the MDA philosophy, all enhancements and modifications of the software system should be performed on the model level, e.g. in the class diagram. Current practice, however, is to change the code directly and to update the corresponding models occasionally or to reverse engineer them from the code. Figure 2 shows the class diagram extracted with reverse engineering techniques from the final application.

In general, software should be developed in teams. In order to support cooperative team work a version management system which supports UML models is absolutely necessary. The essential part of such systems is the ability to calculate differences, present them to the developer and to provide merge operations to come to a consistent model. In our approach we use unified diagrams with difference information to be presented to the developer. For example Figure 3 shows the unified class diagram of Figure 1 and Figure 2. The unified class diagram consists of all parts of both original diagrams; the parts detected as similar are shown in black color, the green (light gray) parts exist only in the second diagram and the red (dark gray) parts exist only in the first diagram. In Figure 3 the red parts are the two compositions called hasNode and hasEdge. They only exist in the Figure

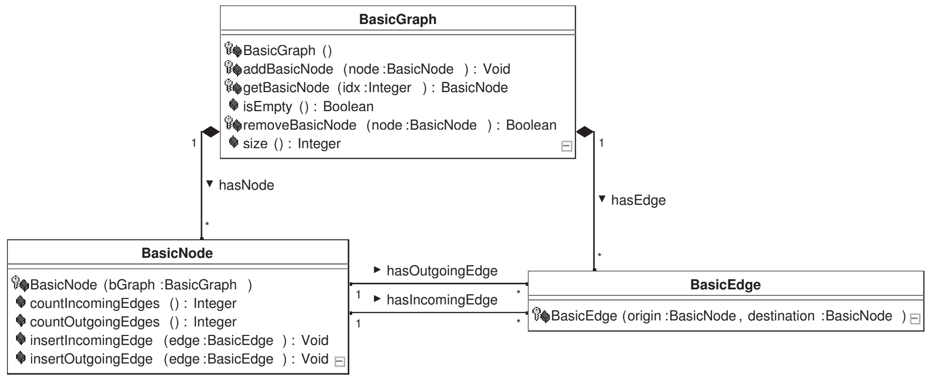
BasicGraph
BasicGraph ()
addBasicNode (node :BasicNode) : Void
getBasicNode (idx :Integer) : BasicNode
isEmpty () : Boolean
removeBasicNode (node :BasicNode) : Boolean
size () : Integer
1
hasNode
*
1
hasEdge
*
BasicNode
BasicNode (bGraph :BasicGraph)
countIncomingEdges () : Integer
countOutgoingEdges () : Integer
insertIncomingEdge (edge :BasicEdge) : Void
insertOutgoingEdge (edge :BasicEdge) : Void
hasOutgoingEdge
hasIncomingEdge
1
1
*
*
BasicEdge
BasicEdge (origin :BasicNode , destination :BasicNode)

Figure 1: Initial model of BasicGraph

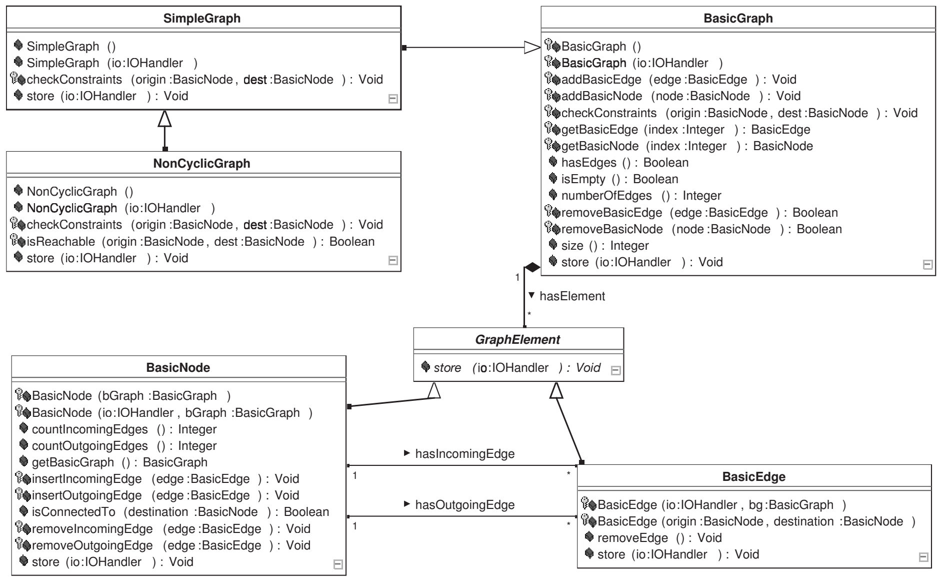
SimpleGraph
SimpleGraph ()
SimpleGraph (io:IOHandler)
checkConstraints (origin :BasicNode , dest :BasicNode) : Void
store (io:IOHandler) : Void
NonCyclicGraph
NonCyclicGraph ()
NonCyclicGraph (io:IOHandler)
checkConstraints (origin :BasicNode , dest :BasicNode) : Void
isReachable (origin :BasicNode , dest :BasicNode) : Boolean
store (io:IOHandler) : Void
BasicGraph
BasicGraph ()
BasicGraph (io:IOHandler)
addBasicEdge (edge :BasicEdge) : Void
addBasicNode (node :BasicNode) : Void
checkConstraints (origin :BasicNode , dest :BasicNode) : Void
getBasicEdge (index :Integer) : BasicEdge
getBasicNode (index :Integer) : BasicNode
hasEdges () : Boolean
isEmpty () : Boolean
numberOfEdges () : Integer
removeBasicEdge (edge :BasicEdge) : Boolean
removeBasicNode (node :BasicNode) : Boolean
size () : Integer
store (io:IOHandler) : Void
1
hasElement
*
GraphElement
store (io:IOHandler) : Void
BasicNode
BasicNode (bGraph :BasicGraph)
BasicNode (io:IOHandler , bGraph :BasicGraph)
countIncomingEdges () : Integer
countOutgoingEdges () : Integer
getBasicGraph () : BasicGraph
insertIncomingEdge (edge :BasicEdge) : Void
insertOutgoingEdge (edge :BasicEdge) : Void
isConnectedTo (destination :BasicNode) : Boolean
removeIncomingEdge (edge :BasicEdge) : Void
removeOutgoingEdge (edge :BasicEdge) : Void
store (io:IOHandler) : Void
hasIncomingEdge
1
*
hasOutgoingEdge
1
*
BasicEdge
BasicEdge (io:IOHandler , bg :BasicGraph)
BasicEdge (origin :BasicNode , destination :BasicNode)
removeEdge () : Void
store (io:IOHandler) : Void

Figure 2: Final model of BasicGraph

1 but not in Figure 2. In general, if the second diagram is 'older' than the first one, the color red expresses that these parts have been deleted and green expresses that these parts have been added. Figure 3 also contains a small u-button, which indicates that the element has been detected as similar, but some parts of it are different. For example name changes are a usual representative, e.g. the parameter name of method getBasicNode in class BasicGraph has changed from idx to index. More details for the visualization aspects can be found in [Nie04, OWK03b].

Although the UML is supported by a large number of tools, their team work support and notably their facilities for calculating and visualizing differences are inadequate [Ros]. Some tools include a proprietary version management system to support team work, which does not allow for exchanging or comparing models produced by other tools [SZN04]. But, usually tools store the developed models as textual XMI files and use standard textual

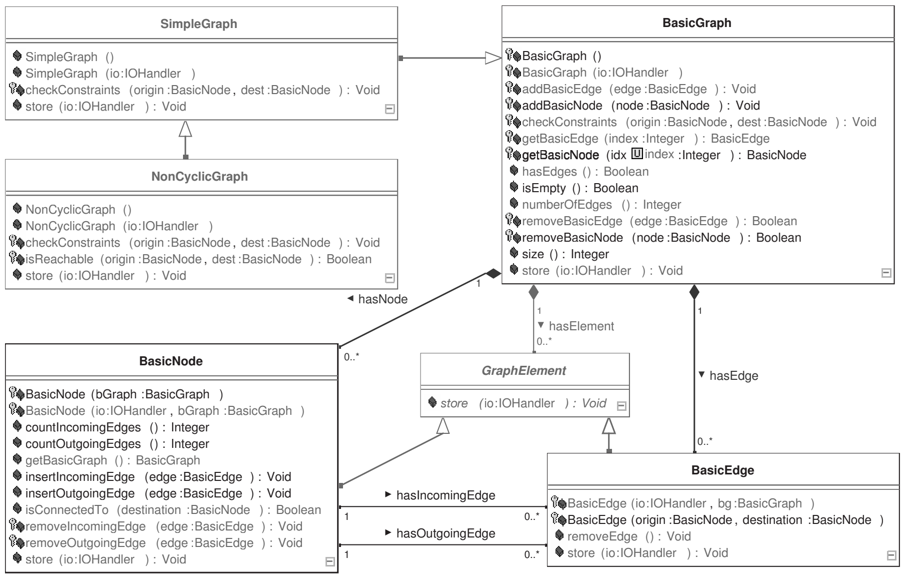
SimpleGraph
SimpleGraph ()
SimpleGraph (io:IOHandler)
checkConstraints (origin :BasicNode, dest :BasicNode) : Void
store (io:IOHandler) : Void
NonCyclicGraph
NonCyclicGraph ()
NonCyclicGraph (io:IOHandler)
checkConstraints (origin :BasicNode, dest :BasicNode) : Void
isReachable (origin :BasicNode, dest :BasicNode) : Boolean
store (io:IOHandler) : Void
BasicGraph
BasicGraph ()
BasicGraph (io:IOHandler)
addBasicEdge (edge :BasicEdge) : Void
addBasicNode (node :BasicNode) : Void
checkConstraints (origin :BasicNode, dest :BasicNode) : Void
getBasicEdge (index :Integer) : BasicEdge
getBasicNode (idx index :Integer) : BasicNode
hasEdges () : Boolean
isEmpty () : Boolean
numberOfEdges () : Integer
removeBasicEdge (edge :BasicEdge) : Boolean
removeBasicNode (node :BasicNode) : Boolean
size () : Integer
store (io:IOHandler) : Void
hasNode
1
0..*
hasElement
1
0..*
hasEdge
1
0..*
GraphElement
store (io:IOHandler) : Void
BasicNode
BasicNode (bGraph :BasicGraph)
BasicNode (io:IOHandler , bGraph :BasicGraph)
countIncomingEdges () : Integer
countOutgoingEdges () : Integer
getBasicGraph () : BasicGraph
insertIncomingEdge (edge :BasicEdge) : Void
insertOutgoingEdge (edge :BasicEdge) : Void
isConnectedTo (destination :BasicNode) : Boolean
removeIncomingEdge (edge :BasicEdge) : Void
removeOutgoingEdge (edge :BasicEdge) : Void
store (io:IOHandler) : Void
hasIncomingEdge
1
0..*
hasOutgoingEdge
1
0..*
BasicEdge
BasicEdge (io:IOHandler , bg :BasicGraph)
BasicEdge (origin :BasicNode, destination :BasicNode)
removeEdge () : Void
store (io:IOHandler) : Void

Figure 3: Document with difference information. Parts detected as similar are shown in black color, the green (light gray) parts exist only in the second diagram and the red (dark gray) parts exist only in the first diagram. Red parts are only the two composition relations called hasNode and hasEdge.

version management software such as CVS to support team work. Storing models as XMI files means that either the XMI files can contain tool-specific and other auxiliary data as well as the order in which elements are stored in XMI files and other details in the formatting of the text are arbitrary and at the discretion of the tool. Both of the above effects can lead to many textual differences, however, these differences are conceptually irrelevant. Obviously, textual representations of UML models are not at an appropriate level of abstraction for computing differences. UML documents, which we call an UML model encoded as XMI file, should rather be regarded as trees when computing differences.

Difference tools must interpret XMI files as graphs. The main structure of such a graph is a tree which contains references (*idref*s in XMI), i.e. additional graph edges beyond a pure tree structure. Generic algorithms for computing graph differences are not appropriate for these graphs, because they do not take the semantics and actual representation of models into account [OWK03b, RW98, ZWR01].

In this paper we present an algorithm that calculates differences of two models given as XMI files. The output is also an XMI file; it contains the unified model of the two original models with additional difference information. In contrast to our previous work, the difference algorithm does not rely on persistent unique identifiers. The calculation itself is configurable to capture the semantics of an actual model or part of the model in the algorithm. As example we use UML class diagrams, but our prototype can currently calculate differences of statecharts as well.

2 Related Work

A large number of algorithms for comparing documents have been proposed. They can roughly be divided into (a) algorithms which can handle only one specific document type and which are fully adapted to this document type (as e.g. proposed in [Gir02]) and (b) generic algorithms which, if at all, require only some configuration data. This paper addresses only the second class of algorithms.

Most of the generic algorithms are intended for documents represented as text, e.g. the LCS (Longest Common Subsequence) algorithm [Mye86]. These algorithms are appropriate for documents such as source programs, LaTeX sources etc., but not for typical models stored in the XMI format; as already mentioned, XMI files are representations of UML documents at a physical level; comparisons at this level lead to many false, conceptually irrelevant differences. Differences between UML documents must be computed on the basis of a conceptual representation [Kel04].

Although UML documents have a primary structure like a tree - they are composed of elements which in turn have sub elements - they are not exactly trees due to cross references; e.g. the type of a parameter of an operation can be another class in a class diagram. Considering UML documents just as graphs is not a viable solution since algorithms for comparing arbitrary graphs are NP complete and fail to exploit the semantics of the models. Algorithms for unordered trees are too inefficient, too.

The only viable approach is to consider UML documents as ordered trees - this is no problem since in all relevant cases, diagram elements are either ordered or they have names, from which an order can be derived. Examples of algorithms for comparing ordered trees are LaDiff [CRGMW96] and XDiff [WDC03].

LaDiff is applicable to ordered typed trees, notable LaTeX documents. It processes the two trees which are to be compared in a bottom-up fashion, i.e. it compares the leaves of the trees pairwise and forms corresponding pairs as soon as their similarity is above a given threshold. The complexity of this algorithm is $O(n^2)$, with n being the number of nodes of the trees, which is still too high for medium to large documents.

XDiff is applicable to typed trees in which elements have names, which are unique in the context of the parent element. Using these names, one can form unique path names of subtrees. XDiff initially processes all elements of the first document (in bottom up order) and computes a hash key which depends on the whole subtree and the path name of the subtree. The hash keys of all elements are collected in a directory (e.g. an AVL tree or a hash table). Then, hash keys of all elements of the second element are computed (again in bottom up order), and for each hash key it is checked whether the same key appears in the directory of the first document. If so, XDiff matches and the two elements form a corresponding pair. Due to the use of a directory, XDiff is very efficient in identifying identical subtrees in both documents, the complexity is in the order of $O(n * log_2(n))$. Identical, but moved subtrees are not identified and there is no notion of similarity at all.

The main task of all algorithms mentioned so far is to identify corresponding pairs of elements in the two diagrams. If the two diagrams are revisions of each other and if the diagram editing tool supports persistent identifiers then one can use these identifiers to form corresponding pairs of elements. Difference tools based on this approach are presented in [AP03, OWK03a]. This approach is more efficient, but only applicable under the conditions mentioned above; moreover, it does not deliver optimal results in the case of significant local modifications in diagrams.

3 Difference Algorithm

Our approach of detecting differences between two UML documents can be divided into two phases, similar to most of the difference algorithms in section 2. In the first phase we have to detect the elements in the first document that have a corresponding element in the second one. Subsequently the differences between the two documents can be deduced and the appropriate output can be created.

3.1 Data Model

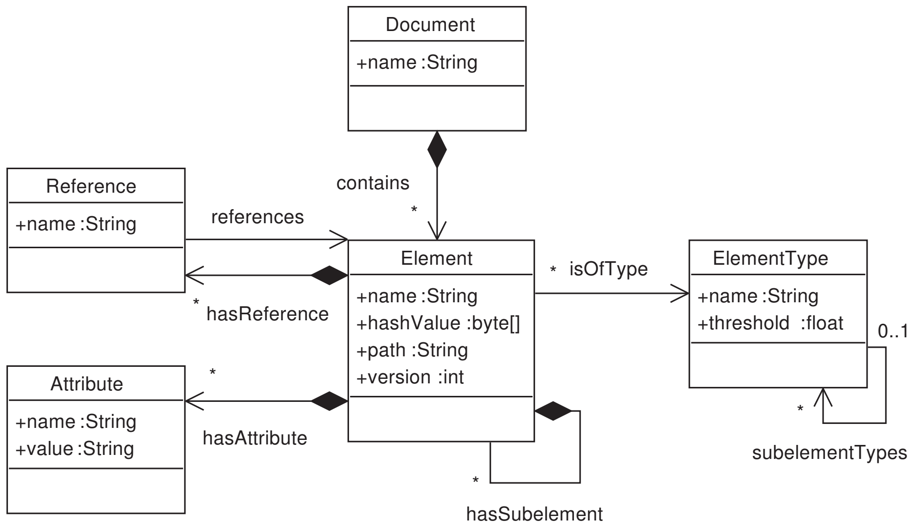

Figure 4: The data model of the difference algorithm

To be independent of an actual existing meta model, e.g. the complex UML meta model, we decided to design a simpler data model for our algorithm. A side-effect is that the algorithm should be able to handle also other models encoded in XMI which are not UML models. The data model is depicted in Figure 4. Basically it is a tree with typed elements, that can be decorated with attributes. In addition to the tree-like elements the data model might also have graph-like cross-references. In detail:

- A Document contains several elements. In a UML class diagram a Document contains all elements of the model.
- Elements have a specific ElementType and can refer to other Elements modeled by the Reference class. Elements might also contain several Attributes. Examples for Elements in a class diagram are: packages, classes, operations, attributes and parameters.
- An Attribute features name and value pairs. For example a UML class has an Attribute 'isAbstract' with values 'true' or 'false'.
- Composite Structure: Elements can be composed of sub elements, notably the tree-structure of the document. Within class diagrams for examples packages contain classes and sub packages, classes contain attributes and operations and operations contain parameters.
- Class Reference models cross-references between elements , e.g. an association features two association ends that both have a Reference referring to a class element.

By reducing the complexity of the model it is possible to use this generic data model for every diagram type within the UML specification, since all elements of an XMI file can be mapped to our data model. Thereby, our data model is comparable to the meta-model internally used by DOM parsers for XML. The mapping between XMI elements and elements in the data model is defined by a configuration file. Our difference algorithm works on an instance of this data model. For example in Figure 5, Class, Parameter, etc. are instances representing classes and parameters and so on of an actual class diagram.

3.2 The difference algorithm for class diagrams

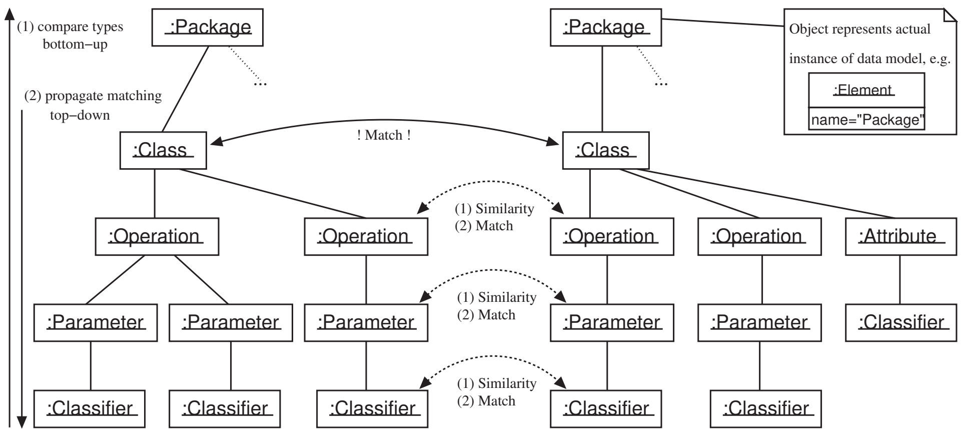

Figure 5: The bottom-up and top-down phase of the algorithm

As we have decided not to rely on persistent identifiers for the model elements, our algorithm detects the matchings by considering the similarity between elements. Most elements of a class diagram are composed of other sub elements, e.g. a class consists of attributes and operations, which itself consist of parameters and so on. So we calculate the similarity between two elements by looking at the sub elements at first. Consequently our algorithm first tries to detect matches in a bottom-up phase.

1. **Bottom-Up:** Similar to the LaDiff algorithm [CRGMW96] within calculating differences of two documents at first all leaf elements are compared. So in Figure 5 first of all the Classifier elements are compared. A Classifier element in a class diagram refers to either a simple datatype or a class element. In the bottom-up phase only elements will be matched, that have a unique similarity to exactly one other element in the second document. If there are similarities between more than two elements no match is detected, as the similarities can change when more and more elements have been compared. A similarity is only noticed when the similarity value is greater than a threshold that is specified for each element type. If a unique similar element is detected the two elements are matched, i.e. they are considered to refer to corresponding elements. In the example given in Figure 5 no match could be found on the Classifier, Parameter and Operation level but on the Class level. In such a case the algorithm switches into top-down phase.

2. **Top-Down:** During the top-down phase the last match in the bottom-up phase is propagated to all child elements of the matched elements referring to the composite structure of our data model. As a result of the additional information due to the fact that now parent-elements and eventually referenced elements have been matched the order of the similar elements can differ from the order in the bottom-up phase. As a consequence the most similar elements are matched and this is propagated top-down further on.

The algorithm stops when all elements have been compared in bottom-up phase. The result of the difference algorithm is a correspondence table consisting of matching element pairs. Therefore the belonging differences can easily be deduced as we will see in section 3.4.

A problem remains when there are more graph-like structures in the document and less composite relations, for example in state-charts. If we want to compare two state-charts, i.e. finding corresponding states and transitions, the transitions between states are very important for detecting the correct correspondence. But if we compare one state, the referenced state has probably not been compared yet. So in this case dependencies can be noticed for the comparing process that contradict the tree-like structure of the diagram. Also cycles can be possible. Cycles and dependencies can be solved by comparing the elements repeatedly. For example classifier, parameter, attribute and operation elements within a class diagram are compared again after the class elements have been compared, since the classifiers refer to class elements. Experiments show that this suffices to detect all correspondences. This procedure lets the similarity flood over the references in a way such as the 'Similarity Flooding'-Algorithm described in [MGMR02].

3.3 The Similarity Function

ElementType	**threshold**	**Criterion**	**weight**
Class	0,4	Similarity of the class names	0,4
		Ratio of similar or matched Operations	0,2
		Ratio of similar or matched Attributes	0,2
		Generalization targets match	0,1
		Packages match	0,1

Table 1: Criteria for comparing class elements

Up to now we have only discussed the similarities between elements without defining them precisely. The elements are compared in a type wise manner, so we have to define a function that compares two elements of the same type and returns a value between 0 and 1, where 0 means no similarity and 1 means mostly similar. Due to our generic approach the similarity function can be easily defined by setting up some criteria for each element type in a configuration file. The criteria consider certain parts of the elements depending on the actual types and structure of the compared models. The values of the different criteria are weighted and the total similarity value is calculated by addition as we can see in the following formula:

$$Sim_{e_1,e_2} = \sum_{c \in C} w_c \cdot \mathrm{compare_c(e_1, e_2)} \qquad (1)$$

Thereby e_1 and e_2 are the elements to be compared, C is the set of criteria, w_c gives the weight of criteria c and $compare_c$ is the compare function for criteria c. Besides the similarity function the threshold defines the minimum similarity value to consider two elements as similar. Actual criteria, weights and the threshold for comparing elements of classes are shown in table 1. The complete table with the criteria for comparing class diagrams can be found in [Weh04]. The similarity of string attributes is calculated by using the text comparing algorithm LCS. The ratio of similar or matched operations and attributes can be easily calculated by counting the sub elements that already match or summing up their similarity values. For the generalization or package criteria the matching of the referenced elements has to be considered.

3.4 Output of the Algorithm

The result of the algorithm, as described in section 3.2, is a correspondence table consisting of all all matched element pairs. To represent the belonging differences we create a unified document, that contains all elements of the two original documents, whereas the elements in the correspondence table are only contained once in the unified document. The differences of the two documents can be easily deduced:

- **structural difference (SD)**: Elements, that have no entry in the correspondence table are considered to be structurally different.
- **attribute difference (AD)**: Corresponding elements that differ in their attribute's values get an attribute difference containing both, the old and the new value.
- **reference difference (RD)**: Corresponding elements, whose references are different in the two original documents have a reference difference with references to both targets.
- **move difference (MD)**: Elements that appear to change their parent element between the two original documents get a move difference with a reference to the other parent element.

The differences between matching elements is annotated at the belonging elements itself. This annotation includes additional information to specify the difference precisely, e.g. an attribute difference reveals both values of the changed attribute. The annotations can be easily stored by using the XMI extension mechanism that allows for adding user specified elements in the unified document.

3.5 Optimization

The difference algorithm as described in section 3.2 has complexity $O(n^2)$, with n being the number of elements in both documents. Note that the number of elements of a certain type increases dramatically level by level regarding to the composite structure of the data model, cf. Figure 4. Consequently in the bottom-up phase, where the difference algorithm tries to find unique correspondences on the lower levels, it fails. For example in class diagrams, the algorithm will not find unique correspondences of datatypes on the lowest level, because datatypes are part of attributes, methods and parameters in class diagrams and therefore massively used. To match datatypes of two models, the algorithm has to investigate the nodes lying on higher levels, e.g. parameters and classes, see Figure 5, and match the actual datatypes in the subsequent top-down phase. Unfortunately, this failure of finding a unique correspondence needs most of the calculation time.

The idea to optimize the algorithm and to reduce the runtime is to use a hashing pre-phase similar to the XDiff algorithm [WDC03]. In this pre-phase we calculate the path of each element regarding to the composite structure of the data-model. The certain parts of the path are usually the names of the elements. For example consider Figure 5, a path of a parameter consists of the package name, the class name, the method name and the parameter name itself. We also calculate a hash value for the element itself and another one for all sub elements. If an element has references to another element, we include the path of the referenced element into the original element's hash value. We concatenate the three values for each element and use it as identification value. Afterwards we match elements with identical identification values. Overall, the hashing pre-phase reduces the number of elements of a certain type that have to be match by the difference algorithm.

The determination of the paths has complexity $O(n)$ with n being the number of all elements and takes place during parsing the XMI files into our data-model. Finding elements with identical paths has worst-case complexity of $O(n*log_2(n))$, with n being the number of all elements, because of unordered sub element lists. Overall this means an overhead of $O(n*log_2(n)$, which is compared to the quadratic complexity of the difference algorithm acceptable and practically reduces the runtime significantly.

The disadvantage of our optimization is that element moves can only be detected by the difference algorithm, because the element paths are different. To detect also moved elements in the pre-phase one solution is to iterate the pre-phase, where in each iteration we decrease the path length by 1 and find matches for all elements that have not been matched in the previous iterations. However, our experiments show that the iterated pre-phase approach results in longer runtime than performing only one pre-phase and detect the moved elements by the difference algorithm.

4 Evaluation

The main assessment criteria of our evaluation are the quality of the calculated solutions and the required runtime. We used several projects to "benchmark" the algorithm. To retrieve different UML documents as input for the evaluation of our algorithm we partly used version management systems to check out different versions of source code. Usually the source code subsequently has been reverse engineered into UML documents. The documents differ in several attributes, notably in size of the models, the elapsed calendar time between the versions, and the number of developers involved in the project. The series of our benchmark projects can be characterized as follows:

- **HTMLPackage:** Three feasibility class diagram examples that model HTML documents, to test and verify our algorithm.
- **UMLDiff Package:** Four diagrams that have been reverse engineered from the source code of our prototype. The versions were taken at the beginning and the end of the implementation of the tool.
- **Fujaba Packages:** Four large class diagrams which were derived from the sources of the UML tool Fujaba. About six months of calendar time was elapsed between the versions.
- **Ritterspiel:** Three versions of a class diagram from a project group at the university of Kassel which designed a board game during six months. These versions stem from the Fujaba internal version management system and have not been reverse engineered from source code.
- **Fujaba BasicPackage Series:** This series consists of 50 consecutive reverse engineered versions of the de.uni_paderborn.fujaba.basic package of Fujaba. On average, a new version has been created every three days. The analysis of the whole series showed that many versions had no or almost no changes in comparison with their predecessor; some had up to 1 % of the elements changed, and in one case, 28 % of the elements were changed. Due to lack of space we will show only the results of 4 comparisons of this series.

The results of the evaluation are shown in table 2. The first column of the table shows the sum of number of XMI elements (Elem.) of both documents. The second column shows the for number of classes (Cl.), respectively. The largest documents contain class diagrams with up to 284 classes and 13.973 elements.

	Quantity			Det. Differences				Errors		
Testdata	**Elem.**	**Cl.**	**SD%**	Σ	**OD**	**SD**	**HM%**	α	β	**RT(s)**
HTMLPackage										
V0 vs. V1	171	17	55,56%	97	2	95	71%	0	0	0,5
V0 vs. V2	204	20	65,69%	136	2	134	66%	0	0	0,5
V1 vs. V2	185	23	21,08%	39	0	39	84%	0	0	0,5
UMLdiff Packages 26.04.04-15.07.04										
diagModel	804	32	30,85%	270	22	248	79%	0	0	0,9
compItems	595	35	46,55%	291	14	277	72%	0	0	0,7
calculator	1545	66	69,45%	1088	15	1073	78%	0	0	1,5
altogether	2929	99	57,87%	1740	45	1695	78%	0	0	3,7
Fujaba Packages 01.01.04-21.07.04										
asg	1755	102	7,69%	148	13	135	94%	0	0	1,1
fsa	7020	199	2,76%	208	14	194	99%	0	0	2,9
basic	8629	237	28,98%	2551	50	2501	87%	0	0	12,69
uml	13973	284	6,13%	971	114	857	93%	-	-	14,95
Ritterspiel V0=21.01.04, V1=29.03.04, V2=27.07.04										
V0 vs. V1	621	36	43,32%	282	13	269	59%	1	1	0,9
V0 vs. V2	1057	44	66,70%	723	18	705	56%	1	1	1,3
V1 vs. V2	1276	48	34,17%	449	13	436	87%	0	1	1,5
Fujaba BasicPackage Series										
01.01.-04.01.	11054	230	0,00%	4	4	0	100%	0	0	3,95
22.01.-25.01.	8648	237	28,05%	2435	9	2426	98%	0	0	5,15
09.02.-12.02.	6224	240	0,16%	18	8	10	98%	0	0	3,2
19.03.-22.03.	7666	267	0,03%	13	11	2	98%	0	0	3,70

Table 2: Test results

The next column (SD%) shows the percentage of elements that have been added or deleted between the two original documents. This value reveals that the extent of change can differ widely among the packages. For example in the Fujaba project the fsa package had only a few changes, whereas the basic package appears to be heavily restructured.

The next columns show the detected types of differences where SD counts the number of structural differences and OD counts all other differences, i.e. reference, attribute and move differences. Obviously, the vast majority of differences are structural differences due to the addition or the removal of elements. The others are much less frequent.

The fraction of matches found in the pre-phase is shown in the column labeled HM%. The figures show that the pre-phase performs very well in general; in most cases more than three quarter of the matches can be found in this phase. In the case of the BasicPackage series, where new versions were created after a short calender time, the fraction of matches found in the pre-phase is even around 98%.

The next columns show the quality of the detected differences. Since we use similarity heuristics for finding correspondences errors are unavoidable. Two typical error types can occur in the computation of a difference: If an element is not matched with another element although it should have been matched we call this an α error. The other way around, if an element is matched with another element, but both do not really correspond to each other, we call this a β error.

We checked for errors by reviewing the unified documents in our visualization tool, manually. By the way the Fujaba UML package was too huge for a review so we cannot give real evidence about the errors here. The figures show that errors appear very rarely. An example for an error shows up at the Ritterspiel series: Here an attribute called damage in class Laser was replaced by two attributes fieldLaserDamage and robotLaserDamage. Since fieldLaserDamage contains a lower case ’d’ and robotLaserDamage does not, the name distance calculated by LCS is smaller for fieldLaserDamage. Consequently our algorithm matches attribute damage with fieldLaserDamage. Although this seems not to be a hard error because the attributes correspond somehow, but one would expect that all attributes are structurally different, i.e., added and removed elements.

The last column presents the runtime in seconds. The runtime primarily depends on the quantity of the elements. Runtime is significantly better if more correspondences are found in the pre-phase, i.e. HM% has a high value. For example, the runtime for the Fujaba BasicPackage series reported in the third section (with HM = 87%) is much higher than all runtime shown in the last section (with HM $>$ 97%).

The tests were performed on a standard PC with an AMD 1600+ and 1 GB of memory. The runtime shown in the table are the average of 10 test runs. For the majority of the diagrams the computation of a difference takes less than 5 seconds. And even the largest diagrams in our test cases are compared within 15 seconds. The memory consumption showed up to be moderate with less than 55 MB memory consumed for the largest documents.

5 Conclusion

This paper has presented an approach for computing differences between UML models encoded as XMI files. The main features of the approach are: (a) it is generic and covers a broad range of UML diagram types; (b) the algorithm used in our approach leads to good runtimes in the case of small documents and acceptable runtimes in the case of large documents; (c) it has a very low error rate, i.e. leads to few false or missed correspondences, the quality of the differences is almost optimal; (d) it does not require persistent identifiers of diagram elements.

We have implemented the difference algorithm as well as a visualization for class diagrams as Fujaba plugins. Fujaba is available under *www.fujaba.de*. The corresponding plugins can be downloaded within Fujaba. The *XMI Support Plugin* provides import and export functions for pure class diagrams in XMI format. The *Difference Calculator Plugin* takes two XMI files as input and produces an XMI file, which contains the unified model with difference information, which may be the input for the *Difference Viewer Plugin*.

The approach has been intensively tested with class diagrams and to some extent with state charts. A significant problem at this time, and a subject of further work, is to get hold of realistic examples, if not benchmarks, of versions histories or groups of variants of documents of other document types. Only with such empirical material, one can fine-tune the configuration parameters for these document types.

Although our algorithm produces errors rarely, they can occur. So a user is probably interested in correcting these errors in view of later having one merged document. To provide correction facilities in a user-friendly way we need to implement an iterative algorithm that is capable of calculating a new partial difference, in the case of a manually changed correspondence.

Another practical problem is the lack of accuracy in the XMI specification. Currently, different tools use different methods of mapping diagram elements onto XML elements. For example, a generalization can be represented in several structurally different ways in XMI files. For each mapping method, an individual import filter must be implemented. This is a general problem for the exchange of documents between different UML tools, and not specific for difference tools.

References

[AP03] Marcus Alanen and Ivan Porres. Difference and Union of Models. Technical Report 527, TUCS - Turku Centre for Computer Science, April 2003.

[CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. pages 493–504, 1996.

[Gir02] Martin Girschick. UMLDiff: Erkennung und Analyse von Unterschieden in Klassendiagrammen und Sequenzdiagrammen. Master's thesis, Technical University of Darmstadt, 2002.

[Kel04] Udo Kelter. Dokumentdifferenzen. In *Softwaretechnik*. Online at: http://pi.informatik.uni-siegen.de/kelter/lehre/03w/lm/lm_dif_info.html, 2004. Lehrmodul zur Vorlesung.

[MGMR02] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In *18th Intl. Conf on Data Engineering (ICDE), San Jose CA*, 2002.

[Mye86] Eugene W. Myers. An O(ND) Difference Algorithm and Its Variations. In *Algorithmica*, volume 1(2):251-266, 1986.

[Nie04] J. Niere. Visualizing Differences of UML Diagrams with Fujaba. In *Proc. of the 2nd Fujaba Days, Darmstadt, Germany*, October 2004.

[OWK03a] Dirk Ohst, Michael Welle, and Udo Kelter. Difference Tools for Analysis and Design Documents. In *Proceedings of the IEEE International Conference on Software Maintenance 2003 (ICSM2003), Amsterdam*, pages 13–22, September 2003.

[OWK03b] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between Versions of UML Diagrams. In *ESEC/FSE'03, September 1-5, 2003, Helsinki, Finland*, 2003.

[Ros] IBM. *Rose, the Rational Rose case tool. Online at http://www.rational.com (last visited June 2004)*.

[RW98] Jungkyu Rho and Chisu 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, December 1998.

[SZN04] C. Schneider, A. Zündorf, and J. Niere. CoObRA - a small step for development tools to collaborative environments. In *Proc. of the Workshop on Directions in Software Engineering Environments (WoDiSEE), Edinburgh, Scotland, UK*, May 2004.

[WDC03] Yuan Wang, David J. DeWitt, and Jin-Yi 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.

[Weh04] Jürgen Wehren. Ein XMI-basiertes Differenzwerkzeug für UML-Diagramme. Master's thesis, Universität Siegen, 2004.

[ZWR01] Albert Zündorf, Jörg Wadsack, and Ingo Rockel. Merging Graph-Like Object Structures. In Andre van der Hoek, editor, *Tenth International Workshop on Software Configuration Management (SCM-10) New Practices, New Challenges, and New Boundaries May 14-15, Toronto, Canada (ICSE workshop)*. http://www.ics.uci.edu/ andre/scm10/, 2001.