Difference Computation of Large Models

Christoph Treude, Stefan Berlik, Sven Wenzel, Udo Kelter
Software Engineering Group
Dept. of Electrical Engineering and Computer Science
University of Siegen, Germany
{treude|berlik|wenzel|kelter}@informatik.uni-siegen.de

ABSTRACT

Modern software engineering practices lead to large models which exist in many versions. Version management systems should offer a service to compare, and possibly merge, these models. The computation of a difference between large models is a big challenge; current algorithms are too inefficient here. We present a new technique for computing differences between models. In practical tests, this technique has been an order of magnitude faster than currently known algorithms. The main idea is to use a high-dimensional search tree for efficiently finding similar model elements. Individual elements are mapped onto a vector of numerical values using a collection of metrics for models and a numerical representation of the names which occur in a model.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; I.7.1 [**Document and text processing**]: Document and Text Editing—*Version control*; E.1 [**Data Structures**]: Trees; F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical Algorithms and Problems—*Sorting and searching*

General Terms

Algorithms, Experimentation, Performance

1. INTRODUCTION

Model-driven engineering (MDE) is expected to become a leading paradigm of software engineering in the 21^{st} century. Assisted by powerful model transformation tools, software developers work mainly or only with models, notably class structure diagrams or activity diagrams. Initial abstract models are stepwise refined and eventually transformed into ready-to-use software. The delivered software product is either source code directly generated from the models or a set of 'executable' models.

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

Another trend is the reduction of time to market. Making release cycles shorter requires more developers to work concurrently. This leads to many different versions of a model. Support of concurrent work by version management systems becomes crucial here. An essential service of these systems is to compute the changes between two documents (i.e. one possible difference) and to merge concurrent changes.

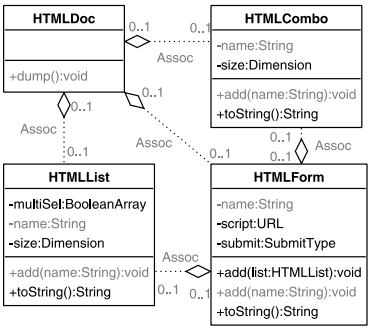
In the context of model-driven engineering, a very common use case for difference tools is to compare two UML class diagrams. Figure 1 shows an example of two small class diagrams which are revisions of each other. This example will be used throughout the remainder of this paper.

Current versioning systems fail to handle models correctly. Storing the models in textual files, e.g. XMI files, and comparing such files is inadequate because the textual representation does not have an appropriate level of abstraction. Textual representations of the same models can have many textual deviations, such as changed identifiers or differently ordered elements.

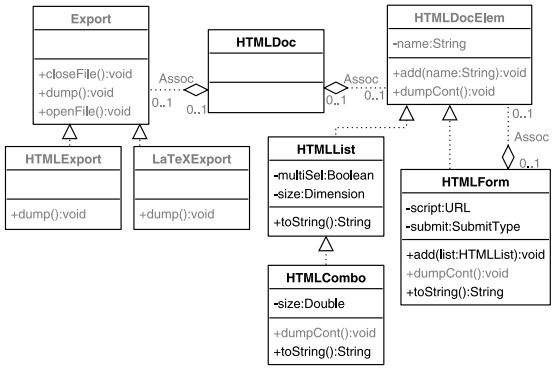
Difference tools should interpret models as attributed typed graphs. The primary structure of such a graph is a tree that contains additional cross references, e.g. the return types of operations can refer to other classes. Virtually all diagram types of the UML and technical models such as Matlab Simulink models have this basic structure.

Only very few algorithms and tools for computing differences between models have been proposed so far (s. Section 1.1). These algorithms initially determine a set of correspondences. A correspondence is a pair of equal or at least similar elements in the two models. A model element not involved in a correspondence is regarded as a local change. Examples of local changes include inserted, deleted or moved elements, or attribute modifications. Tools which display differences often use different colors to highlight local changes (see Figure 1).

The correspondences are basically determined by computing the similarity of pairs of elements and by selecting mutually most similar elements as corresponding pairs. All similarity-based algorithms known so far compute the similarity of each pair of model elements. Effectively, for each element in the first model, a linear search for the most similar element in the second model is performed. This basic approach has inherently $O(n^2)$ complexity, n being the number of elements in a model. Medium or large documents, which are quite likely in MDE, lead to runtimes in the range of 5 minutes to an hour, which are clearly unacceptable for most purposes.



(a) Initial class diagram



(b) Revised class diagram

Figure 1: Example scenario

The main contribution of this paper is a new algorithm which compares models with normal properties in $O(n \log n)$ runtime and which extends the size of models which can be compared in practice by an order of magnitude. The basic idea behind this algorithm is to use a high-dimensional balanced search tree for efficiently finding all elements which are sufficiently similar to a given one. This leads to two technical challenges: (a) to develop a suitable search tree, (b) to represent model elements in such a way in the search tree that a range query retrieves all model elements which have a minimum similarity to a given element.

Our solution to challenge (a) is the S^3V tree, a high-dimensional balanced in-memory search tree, which borrows basic ideas from high-dimensional, disc-oriented search trees for information retrieval systems (notably [7]). The S^3V tree will be presented in detail in Section 3. The solution to challenge (b) is to map model elements onto numerical vectors using a number of well-known and easily computable metrics. Details of this mapping are presented in Section 4.

We have implemented the search tree within a generic difference tool known as *SiDiff* ([9, 14]). The tool is easily configurable for virtually all models with a graph structure. Configurations are currently available for 7 UML diagram types and Matlab Simulink models. We have extensively benchmarked the S^3V tree using large class diagrams; we have measured performance gains up to a factor of 50 with large diagrams, which is consistent with our theoretical analysis. Details of the evaluation can be found in Section 5.

Our approach of efficiently finding similar model elements is not only usable in difference tools, but also for other purposes such as the analysis of version histories or clone detection in models.

1.1 State-of-the-art

A large number of algorithms for comparing documents have been proposed [8]. 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 (e.g. as proposed in [6, 17]) and (b) generic algorithms which require only some configuration data, if any at all. This paper addresses only the second class of algorithms.

Many algorithms are intended for textual documents, e.g. the LCS (Longest Common Subsequence) algorithm [12]. These algorithms are appropriate for documents such as program source code, but not for typical models stored in, say, the XMI format. XMI files are representations of graph-structured models at a physical level; comparisons at this level lead to many false, conceptually irrelevant deviations. Thus, correspondences between models must be computed on the basis of a conceptual representation.

Many difference tools are based on persistent identifiers of model elements (e.g. [1, 13]). Two model elements are determined as corresponding if they have the same identifier; their similarity does not matter. This approach is very efficient, but it fails if tools do not support persistent identifiers or if models have been re-engineered or independently developed. Moreover, it can deliver differences whose quality is questionable.

Similarity-based algorithms are applicable without such restrictions and deliver better quality. Algorithms such as LaDiff [3, 4] and XDiff [16] are only applicable to ordered or unordered trees, respectively; they are not appropriate for models because models have a cyclic structure and a mixture of node types with ordered or unordered children.

In [9] a generic differencing algorithm for models, called SiDiff, was presented. This algorithm can be configured for a wide range of model types. It is similarity-based and does not depend on persistent identifiers or unique element names. It follows the basic schema mentioned above, i.e. it initially compares all pairs of elements of the two models. SiDiff is discussed in greater detail in the next section.

2. SIDIFF – SIMILARITY-BASED DIFFERENCE

The architecture of the SiDiff system is shown in Figure 2. Documents which are to be compared are initially transformed into an internal representation, e.g. from an XML-based file format using XSLT transformations. Alternatively, the models can be created through an API or by implementing pre-defined interfaces, if SiDiff is tightly integrated into another software. The difference calculation of the new, enhanced version of SiDiff consists of three phases:

1. In an initial *hashing phase* a hash value is calculated for each element. Elements with identical hash values are immediately matched[1] and are not considered further during the subsequent phases[2].

[1]Many documents can contain cloned elements, which have equal hash values. The handling of clones is not relevant for this paper.

[2]A similar hashing phase is part of the XDiff algorithm [16].

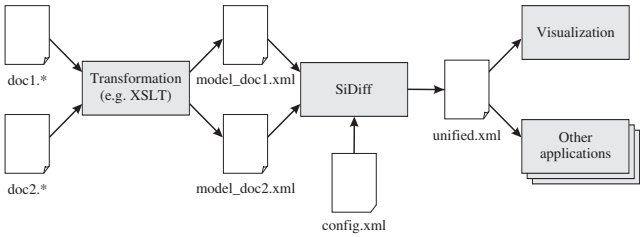


Figure 2: SiDiff Architecture

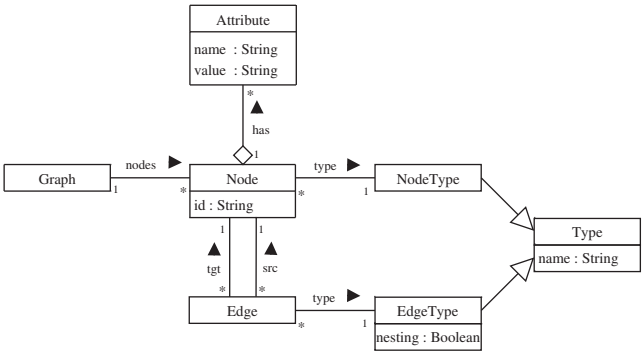


Figure 3: Data model of the SiDiff algorithm

2. In an *indexing phase*, one or several S^3V trees are created. An S^3V tree efficiently finds, for a given model element, the most similar elements in the other model. Typically, the number of returned elements can be kept very small and is independent of the size of the model.

3. The *matching phase* iterates through the elements of one model. For each element, it retrieves the most similar elements of the other model, determines exact similarities, and computes matches.

Subsequently the difference between the two documents is output in an appropriate format (see Section 2.4).

2.1 Internal data model

When initially loading a document of a specific document type (e.g. from an XML file), an internal representation of the model as a directed, typed graph is constructed. This graph is a low-level implementation of the document's graph structure. The nodes and edges of the internal graph use a fixed set of runtime object types in order to make the kernel of the system independent from specific document types. This fixed set of types used in the internal representation is shown in Figure 3.

A document, represented as *graph*, consists of several objects of type *node*. An object of type *node* (a *node* for short) represents one individual element of the model (e.g. the class *HTMLDoc*, the operation *dump()*). *Nodes* have *attributes* (e.g. name, visibility, cardinality). Each *node* is connected to an object of type *node type* which represents an element type (e.g. classes, operations, generalizations). Objects of type *edge* represent associations and similar directed connections between model elements. *Edge* objects contain a reference to an object of type *edge type* which represents the type of the model-level connection. The attribute *nesting* indicates whether connections have part-of semantics (e.g. connections from classes to operations).

2.2 Similarity computation

The properties which are relevant for the similarity of two nodes of the same type are either given by their local attributes (e.g. the names) or by other nodes in the neighborhood of these nodes (e.g. nodes that are referenced). SiDiff uses a set of compare functions to determine the similarity between two nodes, e.g. to compare two attribute values or to compare sets of neighboring nodes. These functions compare two properties of the same type which belong to different nodes. They return a value between 0 and 1; a value of 0 stands for no similarity between the nodes, a value of 1 expresses equality.

Obviously, some properties are more relevant for the similarity of nodes than others; therefore, weights must be assigned to each property. The weights have to be chosen according to the semantics of the model type and according to what users consider a significant change. For each specific model type, a configuration file describes the similarity-relevant properties of the types of model elements. Model elements are compared pairwise using the configuration which is applicable for their element type. The similarity between two elements is defined as the weighted mean of the similarities of the similarity-relevant properties. Additionally the configuration specifies for each element type a minimum similarity for two elements of this type to be eligible as corresponding elements. Table 1 shows a small excerpt of a SiDiff configuration, namely the similarity-relevant properties for class elements within UML class diagrams.

node type = Class **threshold** = 0.5	
Criterion	Weight
Similar value for attribute *name*	0.35
Equal value for attribute *visibility*	0.05
Equal value for attribute *isAbstract*	0.05
Similar set of sub elements of type *attribute*	0.20
Similar set of sub elements of type *operations*	0.20
Similar elements following incoming *generalizations*	0.05
Similar elements following outgoing *generalizations*	0.05
Matched parent element	0.05

Table 1: Criteria for comparing classes

2.3 Comparison procedure

Similarities are computed in a bottom-up/top-down order, according to the tree-like structure of most models. The algorithm starts from the leaves of the models and compares the elements of the same type in bottom-up direction. Two elements are considered similar if their similarity exceeds the given threshold. They are matched immediately if they are not similar to any other elements. Elements which are similar to several other elements are not matched immediately because the similarities might change when further elements are compared. In the following phase of the algorithm elements are matched with their most similar other element. Each match causes the algorithm to switch over to a top-down phase that propagates the new correspondence downwards to the children. The initial similarities stemming from the bottom-up phase can be improved since parent elements or referenced elements can have been matched mean-

while. Consequently, other matchings can be found, which are propagated top-down further on.

Most UML models do not have a real tree structure; they contain cross references between elements, which lead to cycles. The comparison has to handle cycles correctly. Such cycles are dealt with by iterating through a cycle as long as new matches can be found. The similarities between elements are thus propagated through the graphs. This approach is similar to the *Similarity Flooding* algorithm [11]. It allows us to compare documents such as Petri nets, which are not tree-structured and in which the similarity of elements depends mainly on their neighborhood, and not on their compositional structure.

The hashing phase has a significant impact on the efficiency of the algorithm. It has runtime complexity $O(n \log n)$ and is the fastest method of computing correspondences. It also provides trustworthy fix points that speed up the comparison of their neighbor elements.

The quality of the differences produced by our algorithm has been manually evaluated; the evaluation revealed a negligible error rate [9].

2.4 Output of the Algorithm

The result of the algorithm consists of a list of matched element pairs (e.g. the class *HTMLList* in Figure 1) and detailed information about changes. Changes are classified as follows:

Attribute change. An attribute change indicates that two corresponding elements differ in their attributes' values.

Reference change. A reference change indicates that the references between two corresponding elements have changed.

Move. Elements that appear to change their parent element between the two original documents get a move annotation with a reference to the other parent element.

Structural change. Elements that have no entry in the correspondence table are considered to be structurally different, i.e. due to insertion or deletion operations.

After a difference between two models has been computed, the information can be used for several purposes. One option is just to present the difference to a developer, another option is to start an interactive merge process. A difference can be shown in various ways, e.g. in the form of a unified document, which contains common elements only once, or in form of two parallel windows which show one version with the specific parts highlighted. Further details depend on the document type. The output of the algorithm must be adapted to the needs of the final processing of the difference tool.

3. S^{3}V TREES

Our approach to reduce the number of direct comparisons of element pairs is a search tree which arranges similar elements next to each other and which allows us to efficiently identify the most similar elements for a given element.

Within the search tree, all elements are represented by numerical vectors. Each vector index represents a certain characteristic of the model elements. Thus, each element is represented by a point in a multidimensional vector space. The similarity of two elements is defined as their Euclidean distance and the task of finding all similar elements to a given one is handled by range queries. The latter are queries for finding all elements in a specified subspace of the data area. In order to find similar elements, range queries can be called with a circular search space surrounding the particular element where the radius of the search space indicates the maximal Euclidean distance for considering two elements similar.

3.1 High-dimensional search trees

High-dimensional search trees, e.g. according to [7, 2], are access structures optimized for elements with more than three dimensions. To store the elements *buckets* of fixed size are used. Buckets represent subspaces of the entire data space which are pairwise disjoint. The partitioning of the data is governed by a so-called *directory*. The directory is a binary tree wherein each node has a split position and a split dimension. Elements are divided into two partitions according to their value at the particular split dimension. If this value is less than or equal to the split position, they are part of the left subtree, otherwise they belong to the right subtree.

In the kd-tree [2] all nodes within one level of the tree have the same split dimension. For example, the split dimension of the root node is the first dimension, the split dimension of root's children is the second dimension, and so on. If n is number of dimensions then the upper n levels of the tree, the next n levels, etc. use the same sequence of split dimensions. kd-trees do not handle high-dimensional data spaces and sparse vectors adequately.

The LSD (*local split decision*) tree [7] has a more flexible approach of handling splits: in each split, split dimension and split position are chosen independently. The choice can take into account current data and can be optimized. Obviously, it depends on the use case which characteristics describe a good distribution of the data. The original LSD tree is optimized mainly with a view to spatial accesses. The LSD tree model assumes dynamic insertions, i.e. at insertion time of an element it is not known which further elements will be inserted later.

3.1.1 Range Queries

Range queries are the fundamental operation of multidimensional access structures. The aim is to prevent linear search through the elements and to detect all elements in a neighborhood of a given one. The neighborhood of an element can be defined in several ways, most obviously by the maximal Euclidean distance to the element. To process range queries, for every node in the directory and every bucket the range containing all subtree elements is needed. For the root the minimal and maximal value of every dimension is given, thereby defining the data range. The range of a child node or bucket is given by the one of its parent node, being reduced in the split dimension according to the split position.

Figure 4 shows an example where 4-dimensional elements are stored in an LSD tree with bucket capacity of 1. In the directory nodes the split dimension and the split position is recorded. Further the data range maintained by the directory node is annotated.

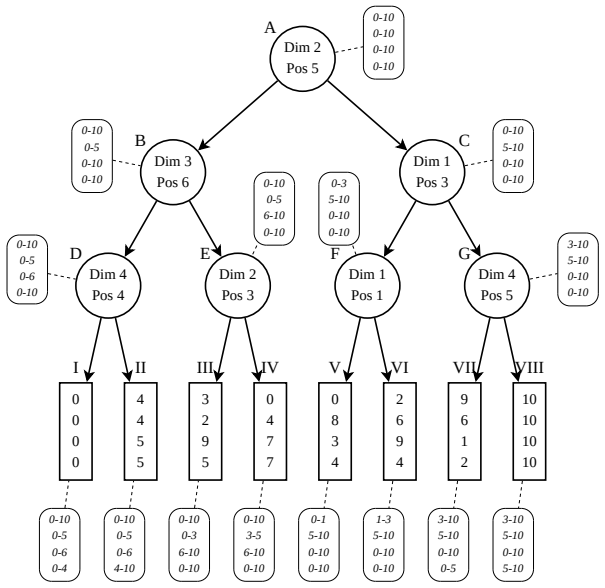


Figure 4: Exemplary range query for the LSD tree

Every range query starts at the root. Subtrees in the directory are followed up only if the search area and the data range of the subtree intersect. If the data range of a node is disjoint with the search area, the same holds per definition also for all sub elements of this node which thus need not to be examined.

3.2 Modifications of S^{3}V trees

The S^{3}V tree uses the same basic ideas as high-dimensional search trees, but has been completely redesigned in order to adapt to, and take advantage of, the conditions in the computation of a document difference.

Within the difference computation of two documents A and B, a separate S^{3}V tree is set up for every relevant element type. First, all elements of A have to be inserted into the trees. Later in the comparison phase, for given elements of B the set of similar elements of A can be found in the corresponding tree. The degree of similarity is defined by the Euclidean distance such that a range query can be applied to determine the set of similar elements. How document elements are mapped to tree elements and hence multidimensional vectors is discussed in Section 4.

The term S^{3}V is the short form of the abbreviation SSSV which stands for *similarity search sparse vector* and describes the two main features of the structure. First, it points out that this data structure is optimized for range queries and thus similarity search. Second, a characteristic of the underlying data is sketched. Elements are indexed multidimensional although many of the values are equal to zero. In the following, differences between high-dimensional search trees and the S^{3}V tree are explained and justified.

3.2.1 Memory Orientation

In contrast to most search trees the S^{3}V tree is supposed to be completely kept in memory since the set of contained objects is comparably small. Hence, limitations concerning external balancing or the bucket size can be dropped. A good choice is a bucket size of one here. The tree is then widely ramified and structured best in such a way that the advantages of the S^{3}V tree can take effect. Moreover, the linear search in buckets within range queries is no longer required.

3.2.2 Tree Construction

In our application context the search tree does not have to support insertions or deletions. All elements to be stored are known right from the beginning – they are the elements of one of the two documents. Thus, the tree should not be re-balanced stepwise with every inserted element but should rather be created systematically on the basis of all elements. Therefore in a first step all elements to be stored are separated into two disjoint sets according to some split strategy. The resulting sets are further divided in the following steps such that the S^{3}V tree evolves. Split strategies have to satisfy the following requirements: Every split should divide the initial set in two almost equally sized result sets to ensure balanced S^{3}V trees. Further, splits should be carried out in such a way that already during the construction of the tree optimizations with respect to range queries are applied.

Algorithm 1 Construction of the S^{3}V tree

```
 1: function constructRecursively(Elements es) : Node
 2:     Node node
 3:     if es.size <= bucketSize then
 4:         node = new Bucket()
 5:         node.addElements (es)
 6:     else
 7:         (sPos, sDim) = computeSplitLine(es, splitter)
 8:         left = new Node ()
 9:         right = new Node()
10:         node = new DictionaryNode(sDim, sPos, left, right)
11:         Elements esLeft = es.getSmaller(sDim, sPos)
12:         Elements esRight = es.getGreater(sDim, sPos)
13:         Elements esEqual = es.getEqual(sDim, sPos)
14:         for all element in esEqual do
15:             if esLeft.size < esRight.size then
16:                 esLeft.addElement(element)
17:             else
18:                 esRight.addElement(element)
19:             end if
20:         end for
21:         left = constructRecursively(esLeft)
22:         right = constructRecursively(esRight)
23:     end if
24:     return node
25: end function
```

The construction of the S^{3}V tree is clarified with the aid of Algorithm 1. At the beginning the function CONSTRUCTRECURSIVELY is called with all elements to be inserted as parameter. The actual construction is realized recursively by this function which returns either a bucket or a new directory node. The latter happens if the number of elements to be inserted exceeds the bucket capacity. Then also a *split line* has to be calculated for the directory node, i.e. a split dimension and split position. The function is generic in the sense that different split strategies can be realized controlled by the *splitter* parameter. As specified in the split line the directory node divides the element set into the two subsets. If an element's value matches exactly the split position it could in principal be allocated to both subtrees. However, for the sake of balance it is assigned to the smaller one. Cal-

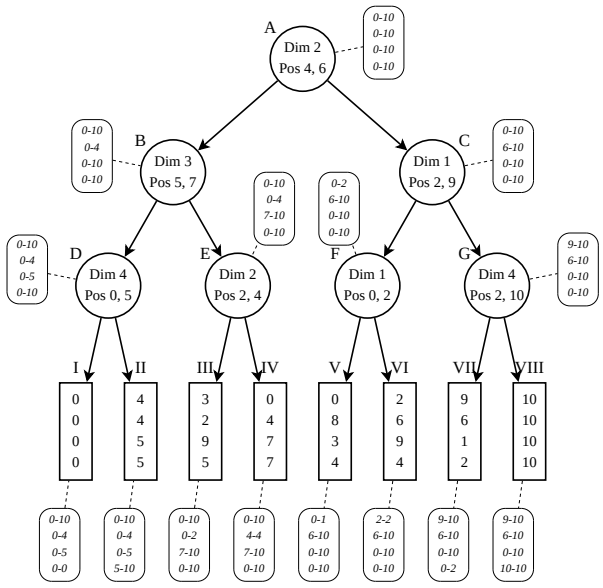


Figure 5: Exemplary range query for the S^{3}V tree

culation of the subtrees is achieved by calling CONSTRUCT-RECURSIVELY for each subtree.

3.2.3 Split Strategy and Range Query Optimization

After the insertion phase the S^{3}V tree is used exclusively for range queries. Optimization should therefore concentrate on this function. It has already been mentioned before that during range queries for every directory node and bucket the range containing all elements of its subtrees has to be calculated. Subtrees will be examined if and only if the search area and the corresponding subspace of the tree intersect. Therefore small subspaces are advantageous. However, as Figure 4 illustrates the calculated subspaces are notably larger than they need to be to store all their elements. This effect can be exploited for optimization in providing a second split position.

The S^{3}V tree has the special feature that a directory node contains the split dimension and both the maximal value of its left subtree and the minimal value of its right subtree. This feature saves runtime since fewer subtrees need to be examined.

Figure 5 shows the S^{3}V tree variant of the LSD tree example given in Figure 4 with thus two split positions instead of one. For example in the root now the split positions 4 and 6 are recorded, since these are the effective maximum and minimum of its subtrees. Compared with the entry 5 in the corresponding LSD tree additional information is gained and the subspaces of the directory nodes B and C and all of their subtrees can be reduced accordingly.

This structural expansion should also be reflected in the split strategy. An optimal split during the construction would divide the element set in two equally sized subsets (balancing) and at the same time show a preferably large distance between the both split positions (reduction of the search area). The two objectives may be conflicting in the individual case and it has to be decided how to proceed case-by-case. One possibility is to sort the element set successively in each dimension and to choose the dimension with maximal difference between the elements $\frac{n}{2}$ and $\frac{n}{2}+1$ as split dimension, when n denotes the number of elements. This way subspaces would be minimal coevally ensuring a balanced tree. In general, this can be achieved better for high dimensional problems which implicitly offer more potential split dimensions. For example, with a central element $(4.5, 4.5, 4.5, 4.5)^T$ and a radial Euclidean search radius of 1 within the trees shown above, the number of nodes to be visited reduces from 9 to 4 and the number of nodes whose subspace has to be calculated declines from 11 to 7.

3.2.4 Runtime Analysis

To conclude this section the runtime of the construction operation and range queries for the S^{3}V tree is discussed.

The critical operation during the construction process is splitting the element set according to some split strategy. It is called once for each directory node. Provided that a split strategy is chosen that guarantees balanced trees there will be $\frac{n}{b}-1$ directory nodes in the S^{3}V tree if n and b denote the number of elements and the bucket capacity respectively. For a bucket capacity of 1 the number of necessary splits is thus $n-1$, i.e. it is linearly dependent of the number of elements.

Only the first split for the root directory node has to consider all elements. Splits for deeper nodes have to consider accordingly smaller sets. Notice that for a complete tree with a bucket capacity of 1 more than half of the splits have simply to divide two elements.

The worst case runtime for range queries is $O(n)$. This is the case if all elements stored in the tree lie inside the search area. Then all buckets and directory nodes have to be evaluated. Provided a bucket capacity of 1 an S^{3}V tree with n elements consists of as many buckets and $n-1$ directory nodes, hence $2n-1$ subspaces have to be calculated. The worst case behavior cannot further be improved. However, this case is rather untypical and the runtime complexity in the average case is clearly better. Using the presented split strategy with two split positions per directory node further reduces the probability that a node of the S^{3}V has to be considered during a range query. Because the construction guarantees balanced trees they cannot degenerate and their maximal height is bounded by $O(\log n)$.

4. ELEMENT MAPPING

In order to be stored in S^{3}V trees each model element is mapped onto a numerical vector. The vectors consist of a *metrical* and a *lexical* part.

4.1 Metrical Indices

We define metrical indices as vector positions that represent numerical values which are obtained by computing metrics for model elements.

Many metrics have been proposed for UML models, see e.g. [10] for an overview. Metrics do not only represent quality features, they can also be used to measure the similarity of model elements.

Metrics are always specific for a document type, at least semantically. However, if one represents models as a typed graph according to the structure given in Figure 3, it turns out that virtually all metrics consist in simply counting certain sets of nodes in the neighborhood of the node which represents the model element to be measured. For instance,

Table 2: Exemplary vector of metrical indices

Index	Meaning	Value	Norm
LON	length of name	8	0.4
NAM	# abstract methods	0	
NAPAC	# package-visible attributes	0	
NAPRI	# private attributes	3	1
NAPRO	# protected attributes	0	
NAPUB	# public attributes	0	
NCV	# class variables	0	
NIV	# instance variables	3	1
NMPAC	# package-visible methods	0	
NMPRI	# private methods	0	
NMPRO	# protected methods	0	
NMPUB	# public methods	2	0.67
NOA	# attributes	3	1
NOC	# children in class hierarchy	0	
NOCM	# constructors	0	
NOM	# methods	2	0.67
SUP	# superclasses	0	

these sets of neighbor nodes are nodes of a given type (s. e.g. NOM in Table 2) or nodes of a given type with a special value of an attribute (e.g. NCV).

Such node sets are easily specified using path expressions (similar to XPath). These expressions can be provided in resource files in order to keep the kernel of the difference tool free of code which is specific for single document types.

Only very few metrics cannot be handled this way, e.g. metrics where inheritance is involved. Our empirical results indicate that it is not necessary to use such metrics in addition to the easily computable ones, because the latter alone yield good hit ratios.

Table 2 shows the metrical part of the vector that results from the UML class *HTMLList* of the diagram given in Figure 1(a).

The choice of metrics to be calculated for a certain element type is somewhat arbitrary. However, given the underlying meta-model and document-type specific semantics, one can deduce a set of significant metrics for each element type easily. A first evaluation usually reveals whether the chosen metrics are sufficient to represent element-specific characteristics. While selecting appropriate metrics is a creative process, our experience has shown that it hardly takes more than a couple of minutes to define a suitable set of metrics.

4.2 Lexical Indices

Similar names are particularly relevant for the similarity of document elements. Metrical indices do not cover such similarities. Thus a method which maps similarities of names onto numerical values is needed.

The easiest way to achieve this is to use a separate index for each name that occurs in the document. The set of all names occurring in a document can be easily determined. Each name is mapped onto a vector position. If a document element has name N the vector for this element contains a 1 at the position for name N. All positions belonging to other names are set to 0. Vector positions that represent names are called *lexical indices*.

Using this approach only exactly matching names can be found. In order to also handle name similarities with the vector approach substrings of names have to be used as lexical indices, too. For UML class diagrams one can exploit the common convention that new substrings of a name begin with capitals (e.g. *getAttributeValue()*). In general similar parts of the element names can be found with the LCS algorithm [12], albeit it is of quadratic complexity when used pairwise.

Table 3: Exemplary vector of lexical indices

Index	Type of index	Value	Norm
HTMLDoc	class name	0	
HTMLCombo	class name	0	
HTMLList	class name	1	
HTMLForm	class name	0	
Export	class name	0	
HTMLDocElem	class name	0	
HTMLExport	class name	0	
LatexExport	class name	0	
HTML	substring of class name	1	
Doc	substring of class name	0	
Combo	substring of class name	0	
List	substring of class name	1	
Form	substring of class name	0	
Elem	substring of class name	0	
Latex	substring of class name	0	
Export	substring of class name	0	
dump	method name	0	
add	method name	1	0.5
toString	method name	1	
closeFile	method name	0	
openFile	method name	0	
dumpCont	method name	0	
to	substring of method name	1	
String	substring of method name	1	
close	substring of method name	0	
File	substring of method name	0	
open	substring of method name	0	
dump	substring of method name	0	
Cont	substring of method name	0	

In addition to mapping element names lexical indices can be utilized to map names of sub elements. For example the vector of an UML class can besides its name also contain the names of its methods.

Lexical indices can generally be used for all document elements that have characteristics represented by character strings.

Table 3 shows the lexical part of the vector that results from the UML class *HTMLList* given in Figure 1(a). Unlike with metrical indices the calculation of lexical values is not limited to the element whose vector is set up. On the contrary the set of all element names in all documents to be compared has to be determined to define the vector indices. For that reason Table 3 also contains the names and parts of names of the other elements given in the example scenario.

4.3 Normalization and Scaling

To prepare a useful basis for the similarity assignment the vectors have to be normalized and scaled. Normalization causes a homogeneous starting situation such that the vectors can be configured with scaling.

4.3.1 Normalization

As can be seen from Table 2 the ranges of the indices vary. Hence changes impact differently on the similarity of vectors, based on the certain index where they appear.

Since these differences result randomly from the definition of the indices and are meaningless with respect to the importance of the changed similarity, the ranges of the indices

have to be normalized. An obvious choice is the range between 0 and 1. Then for every element v_i in a vector $\vec{v} \in V$ the new value v_{i_norm} is given by

$$v_{i\_norm} = \frac{v_i - \min(\{x_i | \vec{x} \in V\})}{\max(\{x_i | \vec{x} \in V\}) - \min(\{x_i | \vec{x} \in V\})}. \quad (1)$$

Equation (1) cannot be evaluated if the minimum and maximum of the range coincide, but in this case normalization is not needed anyway since this index is irrelevant regarding similarity predications.

A problem with the lexical indices is that it cannot a priori be stated how many of them will emerge. Obviously their number is proportional to the total number of elements in both documents. Typically the lexical sub vector contains many zero valued entries (cf. Table 3) such that the resulting distance is small. In the simplest case with only one lexical index type mapping the element names, the maximal distance is independent of the vector length given by $\sqrt{2}$. Therefore, no additional normalization of the lexical sub vectors with regard to their length is needed.

The normalized values are indicated in Tables 2 and 3 in the column *Norm* if they differ from the original values.

4.3.2 Scaling

Normalization provides a homogeneous starting point for an index-based scaling of vector values. The scaling of values at certain indices is a powerful tool to influence Euclidean distances in order to define how much impact these values should have on the similarity of elements. A scaled value then reads $v_{i\_scale} = v_i \cdot f_i$ where v_i and f_i denote values at a certain index and a scaling factor, respectively. The higher the scaling factor, the more influence the corresponding vector index has on the Euclidean distance between vectors.

4.4 Generalization

The concept of metrical and lexical indices can be generalized to almost all types of documents with an attributed typed graph model. Metrical indices can be set up starting at some node for all its neighboring nodes grouped by types or other characteristics derived from attributes. Lexical indices can be defined on every attribute of a node and its neighboring nodes.

Thus an extensive set of possible indices is given that has to be reduced to the meaningful with respect to similarity.

5. EVALUATION

To evaluate our approach for optimizing the difference computation for large model documents, we integrated the concept of S^3V trees into the generic difference tool SiDiff.

With the integration of S^3V trees, the comparison algorithm can be modified as follows. Instead of comparing each element in the first document to each element in the second document, we start a range query for each element in the first document and only compare it to elements from the result set of the query. All other similarity values are set to 0. Since only elements of the same type are to be compared, a separate S^3V tree has to be generated for each element type.

The crucial part of this approach is defining the radii of search areas, i.e. the threshold for range queries. It has to be high enough to return all elements that are candidates for correspondences and it has to be low enough to reduce the amount of elements to be compared significantly. Using two different thresholds has proven to meet these requirements. The first range query is initiated with the lower threshold. Only if this query does not return any elements, a second query with the higher threshold is started. Due to the efficient implementation of range queries in S^3V trees, the higher amount of range queries does not have any considerable impact on the overall runtime.

We selected several test scenarios, mainly different versions of UML class diagrams used before in [9], to test our algorithm. Some of those diagrams have been reverse engineered from source code that was retrieved from version management systems. The diagrams differ in size and in the amount of changes that have been made between different versions. The following test scenarios were used:

HTML Package. The two small class diagrams shown in Figure 1, to test and verify our algorithm. (Test Scenario A)

Ritterspiel. Two versions of a class diagram from a project group at the University of Kassel where a board game was designed over six months. The versions were retrieved from an internal version management system and have not been reverse engineered from source code. (Test Scenario B)

Version histories. Two versions of a class diagram representing a SiDiff plug-in for tracing model elements in version histories. (Test Scenario D)

Fujaba packages. Four large class diagrams with two versions each, derived from the source code of the UML tool Fujaba [5]. The package ASG (Abstract Syntax Graph) was used for Scenario C, while Test Scenario E involves the package FSA (Fujaba Swing Adapter) and Test Scenario F contains the Fujaba Basic package. Scenario G is the largest one. It comprises the Fujaba UML package with all its sub packages.

The evaluation results are shown in Table 4. The first column (*Scen.*) identifies the particular scenario; the next two columns show the size of the class diagrams in terms of the total amount of elements (*Elem.*) and classes (*Cl.*) in the larger of the compared versions. Column *Hash.* indicates whether a hashing phase was used. Each scenario was tested with and without hashing. The thresholds for range queries in the S^3V trees are shown in columns *t1* and *t2*. While the values are set to 1.5 and 2.3 respectively for most scenarios, they are varied in Scenarios C and D.

Column *CS* (Compute Similarities) indicates the total number of direct comparisons between elements of the two documents. This number is crucial because direct comparisons consume the largest part of the total runtime. Column *CS%* sets the values in relation to the number of similarity computations that occurred in the version without S^3V trees. For example, in Scenario G without hashing, the number of direct comparisons was reduced by 97.9% to a value of 118,974. Column *RQ* shows the number of range queries in S^3V trees performed during the calculation. The number of distance calculations between two vectors is shown in column *VD*.

Since heuristics are used for finding correspondences, errors are unavoidable. Compared to the difference results obtained without S^3V trees, there are potentially two kinds of errors:

Table 4: Test results

Scen.	Elem.	Cl.	Hash.	t1	t2	CS	RQ	VD	Match%	CS%	RT_p(s)	RT(s)
A	54	13	true	1.5	2.3	143	112	586	100	32.8	0.2	0.1
			false	1.5	2.3	175	134	779	110.5	25.4	0.3	0.2
B	318	28	true	1.5	2.3	92	32	1099	100	22.1	0.1	0.4
			false	1.5	2.3	558	246	14352	100	5.1	2.2	0.8
C	346	53	true	1.5	2.3	340	120	5939	100	26.6	0.4	0.5
			false	2.3	–	13648	530	31931	100	67.9	2.2	1.7
			false	0.0	2.3	1573	509	5369	100	7.8	2.2	0.8
			false	0.5	2.3	1221	487	10619	100.7	6.1	2.2	0.7
			false	1.0	2.3	1125	493	17010	100	5.6	2.2	1.1
			false	1.5	2.3	1396	512	29673	100	6.9	2.2	0.9
			false	1.8	2.3	5888	536	31990	100	29.3	2.2	1.9
D	425	40	true	2.3	–	2082	205	13966	100	70.6	1.5	1.1
			true	0.0	2.3	1260	314	10793	100	42.7	1.5	1.1
			true	0.5	2.3	1001	296	10858	100	34.0	1.5	0.5
			true	1.0	2.3	735	254	9720	100	24.9	1.5	0.7
			true	1.5	2.3	707	248	14996	100	24.0	1.5	0.9
			true	1.8	2.3	989	242	15663	100	33.5	1.5	0.7
			false	1.5	2.3	1967	685	52368	100	6.0	1.8	1.1
E	1174	102	true	1.5	2.3	19376	425	166679	100	23.9	1.9	1.7
			false	1.5	2.3	25339	2261	827644	99.7	5.7	14.0	4.7
F	1130	122	true	1.5	2.3	61568	4867	1285763	99.7	31.3	3.3	5.9
			false	1.5	2.3	213247	7875	2119694	99.6	26.7	18.2	11.8
G	4777	391	true	1.5	2.3	14204	1382	1152551	99.9	11.7	45.8	7.8
			false	1.5	2.3	118974	8023	9701013	99.8	2.1	1409.4	48.7

- Fewer matches are detected because too many elements are filtered out by the range queries.
- More, or other, matches are detected. Since the original difference algorithm matches two elements immediately if there are no other similar elements, it is possible to find other matches because elements are filtered out by the range queries.

The column *Match%* in Table 4 shows how many of the matches detected by the version without S^3V trees were also detected by the S^3V tree version. The situation that matches were missed and others were added in the same scenario did not occur in our test runs. The slight discrepancies in the results can be attributed to the fact that S^3V trees provide a hashing-like way of pre-assigning elements to each other. In the cases where more matches were detected, the difference results were exactly the same as in the respective test-runs with hashing. Other discrepancies are due to a certain subjectivity of detected deviations.

The last two columns present the runtime advantage achieved by our approach. RT_p gives the runtime with pairwise similarity computation in seconds (i.e. SiDiff without S^3V trees); RT shows the runtime after integrating S^3V trees. The time used to set up the trees is not considered as it increases linearly with the amount of elements and not quadratically like the runtime of the comparison phase. Thus, it is not critical for the difference computation of large model documents. For the largest test documents in Scenario G, an improvement from 45.8 seconds to 7.8 seconds could be reached with hashing, while the improvement without hashing was from more than 23 minutes to less than 50 seconds.

The figures show that errors appear very rarely and that the runtime advantage is significant for large documents and still measurable for smaller documents. In addition, the number of direct similarity computations is reduced substantially. On the other hand, the increasing number of range queries and vector distance calculations for large documents has almost no impact on the overall runtime. The results also indicate that using two thresholds for range queries in S^3V trees is a feasible way to reduce runtime. However, if there is a second threshold, the exact value of the first threshold only plays a minor role – the results with different combinations for Scenarios C and D do not differ much.

Table 5: Dual split positions

Scen.	Splits	Avg.	dist=0	dist=0 in %
A	50	0.67	7	14.0
B	307	0.56	79	25.7
C	335	0.54	56	16.7
D	424	0.60	61	14.4
E	1163	0.47	317	27.3
F	1119	0.51	216	19.3
G	4897	0.54	848	17.3

The comparably long runtime RT in Scenario F is produced by the large number of direct comparisons which is caused by the specific characteristics of the model elements in this scenario. In particular, a lot of elements are very similar to each other. This prevents some of the advantages of S^3V trees from taking effect. The amount of elements that can be sorted out by S^3V trees is considerably smaller than in the other scenarios.

Similar tests have been performed with Matlab Simulink diagrams. The results can be found in [15].

One of the assumptions for constructing the S^3V tree was the ability to find split dimensions such that the distance between the two split positions is considerably higher than 0. Table 5 shows the average distances between the split

positions for all S^3V trees in our test scenarios. With a possible maximum distance of 1 due to scaling, the average distance is usually greater than 0.5. This implies a major improvement for range queries. At each inner node, the data spaces of the subtrees are reduced by more than 0.25 on average. Especially with two thresholds where the first one is relatively small, many subtrees can be ruled out due to dual split positions.

Table 5 also shows that the worst case with no distance between the split positions (i.e. dist=0) does not occur often. An improvement compared to the LSD tree can be reached for about 80% of all splits.

6. CONCLUSION

This paper addresses the problem that currently known algorithms for computing a difference between models are too slow in the case of large models, notably class diagrams. Starting point of our solution is the pairwise comparison of elements which consumes most of the runtime.

We proposed a data structure called S^3V tree which arranges similar model elements next to each other and which allows us to find elements similar to a given one without accessing all candidates. S^3V trees manage numerical vectors and are basically high-dimensional in-memory search trees optimized for range queries.

Our method of mapping model elements onto vectors consists of metrical and lexical indices. Metrical indices generalize the concepts of software metrics in order to describe the characteristics of model elements by numerical values. Lexical indices are used to map name similarities to vectors. Therefore, names of elements along with several substrings are used as indices. We also showed pragmatics of how to use the search tree in the process of computing a difference.

The practical validation shows that the difference calculation becomes an order of magnitude faster for large models.

Our method of comparing large sets of model elements is not only applicable to the computation of differences, but in principle and in an adapted form to other problems such as the detection of clones or patterns within large models; these applications are the subject of current work.

7. REFERENCES

[1] Marcus Alanen and Ivan Porres. Difference and union of models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, Oct. 2003.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[3] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD*, pages 26–37, 1997.

[4] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. *SIGMOD*, 25(2):493–504, 1996.

[5] FUJABA Tool Suite Developer Team. FUJABA Tool Suite. `http://www.fujaba.de/`.

[6] Martin Girschick. Difference detection and visualization in UML class diagrams. Technical report, TU Darmstadt, 2006.

[7] Andreas Henrich, Hans-Werner Six, and Peter Widmayer. The LSD tree: spatial access to multidimensional point and non-point objects. In *Proc. of the 15th Intl. Conf. on Very large data bases*, pages 45–53, Amsterdam, The Netherlands, Aug. 22-25, 1989. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[8] Udo Kelter. Dokumentdifferenzen. In *Softwaretechnik III*. Online at: `http://pi.informatik.uni-siegen.de/kelter/lehre/lm/dif`, 2007.

[9] Udo Kelter, Jürgen Wehren, and Jörg Niere. A generic difference algorithm for UML models. In *Proceedings of the SE 2005*, Essen, Germany, March 2005.

[10] Michele Lanza. Combining metrics and graphs for object oriented reverse engineering. Master's thesis, University of Bern, Switzerland, 1999.

[11] 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.

[12] Eugene W. Myers. An O(ND) difference algorithm and its variations. In *Algorithmica*, volume 1(2):251-266, 1986.

[13] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between Versions of UML Diagrams. In *Proc. of the ESEC/FSE'03, Helsinki*, Sept. 2003.

[14] University of Siegen Software Engineering Group. Homepage of the SiDiff Project. `http://www.sidiff.org/`, 2006.

[15] Christoph Treude. Einsatz multidimensionaler Suchstrukturen zur Optimierung der Bestimmung von Dokumentdifferenzen. Master's thesis, University of Siegen, 2007.

[16] Yuan Wang, David J. DeWitt, and Jin-Yi Cai. X-Diff: An effective change detection algorithm for XML documents. In *19th Intl. Conf. on Data Engineering, March 5-8, 2003 - Bangalore, India*, 2003.

[17] Zhenchang Xing and Eleni Stroulia. UMLDiff: An algorithm for object-oriented design differencing. In *Proc. of the Intl. Conf. on Automated Software Engineering (ASE'05)*, pages 54–65, Long Beach, CA, USA, Nov. 2005.