DIFFERENCING AND MERGING OF SOFTWARE DIAGRAMS
State of the Art and Challenges

Sabrina Förtsch and Bernhard Westfechtel
Bayreuth University, Applied Computer Science I
95440 Bayreuth, Germany

Keywords: Merging, Differencing, UML Diagram.

Abstract: For long, fine-grained version control for software documents has been neglected severely. Typically, software configuration management systems support the management of text or binary files. Unfortunately, text-based tools for fine-grained version control are not adequate for software documents produced in earlier phases in the software life cycle. Frequently, these documents have a graphical syntax; therefore we will call them software diagrams. This paper discusses the current state of the art in fine-grained version control (differencing and merging) for software diagrams with an emphasis on UML diagrams.

1 INTRODUCTION

Software engineers create a large variety of artifacts such as requirements definitions, software architectures, program code, etc. All of these artifacts are subsumed under the generic term *software document*. Throughout its life cycle, a software document evolves into multiple *versions*, each of which records a snapshot of its evolution. Version control has been studied for a long time in the discipline of *software configuration management* (see e.g. (Conradi and Westfechtel, 1998) for an overview).

This paper investigates *fine-grained version control* for *software diagrams*. In the early phases of the software life cycle, software documents with a graphical syntax are used widely; consider e.g. data flow diagrams, entity-relationship diagrams, UML diagrams (the primary focus of this paper), etc. Traditional software configuration management provides version control for text files or binary files. Low-level support of this kind is not sufficient to compare or merge software diagrams on a conceptual level. Therefore, structure-based algorithms and tools are required for *differencing* and *merging* of software diagrams.

The rest of this paper is structured as follows: Section 2 introduces some basic notions, providing the foundation for the following sections. Section 3 states general requirements for differencing and merging. Section 4 briefly reviews previous work on differencing and merging of program code. Section 5, which constitutes the core part of this paper, deals with differencing and merging of software diagrams. Section 6 gives an overview of existing tools for differencing and merging, Section 7 concludes the paper.

2 BASIC NOTIONS[1]

A difference is represented formally as a *delta*. There are two kinds of deltas: A *symmetric delta* of two versions v_1 and v_2 contains all elements which belong to v_1 but not to v_2 and vice versa. Using set notation loosely, the symmetric delta may be written as $\Delta(v_1, v_2) = (v_1 \backslash v_2) \cup (v_2 \backslash v_1)$. In contrast, a *directed delta* starts from one of the versions - say v_1 - and creates the other one (v_2) by applying a sequence of operations. Thus, a directed delta may be formalized as a sequence $\Delta = op_1 \ldots op_m$ such that $\Delta(v_1) = v_2$.

Merging denotes the process of combining n alternative versions $a_1, \ldots, a_n$ into a consolidated version m. Usually, $n = 2$, which will be assumed in the following. *Two-way merging* compares two versions a_1 and a_2 with the help of a diff algorithm which calculates a symmetric delta. When a differing element is

[1] for terminology see (Conradi and Westfechtel, 1998)

Förtsch S. and Westfechtel B. (2007).
DIFFERENCING AND MERGING OF SOFTWARE DIAGRAMS - State of the Art and Challenges.
In *Proceedings of the Second International Conference on Software and Data Technologies - SE*, pages 90-99
DOI: 10.5220/0001342900900099

detected, the user has to decide whether the element is to be included into the merge version. Furthermore, the user may have to decide upon the relative arrangements of elements in the merge version, e.g., when two different lines occur at the same position in two text files.

In the case of two-way merging, any difference requires a user interaction. Frequently, the alternative versions have been derived from a common base version b, and the merge is intended to combine the parallel changes to the base version. Thus, *three-way merging* compares three versions b, a_1, and a_2 and constructs a merge version m incorporating the changes from b to a_1 and a_2, respectively.

Three-way merging increases the level of automation by consulting the base version as an arbitrator in the case of differences. For example, when a line in a text file occurs in only one of the alternative versions, it is inserted into the merge version if and only if it has not been present yet in the base version. A conflict occurs in the case of contradicting changes, e.g., when two lines have been inserted at the same position. User interaction is required only in the case of conflicts.

The task of three-way merging may be characterized as follows: Given a base version b, two alternative versions a_i and two directed deltas $\Delta_i = \Delta(b, a_i)(i = 1, 2)$, construct a merge version m and a merge delta Δ_m such that $\Delta_m(b) = m$ and Δ_m constitutes an order-preserving, complete merge of the operation sequences Δ_1 and Δ_2. Thus, merging builds upon differencing, but adds further complications: In general, it cannot be guaranteed that the input deltas may be merged successfully. For example, an operation op from Δ_1 may be overridden by an operation op' from Δ_2, or op may not be executable any more after the execution of op'. In these cases, a conflict occurs because op and op' do not commute. But even when such conflicts are not detected, the result of the merge may not make sense at all if the merge is performed at a too low level of abstraction.

3 REQUIREMENTS

In this section, we will define potential requirements for diff and merge tools. The attribute "potential" indicates that these requirements may be posed in one application context, but they may also be considered irrelevant in another context. Furthermore, since requirements may contradict each other, one requirement may have to be traded against another requirement.

Table 1 lists requirements for differencing tools. (R1) and (R2) refer to the quality of the result produced by the diff algorithm. (R3) and (R4) ensure reusability. (R5) enables the comparison of diagram versions which were created independently. (R6) re-

Table 1: Requirements for differencing.

(R1)	**Accuracy:** The diff tool should calculate the difference between two versions v_1 and v_2 as precisely as possible.
(R2)	**High conceptual level:** The diff tool should report differences on a high level of abstraction, i.e., it has to operate on a logical rather than a physical level.
(R3)	**Domain independence:** The diff tool should be applicable to a large set of diagram types.
(R4)	**Tool independence:** The diff tool should be independent of the tools which were used to create the diagram versions to be processed.
(R5)	**History independence:** The result produced by the diff tool should depend only on the final states of the diagram versions, but not on the history of edit operations used to create these versions.
(R6)	**Efficiency:** The diff tool should calculate its output as fast as possible, requiring as little space as possible.
(R7)	**User-friendly representation:** The diff tool should represent its output in a user-friendly way.
(R8)	**Lightweight approach:** Implementation of the merge tool should require as little effort as possible.

Table 2: Requirements for merging.

(R9)	**Conflict detection:** The merge tool should detect conflicts between the changes to be merged.
(R10)	**Conflict resolution:** The merge tool should support the resolution of conflicts detected during the merge.
(R11)	**User interaction:** The merge tool should offer an interactive mode, where conflicts are resolved according to user decisions rather than automatically.
(R12)	**Three-way merging:** The merge tool should support three-way merging, which employs a common base version as an arbitrator in order to eliminate unnecessary user interactions.
(R13)	**Preservation of consistency:** The merge tool should preserve the consistency level of the input versions as far as possible when producing the merge version.

quests for a fast response, which is important when processing large volumes of data. (R7) ensures that the result is represented in a user-friendly way. (R8) is motivated by reducing the implementation effort.

Of course, requirements may contradict each other. For example, efficiency may contradict accuracy, domain independence may stand in conflict with operation at a high conceptual level, etc.

Please note that all of these requirements hold for merge tools, too. Thus, Table 2 lists only those requirements which are specific to merge tools. (R9) and (R10) demand conflict detection and conflict resolution, respectively. (R11) calls for conflict resolution by user interaction. We believe that automatic (default) decisions are too dangerous. (R12) prefers three-way merging over two-way merging because the amount of user interaction may be reduced by consulting a common base version. However, two-way merging may still be required if the base version is not known or the alternative versions have been developed independently[2]. Finally, (R13) requires that the result of the merge should be "as consistent as possible".

4 DIFFERENCING AND MERGING OF PROGRAM VERSIONS

For a long time, differencing and merging has been studied primarily for versions of programs, since eventually each software process has to produce an executable program (other artifacts are often considered merely as "documentation"). For a survey of program merging, see (Mens, 2002).

It is interesting to note that up to now *text-based tools* dominate the current state of practice. Text-based tools for differencing and merging have been provided as stand-alone tools; in addition, they have been implemented in both commercial and free software configuration management systems. The underlying technology is cheap, efficient, and widely applicable as it stands.

Text-based tools for differencing and merging are characterized by the following features:

- Usually, text files are treated as sequences of text lines, i.e., text lines are considered as atomic units. However, some tools operate at a more fine-grained level (sequences of characters).

[2]Three-way merging partially conflicts with history independence, since it assumes a common base version. However, change logs are not necessarily assumed for three-way merging.

- Differences are calculated a posteriori without assuming any historical information (logs of changes). Thus, differencing and merging does not rely on tools recording change logs. In particular, no information beyond the actual text (e.g., unique identifiers of text lines) is required.

- Even for text files, there is no unique formal definition of the term *difference*. The actual meaning of this term depends on the level of granularity (lines or characters) and on the set of edit operations which are taken into account (insert, delete, move, copy).

- For text files, there are *exact algorithms* available which calculate the minimal difference with respect to a formally defined *metric*. For example, (Hunt and Szymanski, 1977) calculates the longest common subsequence (lcs), while (Tichy, 1984) additionally covers block moves.

- Text-based merging usually relies on the lcs algorithm for comparing text files line by line (e.g. the well-known Unix utility *diff3*). Thus, changes within one line and moves cannot be handled by such tools.

Remarkably, text-based merging can guarantee no more than a text file as output. As a consequence, the result of the merge may contain syntactic and semantic errors. Furthermore, syntactic and semantic conflicts may go undetected. These shortcomings have triggered numerous research activities at the syntactic or semantic level (e.g. (Buffenbarger, 1995) and (Horwitz et al., 1989)).

So far syntactic or semantic differencing and merging of programs have been implemented not only in some research prototypes but also in widely used development environments (see the eclipse plugin compare). On the one hand, the required technology has proved to be much more sophisticated than for text-based tools. On the other hand, some approaches are severely constrained with respect to the set of programs to which they can be applied (in particular, this statement holds true for semantic merging).

In contrast, text-based tools perform very badly in theory, but fairly well in practice. As noted in (Mens, 2002), empirical evaluations have shown a very high fraction (more than 90%) of successful, non-conflicting merges. Even when the merge result is not consistent, errors injected by the merge are usually caught by the compiler or by failing regression tests. Nevertheless, merging cannot be trusted blindly, and has proved difficult when it is performed after a fairly long time of parallel development (Wiborg-Weber, 1999).

5 DIFFERENCING AND MERGING OF SOFTWARE DIAGRAMS

Treating diagrams as texts is possible when a textual format is defined which may be used as backup or for exchanging data between different tools. Nowadays, all kinds of diagrams may be stored as XML documents, i.e., *structured text*. Effectively, this means that documents are represented as trees, augmented with cross-references. In contrast, the term *plain text* refers to a flat text, consisting of a sequence of text lines.

Viewing diagrams as plain text is not very helpful for differencing and merging (see (Ohst and Kelter, 2002)). Text-based tools for differencing and merging are sensitive to changes of the order in which lines appear in a text file, and they are also sensitive to changes in the layout such as e.g. the applied rules of indentation. To a large extent, the order of text lines and their layout is immaterial to the diagram which is represented by the text. Therefore, applying diff and merge tools at the level of plain text will hardly produce meaningful results. Instead, some suitable structural representation has to be used.

Below, we will explore several *design decisions* which affect functionality, user interface, and efficiency of tools for differencing and merging of software diagrams. In particular, we will discuss the respective trade-offs that have to be taken into account.

5.1 Delineation of the Domain

In the first place, it has to be decided to which types of diagrams the diff or merge tool is going to be applied. For example, the tool may operate on any kind of UML diagrams (Kelter et al., 2005), a specific kind of UML diagram (e.g., class diagrams (Ohst et al., 2003; Xing and Stroulia, 2005)), any diagram processable by a meta-CASE tool (Mehra et al., 2005), etc. The trade-off which has to be made concerns the requirements (R2) and (R3): A tool which is applicable to a large domain can make only basic assumptions with respect to the contents of the diagram to be processed.

5.2 Determination of a Document Model

After having fixed the domain, the tool developer has to design a *document model* defining the elements, relationships, and attributes to be considered. The document model has a strong impact on the capabilities of the diff or merge tool. Via the document model, views are defined on the diagrams to be processed. A simple document model allows for simple (R8) and (relatively) efficient (R6) algorithms, but lowers the conceptual level of differencing or merging (R2).

Considerably differing document models have been proposed for differencing and merging software diagrams. For example, (Engel et al., 2006; Alanen and Porres, 2003), (VVU, 2007, Ahrens) are based on MOF and are thus applicable to MOF instances, including UML diagrams. (Kelter et al., 2005; Xing and Stroulia, 2005) rely on tool-specific document models (trees augmented with cross-references). In (Soto and Münch, 2006) diagrams are transformed into RDF. (VVU, 2007, Störrle) proposes to transform diagrams into Datalog clauses. The motivation to transform diagrams into some generic model is to reuse generic tools and algorithms for differencing and merging. Graphs are another promising document model (VVU, 2007, Ebert et al.).

The document model has to be selected carefully. Differencing and merging have to be performed at an adequate level of abstraction such that a *conceptual mismatch* is avoided. In particular, this issue has to be taken into account when a document is transformed into another representation for the purpose of differencing and merging: The results of differencing and merging have to be translated back into the “native” document model. In the case of differencing, this means that sets of low-level differences have to be aggregated into high-level differences. Likewise, for merging it has to be checked whether combinations of low-level changes may be composed into corresponding high-level changes.

5.3 Definition of Differences

After having determined the document model, the notion of difference has to be defined. A diff tool has to calculate (or at least approximate) a *minimal difference* between two diagrams. In the case of directed deltas, the minimal difference may be defined as a sequence Δ of operations since that $\Delta(v_1) = v_2$ and the cost $c(\Delta)$ is minimal. In the case of a symmetric delta, $v_1 \cap v_2$ has to be maximized.

Thus, defining the difference involves the selection of an appropriate definition of a cost model for calculating the costs of executing a sequence of operations. What is considered “appropriate”, is eventually answered by the user. A formal notion of difference may serve as a *specification* against which the implementation may be verified or tested. In addition, a *validation* is required to check whether the requirements of the user are actually satisfied (i.e., whether the user considers the calculated difference as minimal).

Note that a formal definition of minimal difference introduces a *metric* for measuring the distance between documents. A metric also allows us to assess the quality of the diff algorithm quantitatively. Without a metric, there is no specification of the problem to be solved by the diff algorithm.

On the other hand, it should be noted that an evaluation of some diff algorithm with respect to a given metric alone is not sufficient: Even if the minimal difference with respect to that metric is always found, the user may still complain about missing accuracy (R1). As a simple example, assume that the set of base operations does not contain a move operation. Then, each move is simulated by deletions and insertions, and the user will not consider the calculated difference as minimal.

It is worthwhile to notice that many approaches to differencing do not rely on a formally defined metric. In some cases, the metric may be customized by the user. E.g., (Melnik et al., 2002; Kelter et al., 2005) support customizable similarity functions on which the matching decisions are based.

5.4 Reliance on Unique Identifiers

In order to calculate differences, a criterion of *sameness* is required: Which elements of different versions v_1, v_2 are considered to be the same? In the case of text-based tools, sameness is decided solely with the help of the contents and position of text lines. In this way, history independence (R5) is achieved.

On the other hand, structure-based differencing and merging turns out to be much more difficult. It is not easy to identify elements of different versions in such a way that a minimal delta is computed. Therefore, several diff and merge tools rely on *unique identifiers* (Ohst et al., 2003; Lindholm, 2004; Alanen and Porres, 2003; Rho and Wu, 1998; Mehra et al., 2005; Engel et al., 2006; Soto and Münch, 2006): When an element is created, it is assigned a new unique identifier. When the containing diagram is copied, the identifiers of its elements are retained. In this way, different copies of the “same” element may be located in the versions to be processed.

To a great extent, the calculation of differences is “for free” when unique identifiers are present. Thus unique identifiers simplify algorithms (R8) and make them more efficient. However, unique identifiers make differencing and merging dependent on the history of changes, which implies a contradiction to requirement (R5). In the extreme, it might happen that two versions v_1 and v_2 are considered to have an empty intersection even though they are isomorphic. This situation occurs when both versions have been created with the same contents independently by different users.

Thus, even when using a tool maintaining unique identifiers, differencing and merging may not perform accurately (R1) and even produce counter-intuitive results. It should be noted that the user usually is not aware of unique identifiers and thus might experience anomalies which violate the principle of least possible amazement.

In addition, unique identifiers introduce tool dependencies, contradicting (R4). In the worst case, identifier-based differencing and merging will work only if all versions have been created with the same tool. This situation is improved when multiple tool vendors agree upon the management of unique identifiers, as it is encouraged - yet not enforced - in the XMI standard (xmi, 2005). In (VVU, 2007, Hein and Ritter), an approach is presented which supports diff and merge across tool boundaries by relying on unique identifiers introduced in the MOF versioning standard (mof, 2005).

5.5 Design of Algorithms

Clearly, the previous decisions heavily influence the algorithms for differencing and merging. Unfortunately, structure-based algorithms tend to be more complex and less efficient than text-based algorithms. In particular, this holds true without unique identifiers: Optimal matches may be expensive to compute. In the case of trees, computing a minimal delta is known to be an NP-hard problem.

With respect to a formally defined notion of difference, we may distinguish among *exact algorithms* which are guaranteed to produce a minimal difference, *approximation algorithms* which may miss the minimum only up to a defined maximal distance, and *heuristic algorithms* with no guarantees at all. Accuracy (R1) has to be balanced against efficiency (R6).

So far, we are aware only of heuristic algorithms for differencing and merging. All algorithms assuming unique identifiers fall into this class, since they take the identification for granted and do not search for a better match. But those algorithms which do not build upon unique identifiers are also heuristic algorithms (Kelter et al., 2005; Melnik et al., 2002; Xing and Stroulia, 2005; Chawathe and Garcia-Molina, 1997; Cobena et al., 2002). Accuracy of these algorithms is typically evaluated by human judgment; a metric is not used for this purpose. This makes it difficult to compare these algorithms with respect to their accuracy.

Computational complexity may exclude the application of exact algorithms. For example, let us assume that documents are modeled as graphs. Computation of minimal graph differences includes the search of a graph isomorphism as a special case (if this search is successful, the graphs would be considered to be identical). Graph isomorphism is not known to be a tractable problem. Known algorithms for testing graph isomorphism have a super-exponential worst case behavior. However, this need not be a "killing argument"; consider e.g. the speed up of graph pattern matching achieved in the PROGRES system (Schürr et al., 1999) by exploiting additional information, e.g., from the graph schema.

5.6 Designing the User Interface

Differencing and merging of software diagrams requires a well-designed user interface which in particular relates differences and conflicts to diagram representations the user is familiar with (R7). In the case of differencing, diagrams may be displayed side by side with differences being marked graphically (e.g. by using colors). If not enough space is available, instead a *unified diagram* may be constructed which shows the common and all specific elements contained in only one version (Kelter et al., 2005). However, this representation may easily be overloaded (as an analogy, consider reading a C file with extensive conditional compilation).

Unfortunately, the requirement for a user-friendly representation is neglected in several tools. In (Schneider et al., 2004),(VVU, 2007, Schneider and Zündorf), a merge tool for Fujaba models reports changes in a cryptic textual format. In (Engel et al., 2006) differences between MOF instances are represented as trees rather than graphically. In (Xing and Stroulia, 2005) structural changes are visualized in a more sophisticated way as containment and inheritance change trees.

5.7 Conflict Detection and Resolution

The problem of conceptual mismatch mentioned earlier has to be considered particularly for three-way merging: The user expects that operations are combined and conflicts are detected and resolved at a conceptual level conforming to the document model which (s)he has in mind. When the merge tool operates at a different (physical) level, conflicts reported at that level cannot be understood by the user. Likewise, conflict resolution has to be performed on the conceptual rather than on the physical level.

No merge tool can be blamed for a failing merge if the changes that have been performed concurrently by different users cannot be combined in a meaningful way. While the merge tool has to strive for producing a consistent result (R13), uncoordinated changes may cause inconsistencies. For example, two users may have inserted a class with the same name, resulting in a name clash. Or one user may have made c_1 a subclass of c_2, while another user has defined the inheritance relationship in the opposite direction. In general, we cannot expect that a merge tool detects all kinds of context-sensitive conflicts, reports them to the user, and ensures consistency by rejecting changes causing inconsistencies. Please recall that we do not expect such a behavior when applying text-based program merging; rather, errors introduced by the merge can (partially) be detected by running the compiler. Likewise, merging of diagrams may result in inconsistencies which are fixed in a post-processing step.

However, there is one crucial difference compared to text-based merging: As a result of merging text files, we will get a text file which may be checked by the compiler and which may be viewed and edited with the help of some text editor. In contrast, merging of diagrams may result in *fundamental inconsistencies*: The output produced by the merge may not be processable any more because fundamental constraints are violated. In fact, in many approaches presented in the literature merging may produce an inconsistent result (Lindholm, 2004; Alanen and Porres, 2003; Ohst et al., 2004; Rho and Wu, 1998; Mehra et al., 2005; Chen et al., 2003), (VVU, 2007, Schneider and Zündorf). The merge tool may be blamed for this problem if the merge is performed at the wrong level of abstraction. On the other hand, the consistency constraints enforced by some CASE tool may be too tight. If inconsistencies were tolerated in a diagram editor, the merge tool could create a potentially inconsistent output, which the user can improve subsequently in the editor. To make this work, the underlying document model has to be generalized such that inconsistencies are tolerated (Schneider et al., 2004), (VVU, 2007, Schneider and Zündorf).

6 DESCRIPTION AND CLASSIFICATION OF KNOWN TOOLS

In the following paragraphs, six tools are compared in more detail: one algorithm for calculating a matching, two algorithms for differencing without unique identifiers (see 6.1) and two differencing methods that rely on unique identifiers (see 6.2). The latter are merging tools too, as described in 6.3, where another merging tool that uses logged operations as delta is presented.

6.1 Differencing without Unique Identifiers

Differencing tools that do not rely on unique identifiers need other criteria to identify corresponding diagram elements that are sufficiently similar. In all known concepts the matching and the computation of differences are considered to be separate problems. Since the computation of differences with respect to a given matching is discussed in the next paragraph, we regard the following three algorithms only with respect to the heuristics they use to find a matching.

Similarity Flooding (Melnik et al., 2002) is a general graph matching algorithm operating on directed labeled graphs whose nodes represent diagram elements. For the computation of the similarity of two nodes their neighbourhood is relevant: The similarity of two nodes increases if their adjacent nodes are similar. The computation itself works as follows: Initial similarity values are computed by a pairwise comparison of the node names. These similarity values are then propagated in a fixpoint computation along the edges of the similarity propagation graph. A matching can be selected from these globally computed similarity values by choosing thresholds, constraints and selection metrics.

The generic algorithm *SiDiff* (Kelter et al., 2005) uses an internal data model comparable with a simplified UML meta-model and is configurable for various types of UML diagrams. A diagram is extracted from an XMI file and is represented as a tree consisting of the composition structure augmented with cross-references. Assuming that model elements are characterized by the elements they consist of, the difference algorithm starts with a bottom-up traversal at the leaves of the composition tree. All elements of the same type are compared pairwise using a type-specific similarity function that evaluates the weighted aggregation of type-specific criteria. For instance, in the case of class diagrams the similarity of name, attributes, methods and inheritance relations are considered. If a significant correspondence of two nodes has been identified, these nodes are matched and if they possess child nodes that have not been matched yet, the similarity is propagated top-down into the subtree, i.e. the similarity values of the child nodes are computed anew with respect to the correspondence of their parent nodes. The algorithm ends if all nodes have been processed in the bottom-up phase and all similarities have been propagated downwards (see figure 1). It is important to note

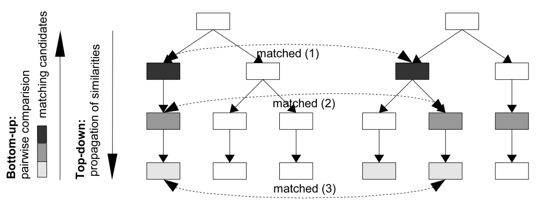

Figure 1: *SiDiff*: Search for Correspondences.

that the matching in the bottom-up phase is not very successful at the lowest levels of the tree, since many leave nodes are nearly identical (see the frequent occurrences of data type integer in class diagrams for instance). To handle this problem the original algorithm has been extended by a pre-phase which tries to match nodes by comparing hash values computed on the paths of the nodes in the composition tree. After the matching has been found, differences are computed and represented in a unified document.

The algorithm *UMLDiff* in (Xing and Stroulia, 2005) operates on class diagrams which have been reverse engineered from Java source code and thus works on a much more fine-grained level than the two algorithms presented above. The data model used consists of a directed graph including a spanning tree of containment relations. In contrast to *SiDiff*, *UMLDiff* starts at the root nodes and compares the nodes of the same logical level pairwise. On the way down only those entities in subtrees are compared whose root nodes have been matched on a higher level (see figure 2). If two objects have the same name, they are identified as equal. If not, their structural similarity is considered, computed from the similarity of names and other criteria specific of the considered entity type. In the case of methods, these are parameter types, fields they read or write and other methods they call or are called by. If a computed structure similarity of two entities exceeds a user-defined threshold and if it is the maximum value of all possible matching candidates, the entities are matched as equal. After all the leaves of the composition tree have been processed, the remaining objects are compared in order to find moved objects. If there are two entities with the same name in different subtrees, they are considered as moved. Then the algorithm tries to identify moved and renamed elements by structural similarity. The top-down traversal with the early restriction of the search space to the subtrees of matched entities and the method of matching the entities by name first make this algorithm efficient. It presumes however, that the two versions that are compared are not too different, i.e. in particular that not many movements and renamings have occurred.

The methods described above seem to work satisfyingly in terms of performance and error rate judging

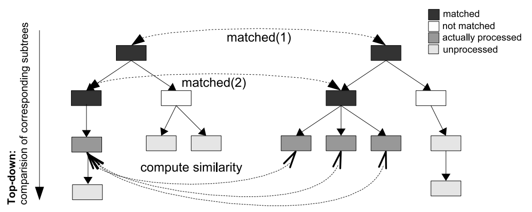

Figure 2: *UMLDiff*: Search for Correspondences.

from the published results. In all three cases, however, the differences and matchings have only been verified by hand. Due to the lack of metrics, it is not possible to compare the different algorithms. It is worth mentioning that the computation in *Similarity Flooding* uses a global concept even if the underlying model is quite simple. On the other hand *SiDiff* and *UMLDiff* make a sequence of local decisions. Further *SiDiff* uses configuration files to adapt the algorithm to the specific diagram types and thus offers a compromise between (R2) and (R3) whereas *Similarity Flooding*, working on a high conceptual level, cannot use diagram type-specific information. *UMLDiff* gives up generality in favour of domain specific optimization. None of the three tools described above supports merging.

6.2 Differencing with Unique Identifiers

The tools presented in this paragraph identify the corresponding diagram elements by comparing their unique identifiers.

In (Ohst et al., 2003) the two diagrams that should be compared are presented as graphs consisting of a spanning tree of composition relations as in the more recent concept presented in (Kelter et al., 2005). In a top-down traversal of each level in the spanning tree the corresponding subtrees which are rooted at nodes with identical identifiers are found. The corresponding nodes are compared with respect to their attributes and relationships and the difference information is recorded in an object created for the unified document representing a symmetric difference. Then the nodes in the matched subtrees are examined further. A move operation is realized as composite operation that deletes one object and inserts an object with the same unique identifier. To identify moved subtrees, all the subtrees that could not be matched are stored in sets and compared once again. If nodes with the same identifier exist in both sets, they represent a moved node. All other nodes have either been deleted or created.

In (Mehra et al., 2005) component-based plug-ins to the meta-CASE tool *Pounamu* for diagram versioning, differencing and merging are presented. Any diagram type, which has been defined in the meta-CASE tool, can be compared to find a directed delta. Instead of traversing the graph-based structure, consisting of shapes and connectors and their properties, the diagrams are compared in two steps: First all shapes are matched by their identifiers. If the properties of corresponding shapes differ, an appropriate change operation is added to the directed delta. If a shape exists in one diagram only, an insert or delete operation is added. Since a connector is defined by its source and target shapes, if a shape is deleted all connected connectors are also deleted and if a shape is inserted, the connectors must be inserted too. In a second step all connectors that have not been processed yet are compared. Moves are not detected.

The set of feasible edit operations used in these two approaches are not equal. In *Pounamu* only insert and delete operations are considered, whereas in (Ohst et al., 2003) a combined delete and insert situation is interpreted as move operation. Further in (Ohst et al., 2003) a symmetric difference is computed whereas in *Pounamu* a directed delta is determined. The two described tools also support merging as shown in the following paragraph.

6.3 Merging

In this paragraph, three merging tools that offer different levels of user interaction are described.

In the *CoObRA* versioning framework (Schneider et al., 2004) all edit operations that are executed on the diagrams are logged by the tool. For this reason no differences must be computed. *CoObRA* uses three-way merging but gives priority to the version that was committed first. The workflow is illustrated in figure 3. A developer has checked version v_1 out of the repository into the local workspace to modify it by applying the operation sequence Δ_2. But if meanwhile the operation sequence Δ_1 has been applied to the version in the repository, the developer fails to commit his changes. He has to update his local version first. This means applying the changes Δ_1 on the origin version v_1 to reach the actual version v_2 stored in the repository, then trying to apply the change operations in Δ_2 again. At this point, con-

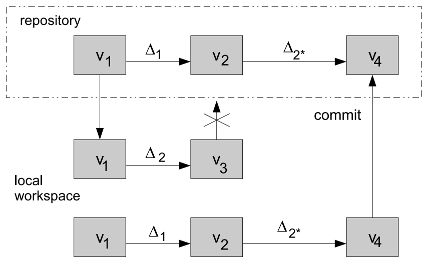

Figure 3: Commit and Update in CoObRA.

flicts may occur if one or more operations in Δ_2 can no longer be applied after the execution of Δ_1. The operation sequence Δ_{2^*} in the figure is a subset of Δ_2, expressing that some operations might not have been applied. Conflicts are reported to the application in a cryptic way, conflict solving is not supported. Further the semantic correctness may be violated: if classes with the same name are created in Δ_1 and in Δ_2, the merged version may contain both classes if it is not filtered by constraints in the application.

The *Pounamu* meta-CASE tool described in (Mehra et al., 2005) offers a plug-in for merging, where the merging is realized interactively. The set of edit operations in the computed directed delta are offered to the user who decides which changes to apply. A difference highlighting plug-in shows the differences graphically based on the local version of the diagram. Additionally the edit operations are presented in a list, where edit operations which are currently not applicable are marked.

(Ohst et al., 2004; Ohst et al., 2003) use three-way merging and split the merging process into three steps. First a pre-merged document is created. In the second step, the conflicts must be resolved manually before the merged document is created in the final step. Conflicts occur if the same attribute has been changed in both versions, if an object has been modified in one version and deleted in the other version and in all derived situations. In case of deletion-modification conflicts the user has to decide whether the object should be deleted or modified. In the case of change conflicts the user is asked which modification should be applied. The pre-merged document is an extended unified document consisting of common parts, automatically merged parts and conflicts. This pre-merged document can be modified in a tool that supports conflict solving and undoing decisions, even decisions that have been made automatically. The merged class diagram may be inconsistent, constraints like uniqueness of names of classes, methods or attributes must be verified after merging (Ohst et al., 2004).

Only in (Ohst et al., 2004) the two versions that have to be merged with respect to a base document have equal relevance. *CoObRA* gives priority to the version stored in the repository, *Pounamu* to the local changes. The differences used for merging are obtained in different ways: in *CoObRA* there is a protocol of the operations, in *Pounamu* the differences are computed as directed delta and in (Ohst et al., 2004) a symmetric difference is calculated. In *CoObRA*, user interaction in the merging process is not possible; modifications must be made manually on the merging result. In the meta-CASE tool *Pounamu* the developer can and must decide which change operations have to be applied. In (Ohst et al., 2004) the non-conflicting transformations are applied automatically leaving only the problematic decisions to the user, including the possibility to interfere in the taken decisions.

7 CONCLUSION

We have defined requirements for algorithms and tools for differencing and merging of software diagrams. Furthermore, we have explored several crucial design decisions which tool developers have to perform. We have also shown how these design decisions have been resolved in a number of approaches published in the literature.

The current state of the art may be characterized as follows:

- There is a common agreement that text-based diff and merge tools are not adequate for software diagrams.
- A number of commercial tools and research prototypes provide support for differencing and merging. However, these approaches suffer from various shortcomings such as non-graphical user interfaces, reliance on unique identifiers, or inconsistent merge results.
- There is no common agreement with respect to the document model as the foundation for differencing and merging, metrics to be used for measuring differences between versions, rules used for merging, etc.
- Published algorithms either assume unique identifiers or are based on heuristics. Evaluations of these algorithms are based on human judgment, and it is hard to compare these algorithms against each other.

Thus, further research is needed to improve the state of the art. However, it is difficult - or even impossible - to meet all of the requirements defined in this paper. From the perspective of software configuration management, it is important to go beyond text-based version control. On the other hand, software configuration management systems need to support version control for a wide variety of software documents. Moreover, they need to handle large volumes of data. From this perspective, general approaches based e.g. on MOF or XML are required. The experiences gained with differencing and merging of program versions indicate that accuracy and sophistication may have to be traded for generality and efficiency.

REFERENCES

(2005). *MOF 2.0/XMI Mapping Specification, v2.1*. Object Management Group, final/05-09-01 edition.

(2005). *MOF2 Versioning Final Adopted Specification*. Object Management Group, ptc/05-08-01 edition.

(2007). Contributions to the workshop "Versionierung und Vergleich von UML-Modellen" on the conference of Software Engineering 2007 in Hamburg. *Softwaretechnik-Trends*, 27(2). (to appear). http://pi.informatik.uni−siegen.de/gi/fg211/VVUM07/.

Alanen, M. and Porres, I. (2003). Difference and union of models. In Stevens, P., Whittle, J., and Booch, G., editors, *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference*, LNCS 2863, pages 2–17. Springer.

Buffenbarger, J. (1995). Syntactic software merging. In Estublier, J., editor, *Software Configuration Management: Selected Papers SCM-4 and SCM-5*, LNCS 1005, pages 153–172.

Chawathe, S. S. and Garcia-Molina, H. (1997). Meaningful change detection in structured data. In Peckman, J. M., editor, *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 26–37. ACM Press.

Chen, P., Critchlow, M., Garg, A., der Westhuizen, C. V., and van der Hoek, A. (2003). Differencing and merging within an evolving product line architecture. In van der Linden, F., editor, *Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5)*, LNCS 3014, Siena, Italy. Springer Verlag.

Cobena, G., Abiteboul, S., and Marian, A. (2002). Detecting changes in XML documents. In *International Conference on Data Engineering*, pages 41–52. IEEE Computer Society.

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

Engel, K.-D., Paige, R. F., and Kolovos, D. S. (2006). Using a model merging language for reconciling model versions. In Rensink, A. and Warmer, J., editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 143–157. Springer.

Horwitz, S., Prins, J., and Reps, T. (1989). Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387.

Hunt, J. and Szymanski, T. (1977). A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353.

Kelter, U., Wehren, J., and Niere, J. (2005). A generic difference algorithm for UML models. In Liggesmeyer, P., Pohl, K., and Goedicke, M., editors, *Software Engineering 2005*, LNI 64, pages 105–116. GI.

Lindholm, T. (2004). A three-way merge for XML documents. In Munson, E. V. and Vion-Dury, J.-Y., editors, *Proceedings of the 2004 ACM Symposium on Document Engineering*, pages 1–10. ACM.

Mehra, A., Grundy, J. C., and Hosking, J. G. (2005). A generic approach to supporting diagram differencing and merging for collaborative design. In Redmiles, D. F., Ellman, T., and Zisman, A., editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 204–213. ACM.

Melnik, S., Garcia-Molina, H., and Rahm, E. (2002). Similarity flooding: A versatile graph matching algorithm and ist application to schema matching. In *Proceedings 18th International Conference on Data Engineering*, pages 117–128, San Jose, CA.

Mens, T. (2002). A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462.

Ohst, D. and Kelter, U. (2002). A fine-grained version and configuration model in analysis and design. In *ICSM*, pages 521–527. IEEE Computer Society.

Ohst, D., Welle, M., and Kelter, U. (2003). Differences between versions of UML diagrams. In *Proceedings ESEC/FSE-11*, pages 227–236, New York, NY, USA. ACM Press.

Ohst, D., Welle, M., and Kelter, U. (2004). Merging UML documents. Internal Report, University of Siegen.

Rho, J. and Wu, C. (1998). An efficient version model of software diagrams. In *Asia Pacific Software Engineering Conference*, pages 236–243. IEEE Computer Society Press.

Schneider, C., Zündorf, A., and Niere, J. (2004). CoObRA - a small step for development tools to collaborative environments. In *Workshop on Directions in Software Engineering Environments; 26th international conference on software engineering*. ICSE 2004, Scotland.

Schürr, A., Winter, A., and Zündorf, A. (1999). The PROGRES approach: Language and environment. In Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G., editors, *Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages, and Tools*, volume 2, pages 487–550. World Scientific.

Soto, M. and Münch, J. (2006). Process model difference analysis for supporting process evolution. In Richardson, I., Runeson, P., and Messnarz, R., editors, *Software Process Improvement, 13th European Conference, EuroSPI 2006*, LNCS 4257, pages 123–134. Springer.

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

Wiborg-Weber, D. (1999). CM strategies for RAD. In Estublier, J., editor, *System Configuration Management: 9th International Symposium (SCM-9)*, LNCS 1675, pages 204–216.

Xing, Z. and Stroulia, E. (2005). UMLDiff: an algorithm for object-oriented design differencing. In Redmiles, D. F., Ellman, T., and Zisman, A., editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 54–65. ACM.