Differencing and Merging of Software Diagrams

Bernhard Westfechtel
Angewandte Informatik 1, Universität Bayreuth, D-95440 Bayreuth
bernhard.westfechtel@uni-bayreuth.de

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. [7] 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 concludes the paper.

2 Basic Notions

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 $\Delta(v_1, v_2) = (v_1 \setminus v_2) \cup (v_2 \setminus v_1)$. 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 sequel. Two-way merging compares two versions $a_1$ and $a_2$ with the help of some diff algorithm which calculates a symmetric delta. When a differing element is 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 mutual 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 $a_1$ and two directed deltas $\Delta_i = \Delta(b, a_i)(i = 1, 2)$, construct a merge version $m$ and a delta $\Delta_m$ such that $\Delta_m(b) = m$ and $\Delta_m$ constitutes an
Table 1: Requirements to 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.

order-preserving, complete merge of the operation sequences \( \Delta_1 \) and \( \Delta_2 \). 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 \( \Delta_1 \) may be overridden by an operation \( op' \) from \( \Delta_2 \), 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 to diff and merge tools. The attribute “potential” indicates that these requirements may be posed in some application context, but they may also be considered irrelevant in that context. Furthermore, since requirements may contradict each other, one requirement may have to be traded against another requirement.

Table 1 lists requirements to 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) requests 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 carry over to merge tools. Thus, Table 2 lists only those requirements which are specific to merge tools. (R9) and (R10) demand for 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\(^1\). 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 even-

\(^1\) 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.
tually each software process has to produce an executable program (other artifacts are often considered merely as “documentation”). For a survey on program merging, see [17].

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 (virtually all industrially used programming languages are text-based).

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).
- 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 metrics. For example, [12] calculates the longest common subsequence (lcs), while [28] additionally covers block moves.
- Text-based merging usually relies on the lcs algorithm for comparing text files line by line (e.g. 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 syntactic or semantic level (e.g. [3] and [11], respectively).

However, so far syntactic or semantic differencing and merging of programs have been implemented only in some research prototypes. 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 [17], 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 [29].

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, virtually 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. 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 diagram [13], a specific kind of UML diagram (e.g., class diagrams [20, 30]), any diagram processable by a meta-CASE tool, 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.
ument 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, [1, 10, 2] are based on MOF and are thus applicable to MOF instances, including UML diagrams. [13, 30] rely on tool-specific document models (trees augmented with cross-references). In [26, 25], diagrams are transformed into RDF; [27] 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. Another promising document model are graphs [8].

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 [27, 25]: 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 $\Delta$ 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 [9]. 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., [16, 13] 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 same-ness 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 [20, 10, 25, 14, 2, 22, 15]: 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) [9]. 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 [18]. In [10], an approach is presented which supports diff and merge across tool boundaries by relying on unique identifiers introduced in the MOF versioning standard [19].

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 [13, 16, 30, 4, 6]. 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 [24] 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 [13]. 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 [10], differences between MOF instances are represented as trees rather than graphically. In [23], a merge tool for Fujaba models reports changes in a cryptic textual format.

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 having 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 a subclass of c2, 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 [23, 14, 2, 21, 22, 15, 5]. 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 [23].

6 Conclusion

We have defined requirements to 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 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


