Tracing Model Elements

Sven Wenzel, Hermann Hutter, Udo Kelter
Software Engineering Group, University of Siegen
Hölderlinstr. 3, D-57068 Siegen, Germany
{wenzel|hutter|kelter}@informatik.uni-siegen.de

Abstract

In model-driven engineering developers work mainly or only with models, which exist in many versions. This paper presents an approach to trace single model elements or groups of elements within a version history of a model. It also offers analysis capabilities such as detection of logical coupling between model elements. The approach uses a differencing algorithm known as SiDiff to identify similar elements in different versions of a model. SiDiff is highly configurable and thus our tracing approach can be adapted to all diagram types of the UML and to a large set of domain specific languages. The approach has been implemented as an Eclipse plug-in that visualizes all relevant information about the traces and it allows developers to interactively explore details. It has been evaluated by several groups of test persons; they considered most of the functions of the tool to be very useful.

1 Introduction

Model-driven engineering (MDE) is proclaimed as new philosophy of software engineering in the 21^{st} century. Sooner or later software developers will not touch any line of code any more, but they will rather work mainly or only with models. The term *model* is not restricted to the diagram types of the widely accepted Unified Modeling Language (UML), but it also includes domain specific languages such as Matlab/Simulink diagrams or web service description languages. Assisted by powerful transformation tools, 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.

Regarding the development process models exist in many versions or variants. Thus, models should be stored in version management systems in the same way as source code. Version management systems offer services such as archiving older revisions of software, and they support concurrent work and collaborative engineering. In general we can distinguish (a) file-based version management systems, such as CVS or Subversion, and (b) dedicated repository systems for specific model types, such as IBM's ClearCase in conjunction with Rational Software Architect. As the example indicates, the latter kind of versioning systems is closely connected with certain modeling tools. They are mostly based on object-oriented databases directly accessed by the tools. Concurrent work is supported in a way that all developers work on a single model that is stored in a fine-grained way and allows locking of individual fragments. Changes in the models can be traced easily.

However, it is common practice that models are managed with files-based versioning systems, e.g. CVS. Usually the models are stored in textual files, e.g. in the XMI format for UML, and checked-in into the versioning system. The systems, which are designed to keep track of versions of files, cannot provide fine-grained information about elements within the files. The integrated tools for computing differences between textual documents work fine with sources files, but fail with textual representations of models, in particular because they do not duly consider the semantics of models. Hence the elements inside the models cannot be traced between different revisions.

Regarding these circumstances and the prevalence of file-based version management systems, daily tasks become a challenge for developers, maintainers, and project managers. Tracing an element from one model revision to another is crucial to answer frequent questions, such as:

- Since when does a certain element exist in the model?
- Where did the element X of the initial model revision disappear?
- Does the given element also exist in other variants of this model?
- What elements bear a relation to the given element?
- Given a constellation of elements, e.g. containing a bug, does this constellation appear in other branches, too?

Furthermore there are questions from the analytical point of view that require trace information, e.g. a differentiation between stable model elements and model elements which are changed quite often.

In this paper we present an approach to provide answers to the above mentioned questions. The approach has been implemented as an Eclipse plug-in that assists developers and maintainers with tracing information. Therefore, we integrated a generic, highly configurable differencing algorithm, called SiDiff, with the analysis of version histories. The difference computation allows building correspondences between subsequent model revisions as described in the subsequent section. Based thereon we can build tracks of model elements and gain tracing information as we will show in Section 3. The implementation of the approach as an Eclipse plug-in and its applicability to the above mentioned questions is described in Section 4. The results of a case-study examining the benefit of our approach against manual tracing are summarized in Section 5. Section 6 summarizes work related to our approach. Finally, in Section 7 we conclude our work and discuss current and future research topics.

2 Computation of Correspondences

The main task of tracing a model element is to locate that element in another model, i.e. the corresponding element. Given a history of successive model revisions, the tracing takes place between a model and its predecessor or successor revision.

The tracing problem becomes trivial if we assume elements to have persistent identifiers [1, 21]. However, most modeling tools do not use persistent identifiers when writing files to disk or they create them in an unfortunate way from element names. Matlab/Simulink, for example, creates identifiers by hierarchical concatenation of element names; thus, simple renaming of some blocks dooms identifier-based tracing to useless results. Even if tools would support persistent identifiers, concurrent work of different developers can lead to identical document elements with different identifiers. Especially if variants of a model were created, i.e. parallel branches in the versioning system, the identifiers would differ and common elements of parallel versions could not be traced.

The comparison of two succeeding documents and the location of corresponding elements respectively are a daily task in software configuration management and known as difference computation. The GNU diff tool is probably the most-known representative in that field; it compares text documents and reveals their changes line-by-line.

However, the textual analysis of models to detect correspondences between them is not sufficient. Working with textual representations of models may lead to too many false, conceptually irrelevant differences. As shown in [16], correspondences must be computed on an appropriate level of abstraction, i.e. the model level taking syntax and semantics into account.

2.1 Difference Tools for Models

Difference tools for models are available, too. They can roughly be divided into (a) algorithms which can handle only one specific model type, to which they are fully adapted, (e.g. as proposed in [13, 31]) and (b) generic algorithms which require only some configuration data, if any at all. Due to the fact that we do not want to restrict our tracing approach to one specific model type, we address only the second class of algorithms.

Therefore, one can assume models to be graphs with a tree-like structure. They are composed of elements which in turn have sub elements. They are not exactly trees due to cross references, e.g. return types of operations referring to other classes in UML class diagrams. This assumption fits to all diagram types of the widely accepted UML [23], as well as to most models defined by domain specific languages[1].

However, the usage of general graph comparison algorithms is inhibited by their NP completeness. Algorithms for unordered trees are also not sufficiently efficient. The only viable approach is to consider model documents as ordered trees, since in all relevant cases, model elements are either ordered or they have attributes, from which an order can be derived.

LaDiff [5] is applicable to ordered typed trees, notably LaTeX documents. It compares the leaves of the trees pairwise and forms corresponding pairs as soon as their similarity exceeds a given threshold. The complexity of this algorithm is $O(n^2)$, with n being the number of nodes of the trees.

XDiff [29] is applicable to typed trees in which elements have names, which are unique in the context of the parent element. Based thereon unique path names of subtrees can be formed. XDiff computes hash keys that depend on subtrees and their path names. Using these hash keys, which are collected in a directory (e.g. a hash table), XDiff is very efficient in identifying identical subtrees in both documents in $O(n \log n)$. However, corresponding subtrees that contain renamed elements are not identified and there is no notion of similarity at all.

Our approach uses the tool suite SiDiff [16, 28] which has proven to be powerful enough to adapt to virtually all kinds of models with an underlying graph structure. It has runtime complexity of $O(n \log n)$ for models with normal properties. It is presented in more detail in the following section.

[1] Although MDE is proclaimed by the OMG, it is not limited to UML.

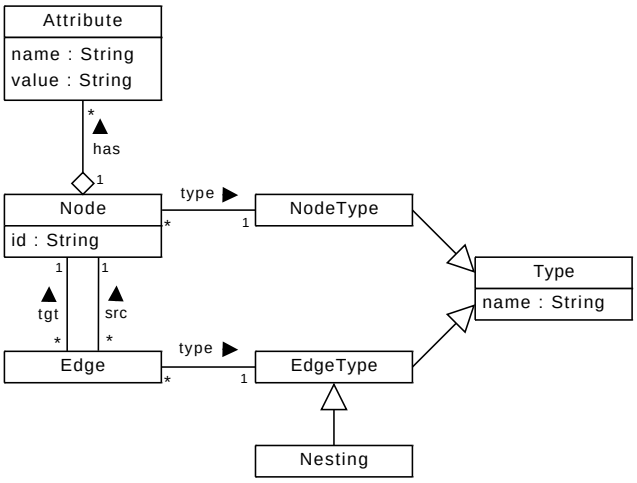

Figure 1. The SiDiff difference computation works on an attributed typed graph

2.2 Excursus: The SiDiff Approach

Similar to the difference algorithms presented above, SiDiff detects corresponding elements. It does not rely on persistent identifiers or unique element names, correspondences are rather formed on the basis of similarities between model elements. It uses an internal graph representation of model documents. Hence, it is not only suitable for UML documents, but also for domain specific languages such as Matlab/Simulink diagrams or web service descriptions.

2.2.1 Internal Model Representation

When initially loading a model of a specific document type (e.g. from an XML file), an internal representation of the model as attributed typed graph is constructed. Figure 1 depicts its structure. A graph consists of nodes and edges with a type assigned to each edge and each node. The specific type *Nesting* can be assigned to edges in order to indicate that these edges are part of a tree representation of the document. A nesting edge points from a parent node to its child. In addition, nodes can have attributes. Each attribute consists of a name-value pair.

Each model to be compared is mapped onto one internal graph. A node type is created for each element type, e.g. class, operation, generalization in UML class diagrams. The actual model elements become nodes that are assigned with the corresponding node type. Attributes of the elements are attached to the nodes. The edges of the internal graph represent relations between the elements; they are assigned with an edge type that relates to connections in the metamodel. The nesting edges are used for part-of relationships, e.g. between classes and operations.

Table 1. Criteria for comparing Classes

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

2.2.2 Similarity Heuristics

SiDiff uses a similarity heuristic to determine whether two model elements correspond or not. The element properties which are relevant for the similarity of two elements of the same type are either local attributes (e.g. names) or other elements in the neighborhood (e.g. elements that are referenced). SiDiff uses a set of compare functions to analyze two properties of the same type which belong to different elements. They return a value between 0 and 1; a value of 0 stands for no similarity between the properties, a value of 1 expresses equality.

In addition each property is assigned with a weight indicating the relevance of the property for the similarity of two elements. 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 threshold, i.e. 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.

2.2.3 Comparison Procedure

Similarities are computed in a bottom-up/top-down order, according to the tree-like structure of most models given by the nesting edges in the internal graph structure. 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 immedi-

ately because the similarities might change when further elements are compared. In a following iteration 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 in the meanwhile. Consequently, other matchings can be found, which are propagated top-down further on.

Most models do not have a real tree structure; they contain cross references between elements, which lead to cycles. The comparison handles such cycles 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 [19]. It enables comparison of 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.

A hashing pre-phase calculates hash values for each element and elements with identical hash values are matched. It gives significant impact on the efficiency of the algorithm and provides trustworthy fix points that speed up the comparison of their neighbor elements.

SiDiff features a very efficient runtime and has a negligible error rate regarding the quality of the differences it computes. For more technical details and the manually evaluated quality we refer to [16, 28].

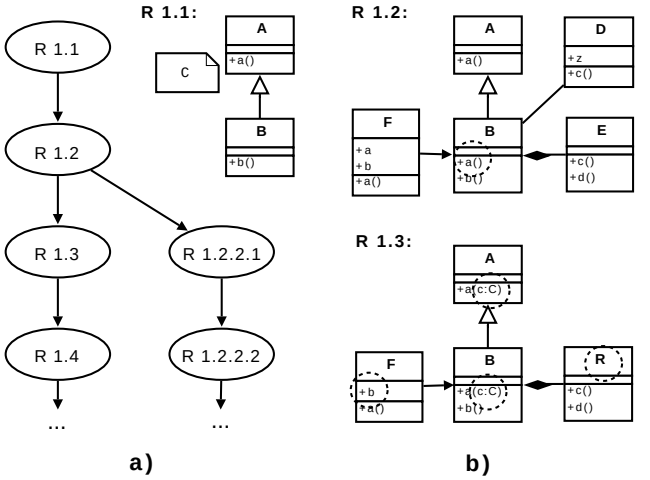

Figure 2. Example of model revisions

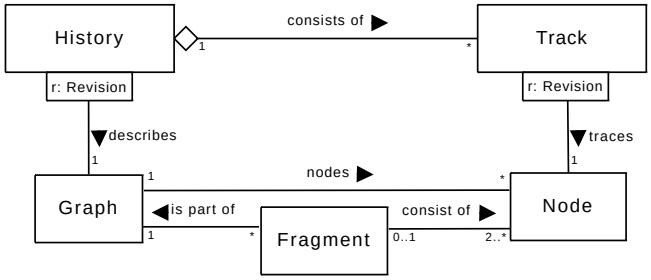

Figure 3. Data model of tracing information

3 Computing Trace Information

The comparison of two models provides information about correspondences, i.e. the same element occurring in both models. Figure 2 shows (a) an example version history of a model and (b) depicts three revisions of that model; differences between model elements in the revisions are marked. In the example, class *E* has been renamed to *R*; the remaining elements correspond by their positions.

In order to compute trace information about an element in a document version, this document version is compared with each direct successor, either in the same branch or in parallel branches. The successors are compared with all their successors, and so on. In the example of Figure 2 we compare revision 1.1 with revision 1.2 which in turn is compared to revision 1.3 and so on; version 1.2 is also compared with version 1.2.2.1, and so on.

3.1 Data Model of Tracks

Following the graph representation of a model document (cf. the internal graph structure of SiDiff in Figure 1), we reveal correspondences between nodes of graphs. These correspondences are stored in a data structure called *History*. As shown in Figure 3, a history describes the evolution of a graph, i.e. a model document, and allows addressing particular revisions. While a graph consists of nodes, the history itself consists of several *tracks*, which in turn allow addressing of individual nodes in a certain revision.

Hence, a track is a chain of model elements. Starting from a basic document, i.e. the model where the observed element occurs the first time, it contains all successor revisions of that element. Due to variants inside the version history the track may split into branches forming the shape of a tree.

Besides tracing single model elements we want to provide support for tracing sets of elements. Such a set is called a *fragment* consisting of several nodes. In contrast to the term model fragment used in MDE-related literature, a fragment does not necessarily cover elements belonging structurally together here. A fragment can rather be seen as a slice of elements inside one model, e.g. several methods fulfilling one requirement of the software.

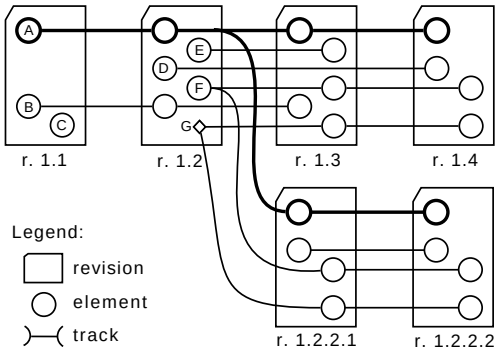

Figure 4. Abstract example of tracks

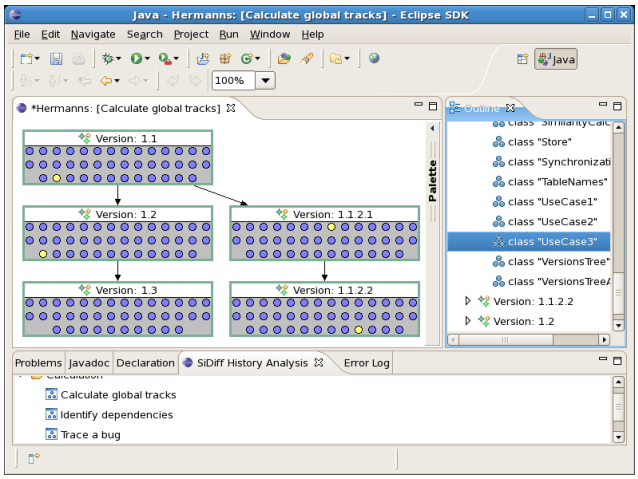

Figure 5. Screenshot of the analysis tool

3.2 Construction of Tracks

A track is computed by pairwise comparison of model documents. Starting from the basic document, we compare a document with its successors. For each element of the first model that occurs also in the succeeding models a track is created, e.g. track *A* in Figure 4. The track represents the same element occurring in different model revisions including branches (e.g. for element *F*). The elements can either occur in the succeeding models without any changes or they have been changed. Unchanged elements are located immediately regarding the hashing functionality of the difference algorithm. Elements that have been changed from one revision to another (e.g. renaming of *E* to *R* in Figure 2) are located by the similarity heuristics. Thereby the threshold value defined for each element type prevents from correspondences between elements with significant changes.

If an element cannot be located in any subsequent revision of a model the track of that element ends; in Figure 4 track *B* ends in revision 1.3. Elements that do not have correspondences in an adjacent revision are compared to the elements of the next following revision. Thereby the traceability of elements is enhanced without including elements into a track that do not reliably correspond. In this case tracks may contain gaps, e.g. track *D* in Figure 4. The maximal size of gaps is configurable.

The case that a gap covers branching points of a model is not supported by default; however, it can be processed. An example is given by the elements on the lower part in Figure 4 which are part of a track in each branch. Although both tracks do not have one predecessor, which would be positioned in *G*, the tracks could be concatenated with *G* being the gap.

Elements without any corresponding partner, e.g. element *C*, are called *day flies* and do not become part of any track.

4 Tool Integration for Tracing Elements

Tracks must be visualized in a meaningful way for developers. We have implemented our approach as a plug-in for the Eclipse platform [8]. Consequently, the visualization has been realized using the GEF framework [9]. A screenshot is shown in Figure 5. It shows an excerpt of the version history of a UML class diagram describing the implementation of our tracing approach.

Tracks are visualized in an abstract representation that is independent of any model type. Rectangles represent different revisions of the given model, inside a rectangle each model element is represented by a small colored circle. The color provides additional information depending on the current analysis task. On the right hand side an outline view shows a list of all revisions and their elements inside. Both the graphical representation and the outline view allow developers to select model elements. Tool tips show further information about the elements. Filters can reduce the set of displayed elements. The panel on the lower part provides different analysis tasks to choose from.

All analysis tasks consist of two steps. First, the requested information is computed and visualized. Afterwards, the user is able to navigate within the visualization to retrieve information one is interested in. The current implementation supports three different tasks according to the questions given in Section 1 and one additional task of finding day flies, which is a by-product of the other analyses.

The tool is independent from any version management system, since the model documents are currently imported from plain XMI files similar to those stored in a file-based versioning system. Direct access to CVS or Subversion is aspired to enhance usability of the prototype.

4.1 Global Tracks Analysis

The analysis of global tracks is the simplest task. It only uses the track information that is computed as described in Section 3. If a user selects a single model element in any revision of the model history, the occurrences in all other revisions are highlighted despite the fact that the elements might have been slightly modified. One can immediately see (a) since when a given element exists in the history, (b) where an element of a revision disappeared or (c) where it occurs in other revisions or variants. Given the exemplary screenshot above, a class named *UseCase3* is selected; it occurs and is marked yellow respectively in each revision except revision 1.3.

4.2 Tracing a Bug

A bug usually affects more than just one model element. Initially, the elements involved in the bug situation must be selected. This set of elements is handled as a fragment (cf. Figure 3). Basically, these elements are traced individually.

We assume that a bug is only present in another version (and should be fixed there) if the set of elements involved in the bug occurs as a whole with only very small changes in the other version. Therefore, similarities of the fragments in different versions as a whole are also computed and must exceed an additional threshold for the other fragment to be considered as a repetition of the bug. Although small changes can already fix the bug, it is better to include false candidates than inspecting only unchanged elements. The exclusion of slightly changed elements (e.g. renamings) would shield a proper bug tracing. However, in case of extensive changes to the probably affected elements no bug occurrence is reported; otherwise the result consists of false candidates only. Obviously, tracing a set of elements without this constraint can be processed by global tracks analysis and is trivial.

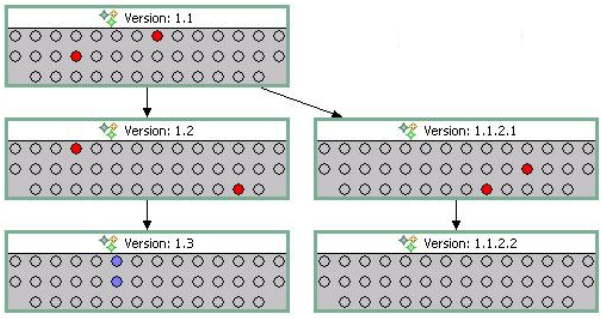

Figure 6. A bug occuring in other revisions

Figure 6 shows a bug tracing analysis. The elements declared to be affected by a bug in revision 1.3 are colored blue, while the bug candidates in the other revisions are colored regarding their probability to be a bug. Unconcerned elements are grayed out. In this example the two elements have been changed in revision 1.1.2.2 so that the occurrence of the bug in that revision cannot be presumed.

4.3 Finding Dependencies

Dependency analysis, formerly known as logical coupling, has been subject of a lot of research in the past (cf. Section 6). It provides information about software elements such as files or methods that bear a relation to each other although that relation is not obvious. These relations are based on the changes applied to the elements, e.g. whenever element *A* has been changed, element *B* was changed, too.

Due to our fine-grained tracing we are able to provide this dependency information for each element within the model history. Once an element has been selected we are able to follow its history track through all revisions and other variants. For each revision where the traced element has been changed, we check other elements that are also affected by modifications. Those elements are stored in a dependency map. The degree of dependency is given by support, frequency and confidence. Support represents the number of changes of one element in the whole history. Frequency is the rate of changes of an element concurrently changed together with the traced element. Confidence is the ratio of frequency and support.

Figure 7. Dependencies to other elements

Within a revision in the history visualization the dependency is displayed using different colors as depicted in Figure 7. The element whose dependencies are of interest is marked white. The dependent elements are colored from green over yellow to red according their degree of dependency. Green states a dependency of at least 50 percent, while an absolute dependency is colored red. Elements with a confidence lower than 50% are declared to be independent and stay grey. The threshold for dependencies can be changed.

4.4 Identifying Day Flies

Day flies are elements that occur only in one revision of the model history. They are usually a sign for models that are changed without taking a long view. From the technical

point of view these elements can be identified very easily; day flies are elements that are not part of any track.

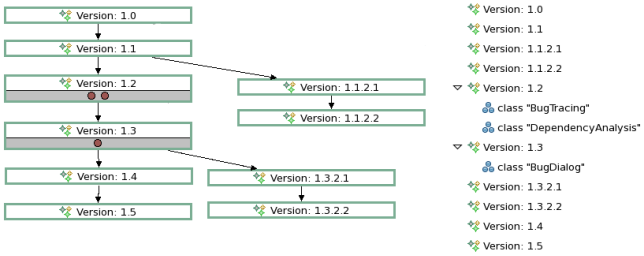

Figure 8. The identification of day flies

In our tool implementation of that analysis task (cf. Figure 8) those elements are visualized by brown circles. This provides a quick overview of model revisions containing day flies. The outline view depicts the elements, if the user wants to examine them in detail.

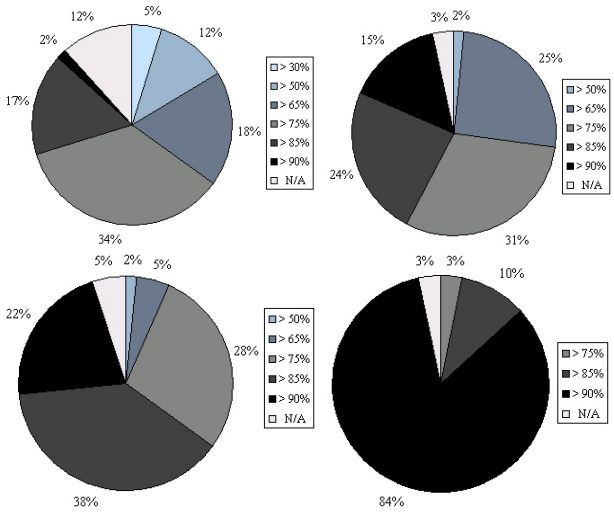

Figure 9. Performance enhancement

5 Evaluation

We evaluated the analysis tool and its visualization methodologies in particular in an empirical case study. The main objective of the study was to assess the applicability of the tracing approach to small histories of UML class model documents. Furthermore we were interested in the users' acceptance of the visualization.

The case study involved 30 developers with different levels of experience, mainly students and university researchers. It turned out that the evaluation results did not differ significantly among the developer types; therefore we do not differentiate between these groups here. Nevertheless, complete detailed results can be found in [15].

Before each test started, the attendees got a short introduction into the analysis tool. During the test they had to analyze model histories; first manually and afterwards with help of the tool. In order to manually analyze the model histories, test persons were provided with standardized XMI files which could be opened in a modeling tool of the test persons' choice and with JPEG files showing the graphical representation of the models. In both phases they had to fill in a questionnaire that asked for time exposure, experiences, preferences, and problems. The questionnaire also contained a general section about scoring the tool.

5.1 Application to an Unknown History

All test persons had to analyze a history of an unknown UML class model. This situation is typical in daily work of reverse engineers. The class models were based on different development stages of our tracing approach. The size of the single model documents ranged between 25 and 30 classes. Although that size is rather small for analysis tasks in daily practice, the different results between manual and tool-assisted analysis are significant.

For each feature of our tool implementation two specific analysis problems were given to the test persons. First point of interest has been the performance comparison of tool-based analysis vs. manual work. Figure 9 depicts the enhancement of performance. As shown in the upper left the time needed to trace single elements was already reduced by at least 50% for 83% of the test persons. The tracing of bugs, i.e. tracing of several elements at a time regarding the degree of changes, was reduced by at least 75% or more in almost 75% of the cases (see upper right). In dependency analysis (lower left) the needed time was halved for 95% of the test persons. Day flies were nearly impossible to be determined by the test persons manually as the performance enhancement states in the lower right.

Beside time reduction, the tool-based analysis produced all results correctly, while the test persons produced erroneous results during their manual analysis. Although the correctness was not considered interesting and has not been recorded, we estimate an error rate of 30% for the manual approach.

Summarized over the four scenarios the developers preferred significantly the tool solution with 108 votes. Only one vote was given to the manual approach, while there were 11 abstentions. These preferences have been explained by several reasons; e.g. performance was mentioned 59 times, 31 test persons commended simplicity. Only one participant of the study believed more in his own experience than in any tool.

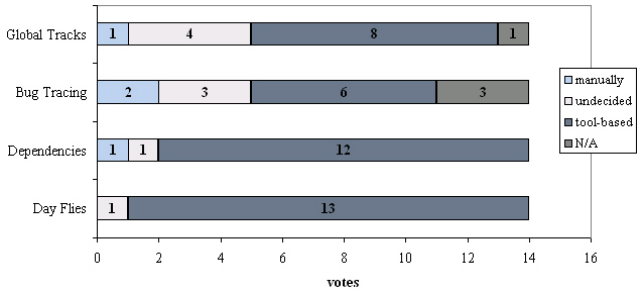

Figure 10. Analysis of well-known histories

5.2 Application to a Well-Known History

Half of the test persons also analyzed a history of UML class models that has been developed by them. The models contained around 20 classes and described the design of an auction and trading system that has been developed in groups of 4–6 persons during a one term software development course. 3 student teams attended the case study.

Despite the joint development of the models and the fairly good knowledge of their history 86% of the test persons preferred the tool-assisted analysis of the model history. Already models with 20 classes are too large to keep an overview of all elements. Figure 10 shows the performance advantages of the tool solution against the manual approach. While the manual approach benefits from knowledge and experiences of developers, the technical solution was superior with performance and correctness on the one side and overview, visualization, and user assistance on the other side.

Beside the normal experiments, the latter group of test persons offered the opportunity to verify the information provided by our analysis tool. The knowledge about the real history of the models allowed a thorough examination of the results computed by our tracing approach. Taken together 100% of the provided information has been judged to be correct. That result was expected, since the correspondence analysis used in the core of the trace computation has been tested intensively in the past [16].

5.3 General Experiences with the Tool

Finally, we were interested in the general experience of developers using our tool for tracing of elements and history analysis. Thereby, the examination was focused on the visualization methodology on the one hand and benefits for developers' work on the other hand.

For the visualization methodology itself, we asked the test persons to evaluate it in three categories. Illustration and graphical controls were declared to be good. Only the handling was not clearly rated to be intuitive. Figure 11

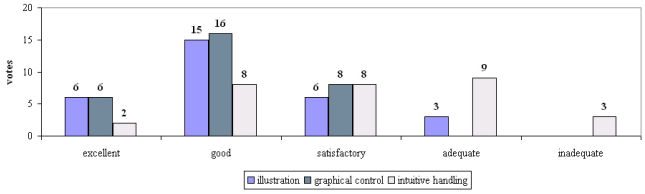

Figure 11. Evaluation of the visualization

shows the distribution in detail.

In addition the developers were asked for the support of their needs during the analysis tasks. In general the benefit was considered very high; especially for bug tracing and dependency analysis which are hard to solve manually. The identification of day flies, however, was rated to be of limited usefulness (cf. Figure 12). We consider this less surprising since mainly developers participated at the case study; analysts or project managers might have a different point of view.

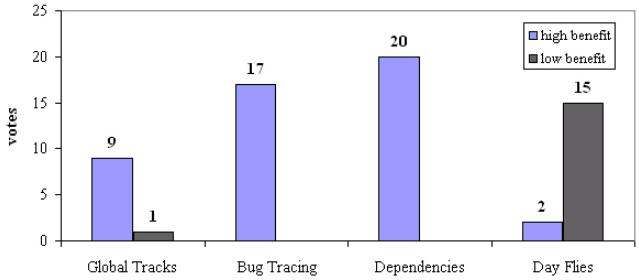

Figure 12. Support of needs

6 Related Work

Software evolution in general has already been part of research activities since many years, e.g. [25, 4, 20]. During maintenance and reengineering the analysis of evolution has become an unavoidable task. Besides learning general laws on evolution, e.g. [18, 27], the knowledge about individual projects becomes central point of interest. There exist many approaches that use history information stored in version management systems to derive statements about software under observation.

Lanza's evolution matrix [17] offers a static visualization of software metrics over time to provide for instance information about growth behavior of modules or day flies, i.e. modules that exist only in one revision. Another visualization approach on statistical history information has been proposed in [11]. Both approaches yield information about status and evolution behavior of a software system to locate points for improvements. Other approaches of that kind are [12, 26].

Another well-researched analysis aspect is relations between elements. These relations, also known as logical coupling, provide information about elements that should be changed together. [3] extracts change-patterns from CVS repositories based on change log information. In [32] a similar approach is proposed that derives association rules. An Eclipse plug-in guides developers during their work interactively. Interactive analysis is also provided by [6]; they allow navigation between coupled elements. These approaches work basically on coarse-grained level and do not support coupling detection between fine-grained software elements.

The approach of Dantas et al. [7] analyses model evolution and concurrently changed elements in particular in a more generic way. Based on a UML aware model repository [22] information about logical coupled elements is collected. Thereby, the approach supports association rules within the same model abstraction level, i.e. intra-model-traces and inter-model-traces across different levels of abstraction, e.g. analysis, design, code. However, the approach is somehow trivial since it relies on a proprietary model repository, which can hardly be found in daily practice. Inter-model-traces are not subject of our approach.

The inter-model-aspect of Dantas' approach, i.e. creating links between different models, leads to the perception of requirements engineering. In the last decades the term traceability has mainly been manifested by the requirements engineering community. Gotel and Finkelstein [14] analyzed the problem of requirements traceability very detailed. Based thereon many approaches towards traceability of requirements have been proposed, e.g. [24, 10]. The latter one, for example, steers towards tracing based on a scenario-driven analysis of runtime information. The approach looks for elements that bear a relation in a scenario. Thereby, relations may interleave several models. However, similar to other approaches it works on a snapshot of a software system; the evolution is not analyzed here.

In [2], Antoniol et al. propose a first approach to tracing individual software elements in object-oriented software. In a first step similarities based on string edit distances and metrics are calculated. In a second step a maximum matching algorithm is used to compute the mapping between two versions. Visualization allows analysis tasks on the traced elements. However, the approach works mainly on class level and is not applied to fine-grained element tracing.

More fine-grained analysis is provided by Xing and Stroulia [30]. Based on the hypothesis that evolutionary information is hardly documented they recover information from a version history to provide an understanding of how and why software has changed. Therefore they compare class models reverse engineered from Java sources with a differencing algorithm. The changes between each version are summarized to infer the development methodology. Nevertheless, the comparison procedure and consequently the analysis capabilities are restricted to class models.

7 Conclusion

This paper addresses the tracing of fine-grained model elements. In model-driven engineering models are part of software and therefore stored in version management systems. The identification of particular elements in different revisions or variants of a model becomes a challenge.

Our approach integrates difference computation with the analysis of version histories. Various differencing algorithms can be used to compute correspondences between elements of two model revisions. Applied to the entire version history of a model we are able to trace single elements, i.e. the location of corresponding elements in each revision. For our purposes we used the highly generic differencing algorithm SiDiff. Due to the fact that it works internally with a graph structure, it is applicable to a wide range of models and we can provide intra-model tracing of elements in UML and domain specific languages similarly. Assuming short versioning cycles, the differencing algorithm supports tracing of even heavily refactored elements.

By visualizing the tracing information we provide a powerful analysis tool that allows model developers to locate single elements or entire fragments in other revisions or variants. In addition we can compute further information based on traces, such as dependencies or day flies on a fine-grained level.

The empirical evaluation of our visualization approach has shown a high benefit for developers and maintainers; not only by accelerating various tasks, but also by supporting them with trustworthy information.

While this work focused on the computation of element traces and used visualization mainly for presenting results, our future work addresses additional analysis features. Interactive model navigation can be used to support developers in model-driven software maintenance. An example is the integration of trace information with polymetric views. Currently, we integrate the above-mentioned approach with clone detection methodologies to provide knowledge about equal model fragments in different variants. The complexity of comparing many models pairwise is currently reduced by enhancing our difference algorithm with capabilities for n-ary comparisons.

References

[1] M. Alanen and I. Porres. Difference and union of models. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, October 2003.

[2] G. Antoniol, G. Canfora, and A. D. Lucia. Maintaining traceability during object-oriented software evolution: a case study. In *Proceedings of the 15th International Conference on Software Maintenance (ICSM '99)*, pages 211–219, Oxford, UK, September 1999.

[3] S. Bouktif, Y.-G. Guéhéneuc, and G. Antoniol. Extracting change-patterns from CVS repositories. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, pages 221–230, Benevento, Italy, October 2006.

[4] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance*, 13(1):3–30, 2001.

[5] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD*, 25(2):493–504, 1996.

[6] M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, pages 189–198, Benevento, Italy, October 2006.

[7] C. R. Dantas, L. G. P. Murta, and C. M. L. Werner. Consistent evolution of UML models by automatic detection of change traces. pages 144–147, September 2005.

[8] Eclipse Foundation. The Eclipse project. `http://www.eclipse.org/eclipse/`.

[9] Eclipse Foundation. Graphical Editor Framework. `http://www.eclipse.org/gef/`.

[10] A. Egyed. A scenario-driven approach to traceability. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, pages 123–132, Toronto, Canada, May 2001.

[11] S. Eick, T. Graves, A. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *Software Engineering*, 28(4):396–412, 2002.

[12] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings of the 13th International Conference on Software Maintenance (ICSM '97)*, pages 160–166, Bari, Italy, October 1997.

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

[14] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the 1st International Conference on Requirements Engineering (ICRE '94)*, pages 94–101, Colorado Springs, USA, April 1994.

[15] H. Hutter. Nachverfolgbarkeit von Modellelementen in Versionshistorien. Master's thesis, University of Siegen, Germany, 2007.

[16] U. Kelter, J. Wehren, and J. Niere. A generic difference algorithm for UML models. In *Proceedings of the SE 2005*, Essen, Germany, March 2005.

[17] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE 2001)*, pages 37–42, 2001.

[18] M. Lehman, J. Ramil, P. Wernick, D. Perry, and W. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings of the 4th International Software Metrics Symposium (METRICS '97)*, Albuquerque, NM, USA, November 1997.

[19] S. Melnik, H. Garcia-Molina, and E. 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.

[20] T. Mens, J. Buckley, M. Zenger, and A. Rashid. Towards a taxonomy of software evolution, 2003.

[21] D. Ohst, M. Welle, and U. Kelter. Differences between Versions of UML Diagrams. In *Proceedings of the ESEC/FSE'03, Helsinki*, September 2003.

[22] H. Oliveira, L. Murta, and C. Werner. Odyssey-VCS: a flexible version control system for UML model elements. In *Proceedings of the 12th International Workshop on Software Configuration Management*, pages 1–16, Lisbon, Portugal, September 2005.

[23] OMG. UML Specification 2.0. `http://www.uml.org/`.

[24] F. Pinheiro and J. Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–66, 1996.

[25] V. Rajlich. Software change and evolution. In *Proc. of the Conference on Current Trends in Theory and Practice of Informatics*, pages 189–202, 1999.

[26] R. Robbes and M. Lanza. Change-based software evolution. In *Proceedings of the 1st International ERCIM Workshop on Challenges in Software Evolution (EVOL '06)*, pages 159–164, Lille, France, April 2006.

[27] G. Robles, J. Amor, J. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE '05)*, pages 165–174, Lisbon, Portugal, September 2005.

[28] C. Treude, S. Berlik, S. Wenzel, and U. Kelter. Difference computation of large models. In *Proceedings of the ESEC/FSE'07*, Dubrovnik, Croatia, September 2007.

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

[30] Z. Xing and E. Stroulia. Understanding phases and styles of object-oriented systems' evolution. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM'04)*, pp. 242–251, Chicago, USA, September 2004.

[31] Z. Xing and E. Stroulia. UMLDiff: An algorithm for object-oriented design differencing. In *Proc. of the International Conference on Automated Software Engineering (ASE'05)*, pages 54–65, Long Beach, USA, November 2005.

[32] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 563–572, Edinburgh, UK, May 2005.