How to Trace Model Elements?

Sven Wenzel
Software Engineering Group, University of Siegen
wenzel@informatik.uni-siegen.de

Motivation In model-driven engineering models enjoy the same status as customary source code. During the development process, many different versions or variants of models are created. Thereby, developers need to trace model elements between different versions to answer questions like *Since when does element X exist in this model?* or *When did element Y disappear?*. Regarding variants developers are often interested if certain elements or groups of elements (e.g. containing a bug) also exist in other variants. Questions from the analytical point of view, such as logical coupling, require traceability of model elements, too.

Traceability of elements becomes a challenge, since in daily practice, models are often stored as textual files, e.g. XMI, in versioning systems such as CVS or SVN. However, these systems are not aware of models, i.e. syntax and semantics inside the textual files. Subsequently, we present an approach for fine-grained element tracing based on difference computation. We furthermore offer a model-independent visualization of our trace information.

Differencing Models The main task of tracing is to locate the correspondence of an element in another model. The comparison of two succeeding documents and the location of correspondences respectively is known as difference computation and a daily task in software configuration management. Modern difference tools are able to compare models on an appropriate level of abstraction, which is basically a graph with tree-like structure; models are composed of elements which in turn have sub elements. They are not exactly trees due to cross references.

For our tracing approach, we use a generic similarity-based algorithm, called SiDiff, which was part of our prior research [1]. Instead of relying on persistent identifiers it computes similarities between model elements in a bottom-up/top-down algorithm, according to the tree-like structure of the models. If two elements reveal a unique similarity and they are not similar to other elements as well, they are matched. A threshold ensures a minimum similarity to avoid unsuitable matches. Elements of cycles are compared repeatedly as long as new matches can be found. Thereby, SiDiff can compare even less tree-like structures, e.g. Petri nets, whose elements are not qualified by their compositional structure but by their neighborhood to other elements. In a pre-phase a hash value is calculated for each element and elements with identical hash values are matched. Thereby, runtime of pairwise comparison is reduced and precision is improved since identical elements provide trustworthy fix points for comparison of neighbor elements.

Computation of Trace Information The comparison of two models provides information about correspondences, i.e. the same element occuring in both models. In order to compute trace information about an element in a model version, this model 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.

Starting from the basic model, for each element that occurs also in the succeeding models a track is created, e.g. for element *A* in Fig. 1. A track is a chain of model elements representing the same element occuring in different model revisions. Due to model variants the track may split into branches forming the shape of a tree.

The elements can either occur in the succeeding models without any changes or they have been changed. Unchanged elements are located immediately by the hashing functionality of the difference algorithm. Elements that have been changed from one revision to another are located by the similarity computation facilities. Thereby the threshold defined for each element type prevents from correspondences between elements with significant changes.

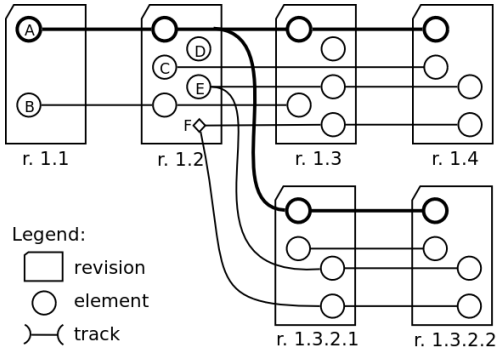

Figure 1. Examples of tracks

If an element cannot be located in any subsequent revision of a model the track of that element ends; e.g. 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 a track may contain gaps, e.g. track *C*. Gaps may also cover branching points of a model, e.g. the one marked with *F* concatenating the tracks of two branches. Elements without any corresponding partner, e.g. element *D*, do not become part of any track.

Tracing of Elements In order to trace elements at daily work, the tracks computed above must be visualized in a meaningful way for developers. Therefore, we have implemented our approach as Eclipse plug-in using the GEF framework. A screenshot is shown in Fig. 2.

Tracks are visualized in an abstract representation independent of any model type. Rectangles represent different revisions of the given model, inside a rectangle

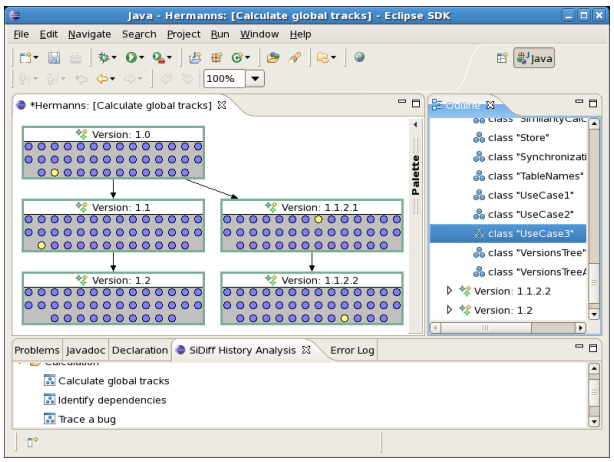

Figure 2. Screenshot of the analysis tool

each model element is represented by a small colored circle. The color provides information depending on the current analysis task. 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 side provides different analysis tasks to choose from. The current implementation supports four different analysis tasks:

Global Track Analysis. The analysis of global tracks is the simpliest task. It only uses the track information computed as described above. If a user selects a single model element in any revision of the model history, the occurences 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 examplary screenshot above, a class named *UseCase3* is selected; it occurs and is marked yellow respectively in each revision except revision 1.2.

Tracing a Bug. A bug usually consists of more than just one model element. 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 small changes in the other version. Therefore, similarities of the element set in different versions as a whole are also computed and must be above an additional threshold for the other fragment to be considered as a repetition of the bug. In bug tracing analysis the selected bug elements are colored blue, while the bug candidates in the other revisions are colored regarding their probability to be a bug. Unconcerned elements are greyed out.

Finding dependencies. Dependency analysis, formerly known as logical coupling, provides information about software elements bearing 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 track through the whole history. For each revision where the traced element has been changed, we check other elements that are also affected by modifications. Those elements are colored according to their degree of dependency.

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. Technically they can be identified very easily as they are not part of any track. In our visualization those elements are shown by brown circles. This provides a quick overview about model revisions containing day flies. The outline view depicts the elements themselves, if the user want to examine them in detail.

Evaluation

We evaluated our approach and its visualization methodologies in an empirical case study involving 30 developers. The probands had to perform different analysis problems on given model histories; first manually with a modeling tool of their choice and afterwards with help of our approach. Both times they had to fill in a questionaire asking for time exposure, experiences, etc.

First analysis covered a history of models with 25 to 30 classes unknown to the probands, which is typical for reverse engineer's work. Although the models are rather small for daily practice, the tool-assisted analysis provided significant enhancements. Summarized over the four analysis scenarios, time exposure was reduced by more than 75% in 75% of the cases. Beside time reduction, our approach computed all results correctly, while the probands produced erroneous results during manual analysis; we estimate an error rate of 30%. Furthermore, day flies were nearly impossible to be detected manually.

Half of the probands also analyzed models designed by themselves during a software development course. Despite the fairly good knowledge of their history 86% of the probands preferred the tool-assisted analysis; already models of 20 classes are too large to keep an overview. The latter group of probands allowed verification of the traces computed by our approach; all information has been judged to be correct as expected, since the correspondence analysis used for the trace computation has been tested intensively in the past.

Summarized over all scenarios the probands preferred essentially the tool solution; mainly explained by performance, trustworthy of results and simplicity. The visualization itself got good ratings for illustration and graphical controls; only the handling was not clearly rated to be intuitive. 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 usefullness.

References

1. U. Kelter, J. Wehren, and J. Niere. A generic difference algorithm for UML models. In *Proc. of SE 2005*, Essen, Germany, March 2005.