Constructing and Visualizing Transformation Chains

Jens von Pilgrim[1], Bert Vanhooff[2], Immo Schulz-Gerlach[1], and Yolande Berbers[2]

[1] FernUniversität in Hagen, 58084 Hagen, Germany
{Jens.vonPilgrim,Immo.Schulz-Gerlach}@FernUni-Hagen.de
[2] DistriNet, K.U.Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium*
{bert.vanhooff,yolande.berbers}@cs.kuleuven.be

Abstract. Model transformations can be defined by a chain or network of sub-transformations, each fulfilling a specific task. Many intermediate models, possibly accompanied by traceability models, are thus generated before reaching the final target(s). There is a need for tools that assist the developer in managing and interpreting this growing amount of MDD artifacts. In this paper we first discuss how a transformation chain can be modeled and executed in a transformation language independent way. We then explore how the available traceability information can be used to generate suitable diagrams for all intermediate and final models. We also propose a technique to visualize all the diagrams along with their traceability information in a single view by using a 3D diagram editor. Finally, we present an example transformation chain that has been modeled, executed and visualized using our tools.

1 Introduction

Monolithic transformations, like most non-modularized software entities have some inherent problems: little reuse opportunities, bad scalability, bad separation-of-concerns, sensitivity to requirement changes, etc. A number of these problems can be solved by decomposing a transformation into a sequence of smaller sub-transformations: a transformation chain or transformation network.

Each stage of a transformation chain produces intermediate models, which means that the total number of models can become very large. In the ideal case this all happens automatically and correctly so we do not have to care much about the intermediate models; only the final target models are to be considered. Unfortunately, we often have to study and possibly modify intermediate models manually in order to get the desired result, if only for debugging purposes. For example, if we detect an unexpected structure in the final output

* Some of the described work is part of the EUREKA-ITEA MARTES project, and is partially funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

I. Schieferdecker and A. Hartman (Eds.): ECMDA-FA 2008, LNCS 5095, pp. 17–32, 2008.

model we must look at the intermediate models to identify the responsible transformation. In another case we might want to optimize the results of a complex sub-transformation manually. In these cases, it may also be useful to inspect the relations between the models under study – i.e. traceability. We have identified three main problems with the scenarios sketched above.

1. Defining, executing and maintaining transformation chains is not a straightforward task. This is particularly difficult if a mixture of different transformation languages and tools is used.
2. Relying on automatic layout mechanisms to generate diagrams for intermediate models is often not satisfactory. Since layout information is neither transformed nor preserved across transformation stages, each model's diagram can look completely different even if subsequent models have only slightly changed.
3. Trying to interpret traceability information manually is difficult since there is no good graphical representation technique for this kind of information. Therefore, this information can often not be fully exploited.

In this paper we explain our approach to construct a transformation chain and present its output to a human user in a clear and comprehensible fashion.

We have extended UniTI (Unified Transformation Infrastructure) [1], a tool for transformation chain modeling and execution, with the ability to keep track of traceability models and their relation to other models in the chain. We discuss how the output of a transformation chain – an intricate network of models connected by traceability links – can be visualized. We explain how a proper diagram layout can be produced for the generated models by leveraging traceability information. Furthermore, we propose a technique to visualize the diagrams, together with traceability links in a 3D editor.

The remainder of this paper is structured as follows. In Section 2 we give a short overview of UniTI. Since traceability plays a crucial role in transformation chains, we discuss our perspective on that subject in Section 3. In Section 4 we describe how to automatically create the layout for generated models and introduce Gef3D, the framework used for implementing a 3D diagram editor. We demonstrate our solution with an example transformation chain in Section 5 and summarize related work in Section 6. Finally, we wrap up by drawing conclusions and identifying future work in Section 7.

2 Setting Up a Transformation Chain

Currently, most transformation technologies do not offer much support for reusing and composing sub-transformations as high-level building blocks. They focus on offering good abstractions and a clear syntax for implementation. Since many transformation languages are now reaching a certain maturity, we need to start focussing on reuse and composition of transformations.

We have developed an Eclipse plugin called UniTI (Unified Transformation Infrastructure) that manages building blocks of a transformation chain. In this

section we give a short overview of UniTI, for a more elaborate discussion we refer to [1].

The basic principles of UniTI are inspired on those of Component Based Software Engineering (CBSE) [2]. We have reformulated them to fit the model transformation domain:

Black-Box. The black-box principle indicates a strict separation of a transformation's public behavior (what it does) and its internal implementation (how it does it). Implementation hiding makes any technique an eligible candidate for implementing the transformation.

External specification. Each transformation should clearly specify what it requires from the environment and what it provides.

Composition. Constructing a transformation with reusable transformation building blocks should be considerably less work than writing it from scratch. A transformation should be a self-contained unit and composition should be easy and should require only a minimum of glue code.

Current transformation technologies do not focus on the principles outlined above; in order to reuse an existing transformation implementation in a chain, a very deep knowledge of that implementation is usually required [1]. This violates the black-box principle and is often consequence of the limited external specification. In the related work section we discuss that QVT does have limited support for the above principles.

In UniTI we try to not only adhere to the three CBSE principles, but also to MDD practices. A transformation chain in UniTI is therefore a model itself, making it a possible subject to MDD techniques such as model transformation.

A typical usage scenario of UniTI goes as follows (see Figure 1). A *Transformation Developer* who is specialized in one or more transformation languages provides a number of executable transformations. The *Transformation Specifier* is responsible for encapsulating these implementations in the the unified UniTI representation (*TFSpecification*). This entails specifying the valid context in which the transformation may be executed by defining pre- and post-conditions. Depending on the transformation language used for the implementation, it might also involve choosing concrete metamodels [3] or a transformation

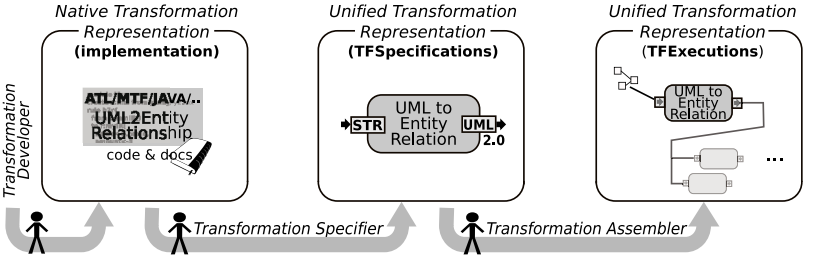

Fig. 1. Overview of UniTI principles

direction [4]. Currently we support semi-automatic generation of such transformation specifications for ATL [3], MTF [4] and Java transformations. Finally, the *Transformation Assembler* creates a transformation chain model by interconnecting an arbitrary number of transformation instances (*TFExecutions*). This can be done without any knowledge of implementation or technology details of the underlying transformations since these are completely hidden by UniTI at this point. Each transformation, be it ATL, MTF or Java, is represented in exactly the same way. Once the transformation chain is completely modeled, UniTI automatically executes all stages and produces the intermediate and final models.

3 Transformation Traceability

Traceability has many applications in software engineering. In requirements engineering, traces make the link between requirement documents and design models or code while program analysis focuses on traces between function calls. From an MDD point of view, traces can be any type of relation between model elements. Traceability information is typically expressed as a model of its own: traces are elements of a traceability model.

For this paper, we consider an MDD-specific kind of traceability: transformation traceability. In this case, traces are produced by automatic model transformations and denote how target elements are related to source elements and vice versa. In this section we discuss how traceability is supported in UniTI (see 3.1) and which traceability metamodels we use (see 3.2).

3.1 Traceability Support in UniTI

Since traceability information can be represented as a model itself, it can easily be produced as an additional output of a transformation. We assume that the creation of traceability models is explicitly implemented by the transformations and that traceability models are available as normal transformation outputs.

While regular models can usually be interpreted independently, traceability models contain cross-references to input and output models and are thus meaningless on their own. These references are usually unidirectional in order to prevent pollution of the regular models. This means that, given a traceability model, we can locate the models to which it refers, but not the other way around. Given only a model, we cannot locate the relevant traceability model(s).

Therefore, we have extended UniTI so it keeps track of the relations between all models and traceability models involved in a transformation chain. Tools such as Gef3D (see Section 4) can hence query the transformation chain model to determine how models and traceability models are related.

Figure 2 shows a part of the UniTI metamodel that introduces dedicated support for traceability. This is a new feature that was not yet described in [1]. A *TFExecution* (an instantiated transformation) has a number of input and output parameters (*TFParameter*), which can be considered as model containers.

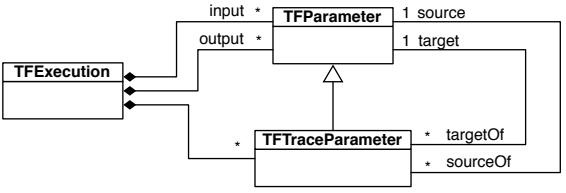

Fig. 2. Traceability Tracking in UniTI

Traceability parameters (*TFTraceParameter*) represent a special type of parameter that encapsulates a traceability model and keeps track of its *source* and *target* model; *TFParameter*s have opposite references: *sourceOf* and *targetOf*. Notice that this structure (multiplicities of 1 for both *source* and *target*) enforces a separate traceability model for each input/output model pair. This decision is made to keep sub-transformations as independent as possible and to prevent sub-transformation implementation details from leaking (see also next subsection).

In summary, traceability models play a first class role a transformation chain modeled in UniTI, which facilitates traceability discovery by model navigation.

3.2 Traceability Metamodel

A lot of different metamodels for traceability information have already been suggested in literature, e.g. in the MDA foundation model [5]. Some of them are specialized, others propose a generalized model. At this point in time there does not seem to be a generally accepted model. One could wonder whether a generic model is necessary since traceability models can be transformed to conform to a different metamodel if required. For this paper we use this transformation approach since both UniTI and Gef3D use different traceability models.

The traceability models that we use are illustrated in Figure 3. On the left, we see the traceability metamodel that is used by UniTI and on the right, the metamodel required by Gef3D. Note that UniTI can also be used without traceability and supports any kind of traceability metamodel, but in order to enable some additional uses of traceability [6], this particular metamodel is required.

The **UniTI Traceability Metamodel** (left part of Figure 3) is designed to minimize exposing implementation details of individual transformations in order to prevent tight coupling within a transformation chain. A sub-transformation should never be able to depend on specific implementation details (not part of its specification) of a previous transformation. However, many transformation traceability models indirectly expose these kind of details by including the (implementation specific) transformation rule name in each trace. Even if this is not the case, an $m : n$ trace that links all elements involved in a single transformation rule, can still expose the rule structure. This means that subsequent transformations can depend on this structure, causing hard coupling within the transformation chain. As a result, sub-transformations cannot easily be replaced by others without breaking the chain.

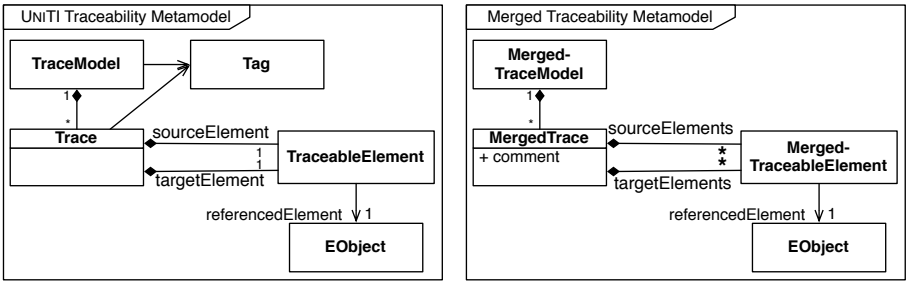

Fig. 3. UniTI Traceability Metamodel and Merged Traceability Metamodel

In order to hide these implementation details, the UniTI traceability metamodel only allows binary trace relations. A tagging mechanism is offered to annotate individual traces with a specific semantic meaning that is independent of the implementation. In the future, these tags will make part of the specification of a transformation. More details on these tags can be found in [6].

The **Merged Traceability Metamodel** (MTM – right part of Figure 3), used in Gef3D, is similar to the UniTI traceability metamodel but has at least two important differences. First, a trace can contain a comment, which can be the transformation rule name. Second, a trace can contain multiple source and target elements. The main design consideration of the merged traceability model is to minimize the amount of traces (*MergedTraces*) that are given to the diagram layout algorithm (see Section 4.1). This is accomplished by creating only a single trace element for each group of related model elements; typically a single source element with a number of target elements.

It is clear that both traceability metamodels have different goals and therefore have a different structure. In order to overcome these differences, we have used a model transformation to translate UniTI traceability models to merged traceability models. By using this technique, we ensure that both tools are loosely coupled and that their traceability metamodels can evolve independently. If one changes, it suffices to change the transformation.

In the next section we show how the merged traceability model is used to generate model diagrams and how traces between different models can be visualized in a 3D editor.

4 Visualization

When visualizing a transformation chain, two main issues can be identified: How to layout diagrams of generated models and how to visualize traces.

UML diagrams such as class diagrams can become quite complex. Often, much time is spent by the user to layout a single model. For example, related elements are grouped together or line crossings are minimized. While automatic layout algorithms may achieve good results with respect to some optimization criteria, users often try to visualize semantics in the layout which is not expressed in the

model itself. For example in Figure 6 important classes are put in the center of the diagram, something that cannot be mimicked by layout algorithms.

A big disadvantage of model transformations is that they don't transform layout information; diagrams are usually not transformed. If we want to take a look at the final output model or at any of the intermediate models, we have to rely on automatic layout algorithms. Since they only consider the current model, the spacial relation to the previous diagrams is often lost. That is, the user has created a mental map of the source diagram, knowing where on the diagram certain elements are located. Common layout algorithms do not take this mental map into account, therefore it can be very difficult to locate corresponding elements in the target diagram. This is especially true for chains in which intermediate models were used to simplify a single transformation – the structure of the different models is usually very similar and thus it is even more important to retain the mental map.

In [7] Kruchten writes: "Visual modeling [..] helps to maintain consistency among a system's artifacts." [7, p. 11]. A base feature to "maintain consistency" of models are traces, but traces are rarely visualized. Displaying traces between elements of two models makes the diagram even more complex. A first approach may be to draw the two models beneath or besides each other and to display the traces as lines connecting related elements. But in this case, the diagrams quickly become cluttered; many traces cross each other and other diagram elements and the whole diagram becomes quite large (difficult to display on a screen). Changing the diagram layout in order to minimize line crossings results in destroying the manual layout and the mental map.

While traces make the problem of visualizing a transformation chain more complex, they can help us in solving the layout problem. We show how to use the traceability models, produced by the transformation chain, to automatically layout intermediate diagrams based on the initial manually created model diagrams, hence preserving the mental map (see 4.1).

For solving the traceability visualization problem we "expand" the dimensions in which we draw the diagrams. Most software engineering models are drawn using two-dimensional diagrams. For drawing models with traces we can use three-dimensional diagrams, using the third dimension to display the traces. In most cases editors for single models already exist and we want to reuse these editors. We therefore adapt existing two-dimensional editors and combine them to a single 3D multieditor. This can be achieved by displaying the 2D diagrams on 3D planes. Traces can then be visualized as connections between elements of the different models displayed on these planes (see 4.2).

4.1 Derived Layout

The transformations used in a transformation chain usually only transform the domain models. In most cases these domain models do not contain any layout information. The layout of diagrams is defined using so called notation models, but these models are not transformed. This is why generated models do not have a layout and why we rely on layout algorithms in order to display these

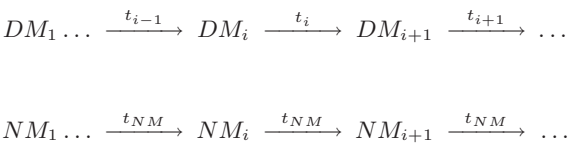

Fig. 4. Transformation of Notation Models

models. What we suggest here is a layout algorithm which is indeed a kind of a transformation. Figure 4 illustrates the idea.

The first row in the figure shows the transformation chain as created by UniTI. Some domain models DM_n are related via transformations t_n. Additional notation models NM_n are used in order to display the domain models. To preserve the structure of a diagram, that is the layout created by a user in the very first notation model NM_1, we have to make use of the information stored in the notation model of the predecessor. Therefore we map the source notation model to a target notation model. This is possible because a path $NM_{i+1} \rightarrow DM_{i+1} \rightarrow DM_i \leftarrow NM_i$ exists.[1] A detailed description of the algorithm can be found in [8].

4.2 2D Diagrams in a 3D Space

A lot of graphical editors are implemented today using the Eclipse Graphical Editing Framework (GEF)[9]. Examples for such editors are the TopCased Tools [10], or the Eclipse UML Tools. For visualizing models and traces in 3D, we do not want to re-implement these editors, even if they are 2D only. The main challenge here is to provide a technique allowing us to use existing editors in a 3D space. Besides minimizing implementation effort, we also retain the graphical syntax used by these editors. Since we actually use the original editor code, the 3D version does not only display the models but also enables us to modify the diagrams in this 3D view.

The concept of displaying models in a 3D space is to project common 2D diagrams on planes and combine these 2D diagrams with real 3D elements. Here, traces are visualized as 3D elements connecting 2D elements of the original diagrams. Figure 5 shows four planes on which 2D digrams are drawn. 3D connections (here traces) connect elements of these diagrams. The models of the figure are explained more detailed in Section 5.

While 3D editors (or visualizations) are used in many domains (e.g. CAD), they are rarely used in software engineering. Most software engineering diagrams are graphs, that is nodes and connections drawn as lines between these nodes. GEF is a framework for implementing editors for this kind of diagrams. Figure 6

[1] While the transformation is usually unidirectional, the trace is always bidirectional ($DM_{i+1} \rightarrow DM_i$). Since domain models are independent from the notation models, i.e. the connection is directed from the notation model to the domain model ($DM_i \leftarrow NM_i$), we have to use a reverse lookup, but this problem can easily be solved.

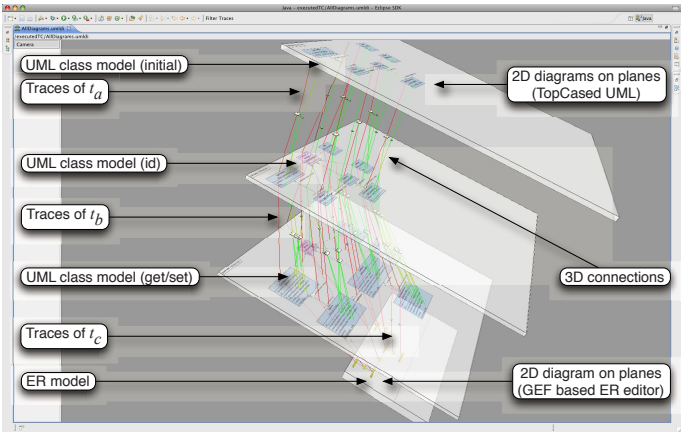

Fig. 5. 2D diagrams on planes and 3D connections

illustrates the main concepts of GEF and Gef3D. GEF is using a Model-View-Controller (MVC) architecture shown in the center of the figure. For each element drawn and edited in a GEF based diagram an MVC triple has to be implemented. The view for each element is called figure (and its class has to implement Draw2D's *IFigure* interface). The controller of a single element is called edit part (and its class has to implement GEF's *(Graphical)EditPart* interface). GEF is independent from any model implementation, but most often models are implemented using the Eclipse Modeling Framework (EMF) [11]. The figures and the edit parts are organized in a tree. The root of the figures is contained in a so called lightweight system which is a bridge between the graphical system of the platform and the figures. The edit parts are usually created by an edit part factory, using the factory pattern. All parts of a graphical editor are held together by an (edit part) viewer.

With Gef3D [12] we provide a 3D extension of GEF, making it possible to display 2D diagrams on planes in a 3D space or even to implement real 3D editors. It provides an OpenGL based implementation for figures, i.e. it provides a new interface for 3D figures (*IFigure3D*, which extends the original interface *IFigure*). The original lightweight system and the canvas, on which the figures are drawn, are replaced by 3D versions too. Instead of a 2D canvas an OpenGL canvas is used, that is we use OpenGL for rendering. These new 3D enabled view elements are instantiated by corresponding 3D versions of the graphical editor and viewer. The core of GEF can be used without further changes.

Adapting an existing GEF based editor is very simple: Instead of rendering the diagram directly on the screen, they are rendered as images wich are used as textures for some 3D elements. Most parts of the 2D editors can be reused, only the top level container figure (i.e. the diagram itself) and its controller have to be replaced. In MDD, UML is often used for modeling. TopCased UML provides

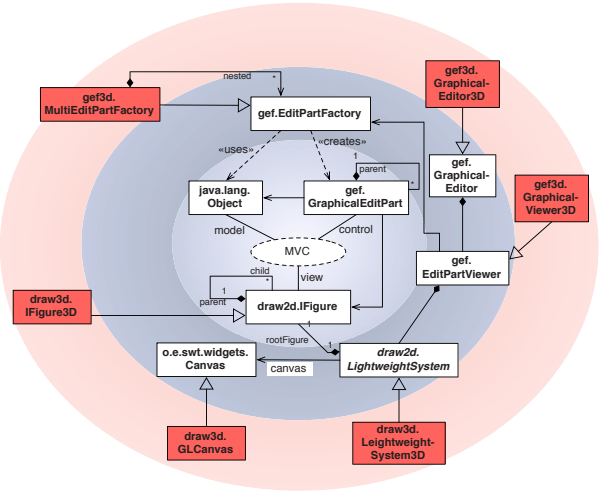

Fig. 6. GEF3D adds 3D to GEF

editors for most UML diagrams, and we adapted TopCased UML so that its diagrams can be rendered in a 3D editor. The result is not only a visualization of UML diagrams, but also a full-featured editor [2].

Since we do not only display a single diagram, multiple editors have to be combined in a single editor called multi editor. This is possible due to GEF's design of the controller components, the edit parts. These parts are created using a so called edit part factory. GEF3D extends this pattern by providing a multi edit part factory. This factory nests already existing factories and creates new elements using a nested factory based on the diagram element in which the new element is contained. As demonstrated in the following example, not only editors displaying the same type of diagrams can be combined, but also completely different editors.

The performance of the 3D visualization is quite well due to the usage of OpenGL. All 2D elements are currently rendered as images (one image per 2D diagram), so if the diagrams aren't too large, this causes no problem. We have also implemented some performance tests, showing that 1000 real 3D nodes with textures can be handled by the engine.

5 Example

In this Section we demonstrate the usage of UNITI and the capabilities of GEF3D by creating and visualizing a typical transformation chain. The transformation chain starts with a simple UML model, creates several UML models, and finally an Entity-Relationship (ER) model. The models represent, even if slightly

[2] Currently not all editor features are implemented, see Section 7.

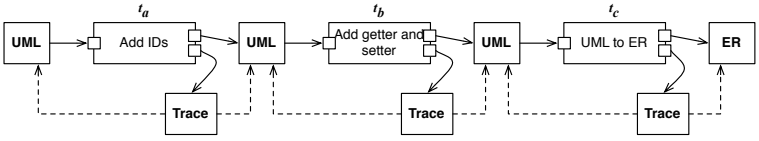

Fig. 7. Schematic of the example transformation chain

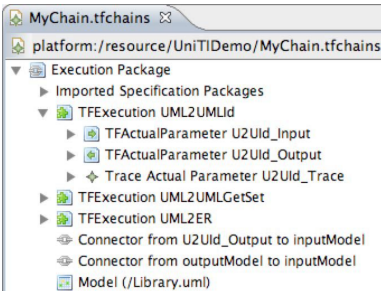

Fig. 8. Part of the transformation chain as modeled in UniTI

simplified, typical artefacts used when creating an information system. We first set up the transformation chain with UniTI and then visualize the whole chain with a Gef3D based editor.

The example transformation chain is schematically represented in Figure 7. It contains three transformation stages:

1. UML refinement – Id attribute (t_a): The first (UML) model represents a domain model of a library. Since it is a domain model, implementation details such as operations and persistency properties are omitted. The first refinement transformation t_a adds a unique identifier property to each class. For the initial model, we have manually created a model diagram so that further diagrams can be generated automatically (as described in 4.1).
2. UML refinement – getter and setter (t_b): For each public property, a public get and set method is added to control access to that property (which can also refer to an association). The property itself is then marked as private. This is useful if we want to add further transformation towards, e.g., a Java implementation.
3. UML to ER (t_c): Finally we transform the UML model to a corresponding ER model.

This transformation chain was modeled in UniTI (see Figure 8) and the results were visualized using Gef3D (see Figure 5).

The traces are displayed in different colors, depending on the level of the connecting elements. That is, traces connecting top level elements (e.g. classes or associations) are drawn red, while traces connecting nested elements (like operations) are drawn green. We have implemented a filter to hide traces of the

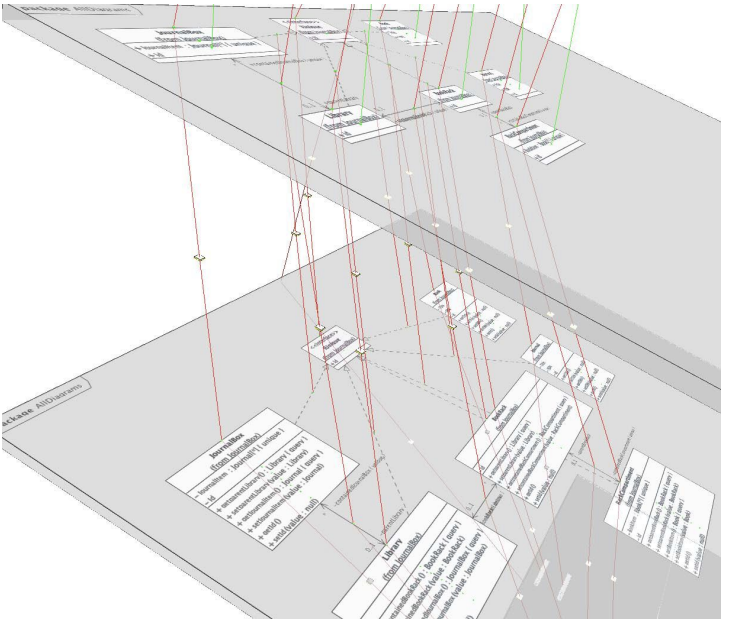

Fig. 9. Transformation chain output with filtered trace visualization

latter kind making the traceability much more clear. Figure 9 shows some filtered traces. Since this paper leaves no room for more pictures, we refer to GEF3D's[3] and UNITI's[4] website, which provide some screenshots and screencasts.

6 Related Work

The approach that was presented in this paper consists of three main areas: transformation chaining, traceability and visualization of models. In this section we identify related approaches in each of these areas.

6.1 Transformation Chains

There are a number of alternatives to UNITI that allow to build and execute transformation chains.

In [13], a transformation chain is seen as the composition of different tools that each support one or more transformations. They make a strong separation between transformation definition and execution. Transformation building blocks are represented as plugins to the Eclipse framework. Another approach similar to ours is described in [14]. They reuse classical component middleware as a

[3] http://gef3d.org/
[4] http://www.cs.kuleuven.be/~bertvh/uniti.shtml

Transformation Composition Framework (TCF); transformations, models and metamodels are all represented as components.

These systems have a similar goal as UniTI, but there are some significant differences. UniTI offers a model-based approach to specify and execute transformation chains, while the former systems use a scripting language. A model-based approach has the advantage that transformation chain models can easily be explored by other MDD tools (e.g. the Gef3D editor in Section 5) and can even be transformed themselves, for example to insert additional transformations. UniTI also integrates first class support for traceability models, which is not present in the other approaches.

Similar to UniTI, QVT [15] supports chaining of black-box transformations. However, it is up to each QVT implementation to provide concrete mechanisms to call out to external transformations. At the moment of writing, existing implementations of Borland and SmartQVT only enable Java to specify black-box transformations.

The AM3 Global Model Management approach [16] aims to build a generic model management infrastructure, including support for transformations, metamodels and other MDD artifacts. A similar approach, though more specialized towards model merging, is pursued in [17]. UniTI can be considered as one application of model management, focused on managing transformation related MDD resources.

6.2 Traceability

In our approach, we generate and visualize traceability models and use these models to generate diagram layouts. Related work concerning the generation of traceability and traceability in general is presented in this subsection, visualization related work is presented in the next subsection.

Traceability Metamodels. Many traceability metamodels have already been proposed for this purpose [18]. Some of them have a very specific purpose, others try to come to a generic model that is suited for every situation. We do not believe that such a generic model is feasible. Instead we were confronted ourselves with two different traceability metamodels when integrating UniTI and Gef3D. We have not attempted to adapt both tools to use a common traceability metamodel but rather used a model transformation to convert the traceability model when needed (see Section 3.2).

Traceability Generation. Most current transformation languages [15,3] build an internal traceability model that can be interrogated at execution time, for example, to check if a target element was already created for a given source element. Although this kind of traceability is generated without any additional required effort, there are some issues. Firstly, not all languages allow to export this information into an additional model, this is the case for ATL and MTF. Secondly, the traces cannot be adapted from within the transformation rules, which is required for our trace tagging approach. For these reasons, it is often

necessary to explicitly incorporate the generation of an addition traceability model into each transformation, as we did in our approach.

Manual implementation of traceability generation can be very cumbersome since very similar trace generating statements must be reproduced for each transformation rule. A solution for automating parts of this work has been proposed in [19]. They explain how trace generation statements can be added to an existing transformation by a model transformation – i.e. the transformation is transformed itself. This technique can be used for transformations in UniTI as well.

In [20], a distinction is made between traceability in the small (between model elements) and traceability in the large (between models). In Section 3.1, we make a similar distinction: traceability models contain links between model elements and UniTI keeps links between models and their traceability models.

6.3 Visualization

There are three kinds of related work: Work related to visualizing (software engineering) models in 3D, work related to visualizing diagrams on planes, and work related to visualizing traces.

Software engineering models like class diagrams are visualized in a 3D manner for example in [21] This work is not using common notations, but invents new ones optimized for 3D visualization. The new notation syntax has to be learned and it differs from common notations like UML, hence existing editors cannot be reused. Often the visualization tools can only present, but not edit the diagrams, while Gef3D has the potential of a real full-feature editor based on a widely used framework.

The idea of visualizing 2D diagrams on planes is described in [22] already. In this work, 3D diagrams that look quite similar to the ones presented here, are shown. However, the diagrams here were produced by an existing tool. In [22] the diagrams are only illustrations of an idea, but a tool was not yet implemented. As far as we know there still is no tool, at least publicly, available.

Most work about visualizing traces examines requirement or execution traces. In the first case, often matrices are used, a column for each requirement and a row for each artifact [23]. In [24] execution traces are visualized in 3D diagrams, but again new 3D notation syntax is used. Since usually a lot of execution traces are produced in a software system, execution traceability becomes more a quantitative information. The effect of this is that a single trace cannot be identified anymore.

7 Conclusion and Future Work

Complex model transformations can often be split up into smaller sub-transformations in order to provide better robustness to changes, to allow more reuse opportunities and to simplify the implementation of individual transformations. A transformation chain thus produces many intermediate models before delivering the final output model(s).

While this approach seems very attractive, we have identified three issues. First, we need an approach to manage all the high level MDD artifacts in a transformation chain: models, metamodels and transformations. Second, model diagram information must also be preserved throughout the transformation chain to allow manual inspection of the intermediate and final models. Third, we need a good representation for traceability information between different models.

We have proposed a solution to these problems by integrating two tools. UniTI allows to model and execute transformation chains in a technology independent way. It also keeps track of traceability models and their relation to other models. A Gef3D based tool queries the transformation chain model to find models and traceability models. Based on the traceabiltiy models, it can generate a consistent diagram layout for all models throughout the chain. Furthermore, it visualizes all these diagrams in a 3D model editor where also the traceability links themselves can be shown.

This integrated approach hence facilitates both the specification of transformation chains and their visualization. The visualization provides a concrete representation of otherwise abstract traceability information. It can for example be used to quickly localize a problem in the transformation chain in order to solve it locally in the identified sub-transformation.

We want to keep improving our approach in the future. UniTI will be extended so that traceability information of the whole chain can accessed transparently by subsequent transformations. Gef3D will be further developed in order to provide full-featured editors, i.e. the diagrams should not only be visualized but also be editable in the 3D view. Besides, the usability should be improved, e.g. by highlighting search results (or all elements connected by a trace).

References

1. Vanhooff, B., Ayed, D., Baelen, S.V., Joosen, W., Berbers, Y.: Uniti: A unified transformation infrastructure. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 31–45. Springer, Heidelberg (2007)
2. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Professional, Reading (1997)
3. Jouault, F., Kurtev, I.: Transforming models with atl. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844. Springer, Heidelberg (2006)
4. IBM Alphaworks: Model transformation framework. Misc (2004), `http://www.alphaworks.ibm.com/tech/mtf`
5. OMG: A Proposal for an MDA Foundation Model. Object Management Group, Needham, MA. ormsc/05-04-01 edn. (2005)
6. Vanhooff, B., Van Baelen, S., Joosen, W., Berbers, Y.: Traceability as input for model transformations. In: Proceedings of the European Conference on MDA Traceability Workshop, Nuremberg, Germany (2007)
7. Kruchten, P.: The Rational Unified Process. Object Technology Series. Addison-Wesley, Reading (2004)
8. von Pilgrim, J.: Mental map and model driven development. In: Fish, A., Knapp, A., Störrle, H. (eds.) Proceedings of the Workshop on the Layout of (Software) Engineering Diagrams (LED 2007). Electronic Communications of the EASST, vol. 7, pp. 17–32 (2007)

9. Eclipse Foundation: Graphical Editing Framework (GEF), Project Website (2008), http://www.eclipse.org/gef
10. Topcased: Topcased Tools, Project Website (2008), http://www.topcased.org/
11. Eclipse Foundation: Eclipse Modeling Framework (EMF), Project Website (2008), http://www.eclipse.org/modeling/emf/
12. von Pilgrim, J.: Graphical Editing Framework 3D (GEF3D), Project Website (2008), http://gef3d.org
13. Kleppe, A.: Mcc: A model transformation environment. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 173–187. Springer, Heidelberg (2006)
14. Marvie, R.: A transformation composition framework for model driven engineering. Technical Report LIFL-2004-10, LIFL (2004)
15. Object Management Group: Qvt-merge group submission for mof 2.0 query/view/transformation. Misc (2005)
16. Allilaire, F., Bezivin, J., Bruneliere, H., Jouault, F.: Global model management in eclipse gmt/am3. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067. Springer, Heidelberg (2006)
17. Salay, R., Chechik, M., Easterbrook, S., Diskin, Z., McCormick, P., Nejati, S., Sabetzadeh, M., Viriyakattiyaporn, P.: An eclipse-based tool framework for software model management. In: Eclipse 2007: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange, pp. 55–59. ACM, New York (2007)
18. Gills, M.: Survey of traceability models in it projects. In: ECMDA-TW Workshop (2005)
19. Jouault, F.: Loosely coupled traceability for atl. In: Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany (2005)
20. Barbero, M., Fabro, M.D.D., Bézivin, J.: Traceability and provenance issues in global model management. In: 3rd ECMDA-Traceability Workshop (2007)
21. Alfert, K., Engelen, F., Fronk, A.: Experiences in three-dimensional visualization of java class relations. SDPS Journal of Design & Process Science 5, 91–106 (2001)
22. Gil, J., Kent, S.: Three dimensional software modelling. In: 20th International Conference on Software Engineering (ICSE 1998), Los Alamitos, CA, USA, p. 105. IEEE Computer Society, Los Alamitos (1998)
23. Duan, C., Cleland-Huang, J.: Visualization and analysis in automated trace retrieval. In: First International Workshop on Requirements Engineering Visualization (REV 2006 - RE 2006 Workshop), Los Alamitos, CA, USA, vol. 5. IEEE Computer Society, Los Alamitos (2006)
24. Greevy, O., Lanza, M., Wysseier, C.: Visualizing live software systems in 3d. In: SoftVis 2006: Proceedings of the 2006 ACM symposium on Software visualization, pp. 47–56. ACM Press, New York (2006)