Using Visual Dataflow Programming for Interactive Model Comparison

Rainer Lutz and Stephan Diehl
Computer Science Department
University of Trier
Trier, Germany
{lutzr, diehl}@uni-trier.de

ABSTRACT

In software engineering the comparison of graph-based models is a well-known problem. Although different comparison metrics have been proposed, there are situations in which automatic or pre-configured approaches do not provide reasonable results. Especially when models contain semantic similarities or differences, additional human knowledge is often required. However, only few approaches tackle the problem of how to support humans when comparing models.

In this paper, we propose a tool for interactive model comparison. Its design was informed by a set of guidelines that we identified in previous work. In particular, our prototype leverages visual dataflow programming to allow users to implement custom comparison strategies. To this end, they can combine metrics and graph operations to compute similarities and differences, and use color coding to visualize the gained results. Additionally, a qualitative user study allowed to assess whether and how our tool facilitates iterative exploration of similarities and differences between models.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

Keywords

graphs; models; comparison; human-centered; visual dataflow programming; tools

1. INTRODUCTION

The comparison of graphs or graph-based models is of importance for different application scenarios. In software engineering, where models are often used during early phases of the development process or to provide an overview of an existing system, there is also a need for graph comparison. Software designers, for example, would like to find similar components of a system that might be replaced or even merged into a single one. Research in this area often focuses on the algorithmic challenge posed by sophisticated comparison metrics, which are supposed to compute a reasonable set of corresponding model elements. However, there are situations when automatic or pre-configured approaches do not provide satisfying results such that human interaction is required. For instance, when semantic aspects of models are compared, it becomes increasingly important to involve developers instead of entirely relying on the results of an automatic approach. While some works present visualizations to show similarities or differences between models, only few cover the issue of how those may support developers when manually comparing software models. Especially in the field of model-driven software development several authors (cf. [1, 4, 7, 30]) discuss the need for suitable visual representations of similarities or differences and the evaluation of such visualizations. Also, there seems to be a trend towards adaptable comparison metrics (cf. [2, 16]).

When designing interactive software it is of particular importance to project the work habits of potential users to the graphical user interface and the corresponding interactions. In order to learn more about practices and strategies for model comparison, in a recent study [21] we observed humans comparing UML class diagrams in the context of model merging. In summary, we found the entire process of comparing (and merging) these models to be iterative and exploratory. Our participants were often adjusting or alternating their comparison strategies in order to find different types of similarities, which could, for instance, help them to verify a previous observation. Based on the results of our study, we formulated design guidelines for tools that are intended to support users when comparing UML class diagrams with respect to a subsequent merging.

In this paper we present a tool that was informed by these guidelines. Thus, we formulate the following contributions:

- We provide a tool that follows a visual dataflow programming approach in order to allow users to visually configure and combine model comparison strategies. In addition, users may explore the results of their comparison strategies by using different visualizations.
- By conducting a qualitative user study we were able to find out if and how our tool facilitates iterative exploration of similarities and differences between two UML class diagrams and to assess whether it conforms to the guidelines from our previous work.

In the subsequent section we summarize our previous study, where humans compared UML class diagrams without tool-

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642984.

Table 1: Guidelines for tool design in the context of model comparison.

G1	**Support individual workflow:** In [21] we found that there have been many individual differences in the way humans compare UML class diagrams. Therefore, a tool that allows users to interactively compare two models must not restrict users to a certain workflow. In particular, it should not only support them when comparing models, it might also enable to explore individual ideas by, for instance, adapting available comparison metrics.
G2	**Allow model extensions:** In the first place, extending models becomes increasingly important to produce valid results during merging (e.g., resolving inheritance cycles). However, changing the layout of a model may also be considered as an extension and, moreover, can support the comparison process (e.g., moving similar model elements next to each other). Thus, a tool should at least allow to modify the layout of the compared models or even provide a model editor that temporarily accepts invalid models.
G3	**Support annotations:** During our study we discovered that humans may resort to a variety of different visualizations and annotation techniques, which should be preserved by a tool for model comparison. For instance, they used color coding to distinguish between several matches, but also to mark different types of similarities. Furthermore, a tool should allow to write additional comments such that users are able to capture ideas (e.g., hints on how they found a certain similarity) for later use.
G4	**Support grouping:** According to [21] grouping is an adequate means for different reasons. For instance, grouping may reduce the complexity of UML class diagrams and enables users to work more efficiently [34]. It also allows humans to find model elements that share a certain property (e.g., have been matched by the same metric). Thus, grouping does not necessarily mean that all elements a rendered in a single box. In a broader sense, elements that share the same color may also be seen as a group.
G5	**Raise awareness:** Especially when humans tended to focus on one of the models, they sometimes overlooked or ignored important differences. Therefore, in order to raise their awareness for possible alternatives, a tool should not only allow users to access the best matches, but also those of lower similarity.
G6	**Provide algorithmic support:** Our participants often applied straightforward comparison strategies (e.g., name or layout matching) to identify similarities between the UML class diagrams. Thus, a repertory of comprehensible comparison metrics could already help to simplify the problem. Also, we found that humans tend to explore and combine such simple metrics; often in an iterative process from a coarse- (overview of all similarities) to a fine-grained level (differences between a particular match).
G7	**Help to keep track:** During our study the participants used various gestures and annotations to keep track of the current state of their analysis. Although they often followed a certain strategy, this was not always a strict process. Thus, a tool should, besides allowing users to browse an interaction history, also enable them to interrupt their workflow and explore alternative ways for comparing both models.

support, and introduce the resulting guidelines. In Section 3 we describe our prototype implementation in detail and show how users can create and execute comparison strategies. Section 4 analyzes how the participants of our study perceived and applied our tool while Section 5 summarizes important related work. Section 6 concludes this paper.

2. REQUIREMENTS

When developing interactive software systems it is crucial to involve potential users to gather requirements for the design of such systems. Common strategies for requirements elicitation are interviews and questionnaires, brainstorming sessions, use cases and role playing, or user observation.

To find requirements for an interactive tool that allows to compare models and especially UML class diagrams for a subsequent merging, we resorted to the last method—user observation. To this end, we explored how humans accomplish such a task without any tool-support by conducting a Think Aloud study [21] following the approach of Boren and Ramey [6]. In particular, we recruited 13 participants and asked them to manually compare and merge two pairs of printed UML class diagrams by using only colored pens and a sheet of paper. As the participants verbalized their thoughts, we could videotape each session for later analysis.

In a subsequent step, the Grounded Theory method, as described by Strauss and Corbin [33], allowed us to systematically analyze the videotaped experiments in an iterative procedure. The result of this analysis comprises a network of six top level categories and about 75 subcategories, their descriptions, and dependencies among each other. Details on this network can be retrieved from [21]. In general, it reflects both the behavior of the participants as well as the strategies they developed when manually comparing and merging UML class diagrams (e.g., dividing a problem into several subproblems or handling conflicts). Although the entire information captured by this network provides a detailed source for user-defined comparison metrics, visualizations, or interactions, which might be included into a prototype implementation, these details are difficult to communicate in the first place. Thus, in order to provide important ideas and observations from our study, in [21] we formulated seven guidelines for tool design. However, as these are related to observations described in our previous work, one may not entirely understand them without this information. To overcome this problem, we rephrased the guidelines in the context of model comparison such that readers are able to understand them without knowing our previous work in detail. Table 1 presents these adapted guidelines. Although they provide important insights on the problem of manual model comparison, not every single design decision can be motivated by them. In other words, when developing our tool we also included results from the more fine-grained network of categories and their descriptions.

3. TIGAM

Based on the guidelines presented above, we were able to develop a first prototype, which is introduced in this section in detail. TIGAM—a Tool for Interactive Graph Analysis and Matching—allows to iteratively discover similarities and differences between two graphs by applying user-defined comparison strategies and related visualizations. Moreover, it supports users when merging both graphs into a single one. In this paper, however, we focus on the comparison of two graphs with user-defined strategies and visualizations, which in the following are referred to as *analysis programs*.

Besides UML class diagrams our tool also allows to import other types of graphs like mindmaps or social networks. For the sake of simplicity, we use the term *model* for the re-

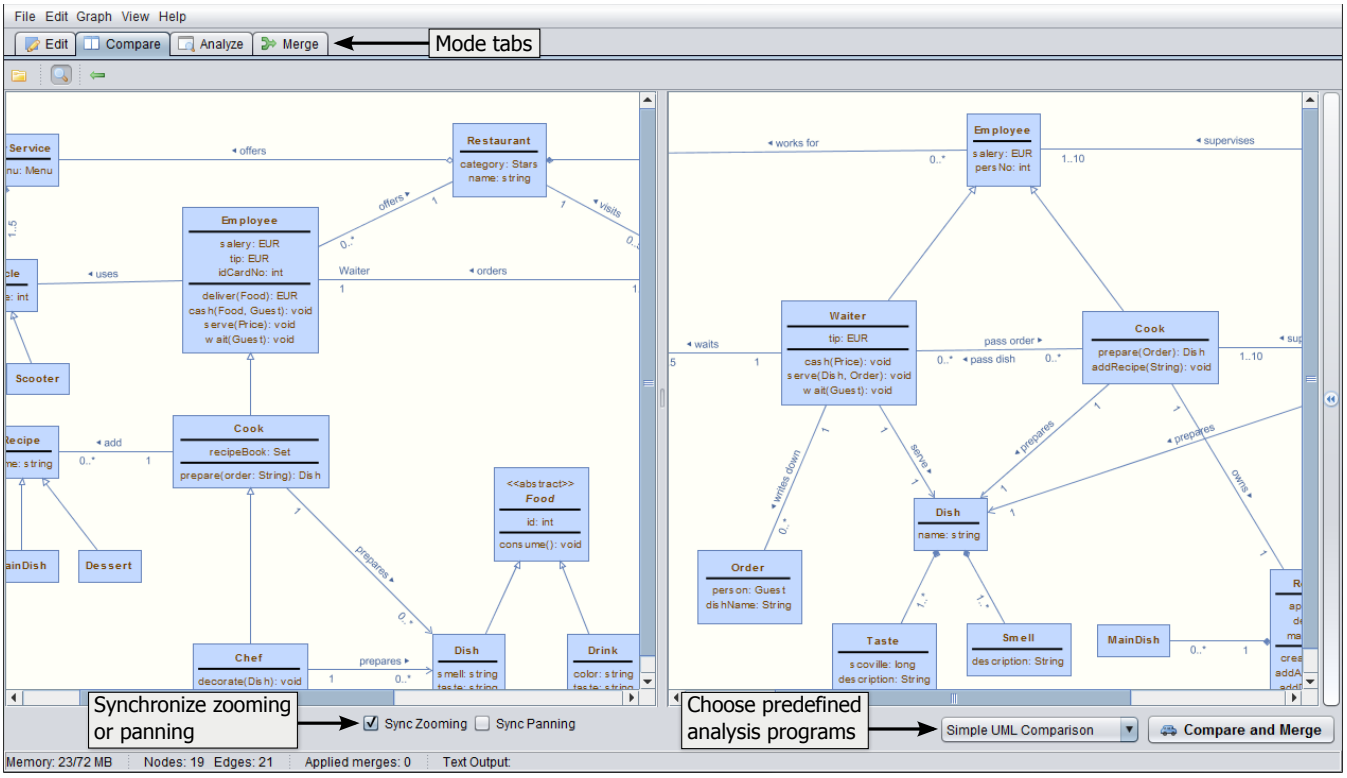

Figure 1: The Compare Mode allows to load two source models, which can be compared with a predefined analysis programs. Both models are read-only such that users may always view their initial layout.

mainder of the paper. Also, a *model element* may be either a node or an edge. Finally, we denote the models that are supposed to be compared as *source models* and the result of an analysis as *result model*.

In [21] we observed that the process of manually comparing and merging two models involves several different tasks like getting an overview and navigating through the models, comparing them by developing adapted comparison strategies, and finally merging them into a single model while solving possible conflicts. In TIGAM we support these top level tasks by introducing different modes that are accessible via the tabs at the top of the main component (cf. Fig. 1). We distinguish four modes, which are briefly described below:

Edit Mode: In this mode a single model can be created or modified. Basically, it resembles a full featured graph editor due to the underlying JGraphX [14] library. In particular, it allows to add or remove nodes and edges, alternate their appearances, apply layout algorithms to automatically place nodes and edges of the model on the screen, and also provides an undo history. We extended the JGraphX library such that users are able to create UML class diagrams. The finished model can be compared with another model in a subsequent step.

Compare Mode: As shown in Figure 1, this mode allows to load two source models in a side-by-side view in order to compare them. To this end, both subviews may be panned and zoomed individually or even synchronized. However, it is not possible to modify the models in this mode because in [21] we observed that it can be important to examine the content and the layout of the initial source models which is why both of them should stay untouched. The Compare Mode also allows to choose from a list of predefined or previously saved analysis programs that may be used for a quick comparison. A simple description of the chosen program can be obtained by opening the panel at the right.

Analyze Mode: This mode allows to develop custom comparison strategies and their visualizations. In order to facilitate the creation of such a strategy, we chose a visual programming approach, in particular, this mode is based on the idea of dataflow programming [15, 28]. Users may choose from a variety of high level programming blocks that enable them to compare the elements (or an arbitrary subset) of both models with each other and visualize the computed result. Figure 2 shows an example of such an analysis program. A detailed description of this program is given in Section 3.1.

Merge Mode: As depicted in Figure 3, this mode shows the results of an analysis program. In particular, when a valid program is executed, copies of both source models are transfered to the Merge Mode and placed next to each other on a single canvas. This allows users to rearrange nodes and edges freely such that the layouts of both models may also intersect and are not restricted to a certain area. All visualizations included in such a program are displayed (some of them only on demand) and may help users to explore similarities and differences of the compared models. Moreover, this mode allows to combine both source models into a single one. Therefore, it provides semi-automatic features

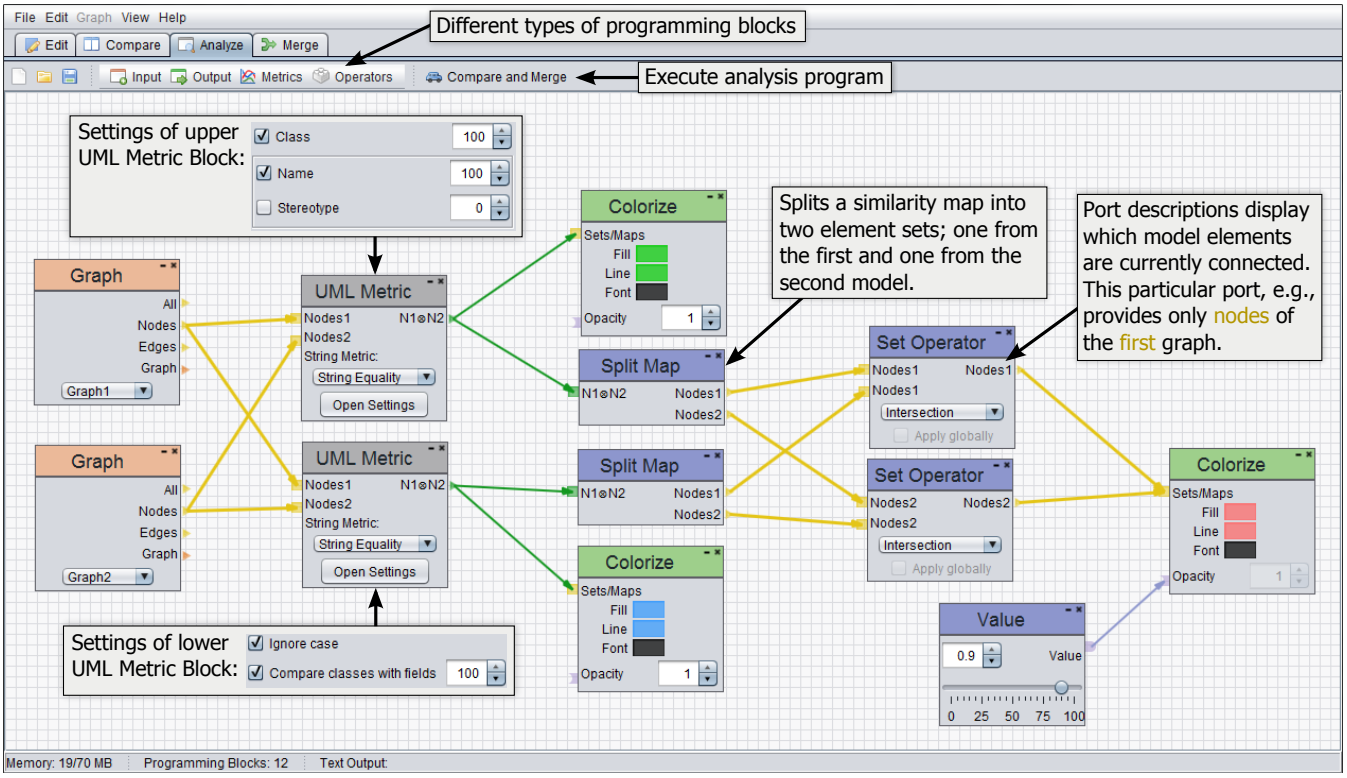

Figure 2: The Analyze Mode allows to develop custom comparison strategies and visualizations by combining different visual programming blocks.

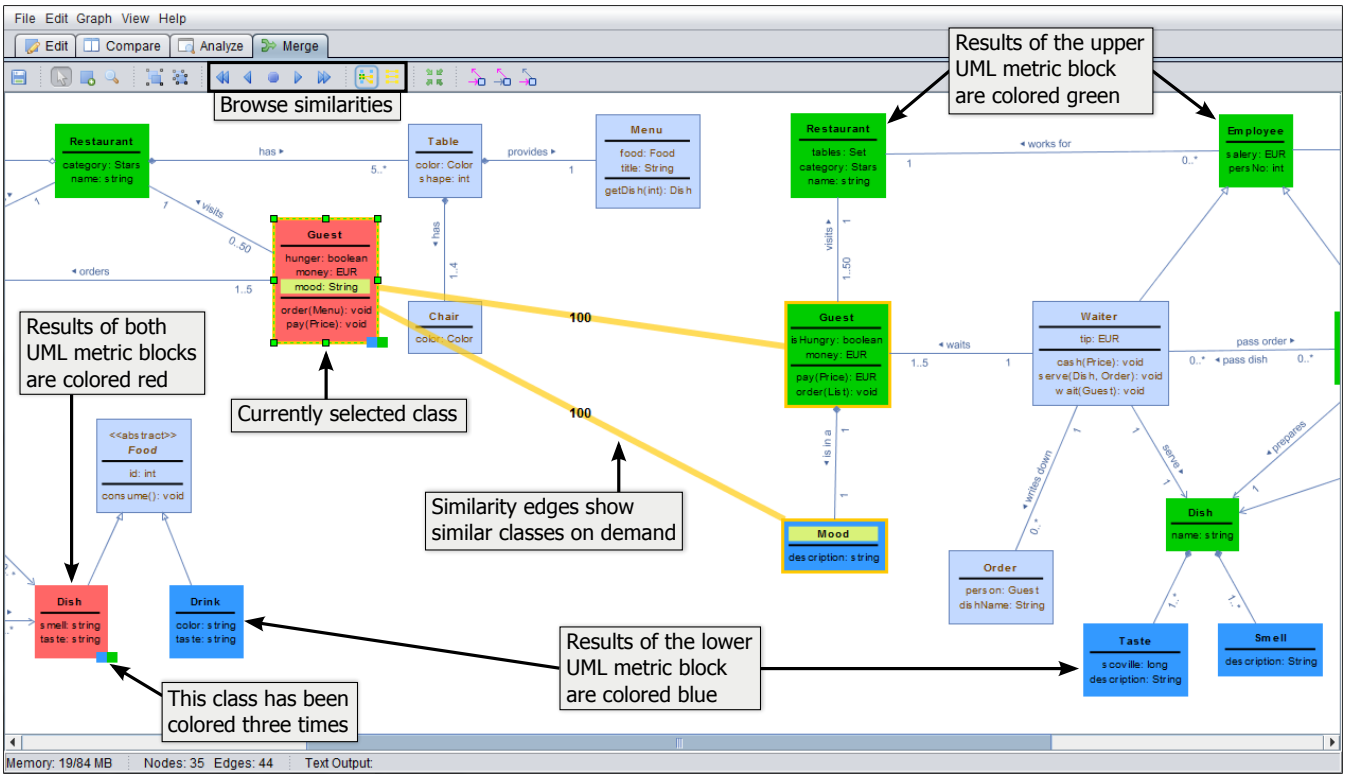

Figure 3: The Merge Mode displays the results of a custom comparison strategy, allows to browse the discovered similarities, and enables users to merge both models into a single one.

to merge overlapping parts of both models and to resolve common conflicts that might occur during that process. For more complex merge operations or extensions to the result model, users may resort to common graph editor features as described earlier.

In the following we focus on the last two modes as these contain the most interesting and novel features of our tool. To this end, we assume that two source models have already been opened like shown in Figure 1. As a next step, an appropriate comparison metric is required. This is where the Analyze Mode comes into play.

3.1 Creating Analysis Programs

The Analyze Mode allows to create custom analysis programs. Therefore, users may switch to this mode, which initially displays an empty canvas, and create a new program from scratch. As an alternative, they can also load an existing analysis program, use it as a starting point or even integrate it into another one. In any case, users can access a repertory of visual programming blocks through the menu beneath the tab bar and add them to their program. A single programming block usually consists of a short name, a number of input and output ports, which allow to connect blocks to each other, and, depending on its purpose, a list of settings that may be used to adjust the outcome of a block. Several types of blocks are available which are listed in Table 2 and introduced below.

Directed links denote a dataflow from a source to a target block. In general, we distinguish two types of data that flow along these links: model-based data and configuration data. The first type comprises element sets and similarity maps. Element sets, on the one hand, flow along thick yellow links (cf. Fig. 2) and may contain any combination of nodes and edges from a source model. Similarity maps, on the other hand, flow along thick green links and store discovered similarities between two element sets. In particular, for each element from one set, which is stored in a hash map, a separate priority queue collects matches from the other set ordered by their similarity. The latter type, configuration data, is used to modify the settings of a particular block and flows along thin blue or purple links like depicted for the *Value* block in Figure 2. Cycles in the data flow are not allowed and thus rejected by our tool.

Table 2: Available visual programming blocks

Category	Blocks
Input	Graph, Normalized Value
Metric	String Metric, UML Metric, Node Degree, Similarity Coverage
Operator	Set Operator, Split Set, Find Neighbors, Split Map, Merge Maps, Filter Map, String Normalizer, String Tokenizer
Output	Colorize, Hide, Colorize Map, Similarity Edges, Graph Layout, Comparison Layout, Text Output

The visual programming blocks included in our tool are organized into four categories: *Input*, *Metrics*, *Operators*, and *Output*. Due to space limitations, we cannot provide particular examples for all blocks. However, Table 2 gives an overview of all currently available blocks.

Input blocks, which are also called sources in visual programming, are supposed to provide certain data from the underlying data model. With *Graph* blocks, for instance, users may access different aspects of a chosen source model. In Figure 2 the orange *Graph* blocks and their connected *"Nodes"* output ports indicate that the user decided to consider only nodes for her custom analysis program.

Metric blocks: One can also see that both *Graph* blocks are connected to gray *UML Metric* blocks, which allow to compare model elements from the provided element sets. Furthermore, the settings of each *UML Metric* block may be adjusted such that different aspects of the source models can be observed (cf. Fig. 2). TIGAM also provides other metric blocks, e.g., for label-based comparison of other graph types like mindmaps or a simple structural analysis. All of these metric blocks have in common that they accept two element sets and produce a similarity map that captures all discovered similarities between the given sets. Individual settings allow users to find different types of similarities and may help them to make a decision on how to combine similar parts of both source models (cf. [21]).

Operator blocks: A variety of different operator blocks allow to split, combine, filter, or preprocess arbitrary element sets. In Figure 2, for example, the resulting similarity maps of both metric blocks have been split such that the matched nodes from each source model are separately available. In a next step, the computed node sets are pairwise intersected by using two *Set Operator* blocks in order to get only those nodes that where found by both *UML Metric* blocks. Other blocks, for instance, enable to combine similarity maps, inspect the neighborhood of nodes, or allow to preprocess labels by removing blanks or special characters.

Output blocks: In order to view the results of custom comparison strategies in Merge Mode, users have to include visualization blocks into their analysis program. Figure 2 shows the usage of so called *Colorize* blocks, which allow to style incoming model elements by changing their colors or adjusting their opacity. Besides element sets, these blocks can also handle similarity maps. In this case, all elements contained in a given similarity map are colored. Although this visualization seems to be rather simple, it may be used in several situations. For instance, to color both models differently such that they can be told apart during a subsequent analysis. Or, like in the program depicted in Figure 2, to highlight several types of similarity with separate colors. The result of this program is shown in Figure 3. In general, visualization blocks are often useful to get an overview of which model elements have been found to be similar by a certain metric block or still remain after a certain filtering. However, such a visualization does not tell us which elements in the first model correspond to one or more elements in the second one or vice versa. To this end, we included two additional blocks that allow to visualize matched elements. While the *Colorize Map* block assigns equal colors to matched elements of a connected similarity map, the *Similarity Edges* block enables to display additional lines between similar model elements on demand such that users may browse the discovered matches one by one. The latter block also allows to select an arbitrary number of elements in Merge Mode and examine their matches like shown in Figure 3 for the *Guest* class on the left side.

Finally, we also want to explore whether certain layouts facilitate the comparison of two models. To this end, users may include *Graph Layout* blocks into their analysis program. For instance, we implemented a layout that bene-

fits the comparison of mindmaps, which in general follow a tree-like structure, by facing their leave nodes to each other. In [20] we used a similar approach to compare directory structures. Furthermore, it could also be interesting to investigate whether it is suitable to contrast class or even package hierarchies in UML class diagrams.

3.2 Executing Analysis Programs

In [21] we observed that two UML class diagrams are often compared iteratively, i.e., new insights about the similarity of two diagrams are collected step-by-step. Such an iterative analysis is also supported by TIGAM as it allows users to modify and re-execute their analysis programs at any time.

After the execution of an analysis program, the Merge Mode displays those visualizations that have been integrated into this program. For instance, nodes or edges contained in a set that has been plugged into a *Colorize* block are rendered in the chosen color, while the remaining model elements keep their default style (cf. Fig 3).

If two or more colors have been assigned to an element, it primarily shows that color which has been applied last. In other words, the visualization block with the highest topological order provides the main color for a node. Here, we simply assume that users consider this visualization block the most important one. The remaining colors, however, are not discarded, they are displayed in a small overlay at the bottom right of a model element. Users may click it in order to swap the main color of a node.

In addition, users are able to browse and examine the discovered matches one by one via the single arrow buttons beneath the tab bar or the assigned keyboard shortcuts. In order to highlight the currently selected match, TIGAM draws temporary lines between the particular elements, which we call *similarity edges*. Figure 3 includes two of those edges. There, one can also see that their labels show the similarity that has been computed by a particular metric block. If more than a single metric block is used in an analysis program, users may directly switch between their results (similarity maps) by clicking the double arrow buttons. Moreover, when users browse corresponding UML classes, matched fields and methods are also highlighted in order to indicate which parts of the classes are similar.

In addition to browsing the discovered similarities one by one, users may also view all of them at once, or restrict the displayed similarities to the currently selected elements. This way, users can focus on a certain part of a model.

3.3 Example

The analysis program shown in Figure 2 was extracted from a program created by a participant of our user study. In general, it was designed to explore different types of similarities between the source models and was developed in an iterative manner. First, a program using only a single *UML Metric* has been created and executed. Then, an additional metric block with different settings was added and a second execution was triggered in order to uncover new similarities. Finally, the user wanted to observe which matches have been detected by both metrics as this may suggest more reliable candidates, added a few more programming blocks and executed the analysis program a third time.

In particular, the final program uses two separate *Graph* blocks to access the nodes of each particular source model. These node sets are then connected with two *UML Metric* blocks such that each metric block compares the nodes of the first model with those of the second one. By configuring the *UML Metric* blocks individually (cf. Fig. 2), different similarities between both models can be detected. The first metric block only searches for nodes that share the exact same class name. The second one allows to find fields that have been named like classes, i.e., for two classes the metric compares the class name of the first one to all field names of the second and vice versa. In addition, the *Ignore Case* option produces better results as classes (at least in Java) typically begin with a capital letter.

The similarity maps of both metric blocks are then plugged into separate *Colorize* blocks such that the found matches can be distinguish in Merge Mode. Furthermore, both maps are also split into two sets of nodes, one from each model. While the upper set contains those nodes that only belong to the first model and have been matched by the particular *UML Metric* block, the lower set only includes matched nodes from the second model. This way, one can create subsets which may be analyzed in subsequent steps. In Figure 2 this is done by using intersections in order to get only such nodes that where found by both *UML Metric* blocks. To this end, the resulting sets are intersected by connecting them with two *Set Operator* blocks, one for each model. In order to distinguish nodes that are produced by this intersection, the gained sets are plugged into a third *Colorize* block.

Figure 3 depicts the visualized similarities after executing the last iteration of this analysis program. There, nodes with a green color share the same class name. Blue nodes, on the other hand, show a match between a class name and a field. Finally, nodes that have been detected by both metrics are colored red. The red *Guest* class from the left diagram, for instance, was matched with two classes of the right diagram. In order to access this information the user selected the red class and demanded to display all similarity edges. The upper edge shows that there exists a class with the name *Guest* in the right diagram. The lower one indicates that a field named *mood* was matched with the *Mood* class of the right diagram. As the *Ignore case* checkbox was selected for the metric block (cf. Fig. 2), even this edge depicts a similarity of 100%. In addition, the field itself and the class name are highlighted with the same color such that the user is directly able to recall the type of the similarity.

Although this example shows how to distinguish different types of similarities, out tool is not limited to this visualization. For instance, it is also possible to use individual colors for each particular match (cf. Sec. 3.1).

4. EVALUATION

In order to find out whether our prototype implementation is suitable for an iterative analysis and comparison of two source models and whether it complies with the guidelines presented in Section 2, we conducted a qualitative user study. Also, we wanted to observe how people work with TIGAM, i.e., which features they use, which workflow they follow, and where they could imagine improvements and extensions to our tool.

4.1 Experimental Design

For our study we chose the hallway testing method, where you randomly invite four to six non-expert participants in order to find usability flaws. Nielson and Landauer [24] showed that this number of subjects is sufficient to discover most of

these flaws. Thus, we asked two students (S), two academic researchers (R), and two software developers (D) to participate in our study. Detailed information about our subjects is shown in Table 3 including their general experience and the time that they have spent working with our tool. We tested the version of TIGAM that is described in Section 3 on a Windows 7 computer with a 23 inch widescreen monitor and full HD resolution. Besides audio and screen capturing, we also logged all user interactions with our prototype.

Table 3: Participants' occupation (see above), experience (1=none to 5=expert), and time spent with our tool for briefing, training, and free exploration.

		P1	P2	P3	P4	P5	P6	Avg
Occupation		S	S	R	R	D	D	
Experience	Object-orientation	3	4	5	4	5	4	4.17
	UML modeling	2	3	4	3	3	3	3
	Textual diff tools	3	3	5	4	4	4	3.83
	Graph or tree comp.	3	2	5	2	4	1	2.83
	TIGAM	3	1	2	2	2	1	1.83
Time spent (in min)		74	72	79	53	59	78	67
	Briefing	9	14	9	8	9	11	10
	Training	28	32	32	25	29	37	30
	Free exploration	37	26	38	20	21	30	28

After a short briefing, where we explained the general procedure of our experiment and introduced basic features of our tool, the training phase began. There, the participants were asked to solve six well-defined analysis tasks, in which they learned common controls of TIGAM and were introduced to the most important visual programming blocks. Due to time constraints and as we wanted to allow the participants to explore the possibilities of our tool, we did not include all programming blocks into this training phase. We also decided to use non-UML models during the first phase of the experiment, which represented a small portion of two partly similar social networks. This way, the participants could rather focus on their analysis programs than on the complexity of the graphs. Moreover, the programs created during the training phase are not directly applicable to UML class diagrams such that the participants were required to develop their own analysis programs.

When all six tasks had been solved, we moved to the second phase where our participants got a short introduction to the UML-specific features of our tool. Then, we provided two UML class diagrams (excerpts of them are shown in Figure 1 and 3) together with the task to implement custom comparison strategies and visualizations that would allow them to merge both diagrams in a later step. Otherwise, our participants were free to explore the tool and combine visual programming blocks to obtain information on the similarity of the provided UML class diagrams.

After each experiment we asked our participants to answer a questionnaire, which was designed to assess the overall usability of our tool, to discover missing features and opportunities for improvement, and to evaluate whether and how our tool conforms to the guidelines presented in Section 2.

4.2 Findings

When analyzing the results of our study, we had to handle three different types of data sources: screen captures along with audio recordings, logged interactions with our prototype, and the post-study questionnaire. As the first two data sources are strong interrelated, we discuss both of them in the following subsection. The results of the questionnaire are described afterwards.

4.2.1 *Interactions and Strategies*

By analyzing the screen captures and audio recordings as well as the logged interactions with our tool, we wanted to uncover how users work with our prototype, which strategies they follow, and which features they use and appreciate. To this end, we watched all screen captures and transcribed important activities and statements to text files along with their individual timestamps. These text files allowed us to quickly browse the according videos in case we wanted to inspect a certain activity in detail. Moreover, we used the recorded log files to generate an overview of the interactions with TIGAM for each particular experiment. These visualizations are shown in Figure 4 and allowed us to search for interesting patterns on a higher level of abstraction. In a next step, we went back to the screen captures in order to inspect these particular patterns in detail.

Before generating the log file visualizations shown in Figure 4, we classified all possible interactions into five categories. Next, we provided these categories to our data analysis tool, which divided each experiment into time spans of 30 seconds[1] and counted the number of interactions for each category during a given time span. While blue bars show the amount of modifications to an analysis program, red ones occur every time a participant executed it. The green category comprises interactions that helped our participants to compare nodes like browsing the discovered matches or displaying similarity edges for a certain selection of nodes. Yellow bars show those situations in which our participants modified the result model (e.g., when they changed its layout or even applied a merging operation). Finally, the navigation category includes selection as well as zooming and panning interactions. For referencing purposes, we numbered the time spans and assigned a letter to each category such that, e.g., P2.C4 refers to the second participant and there to the fourth time span of the comparison category while P2.4 and P2.E refer to the entire fourth time span and the entire execution category, respectively.

In general, Figure 4 shows that, although all participants got the same briefing, solved the same tasks during the training phase, and were asked to compare the same UML class diagrams, they developed individual comparison strategies and freely explored the possibilities of our tool. For instance, finding an initial metric was accomplished differently. While some participants customized their metric already before the first execution of their analysis program (e.g., P1.P0 to P1.P7), others simply kept the default settings (e.g., P4.P0 to P4.P1) assuming that these have been chosen well. Another example is the frequency of the program executions. Although all participants compared both UML class diagrams in a iterative process, which can be seen in Figure 4 by alternating blue and red bars, some of them executed their program less frequently (P2.E, P3.E, or P4.E). Sometimes due to a more detailed inspection of the obtained results (e.g., P3.C14 to P3.C20) or even initial merging operations (e.g., P4.G5 to P4.G17). Others like P2.P12 to P2.P23 spent their time on creating a more complex analysis program.

[1]We experimented with different time spans, but found that a length of 30 sec. produces the most comprehensible results.

Nevertheless, we could also identify similarities between the workflows of our participants. For all of them, we observed the Visual Information Seeking Mantra introduced by Shneiderman [32] when examining the result model for similarities and differences: Overview first, zoom and filter, then details-on-demand.

Overview: After executing an analysis program, our participants first examined the result model from a zoomed out perspective. Through different color codings they were able to distinguish multiple metrics. In addition, some participants quickly browsed the discovered matches (e.g., Fig. 4, P1.C47) or displayed similarity edges once in order to get a first idea of how both models are related to each other.

Zoom and Filter: In a subsequent step, our participants focused on a certain part of the model by zooming and panning the viewport or displaying particular similarity edges to view only a subset of the discovered matches. These interactions were often closely related to each other which can be seen in Figure 4 where green and gray bars have a similar distribution. Three of the participants also used *Hide* blocks or the opacity settings of *Colorize* blocks in order to temporarily filter from uninteresting classes. For instance, classes, for which an exact match was found, seemed to be less interesting than those that are only similar but not equal.

Details-on-Demand: Finally, the participants investigated interesting classes in detail, for instance, by exploring alternative matches, comparing their fields and methods, or inspecting the similarity value of a specific similarity edge. During this process some participants also found weak matches or mismatched classes that helped them to refine or improve on their comparison strategy. Also, one participant tended to spatially separate matched classes from their neighbors before inspecting them in detail.

4.2.2 Questionnaire

After having worked with our tool for about an hour, each participant was asked to answer a questionnaire, which was designed to gain insights on the usability of TIGAM. Our main goal was to find out whether our prototype conforms to the guidelines shown in Table 1. As these provide a high-level view on the requirements of our tool, they were suitable to derive questions for our survey. In particular, we asked each participant if and why our tool corresponds to these guidelines. To this end, we formulated statements that should be rated on a 5-point Likert scale (1=strongly disagree to 5=strongly agree). The statement for G1, e.g., reads as follows: *"I have not been restricted in my individual workflow"*. In addition, we allowed the participants to comment on these statements. In case of G1, they should explain where they felt restricted in their workflow.

Table 4 provides the scores for all guidelines. One can quickly see that all of their means are above average, which suggests that our tool generally conforms to the guidelines presented in Table 1. However, there are some guidelines (namely G1, G2, and G7) that got lower scores than others. By browsing the comments related to the guidelines as well as the screen captures of each experiment, we were able to find explanations for these scores.

In particular, Participants 1 and 2 stated with respect to G1 that a certain training with our tool is required in order to manage all of its features and that new users can be slowed down in their individual workflow. Participant 1 added that this might also be due to his lack of experience in comparing

Table 4: Participants' opinion on the conformance of our prototype to the guidelines from Table 1.

	P1	P2	P3	P4	P5	P6	Avg
G1 Support individual workflow	3	2	3	4	4	4	3.33
G2 Allow model extensions	2	4	3	3	5	3	3.33
G3 Support annotations	4	4	4	5	5	4	4.33
G4 Support grouping	4	5	3	4	4	5	4.17
G5 Raise awareness	4	4	5	4	4	2	3.83
G6 Provide algorithmic support	4	5	3	5	5	4	4.33
G7 Help to keep track	2	4	2	4	5	4	3.5

UML class diagrams. As our tool also allows to merge two models, Participant 3 would have normally integrated such activities into his workflow.

As extensions to the UML class diagrams were not explicitly mentioned in our task description, our participants did not have the intention to improve the result model. This is why most scores for G2 are around average. Solely Participant 5 explicitly investigated at the end of his session how UML class diagrams can be modified and extended in Merge Mode (cf. Fig. 4, P5.G36 to P5.G38) and found these features to be of decent quality.

Scores for the seventh guideline are twofold: While some participants reported that they lost the overview of the considered classes at times, others actively tried to keep the Merge Mode organized. This was done by either using additional *Hide* blocks or lowering the opacity of a specific *Colorize* block in order to filter from already observed classes.

In addition to the questions about the guidelines, we also asked our participants to assess whether a stepwise exploration facilitated the process of comparing the given UML class diagrams. Four out of six participants answered this question and stated that an iterative or stepwise exploration is a crucial technique when comparing UML class diagrams. It allowed them to begin on a coarse-grained level using simple comparison strategies and refine them iteratively. Also, one participant stated that a stepwise exploration helped him to keep track of those classes he had already visited. Moreover, we wanted to know whether our participants could imagine further scenarios in which TIGAM may be applied. The participants' suggestions show that the application of TIGAM is not limited to UML class diagrams. For instance, due to his background in code clone detection, Participant 3 found that our tool could be used to find and especially present code clones to developers. Participant 4, on the other hand, could imagine to use an adapted version of our tool in his every day workflow to compare XML files or hierarchical directory structures. Other ideas were to compare business process models or to detect plagiarism in student exercises.

4.2.3 Ideas for Improvement

During the course of our user study as well as by evaluating our questionnaire, we found several interesting suggestions to improve TIGAM.

In Analyze Mode, for instance, our tool could allow users to define and reuse their own programming blocks by grouping existing ones. Also, Participant 5 suggested to introduce switches such that different subprograms can be turned on or off. Although most of our participants have been satisfied with the metrics provided by our tool (cf. Table 4, G6), we

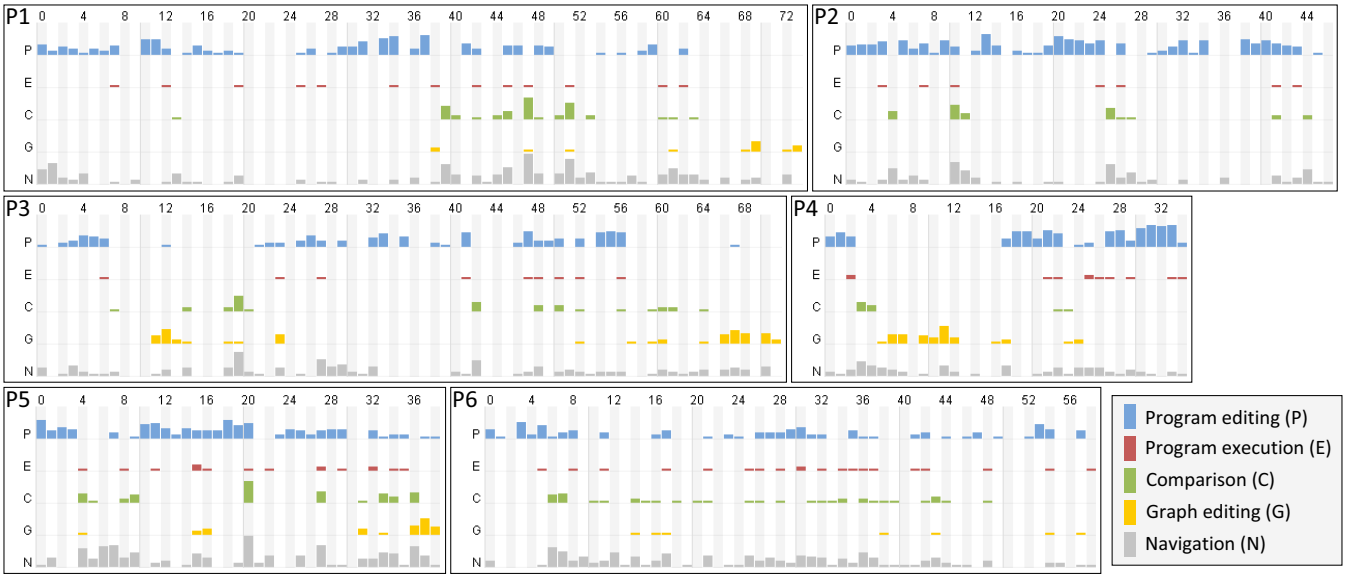

Figure 4: Visualization of logged interactions with our prototype for each participant (P1-P6).

plan to add more sophisticated ones, which allow to discover elaborate structural or semantic matches. However, a challenge is to find intuitive metrics that can be parameterized or customized to be applicable in different scenarios while they are still comprehensible.

In Merge Mode, Participant 6 reported that it sometimes became tedious to compare such nodes in detail that are spatially distributed as a lot of panning and zooming might be required. Thus, we plan to extend the similarity browsing feature such that users may instruct TIGAM to pan and zoom to the next corresponding node automatically. Furthermore, Participant 3 suggested that, in Merge Mode, users should be able to manually define nodes that are treated as matches or that will not be matched independently of which result a particular metric computes. Such manually defined data could also be made available in Analyze Mode via a specific input block.

4.2.4 *Validity and Limitations*

The qualitative nature of our evaluation allowed us to explore which strategies users develop when working with our prototype. However, due to the low number of participants, we cannot draw any quantifiable conclusions from our experiments. Moreover, the interpretation of the observed user behavior could be biased by the authors as well as the results of our previous work. Also, people that work in the field of model-driven software development could have perceived our tool differently. Nevertheless, the exploratory character of our evaluation allowed us to investigate different aspects of computer-supported model comparison and provides preliminary evidence of the usability of our tool.

5. RELATED WORK

In general, works related to our approach are originated in two areas of research: graph comparison and visual programming. In the following we present a brief overview of these areas. However, we do not attempt to cover all existing works, but rather want to show a broad field of application.

5.1 Graph Comparison

The comparison of graphs or graph-based models is of importance for different application scenarios. In information visualization, for example, several approaches have been proposed [10]. For instance, the evolution of software is visualized by computing differences between consecutive versions of the system or genomic sequences are aligned next to each other. Also, the comparison of social networks seems to be of increasing importance. Users probably want to know which friends they share among several networks in order to store them as a single contact. In [19] Ley describes similar problems on how to find homonyms and synonyms for authors included in the DBLP computer science bibliography. In software engineering and especially in model-driven software development it is of particular importance to find differences between versions of UML or domain-specific models. But, whenever users have to decide or verify whether certain parts of two or more models are similar, an appropriate representation of similarities or differences is required.

Although several approaches for various applications have been proposed, to the best of our knowledge, non of these allows users to iteratively explore the similarities and difference between models and visually configure comparison strategies and the according visualizations. Nevertheless, the proposed approaches may coarsely be categorized into those that explicitly show differences and those that point users to similarities between graphs. To this end, some approaches use additional space, views, or even files to show or store the results of a comparison. Others augment the models themselves with additional information about their similarity. In the latter case, one may also distinguish between so called *unified views* where a precomputed combined graph is used to show differences or similarities and so called *side-by-side views* that display graphs next to each other.

5.1.1 *Visualization of Differences*

Approaches in this category are supposed to find differences between graphs or models. Thus, similarities are not highlighted explicitly. In order to visualize such differences,

for instance, Girschick [9], Selonen und Kettunen [31], or Kelter et al. [17] first compute a difference model based on their individual comparison metrics. Besides equal elements, which are only represented once, it contains all other elements of both source models (unified view). Then, they use colors to highlight those elements that were added to the second, removed from the first, or changed in between both source models. Such approaches are often designed for the application in model-based version control systems. There, developers may derive two variants from a common base model which need to be compared and merged in order to integrate the changes of both variants into the latest version of a model. However, these visualizations often depend on a small number of differences between both variants. In case this number grows, such a visualization might become confusing. This is why users should be able to adjust or limit the visualization to areas that are currently of interest. For instance, Ohst et al. [25] observed how they could define interesting areas of a difference model. Moreover, besides color coding, which is often used to show differences between two models, further graphical elements may provide additional information. Scharf and Zündorf [27] augment differences in a unified view with additional markers that allow to access details of the detected difference and provide a way to interact with the unified model.

Side-by-side views are another approach to compare graphs or models. For example, KIELER [29] displays two UML models next to each other and uses colors to show additions, deletions, or changes. Through sophisticated interaction techniques like automatic panning or zooming, users may browse both models and explore the detected differences. An additional hierarchical representation helps users to get an overview of the displayed results. DARLS [35] follows a hybrid approach to compare not only UML class diagrams, but also other types of graphs. On the one hand, it displays two graphs in a side-by-side view, on the other hand, it augments each graph such that it shows all differences to its counterpart by using semi-transparent nodes and edges. In addition, it uses colors or strikethrough text to highlight differences between fields and methods of UML classes. A timeline visualization above the models allows to browse different versions and open them in the side-by-side view. The approach of Hascoët und Dragicevic [12, 13] renders each graph in a semi-transparent layer such that they may also be laid on top of each other. By applying certain layout algorithms and zooming technique users are able to visually compare the stacked graphs. Finally, Andrews et al. [3] present an approach that integrate three views in order to compare common graphs. Two for displaying the source graphs and the third one to show a combined graph, which is based on the results of a precomputed comparison.

5.1.2 Visualization of Similarities

In contrast, users may also explicitly be made aware of similarities instead of differences. However, only a few tools follow this approach. Rondo [22], e.g., uses colors and additional lines to show similarities between two graphs that were laid out next to each other. Dadgari and Stuerzlinger [8] present a tool that allows users to explore both similarities and differences. To this end, each graph is displayed in a separate layer. Further layers may be created by intersecting two graphs to display only common nodes and edges or differencing them to show nodes or edges that are unique. These kinds of set operations can be applied based on the pure structure of the source graphs or even their layout.

Unified views are also used to find similarities between graphs. Koop et al. [18], e.g., recently presented an approach that compares multiple graphs and computes a so called summary graph. To this end, similar nodes are merged virtually such that users can explore a unified graph. Particular parts of this graph may also be interactively separated in order examine the original structure of the source graphs.

5.2 Visual Dataflow Programming

According to Johnston et al. [15] visual dataflow programming has its origins in the mid 1980s and experienced a growth in the following years. LabVIEW [23] was one of the early adopters of visual dataflow programming in the industry and is still of importance today. It is among others used for instrument control and industrial automation and allows engineers to create applications for these purposes.

In general, visual programming or visual scripting languages are nowadays often applied in the domain of end-user programming in order to abstract from text-based languages. For instance, in 3D software suites and game development visual programming approaches allow users to create custom materials and shaders for three-dimensional objects and facilitate the compositing and post-precessing of rendered images or movies. The Analyze Mode of TIGAM, e.g., was inspired by the visual programming system of Blender [5], an open-source 3D computer graphics software. Also, visual programming has been applied in educational settings in order to facilitate teaching of programming languages to kids and teenagers. For example, the MIT App Inventor [26] has been successfully used in computer science education [11].

6. CONCLUSION

In this paper we presented TIGAM, a tool for interactive graph analysis and matching. The design of this tool was informed by a set of guidelines we derived from previous work and which we put in the context of model comparison. TIGAM allows users to compare models in an iterative procedure by visually configuring and combining custom comparison strategies. To this end, it leverages visual dataflow programming and, additionally, provides different visualizations that may be used to explore the results of a created comparison strategy.

The findings of a qualitative user study provide preliminary evidence that our tool facilitates iterative exploration of similarities and differences between UML class diagrams and, moreover, that its design conforms to the guidelines from our previous work. Furthermore, our evaluation suggests that TIGAM already provides a decent set of programming blocks for model comparison and that users are able to combine these blocks in order to create convenient comparison strategies and appropriate visualizations. Nevertheless, our evaluation also revealed interesting ideas to extend and improve on the current prototype.

7. ACKNOWLEDGMENTS

We want to thank Marco Schuh, Xin Zhou and Marc Gelhausen who implemented parts of the tool. This work was partially supported by the German Research Foundation DFG Grant No. DI 728/12-1.

8. REFERENCES

[1] K. Altmanninger, P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. Why Model Versioning Research is Needed!? An Experience Report. In *Proceedings of the Joint Workshop on Model-Driven Software Evolution (MoDSE) and Model Co-Evolution and Consistency Management (MCCM), Denver, CO, USA, October 4*. Springer, 2009.

[2] K. Altmanninger, G. Kappel, A. Kusel, W. R. Martina, Seidl, W. Schwinger, and M. Wimmer. AMOR - Towards Adaptable Model Versioning. In *Proceedings of the International Workshop on Model Co-Evolution and Consistency Management, in conjunction with Models*, 2008.

[3] K. Andrews, M. Wohlfahrt, and G. Wurzinger. Visual Graph Comparison. In *Proceedings of the International Conference on Information Visualisation, Barcelona, Spain, July 15-17*, pages 62–67. IEEE Computer Society, 2009.

[4] L. Bendix and P. Emanuelsson. Requirements for Practical Model Merge - An Industrial Perspective. In *Proceedings of the Internationl Conference on Model Driven Engineering Languages and Systems, Denver, CO, USA, October 4-9*, pages 167–180. Springer, 2009.

[5] Blender Foundation. Blender - Free and Open 3D Creation Software. http://www.blender.org/. (accessed April 2014).

[6] M. Boren and J. Ramey. Thinking aloud: Reconciling theory and practice. *IEEE Transactions on Professional Communication*, 43(3):261–278, 2000.

[7] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[8] D. Dadgari and W. Stuerzlinger. Novel User Interfaces for Diagram Versioning and Differencing. In *Proceedings of the British Computer Society Conference on Human-Computer Interaction, Dundee, UK, September 6-10*, 2010.

[9] M. Girschick. Difference Detection and Visualization in UML Class Diagrams. Technical report, TU Darmstadt, 2006.

[10] M. Gleicher, D. Albers, R. Walker, I. Jusufi, C. D. Hansen, and J. C. Roberts. Visual comparison for information visualization. *Information Visualization*, 10(4):289–309, Sept. 2011.

[11] J. Gray, H. Abelson, D. Wolber, and M. Friend. Teaching CS Principles with App Inventor. In *Proceedings of the Annual Southeast Regional Conference, Tuscaloosa, AL, USA, March 29-31*, pages 405–406. ACM, 2012.

[12] M. Hascoët and P. Dragicevic. Visual Comparison of Document Collections Using Multi-Layered Graphs. Technical report, Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (LIRMM), AVIZ (INRIA Saclay - Ile de France), 2011.

[13] M. Hascoët and P. Dragicevic. Interactive graph matching and visual comparison of graphs and clustered graphs. In *Proceedings of the International Working Conference on Advanced Visual Interfaces, Capri Island, Naples, Italy, May 22-25*, pages 522–529. ACM, 2012.

[14] JGraph Ltd. JGraphX. https://github.com/jgraph/jgraphx. (accessed April 2014).

[15] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36(1):1–34, Mar. 2004.

[16] T. Kehrer, U. Kelter, P. Pietsch, and M. Schmidt. Adaptability of Model Comparison Tools. In *Proceedings of the International Conference on Automated Software Engineering, Essen, Germany, September 3-7*, pages 306–309. ACM, 2012.

[17] U. Kelter, J. Wehren, and J. Niere. A Generic Difference Algorithm for UML Models. In *Software Engineering, Fachtagung des GI-Fachbereichs Softwaretechnik, Essen, Germany, March 8-11*, pages 105–116, 2005.

[18] D. Koop, J. Freire, and C. T. Silva. Visual Summaries for Graph Collections. In *Proceedings of the Pacific Visualization Symposium, Songdo, South Korea, February 28 - March 2*. IEEE Computer Society, 2012.

[19] M. Ley. DBLP - Some Lessons Learned. In *Proceedings of the International Conference on Very Large Data Bases, Lyon, France, August 24-28*, volume 2, pages 1493–1500. ACM, 2009.

[20] R. Lutz, D. Rausch, F. Beck, and S. Diehl. Get Your Directories Right: From Hierarchy Visualization to Hierarchy Manipulation. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing, Melbourne, Australia, July 28 - August 1*. IEEE Computer Society, 2014.

[21] R. Lutz, D. Würfel, and S. Diehl. How Humans merge UML-Models. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement, Banff, AB, Canada, September 22-23*. IEEE Computer Society, 2011.

[22] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *Proceedings of the International Conference on Management of Data, San Diego, CA, USA, June 9-12*, pages 193–204. ACM, 2003.

[23] National Instruments. LabVIEW System Design Software. http://www.ni.com/labview/. (accessed April 2014).

[24] J. Nielsen and T. K. Landauer. A Mathematical Model of the Finding of Usability Problems. In *Proceedings of the Conference on Human Factors in Computing Systems, Amsterdam, The Netherlands April 24-29*, pages 206–213. ACM, 1993.

[25] D. Ohst, M. Welle, and U. Kelter. Differences between versions of UML diagrams. In *Proceedings of the Symposium on Foundations of Software Engineering, Helsinki, Finland, September 1-5*, pages 227–236. ACM, 2003.

[26] S. C. Pokress and J. J. D. Veiga. MIT App Inventor: Enabling Personal Mobile Computing. In *Proceedings of the Workshop on Programming for Mobile and Touch, Indianapolis, IN, USA, October 27*, 2013.

[27] A. Scharf and A. Zündorf. Difference Visualization for Models (DVM) - Visualizing model changes directly within diagrams. In *Proceedings of the International Fujaba Days, May 11-13, Tartu, Estonia*, 2011.

[28] S. Schiffer. *Visual Programming - Foundations and Applications*. Addison-Wesley-Longman, 1998. (in German).

[29] A. Schipper, H. Fuhrmann, and R. von Hanxleden. Visual Comparison of Graphical Models. In *Proceedings of the International Conference on Engineering of Complex Computer Systems, Potsdam, Germany, June 2-4*, pages 335–340. IEEE Computer Society, 2009.

[30] P. Selonen. A Review of UML Model Comparison Approaches. In *Proceedings of the Nordic Workshop on Model Driven Engineering, Ronneby, Sweden, August 27-29*, pages 37–51, 2007.

[31] P. Selonen and M. Kettunen. Metamodel-Based Inference of Inter-Model Correspondence. In *Proceedings of the European Conference on Software Maintenance and Reengineering, Amsterdam, The Netherlands, March 21-23*, pages 71–80. IEEE Computer Society, 2007.

[32] B. Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the Symposium on Visual Languages, Boulder, CO, USA, September 3-6,*, pages 336–344. IEEE Computer Society, 1996.

[33] A. L. Strauss and J. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, 2008.

[34] S. Yusuf, H. H. Kagdi, and J. I. Maletic. Assessing the Comprehension of UML Class Diagrams via Eye Tracking. In *Proceedings of the International Conference on Program Comprehension, Banff, AB, Canada, June 26-29*, pages 113–122. IEEE Computer Society, 2007.

[35] L. Zaman, A. Kalra, and W. Stuerzlinger. DARLS: Differencing and Merging Diagrams Using Dual View, Animation, Re-Layout, Layers and a Storyboard. In *Extended Abstracts on Human Factors in Computing Systems, Vancouver, BC, Canada, May 07-12*, pages 1657–1662. ACM, 2011.