Recommending Model Transformations via Subgraph Matching

Bachelor Thesis Proposal

1 Introduction and Motivation

Model-driven engineering is increasing in popularity. Unlike most common text based programming languages, most models don't have IDEs that aid the user when editing. In particular most model editors don't include auto-completion or refactoring tools or only with very limited functionality. A reason for this is that it is easier to create IDEs for most common text based programming languages because they follow the same basic principles (e.g. there are a lot of C-like programming languages). Creating IDEs for models is harder because while most of them are simple graphs they all follow different rules and constraints.

Model transformations represent the changes that need to be applied to a model to arrive at a target model. They can be expressed uniformly for different types of models. For example using Henshin rules [1]. This makes it easier to create model transformations automatically. [2] But the user still has to choose the transformation he wants to apply from a potentially very long list and fill out the parameters of the transformation by hand.

It is not always clear to the user what model transformation he wants to apply next or how the end result should look like. In those scenarios an auto-complete focused workflow is easier and faster to follow. To help create such a workflow the model editor needs to generate and find useful model transformation recommendations based on the changes the user has already made to the model.

Because these model transformations are very domain and model dependent it is hard to define rules to generate them on the fly. However if there are already pre generated model transformations it is possible to search for useful model transformations in this list and recommend them to the user. This also allows the recommendation algorithm to be reused for different types of models and makes creating an universal model editor/IDE easier.

This thesis aims to provide a reusable algorithm to generate recommendation from user changes and a list of typical model transformations.

2 Problem Statement

Given a list of typical model transformations (further also called `AbstractEditScript`s) and the user changes to a model (further also called `EditScript`), generate model transformation recommendations. The `AbstractEditScript`s and the `EditScript` are both model transformations represented as graphs. The list of typical model transformations is pre generated and the user changes are provided by e.g. an IDE. See Figure 1.

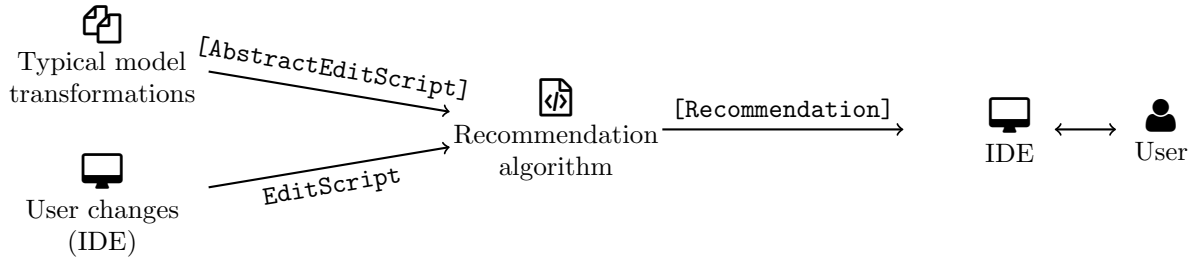

Figure 1: Dataflow of the recommendation algorithm. The edges use haskell type notation. `[X]` means *list of* `X`.

The problem contains the following subproblems:

To generate recommendations we need to (1) find imperfect matchings between the `EditScript` and the `AbstractEditScript`s, (2) convert these matchings to recommendations and (3) rank the recommendations to help the user choose a relevant recommendation.

There are several algorithms that find a matching between the nodes of two graphs [3]–[5]. Most of the algorithms don't guarantee to find a common subgraph. That means they are not applicable to finding model transformation recommendations because the matching can not always be converted to a valid model transformation for the used model. Also none of these algorithms have been tested on model transformations. So it is not clear which of them give good results for model transformations.

The ranking of recommendations can be done with some sort of similarity score between the `EditScript` and `AbstractEditScript`s. The most straight forward similarity score for graphs is graph edit distance. However the problem of calculating the edit distance between two graphs is NP-complete [6] which leads to long run times. This is especially bad because the algorithm should aid the user in real time while he's editing a model. There are also some algorithms which find an approximate edit distance but none of them have been tested on model transformations so it is unknown which of them would give the best results.

For more information about graph matching and graph similarity algorithms see section 4.

3 Purpose of the study

The purpose of this study is to provide an algorithm to find model transformation recommendations based on changes made by an user to a model. Further the algorithm creates a matching between the user changes and the list of typical model transformations so that the recommendation can be applied to the model that is being edited. The recommendations are also ranked by relevance. The algorithm is model and domain independent and utilizes a pre-generated list of model transformations but can potentially be used on arbitrary graphs.

4 Review of the literature & previous work

Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching (by Melnik, Garcia-Molina and Rahm) [7]

Melnik et al. describe an iterative graph matching algorithm. They use graphs representing schemas of SQL databases for testing and explanations. Given two directed labeled graphs G_1 and G_2 the result of

the similarity flooding algorithm is the list of similarity scores between every node in G_1 and every node in G_2. They then describe how to select node pairs from this list to get the most plausible matches. For this they first filter out nonsensical node pairs using domain constraints and then select matches. The selection of node matches is closely related to matching problems in bipartite graphs and they solve this using the stable marriage problem in a polygamous society. Which means no nodes in the chosen node pairs can be matched with another node that has higher relative similarity.

The similarity flooding algorithm works as follows:

1. Find an initial similarity score for all node pairs in $G_1 \times G_2$. Melnik et al. do this using string matching.

2. Create a *pairwise connectivity graph* (PCG) with nodes from $G_1 \times G_2$ (node pairs) and labeled edges between two node pairs if there is an edge with the same label between the nodes in both graphs. I.e. there is an edge with label l pointing from (a_1, a_2) to (b_1, b_2) in the PCG if there is an edge with label l in G_1 pointing from a_1 to b_1 and an edge with label l in G_2 pointing from a_2 to b_2.

3. Attach weights to every edge in the PCG. Melnik et al. distribute the total weight of 1 equally between all outgoing edges of a node. But they mention that the weights can be computed differently. They call the weight of an edge from node x to node y: $w(x, y)$.

4. For every edge in the PCG add an edge in the opposite direction (if it does not exist already) and attach weights with the same method. The resulting graph is called the *propagation graph* and the weights *propagation coefficients*.

5. Let $\sigma^i(x, y)$ be the similarity between node $x \in G_1$ and node $y \in G_2$ in iteration i. With σ^0 set to the initial similarity scores as determined in step 1. In every iteration increment the σ-value using a fixpoint formula. Melnik et al. describe using

$$\sigma^{i+1} = normalize(\sigma^i + \phi(\sigma^i))$$

   where ϕ is the similarity score of the neighboring nodes in the PCG multiplied by their propagation coefficients.

$$\phi(\sigma^i)(x, y) = \sum_{(a_1,l,x)\in A,(b_1,l,y)\in B} \sigma^i(a_1, b_1) \cdot w((a_1, b_1), (x, y)) + \sum_{(x,l,a_2)\in A,(y,l,b_2)\in B} \sigma^i(a_2, b_2) \cdot w((a_2, b_2), (x, y))$$

   The new similarity scores σ^{i+1} are then normalized by dividing by the maximum σ in that iteration. This is denoted by *normalize*.

   Melnik et al. also test other fixpoint formulas and find that

$$\sigma^{i+1} = normalize(\sigma^0 + \sigma^i + \phi(\sigma^0 + \sigma^i))$$

   performs the best.

6. Terminate when we reach a fixpoint (i.e. $\|\Delta(\sigma^n, \sigma^{n-1})\| < \epsilon$) or when a maximum iteration count is reached.

This fixpoint iteration does not necessarily converge therefore we have to terminate the algorithm after a maximum number of iterations. This might give bad or useless results. Melnik et al. also don't talk about how fast the fixpoint iteration converges.

This algorithm also does not guarantee that the given matching is a common subgraph of both input graphs. If the result of the matching is not a common subgraph the resulting model transformation can't be applied to the model if it is even a valid model transformation. Which means the last part of the algorithm (selection of matches from the list of node pairs with their similarity scores) has to be changed to only allow common subgraphs. This might degrade performance significantly because subgraph matching is NP-complete. [6]

The similarity flooding algorithm only solves the matching part of the recommendation algorithm. Even though it calculates node similarities it does not calculate a similarity score between two graphs. Which means we need another algorithm to do the graph similarity calculation.

Graph similarity scoring and matching (by Zager and Verghese) [4]

Zager et al. describe a generalized version of the HITS (*hypertext-induced topic selection*) algorithm by Kleinberg [8]. This is an iterative algorithm that is based on the notion that an element in Graph G_A is similar to an element in Graph G_B if their respective neighborhoods are similar.

This algorithm uses unlabeled directed graphs. It is defined by the following iteration function:

Let $x_{ij}(k)$ be the similarity between node i from graph G_B and node j from graph G_A in iteration k and let E be the set of edges of a graph.

$$\tilde{x}_{ij}(k) = \sum_{r:(r,i)\in E_B, s:(s,j)\in E_A} x_{rs}(k-1) + \sum_{r:(i,r)\in E_B, s:(j,s)\in E_A} x_{rs}(k-1) \tag{1}$$

This calculates the sum of all similarity scores x_{rs} where r is a neighbor of i in graph G_B and s is a neighbor of j in graph G_A from the previous iteration. This is split into two sums because the graphs are directed.

These similarity scores are normalized after every iteration:

$$x_{ij} = \frac{\tilde{x}_{ij}}{\sum_{r,s} \tilde{x}_{rs}^2} \tag{2}$$

Zager et al. extend this to edge similarity. This is not very useful for matching graphs consisting of Henshin Rules. The same Henshin Rule always has the same label and labeled edges. This means calculating the similarity score of the edges is not very useful.

The algorithm describe by Zager et al. always converges. However there is no mention of how fast it converges. They provide performance data that suggests that matching graphs with few edges, many labels and more equally distributed labels is faster. This algorithm also has the same problem as [7] in that it does not guarantee the matching to be a common subgraph. It also does not calculate a similarity score between the graphs.

Reactive Tabu Search for Measuring Graph Similarity (by Sorlin and Solnon) [3]

Subgraph isomorphism, maximum common subgraph and graph edit distance are NP-hard and NP-complete problems [6]. Sorlin et al. outline a greedy algorithm, a tabu search algorithm and a reactive tabu search algorithm to try and solve these problems. The described algorithms are non deterministic.

The greedy algorithm only solves the problems for approx. half of the tested graphs. For most tested graphs the reactive and non reactive tabu search algorithms solve the problems easily. For some hard graphs the algorithms take a long time and don't solve the problems on every try. The reactive tabu search algorithm solves these hard graphs in 79% of the runs.

Sorlin et al. say that these algorithms can compute a similarity measure of two graphs having 100 vertices in a reasonable amount of time. They don't specify further what "reasonable amount of time" means. It might not be reasonable for real time recommendations.

If the algorithms find a solution it is always a common subgraph. The reactive tabu search algorithm has the highest success rate in finding solutions. But it still does not always give a result for hard graphs. This makes the algorithms less useful because you would potentially have to run them several times on every pair of graphs. That further degrades the "reasonable" running time. However this algorithm calculates both a maximum common subgraph and a similarity score.

RASCAL: Calculation of Graph Similarity using Maximum Common Edge Subgraphs (by Raymond, Gardiner and Willett) [5]

Raymond et al. provide an inexact graph matching algorithm called RASCAL (RApid Similarity CALculation). The algorithm is divided into two parts: screening and the actual matching and similarity calculation. The result of the screening is an upper bound on the similarity score. This part of the algorithm is very fast. The screening has a total complexity of $O(n^3)$. It also takes edge labels into account. The goal of the screening is to reduce the number of graphs the actual matching algorithm has to be run. For this they specify a minimum acceptable similarity score and only match graphs that pass the screening with a higher upper bound.

The actual matching algorithm is a branch and bound algorithm and is based on finding maximum cliques. This algorithm is significantly slower than the screening. The running time further degenerates if the graphs are symmetrical. This would be fixable if the graphs are planar. However model transformations are not always planar.

Raymond et al. test their algorithm on chemical databases containing molecules. Their test machine has a 450MHz Intel Celeron and 128 MB of RAM. The running time of their algorithm is relatively fast. However the screening filters out a lot of graphs and all of their graphs are planar so the running time does not degenerate when the graphs are symmetrical. Picking a minimum acceptable similarity score might not be optimal for recommendations. Instead it is probably better to pick the graphs with the highest upper bound after the screening.

Because the amount of data used by Reymond et al. is heavily reduced by the screening, the numbers given can't really be transfered to other sets of graphs. To qualify the running time a bit better I assumed the screening algorithm takes no time and calculated the running time for the matching algorithm using the number of graphs left after the screening. I.e. $\frac{Total\ running\ time}{(No.\ of\ comparisons)\cdot(Percentage\ of\ graphs\ after\ screening)}$. Table 1 shows the approx. running time per comparison in milliseconds. The running time is an overestimation because it still contains the time it takes to screen all graphs and not just the graph which are actually matched. Even though they tested the algorithm on old hardware the running time is totally acceptable for real time recommendations.

The RASCAL algorithm always gives common subgraphs which we need for model transformation recommendations. It also gives both a matching and a similarity score for the graphs. According to the data given by Raymond et al. it is reasonably fast on a relatively old machine. Current machines have usually almost 10× the clockspeed and at least 16× the RAM. Because the test data they used was all planar it is expected that the running time for model transformations will be slower if they are symmetric graphs.

This algorithm seems the most promising because it calculates both a common subgraph and a similarity score and the running time is totally acceptable on old hardware.

Table 1: RASCAL time trial results (percentage of graphs remaining after screening, total running time including screening in seconds, running time per matching ignoring screening in milliseconds).

Data set	No. of comparisons	Avg. $\|V(G)\|$		Minimum similarity index 0.7	0.75	0.8	0.85
ASX	14,501,805	32.5	Percentage after screening	11.7%	6.15%	2.91%	1.04%
			Total time in seconds	71,8065	37,299	17,570	7058
			Time per matching in ms	42	41	41	46
CNS	127,992,000	23.3	Percentage after screening	1.11%	0.38%	0.13%	0.040%
			Total time in seconds	55,340	22,216	8462	3269
			Time per matching in ms	38	45	50	46
NAN	3,684,255	26.3	Percentage after screening	1.31%	0.60%	0.26%	0.096%
			Total time in seconds	1883	826	349	150
			Time per matching in ms	39	37	36	42

Related work: Automatic Change Recommendation of Models and MetaModels Based on Change Histories (by Kögel, Groner and Tichy) [9]

Kögel et al. descibe an algorithm to find live recommendations based on the last user change and change histories. They compare the last user change which is one henshin rule and its arguments with the change history and find historic precedents for this rule. They then aggregate related rules with the same arguments and finally substitude the arguments of historic rules with arguments of the user change where possible. The algorithm can further be extended by aggregating related recommendations, substituting the arguments of historic rules with placeholders, and also take indirectly related rules into consideration. They evaluate their algorithm by counting true and false positives and calculate the precision which is between 0.43 and 0.82 for the tested datasets.

This algorithm is limited in several ways:

- it only takes a single user change into consideration
- recommendations are not checked for their validity
- the aggregation of henshin rules can lead to too specific recommendations or discard correct recommendations
- only change history is considered to generate recommendations

The algorithm proposed in this proposal tries to be more general in that it abstracts from the data source. Typical model transformations can be generated from change history, specified manually or using any other method. The algorithm also won't be limited to a single user change. Instead it will use the complete user changes which can include many henshin rules. The algorithm also tries to be more general when searching for related model transformations. It does not only use the list of henshin rules with their arguments but the whole graph structure.

During the evaluation phase of this thesis the results can be compared with the results achieved by Kögel et al.

5 Research questions and/or Hypotheses

The following will describe the research questions I attempt to answer with this thesis.

RQ1 What are objective quality criteria for model transformation recommendations?

Objective quality criteria are needed to correctly evaluate the result of the algorithm and draw meaningful conclusions.

One such criteria could be randomly generate input graphs for the algorithm and create a confusion matrix with false/true positives/negatives and evaluate according to the confusion matrix.

Another important criteria is the running time of the algorithm.

RQ2 How can we find model transformation recommendations that are good according to our quality criteria?

This is the main goal of the thesis and the answer to this question will be an algorithm that attempts to give good result according to the quality criteria.

A possible way of arriving at a good algorithm is choosing one algorithm in literature and modify it to make it work (better) with model transformations.

This involves testing the algorithm with different inputs and parameters and evaluating the results according to the quality criteria of **RQ1**.

RQ3 What is a good compromise between running time and quality of the results?

After evaluating the quality of the results and the running time the next step is to relate the two and find a good compromise between them.

The result will probably be one or more diagrams to show how the running time relates to the quality of the recommendations with differen input sizes and parameters.

6 The Design - Methods and Procedures

This thesis will follow Design Science Research. [10] Figure 2 shows an overview of the workflow and resulting artifacts.

This proposal is part of forming an *Awareness of the Problem*. The following also gives a *Tentative Design* which contains a description of the *Development*, *Evaluation* and *Conclusion*.

The first step **M1** is a review of literature to find one or more promising algorithms or combination of algorithms that solve the problem (inexact graph matching and finding a similarity score). Some literature review has already been done during the writing of this proposal.

The second step **M2** is to define objective quality criteria for the recommendations (**RQ1**). This is done before seeing how the recommendations look like to guarantee that they are objective. These criteria will probably be similar to the once used in literature to evaluate other graph matching and similarity scoring algorithms.

After defining quality criteria and doing a literature review the next steps **M3-M5** are to implement, test and evaluate an algorithm to solve the problem. The development, testing and evaluation will be done iteratively first for unlabeled graphs, then for labeled graphs and finally for *real data* [11] (graphs consisting of henshin rules). These iteration steps will be split up into even more steps as necessary to tune and improve the algorithm iteratively. If the result of one of the first evaluations shows the algorithm gives bad results or has poor performance and the algorithm is not easily fixable I will start over using another algorithm or combination of algorithms. This answers **RQ2-RQ3**.

Development

The algorithm will be developed in Haskell because it offers good type safety (which aids the correctness of the code). Complex algorithms are more readable and it is easy to prototype in Haskell [12]. I will

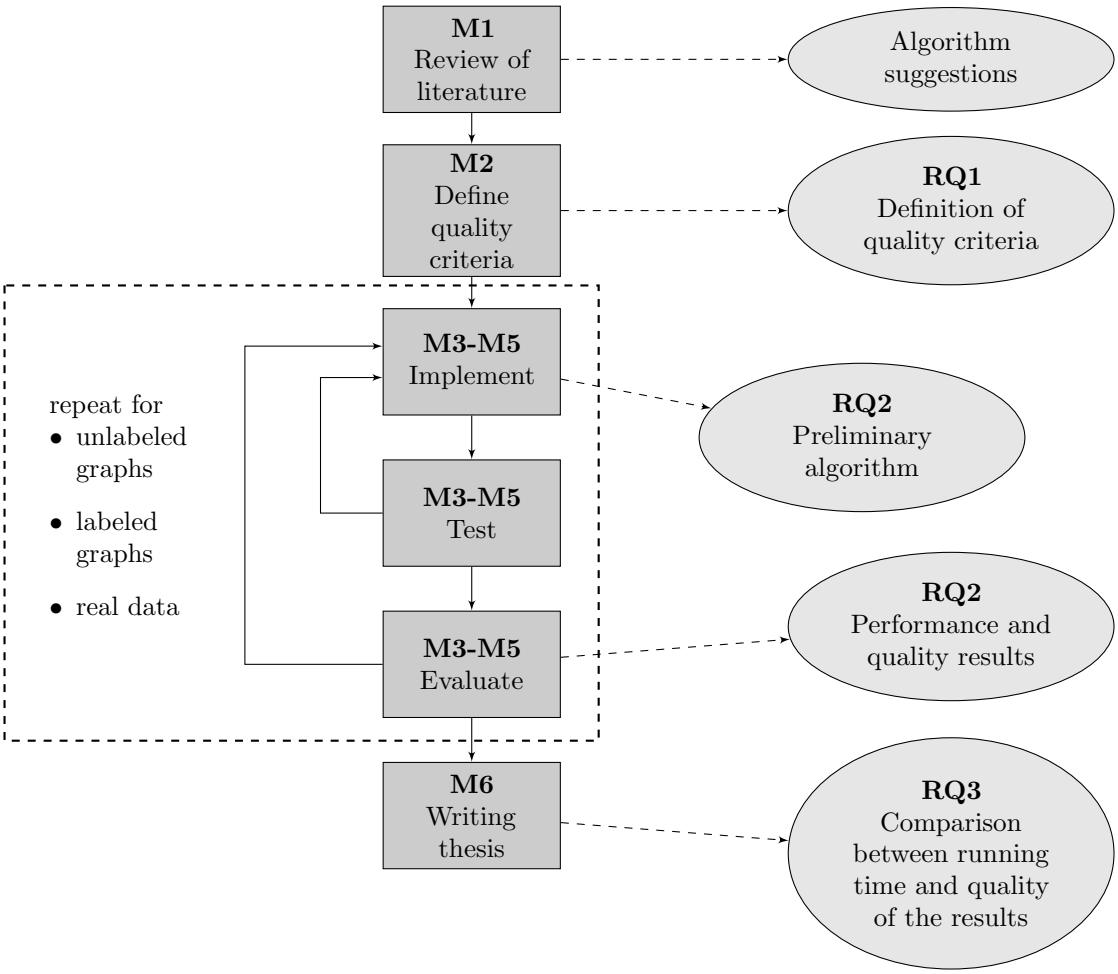

Figure 2: Workflow

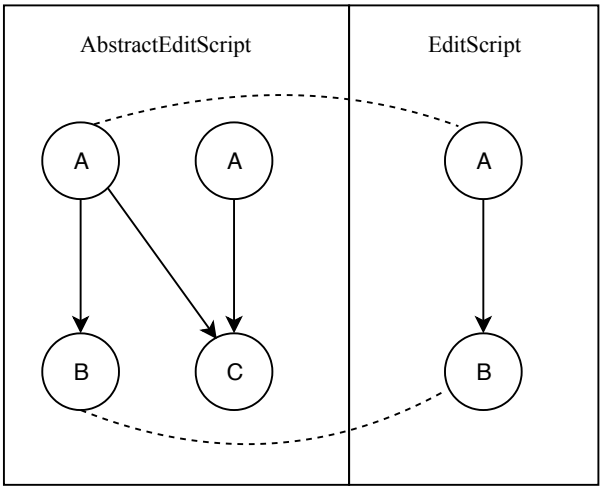

Figure 3: Mapping between an `EditScript` and an `AbstractEditScript`

utilize unit tests and quick check to make sure the algorithm is correct. The following descibes a rough outline of the algorithm:

The recommendations are generated by first calculating an imperfect matching and a similarity score between the user changes and the typical model transformations. The similarity score is later used to rank the recommendations. The imperfect matching between an `EditScript` and an `AbstractEditScript` is informally a common subgraph matching between the two graphs as well as a mapping between the nodes of the `EditScript` to the nodes of the `AbstractEditScript`. See figure 3. The mapping must also retain the graph structure which means the matched nodes must have the same edges between them. But additional edges are allowed. Finally these imperfect matchings are converted to `Recommendation`s which can be applied to the model.

`EditScript`s and `AbstractEditScript`s are graphs consisting of *Henshin Rules* [1] and parameters as nodes with edges pointing from henshin rules to their parameters. All parameters and henshin rules have attached labels. E.g. *"addMethod"* for henshin rules or *"class"* for parameters. In `EditScript`s the parameters are filled with actual values from the model while in `AbstractEditScript`s the parameters are only placeholders.

The mapping between an `EditScript` and an `AbstractEditScript` points from parameters to parameters and from henshin rules to henshin rules. A parameter in the `EditScript` can point to multiple (or no) parameters in the `AbstractEditScript` while a henshin rule in the `EditScript` points to at most one henshin rule in the `AbstractEditScript`. The mapped nodes also have the same label. E.g. a *"class"* can only point to a *"class"*.

A `Recommendation` is created from an `AbstractEditScript` where the values in the placeholder nodes are filled in by values of the mapped nodes from the `EditScript`. This is only possible for the nodes in the common subgraph which means not all nodes are necessarily filled. That means the `Recommendation` is not necessarily an `EditScript` but somewhere in between an `EditScript` and an `AbstractEditScript`.

Testing and Evaluation

For the first two evaluations (unlabeled graphs and labeled graphs) I will test and evaluate the algorithm with randomly generated graphs. These could be created as follows:

1. Generate a random graph representing the `EditScript`

2. Manipulate the graph to create `AbstractEditScript`s. For example like the following:
   - Delete some nodes or edges
   - Combine the graph with another randomly generated graph representing an `EditScript`
   - Swap some node or edge labels
   - Change the direction of edges
   - Arbitrarily change labels

   While creating the `AbstractEditScript`s count the number of changes to get an expected ranking that can be used in the confusion matrix.

The third iteration will also use *real data* from [11] in addition to randomly generated graphs for testing and evaluation.

Evaluation will be done according to the defined quality criteria and will roughly work like this:

1. Generate a list of `(EditScript, [AbstractEditScript, ExpectedRanking])` pairs using random graphs or real data
2. run and time the algorithm on each of the test pairs
3. analyze the recommendations according to the previously defined quality criteria
4. plot and evaluate the resulting data and find out what needs to be improved if necessary

Additionally the results can also be compared to the results of [9].

Conclusion and Artifacts

The bachelor thesis will contain a description of the problem as well as a description of the solution and the conlusion. Besides the bachelor thesis and the algorithm in Haskell there will be performance and quality measurements for each iteration of the algorithm.

7 Limitations and Delimitations

The quality of the recommendations is dependent on the pre-generated list of typical model transformations, which is not part of this thesis. The research will be limited to randomly generated graphs and model transformations [13] generated from change history by Stefan Kögel. Randomly generated graphs have the benefit that the expected result is known because it is generated as well while the expected result of *real* model transformations in not known.

Only model transformation in form of graphs containing henshin rules will be used to test the algorithm. As long as other representations for model transformations use the same overall structure as henshin rules (labeled, directed graphs with actions and parameters as nodes and edges pointing from actions to their parameters) the recommendation algorithm should yield similar quality for recommendations.

This thesis will only include one algorithm. A comparison to other algorithms is not directly possible because there are no similar algorithms for model transformations and developing and evaluating different algorithms would take to much time.

8 Significance of the study

While there are some algorithms and tools that can, given an input graph, find similar graphs. [7], [14] There are no algorithms that are directly applied to model transformation recommendations. Also most of the algorithms only find a mapping and not a similarity score, which is needed for ranking recommendations.

Providing an algorithm to find model transformation recommendations could dramatically improve the user experience of model editors and speed up model driven development. Because it would make it much easier to apply big changes from the recommendations instead of applying them manually.

9 Planning

9.1 Own Background

Completed *Paradigmen der Programmierung* and *Funktionale Programmierung* lectures, which teach the basics of Haskell and functional programming.

9.2 Required Resources

Cpu time on a server is needed to test the results of the algorithm and evaluate its running time on big data sets. A server will provide a more consistent execution environment than just running it on a laptop and therefore improve the quality of the evaluation. This will probably be needed after every iteration of the development. The test will probably be run on BW Grid and BW Cloud.

9.3 Work packages

M1 Review of literature (answers **RQ2**)

M2 Define quality criteria (answers **RQ1**)

M3 Development and evaluation (answers **RQ2** and **RQ3**)

M4 Development and evaluation (answers **RQ2** and **RQ3**)

M5 Analyze results (answers **RQ3**)

M6 Write thesis

9.4 Contingency plan

- Chosen algorithm is to slow or gives bad results

  In one of the first development / evaluation iterations this can be resolved by either optimizing the algorithm and if that is not possible search for another algorithm. In the later iterations it is not feasible to start developing a new algorithm so only optimization will be attempted.

References

[1] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced concepts and tools for in-place emf model transformations," in *Model Driven Engineering Languages and Systems*, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 121–135, ISBN: 978-3-642-16145-2.

[2] S. Kögel, "Recommender system for model driven software development," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ACM, 2017, pp. 1026–1029.

[3] S. Sorlin and C. Solnon, "Reactive tabu search for measuring graph similarity," in *Graph-Based Representations in Pattern Recognition*, L. Brun and M. Vento, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 172–182, ISBN: 978-3-540-31988-7.

[4] L. A. Zager and G. C. Verghese, "Graph similarity scoring and matching," *Applied Mathematics Letters*, vol. 21, no. 1, pp. 86 –94, 2008, ISSN: 0893-9659. DOI: https://doi.org/10.1016/j.aml.2007.01.006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893965907001012.

[5] E. J. Gardiner, J. W. Raymond, and P. Willett, "RASCAL: Calculation of Graph Similarity using Maximum Common Edge Subgraphs," *The Computer Journal*, vol. 45, no. 6, pp. 631–644, Jan. 2002, ISSN: 0010-4620. DOI: 10.1093/comjnl/45.6.631. eprint: http://oup.prod.sis.lan/comjnl/article-pdf/45/6/631/1184782/450631.pdf. [Online]. Available: https://doi.org/10.1093/comjnl/45.6.631.

[6] M. R. Gary and D. S. Johnson, *Computers and intractability: A guide to the theory of np-completeness*, 1979.

[7] S. Melnik, H. Garcia-Molina, and E. Rahm, "Similarity flooding: A versatile graph matching algorithm and its application to schema matching," in *Proceedings 18th International Conference on Data Engineering*, 2002, pp. 117–128. DOI: 10.1109/ICDE.2002.994702.

[8] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, no. 5, pp. 604–632, Sep. 1999, ISSN: 0004-5411. DOI: 10.1145/324133.324140. [Online]. Available: http://doi.acm.org/10.1145/324133.324140.

[9] S. Kögel, R. Groner, and M. Tichy, "Automatic change recommendation of models and meta models based on change histories.," in *ME@ MODELS*, 2016, pp. 14–19.

[10] V. Vaishnavi and W. Kuechler. (Jan. 2004). Design Research in Information Systems, [Online]. Available: http://desrist.org/design-research-in-information-systems.

[11] S. Kögel and M. Tichy, "A dataset of emf models from eclipse projects," *Universität Ulm*, 2018. DOI: 10.18725/oparu-9850. [Online]. Available: https://oparu.uni-ulm.de/xmlui/handle/123456789/9907.

[12] P. Hudak and M. P. Jones, "Haskell vs. ada vs. c++ vs. awk vs.... an experiment in software prototyping productivity," *Contract*, vol. 14, no. 92-C, p. 0153, 1994.

[13] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[14] P. Yi, B. Choi, S. S. Bhowmick, and J. Xu, "Autog: A visual query autocompletion framework for graph databases," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1505–1508, 2016.

[15] V. Blondel, A. Gajardo, M. Heymans, P. Senellart, and P. Van Dooren, "A measure of similarity between graph vertices: Applications to synonym extraction and web searching," *SIAM Review*, vol. 46, no. 4, pp. 647–666, 2004. DOI: 10.1137/S0036144502415960. eprint: https://doi.org/10.1137/S0036144502415960. [Online]. Available: https://doi.org/10.1137/S0036144502415960.

[16] H. Österle, J. Becker, U. Frank, T. Hess, D. Karagiannis, H. Krcmar, P. Loos, P. Mertens, A. Oberweis, and E. J. Sinz, "Memorandum on design-oriented information systems research," *European Journal of Information Systems*, vol. 20, no. 1, pp. 7–10, 2011.

[17] V. R. Basili, R. W. Selby, and D. H. Hutchens, "Experimentation in software engineering," *IEEE Trans. Softw. Eng.*, vol. 12, no. 7, pp. 733–743, 1986.

[18] W. C. Booth, G. G. Colomb, and J. M. Williams, *The craft of research*, 2nd. University of Chicago press, 2003, ISBN: 0-226-06567-7.

[19] A. Collins, D. Joseph, and K. Bielaczyc, "Design research: Theoretical and methodological issues," *Journal of the Learning Sciences*, vol. 13, no. 1, pp. 15–42, 2004.

[20] J. W. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*, 3rd. London: Sage Publications Ltd., 2008.

[21] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," *IEEE Software*, vol. 12, no. 4, pp. 52–62, 1995, ISSN: 0740-7459.

[22] M. K. Malhotra and V. Grover, "An assessment of survey research in pom: From constructs to theory," *Journal of Operations Management*, vol. 16, no. 4, pp. 407–425, 1998, ISSN: 0272-6963.

[23] J. McKay and P. Marshall, "Shaping a process model for action research," in *PACIS*, 2001.

[24] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2008, ISSN: 1573-7616.

[25] A. Wilkinson, *The Scientist's Handbook for Writing Papers and Dissertations*, ser. Prentice Hall Advanced Reference Series. Prentice Hall, 1991, ISBN: 9780139694110.

[26] R. Yin, *Case Study Research: Design and Methods*, 4th. SAGE Publications, 2008, ISBN: 9781412960991.