Automatic Change Recommendation using Decision Trees

Bachelor Thesis Proposal

Valentin Reyes Häusler

April 8, 2018

1 Introduction and Motivation

- Design phase is a core step in any software development process
- Objective of design phase: creation of design for architecture, software components, interfaces and data (IEEE-Standard "design phase")
- This is accomplished by creating models and meta-models describing the solution precisely and completely while maintaining a necessary level of abstraction. (Skript 07.1)
- For example: In object-oriented modeling class diagrams are indispensable. **??** shows classes and their relationships such as dependencies and associations.
- Importance of models: Recognition of patterns and redundancies allowing simplification (creation of a super class) ... (TODO: more examples)
- It is clear that a precise and complete design serves as a stable base for an efficient implementation.
- Tools for the latter phase are vastly available. They range from code completion and action recommendation to generation of complete blocks of code.
- This allows the engineer to concentrate on the core tasks of the implementation phase.
- Although the design phase is equally important, comparable tools are often lacking or non existent.
- A possible improvement in this aspect are automatic action proposals for the creation of models.
- Basic to semi-complex actions such as the creation of a super class, separation of packages and so on, could be proposed to the designer.
- Such a tool would again allow the engineer to concentrate on the core tasks of a consistent and complete design.
- This thesis aims to create and evaluate such a tool.
- We intend to do this by analyzing the versionized histories of models
- Using machine learning algorithms and data structures we look for reoccurring patterns
- The patterns are then matched to a model which is being worked on.
- If a fit is found a recommendation can be done.
- Example for class diagrams:

Pattern: State: Two classes X and Y share some attributes and functions
Actions:

- Create superclass Z for X and Y
- Move shared attributes and functions from Y to Z

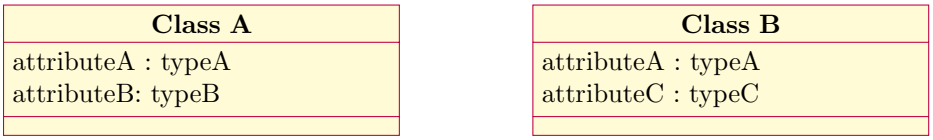

Figure 1: Class diagram. Class A and class B share attributeA.

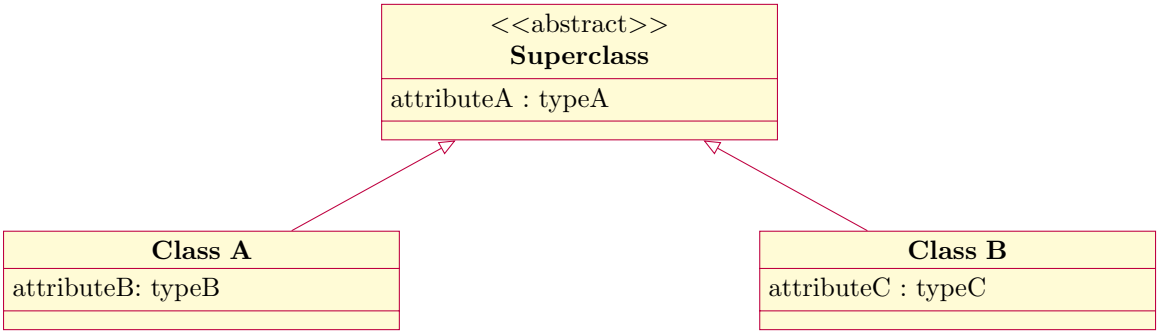

Figure 2: attributeA is "pulled up" from class A and B to Superclass.

- Delete the same attributes and functions from X

Abstraction: Pull up shared attributes/functions to superclass

2 Problem Statement

- We use SiLift
- SiLift is able to compare different versions of a model
- It recognizes *low-level* actions and dependencies between those actions.
- Low-level actions are simple internal operations for the deletion and creation of objects and their references.
  - Figure 3 shows the low-level differences between two versions of the model *company*.
- These low-level actions can be combined to *user-level* actions. This can be seen in 3.
-Creating a new string-attribute *section* in the class *Developer* consists of:

1. Create attribute: section
2. Add containingClass-reference from section to Developer
3. Add structuralFeatures-reference from Developer to section
4. Add type-reference from section to String
5. Change the value of section.

- Abstracting the individual actions we can create a *recognition rule* for the user-level action INSERT <type><attr>=<val>IN <class>:

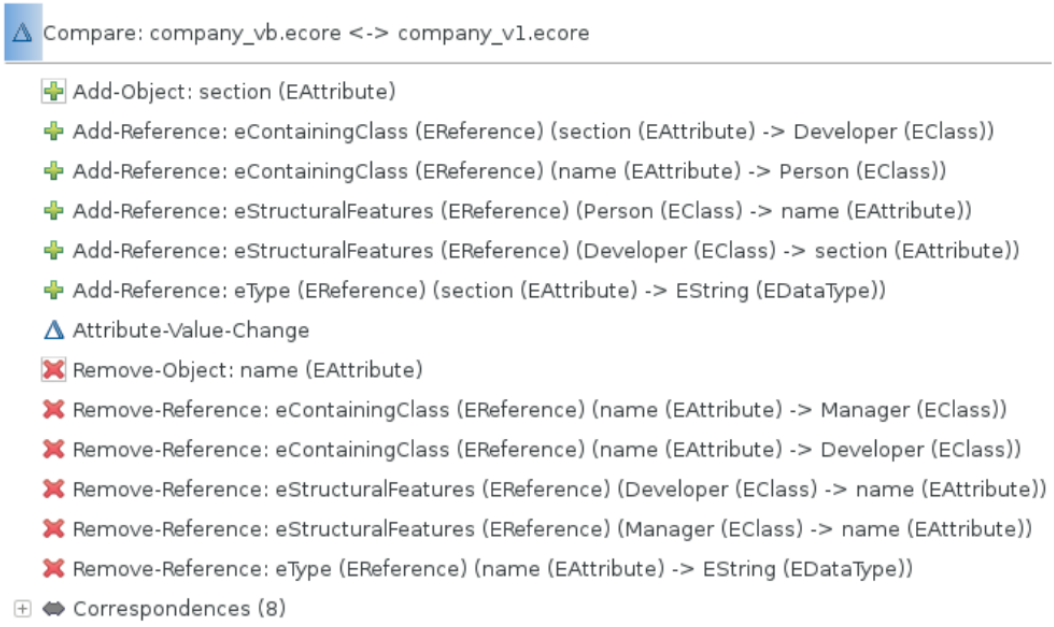

Figure 3: Low-level differences between two models as recognized by SiLift

1. Create attribute: <attr>
2. Add containingClass-reference from <attr>to <class>
3. Add structuralFeatures-reference from <class>to <attr>
4. Add type-reference from <attr>to <type>
5. Change value of <attr>to <val>

- SiLift uses these recognition rules and *lifts* matching sets of low-level actions to user-level actions.

- We aim to automatically generate recognition rules from a models history.
- The initial data is given in from of a dependency graph.
- Nodes represent actions; edges dependencies.
- Figure 4 represents a dependency graph.
- Note that since not every actions has a dependency there can be multiple nodes without parent nodes. And multiple node disjunct graphs.
- We call this nodes orphan-nodes.
- Generating the recognition rule for *INSERT* <*type*><*attr*>=<*val*>*IN* <*class*> we applied an abstraction to the low-level actions.
-The abstraction consists of mapping the arguments of an action to variables.
- For example: Create attribute: section becomes Create attribute: <attr0>with <attr0>= section.
- Applying this abstraction allows us to compare actions and action sequences to one another.
- We use this to find reoccurring patterns of action sequences in a model's history.
- To do this we create multiple decision trees.
- The generation of a decision tree is as follows:

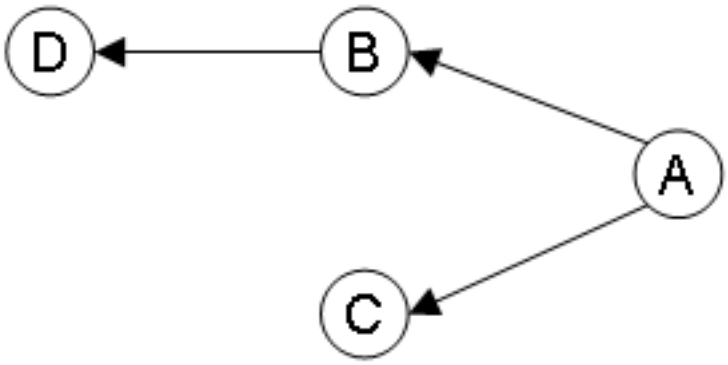

Figure 4: Dependency graph. A has to happen before B and C; B has to happen before D.

1. Find every orphan node representing the same abstract action.
2. Create a decision tree with that action as it's root.
3. For every orphan node do:
   a) Breadth first search every possible path starting at the orphan node.
   b) If there is no equivalent path in the decision tree starting at the root, find the largest matching path starting at the root and expand it to match.

- The construction of the decision trees allows us to calculate the frequency and with that the probability of a certain path happening.
- With this data structure we can match the last actions of a user to the corresponding decision (sub-)tree and giving subsequent paths as possible actions.

- It is easy to see that this format could be expanded to be used in other applications where access to a versionied history is available.

3 Purpose of the study

The purpose of this study is to create a tool for change proposal during the creation of models and meta models. Such a tool, if efficient, could provide support for engineers during the design phase of software comparable to autocompletion tools used during the implementation phase.

4 Review of the literature

Until now only the comparable paper on automatic change recommendation has be studied. Further reading can be done for autocompletion algorithms for other applications. Decision trees, random forests and other regression data structures should be contemplated as well.

5 Research questions and/or Hypotheses

RQ1: What are the problems when implementing change recommendation tools?

RQ2: How helpful is such a tool realistically?

RQ3: How well does the tool perform?

RQ5: How much data does the tool need to perform relatively well?

RQ4: How does the tool compare to other implementations?

6 The Design - Methods and Procedures

7 Limitations and Delimitations

The goal of this study is to create an effective and helpful change recommendation tool to aid during the creation of models. We will still aim to keep the tool as broad as possible so that it can be used for other application in which a change history is available.

Although a user interface and plugin for different model creation software are certainly imaginable, we won't try to archive those goals. Instead a command line interface will be implemented.

8 Significance of the study

The significance of this study lies within its usability and helpfulness. Streamlining the process of model creation aids to increase the quality of software and its design. The ability to bypass mindless repetition during the creation of a model helps the engineer to focus on the important aspects of that model.

The creation of this study will also help in deepening my knowledge in functional programming and machine learning. Both of which serve as useful tools for any programmer. With the former being an additional paradigm with which to program and the latter being a relevant topic in almost every aspect of computer science.

9 Planning

9.1 Own Background

My knowledge in functional programming, while lean, serves as a good start for the creation of this study. Especially when supported by my background with lectures such as "Programmierung von Systemen" and "Algorithmen und Datenstrukturen" as a student as well as a tutor. Both of which are highly applicable in this study. My participation in "Sopra 16/17" provides me with experience for the designing, implementation and testing of the tool which we aim to create.

Having said that, extensive practice in practical functional programming will be necessary as well as attaining knowledge in machine learning, especially regression algorithms. My successful completion of the lecture "Einführung in die künstliche Intelligenz" and the proseminar "Algorithmen" will hopefully provide a strong base for that.

9.2 Required Resources

No especial resources will be necessary. Only access to data in form of model histories and the tool SiLift should suffice for the creation of this study.

9.3 Work packages

M1 Extensive research into regression learning and practice with practical Haskell.

M2 Design and planning of the tool.

M3 Implementation and testing.

M4 Implementation and testing.

M5 Review and possible correction of the tool.

M6 Collection of results on paper, answering all research questions, including the comparison with comparable tools.

9.4 Contingency plan

1. Failing to create such a tool with Haskell due to lacking knowledge. This eventuality will be discussed with the creators of the study.

References

[1] (). Eclipse modeling framework (emf) - tutorial, [Online]. Available: `http://www.vogella.com/tutorials/EclipseEMF/article.html`.

[2] J. Church, *Learning Haskell Data*, ser. Community experience distilled. Packt Publishing, 2015, ISBN: 9781784394707. [Online]. Available: `https://books.google.de/books?id=9vPXsgEACAAJ`.

[3] M. O.T. S. Timo Kehrer Udo Kelter, "Understanding model evolution through semanticallylifting model differences with silift," Software Maintenance (ICSM).

[4] M. T. Stefan Kögel Raffaela Groner, "Automatic change recommendation of models and meta models based on change histories," Institute of Software Engineering and Programming Languages, Ulm University.