Decision reuse in an interactive model transformation

Mika Siikarla  Tarja Systä
Tampere University of Technology
Institute of Software Systems
P. O. Box 553, FI-33101 Tampere, Finland
E-mail: {mika.siikarla,tarja.systa}@tut.fi

Abstract

Propagating incremental changes and maintaining traceability are challenges for interactive model transformations, i.e. ones that combine automation with user decisions. After evolutionary changes to the source models the transformations have to be rerun. Earlier decisions cannot be used directly, because they may have been affected by the changes. Re-doing or verifying each decision manually is error-prone and burdensome.

We present a way to model user interaction for transformations that are well (but not fully) understood. We model each decision as a set of options and their consequences. Also, we model the decision context, i.e. the circumstances (including model elements) affecting the decision. When a transformation is run, user decisions and their context are recorded. After a model change, a decision can be safely reused without burdening the user, if its context has not changed. The context maps source model elements to a decision, and thus provides traceability across the decision.

1 Introduction

The Model Driven Development (MDD) paradigm [12] proposes to use model transformations to keep all the models of a software system consistent and up-to-date during maintenance and development. One goal is that evolutionary changes could be made to high-level models (instead of to the code) and the other models and code would be derived with model transformations. A model transformation constructs one set of models (target models) based on another set of models (source models). A model transformation is built by a transformation architect and executed by a transformation engineer.

We say a transformation is semi-automatic if the transformation engineer makes decisions that guide an incomplete or ambiguous automatic transformation. For example, he first annotates the source model by adding markings (e.g. stereotypes) and then runs an automatic transformation that uses the markings to resolve ambiguity. We call a semi-automatic model transformation interactive if the engineer makes decisions during the transformation, as need rises, instead of as a separate step.

Semi-automatic transformations enable the MDD approach even when full automation is not possible, for example because the mapping between the source and target domains is not thoroughly understood. Also, developing a complete fully automated transformation may take too much time or effort, especially if it is built for a single project.

Software evolution and maintenance cause changes to the source models. Also, in iterative development the models undergo several rounds of changes. The transformation is executed to update the target models. Some of the human decisions from the previous run may have become invalid while others remain valid. Having to repeat the manual effort for the unaffected decisions is highly undesirable; it burdens the engineer, slows down the process and the repetitious nature increases the risk of human error. Of course, the more frequent and smaller the changes are, the larger portion of the total effort is redundant.

It is not possible to simply record all the decisions during one run and reuse them indiscriminately in the next run, because some of them might be invalid. If the decisions are recorded and offered as defaults, the transformation engineer has to manually verify the decisions’ validity. This does not reduce the engineer’s work very much, because verifying a decisions can take almost as much effort as making the decision in the first place. For example, in order to tag a class with Unique_DB_Table the transformation engineer might have to examine the attributes and associations of the classes within the inheritance hierarchy.

In this paper we introduce a way to model a class of decisions in interactive transformations so that decision validity can be automatically verified thus reducing the human burden in decision reuse. We propose that for each decision the transformation architect models its decision context, i.e. the
circumstances that affect the decision, a set of source model elements. When the transformation is run, user decisions and their context are recorded. When the transformation is re-run after a model change, a decision can be safely reused if its context has not changed.

Modeling the decisions helps document which decisions were available and which were chosen at development time, not just the consequences of the decisions. Decision context also provides some traceability across decisions. Features in the target model can be traced back to the decision(s) that caused them. The decision context maps features in the source model to the decision and thus to parts of the target model.

Our approach is useful in cases where the mapping between the source and target domains is understood well enough to define which circumstances affect a decision, but not well enough to define precisely how. We limit to interactive model transformations where the user actions are limited to decisions with predefined options and consequences. Arbitrary modifications to the model are not allowed.

We present an interactive transformation mechanism and its implementation. It utilizes decision context to achieve automatic decision reuse. The mechanism uses graph transformation to process the source models and creates a task graph, i.e. a description of the interactive transformation process. An interactive tool executes the task graph and records decisions and their context and creates the target model. After a change in the models the transformation is rerun and valid decisions are automatically reused.

We have presented the basic interactive transformation mechanism using graph grammar and task graph in [15]. In this paper we introduce the concept of decision context. We extend the graph grammar to use context definitions and to automatically reuse valid decisions from an earlier transformation, thus enabling evolutionary changes to the source models. We present and discuss a transformation from an entity relationship (ER) diagram to a database schema.

2 Related work

We have not been able to find any other generic model transformation approach that supports defining the conditions under which a decision can be reused. There are many approaches where decisions are recorded and at the transformation engineer’s disposal, but checking the decisions’ reusability is left to the engineer.

One of the most common ways to implement semi-automatic model transformations is based on model marking [11, p. 3-6]. Source model elements are manually marked to indicate which transformation rule should be applied and what role the element should play in the execution of the rule. In UML models stereotypes are typically used for marking. The markings provide the additional information required for unambiguous transformation. An automatic transformation is then used to produce the target models. Any automatic model transformation mechanism can be used in this manner. Decisions (the markings) are preserved over source model changes, but unlike in our approach there is no way to automatically determine if the decisions can be reused.

Other approaches preserving or keeping track of user’s decisions during model transformations have been developed especially to support reverse engineering. In Rigi[17], for instance, the user can manipulate and query the reverse engineered model by (writing and) running scripts. The tool saves the scripts in the order they have been run, thus keeping track of the user’s model analysis steps. The user’s decisions are preserved rather implicitly in the scripts. The decision context are not necessarily saved.

The work by Gorp et al. on supporting traceability in interactive consistency maintenance of software models [6] comes, perhaps, closest to our work. As we do, they aim to interactively and semi-automatically solve inconsistency problems due to changes in one of the models. The proposed tool, ICONS, combines two existing methods and tools: ToolNet[1] for manual and CAViT[7] for automated consistency maintenance. CAViT can be used to assess a certain consistency violation and repair it automatically. In ICONS violating model elements are presented to the user, who can solve the inconsistency problems either manually using ToolNet or in an automated fashion using CAViT. The user can also choose to ignore the inconsistency.

In [5], Embley and Xu propose an interactive model-driven approach to transform SQL databases to Object-Relationship Model (ORM) instances and to further revise or abstract the ORM instances. While the approach by Embley and Xu is solely targeted to database reverse engineering, our approach is a generic model transformation approach. Namely, our approach does not use any fixed transformation rules. Instead, the user can define her own transformation patterns for her own specific purposes.

The transformations in the approach by Embley and Xu consists of a set of rules, e.g., how to handle primary and foreign keys, and how to create generalization/specialization hierarchies. In cases where the rules do not yield unique results, the transformation tool SQL2ORM consults the user. Support for modeling user decisions and decision context is not, however, provided.

3 Overview of the transformation process

3.1 Overview

When constructing a transformation, the transformation architect first identifies tasks or task sets that are repeated during the transformation. He groups similar task sets
together and identifies the degree and kind of variation within the groups. The transformation architect chooses what kind of decisions to use for expressing the variation. These steps are necessarily vague; they are normal design work. The transformation architect relies on his experience, skills and possibly a domain expert to understand the problem domain.

Our approach helps the transformation architect to model his understanding of how the transformation should work. He models each task set and its decisions as a task graph fragment, which describes what actions are performed. He models to which source model elements the task set should be applied as an application condition. For each decision he models the condition for reusability as a decision context.

The transformation implementation consists of the task graph fragments (which may contain decisions), the decision context definitions and a graph grammar containing the application conditions.

When a transformation engineer executes the transformation on some source model, the tool runs the automatic graph grammar which combines task graph fragments into a task graph. The task graph is an executable description of the transformation process customized for that specific source model. The task graph contains automatic and manual tasks. The manual tasks are decisions and the graph grammar has marked their decision context in the task graph. The tool executes the task graph and presents manual tasks to the user and records the decisions in the task graph.

After a change in the source model the transformation engineer executes the transformation again. The tool runs the graph grammar to generate a new task graph. At this point the graph grammar engine will attempt to reuse decisions from the previous task graph. Figure 1 shows a model of performing a transformation activity.

### 3.2 Task graph

A task graph is a directed acyclic graph with labeled nodes and edges. A node represents an atomic task that is performed as part of carrying out the transformation process as a whole. The edges represent relations and ordering between the tasks. When a task is performed it is bound to a value. For example, when a task for selecting or creating a UML class is performed, the task is bound to the class (its unique identifier). The task graph can then be seen as a collection of variables and the transformation process as a problem of finding a solution to the multi-variable “equation”.

Figure 2 shows a small task graph at a certain point of time during a transformation from a class diagram to a relational model. Tasks are represented by filled circles in the center. The class symbols on the left are source model elements and the squares on the right are target model elements (database tables). The oval shapes labeled $cl-tbl$ have to do with marking decision context. All tasks except for task $T3$ have been performed. Tasks $C1$, $C2$, $C3$ and $C4$ are bound to source model elements $a$, $b$, $c$ and $d$ respectively. Task $T1$ is bound to table $x$ and $T2$ and $T4$ to table $y$. Task $T3$ has not yet been performed and is unbound.

### 3.3 Decision context

When the transformation architect constructs the transformation, he defines what kind of decisions there are and how their decision context is formed. He also chooses the application condition. For example, he might choose “to which table should this class be mapped” as one decision kind. He decides that any such decision depends on which class “this class” is and which class, if any, it inherits. In our approach this knowledge is modeled as a mapping definition, in this case $cl-tbl$: $(class, superclass) \rightarrow (table)$. $table$ is the decision and $class$ and $superclass$ form the context. This decision must be applied once for each class in the source model. Figure 3 visualizes this mapping.

If there are multiple decisions that will always appear together and have the same context, they can be modeled as a single mapping. For example, $inherit: (class, superclass) \rightarrow (dec1, dec2, dec3)$.

The elements of the mappings are realized in the task graph as tasks that play a role in the mapping. The values of
those tasks, once bound, then determine the concrete value of the mapping. In the task graph in Figure 2 the roles of class, superclass and table are played by, for example, tasks C2, C1 and T2 as well as tasks C4, C2 and T4. The two task triplets represent two elements in the mapping: \((b,a) \rightarrow (y)\) and \((d,b) \rightarrow (y)\).

When a graph grammar rule adds a decision to the task graph, it must also provide tasks to act as the decision context. So, a rule that applies \(cl-tbl\) adds three tasks, one for the decision and two for the context. The rule also binds the two context tasks and thus defines the value for the context. A context task can be bound either explicitly to a specific model element or implicitly by referencing the value of another bound task. Tasks can be merged to reduce the size of the task graph.

For example, applying \(cl-tbl\) to each class in the source model can be done with three graph grammar rules. The first rule is executed once for each class \(c\), creating a task and binding it to \(c\). The second rule is executed once for each subclass \(i\) and its superclass \(s\). The rule applies \(cl-tbl\) by adding tasks \(cl\_{table}, cl\_{class}\) and \(cl\_{superclass}\) for table, class and superclass, respectively. \(cl\_{class}\) is merged with the task the first rule created for \(i\) and \(cl\_{superclass}\) with the task created for \(s\). The third rule is executed once for each class that has no superclass. It differs from the second rule in that \(cl\_{superclass}\) is left unbound.

Figure 2 shows the task graph resulting from executing the graph grammar on four classes. Initially tasks \(T1\)–\(T4\) are unbound, unlike in the figure. Due to task merging, there are five class tasks instead of eight. For example, tasks C2 acts as class in one \(cl-tbl\) element and superclass in another. The oval \(cl-tbl\) shapes and the arrows from tasks point out the decisions and their context. Only two of the four \(cl-tbl\) elements are visualized to keep the picture uncluttered.

### 3.4 Decision reuse

When the transformation engineer starts executing a transformation there are unperformed tasks, i.e. unbound variables. The decision context \(\rightarrow\) decision mappings are not concrete. After all the tasks have been performed, all variables are bound. The task graph now is a record of the concrete mappings. Table 1 contains the values for \(cl-tbl\) mapping from Figure 2 if task \(T3\) is bound to table \(x\).

When the source model is changed the transformation engineer executes the transformation again. The tool first retrieves the previous task graph and recovers the decision context \(\rightarrow\) decision mappings. Next the tool executes the graph grammar. When the grammar adds a decision to the task graph it will check the corresponding recovered mapping for an entity matching the decision context. If a match is found the value that is mapped to that context is used as the value for the new decision. The user does not have to make that decision again, i.e. the decision is reused.

Consider the example above. The transformation engineer performs the transformation on the class diagram in Figure 2 and after he has made all the decisions, the task graph records the mapping in Table 1. Now, the source model is changed so that class \(d\) no longer is a subclass of class \(b\). When the transformation engineer re-launches the transformation, the graph grammar applies the \(cl-tbl\) mapping to the task graph.

Without decision reuse it would create an incomplete mapping \(\{ (a,-) \rightarrow (?, (b,a) \rightarrow (?), (c,a) \rightarrow (?), (d,-) \rightarrow (?) \}\). However, the recovered mapping (Table 1) contains entries for decision context \((a,-), (b,a)\) and \((c,a)\), so those decisions are reused and the decision tasks are bound. There is no entry for \((d,-)\) (only for \((d,b)\)) so that decision task is left unbound. The actual mapping is \(\{ (a,-) \rightarrow (x), (b,a) \rightarrow (y), (c,a) \rightarrow (x), (d,-) \rightarrow (?) \}\).

It should be noted, that the transformation can be re-launched even if the previous transformation has not been completed yet. Decision reuse works normally for the decisions that had been made at that point.

### 3.5 Storing decision context \(\rightarrow\) decision mappings

When the graph grammar adds a decision to a task graph, it marks the context and decision tasks with the mapping name and the name of the role the task plays. When the task graph is stored, the markings are stored along with the task graph structure and the values bound to the tasks. In order to differentiate between different elements of a mapping, each element is associated with a unique number. Since each task corresponds to a value (or lack of value) the entire mappings can be reconstructed from the tasks, values and markings.

For example, in Figure 2 task C1 plays roles in three different elements, so it has markings \((cl-tbl,1,class), (cl-tbl,2,superclass)\) and \((cl-tbl,3,superclass)\). Table 2 shows all the values and markings stored in the task graph from Figure 2.
Table 2: Task markings for cl-tbl

<table>
<thead>
<tr>
<th>Task</th>
<th>Value</th>
<th>Roles (instance/role)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>a</td>
<td>1/class, 2/superclass, 3/superclass</td>
</tr>
<tr>
<td>C2</td>
<td>b</td>
<td>2/class, 4/superclass</td>
</tr>
<tr>
<td>C3</td>
<td>c</td>
<td>3/class</td>
</tr>
<tr>
<td>C4</td>
<td>d</td>
<td>4/class</td>
</tr>
<tr>
<td>T1</td>
<td>x</td>
<td>1/table</td>
</tr>
<tr>
<td>T2</td>
<td>y</td>
<td>2/table</td>
</tr>
<tr>
<td>T3</td>
<td></td>
<td>3/table</td>
</tr>
<tr>
<td>T4</td>
<td>y</td>
<td>4/table</td>
</tr>
</tbody>
</table>

4 Implementation

4.1 Execution environment

The transformation execution environment is built on top of MADE [9], an interactive software design tool developed at Tampere University of Technology. MADE itself is integrated with IBM Rational Software Architect (RSA) [10], an Eclipse [4] based UML CASE-tool.

The transformation engineer can start a transformation from inside MADE/RSA. The model transformation engine executes the graph grammar and passes the resulting task graph to MADE. MADE executes the automatic tasks that do not depend on decisions and presents the currently available human tasks to the transformation engineer. When the transformation engineer makes a decision, MADE records it by binding the corresponding decision task.

The graph grammar might not be able to produce the entire task graph representing all possible consequences of decisions at once. The incomplete task graph is then executed until some decisions have been made. The graph grammar continues and uses the decisions as feedback. This may have to be repeated several times before the transformation process is finished.

MADE presents currently available human tasks as a list. A task is available, if it does not depend on any incomplete task. There is some support for grouping tasks hierarchically, so that the task list does not grow too long. The task list does not block the normal use of the CASE-tool. The transformation engineer is free to view the models in the same way he would in RSA without MADE. The transformation engineer can also choose which of the tasks he performs next. The tool does not force any particular order of performing the tasks.

The transformation engineer can save the task graph in the middle of a transformation action and continue at a later time. The task graph with bindings and decision context markings is stored into its own file separate from the models. When the task graph is loaded, the work can continue.

4.2 Graph rewrite system

The graph grammar of a transformation is executed by a graph rewrite system (GRS) [13] that we developed for this purpose only. Roughly speaking, a graph rewrite system consists of rewrite rules, so called productions. Each production consists of a left-hand side (LHS) and a right-hand side (RHS), which are graphs. When applied to a graph, the rewrite system finds a production whose left-hand side matches a subgraph. The matching subgraph is then replaced with a copy of the production’s right-hand side. This is repeated until there are no productions with matching LHS left.

In our implementation the productions are prioritized. A production can only be applied if no other production with higher priority is applicable. The transformation architect gives the graph grammar in a textual form containing the productions and the decision context mappings. The priority is simply the order in which the productions are introduced, the first production having the highest priority. The syntax is loosely based on a file format for defining graphs used in Graphviz[8]. The following is an example of a production.

```plaintext
rule inh_opt_i {
    cls:ClassRole, inh:inherit
    optI:SingleChoiceRole
    cls <--sub--> inh --(i)---> optI
    nac // neg. app. cond. follows
    cls <--cls-- cls_tbl:ClsToTbl
    =>
    cls:ClassRole, inh:inherit
    optI:SingleChoiceRole
    cls <--sub--> inh --(i)---> optI
    patt:ClsToTbl // create ClsToTbl
    patt.cls --> cls // merge
}
```

The lines up to “=>” describe the left-hand side and the lines after the right-hand side of the production. The first three lines of the LHS describe the application condition, i.e. the subgraph to look for in the graph. Here it is an inherit mapping node attached to a (class) task and a single-option decision task. The line after “nac” describe the negative application condition, i.e. what must not be found in the graph. In this case the class task must not already be attached to a ClsToTbl mapping node.

The transformation architect compiles the grammar file into a Java class implementing a specific interface. The left-hand side of each production is translated into a match finding method and the right hand-side into a graph modifying method. The Java class is compiled as part of the model transformation plugin for MADE.
In our approach the graph rewrite system does not create the target model directly, but rather a task graph. When all the tasks have been performed, the task graph forms a mapping between the source and target models. This resembles correspondence graphs in triple graph grammars (TGG) [14]. Briefly, a TGG is a kind of a graph rewrite system that produces the target model but also a correspondence graph to express relationships between the models. An undeterministic TGG can produce different target models from the same source model, but the grammar has to be complete. The productions alone define all possible outcomes. In our approach, user decisions augment the grammar as “hidden” productions. Also, an unfinished task graph represents several possible mappings. In a sense it is an abstract correspondence graph.

5 Example: ER → DB transformation

5.1 Logical database design

Database design can be divided into conceptual, logical and physical database design phases [2, p. 419]. Logical database design maps a conceptual model (from the conceptual design phase) onto a logical model, which is based on the chosen data model, but independent of any particular database management system. For instance, if the relational model is chosen, the result is a relational schema.

Logical database design is well-suited for interactive transformations because the field is mature and well-studied, yet there is no consensus on how the process should be automated. As Date writes: “[Database] design is still very much an art, not a science. There are (to repeat) some scientific principles … [However], there are many, many design issues that those principles simply do not address at all. There are many design methodologies—some fairly rigorous, other less so, but all of them ad hoc to a degree—that can be used as an attack on what at the time of writing is still a rather intractable problem, viz., the problem of finding “the” logical design that is incontestably the right one.” [3, p. 328]

In this paper the entity relation model from the conceptual phase model is given as a UML class diagram. Figure 4 shows a class diagram with eight classes. We look at the options for transforming inheritance into relational model and show in full detail how the options and the decision making can be modeled using our interactive transformation mechanism. We also implement the decision for selecting primary keys for tables. We implement only two decision kinds, so that we can present them in sufficient detail.

The literature presents several design methodologies for logical database design. We chose one [16, pp. 241–304] that is explained in detail and gives the designer several design options. There are three alternatives for transforming inheritance [16, pp. 282–283].

1. A table is created for the superclass and each subclass participating in the inheritance relationship. A reference to the superclass’s table is added to each subclass’s table.

2. All the subclasses are mapped to the table of the superclass. All attributes from the subclasses map to columns in the superclass’s table. It might be necessary to add one or more discriminator fields in the table to tell to which class(es) a row corresponds.

3. The superclass is split into several tables based on the subclass. Each subclass is translated into its own table and the attributes from the superclass are copied into that table.

The method does not give strict rules on which design to use. The designer must consider the benefits and drawbacks of each option. The decision is made for each inheritance relationship separately, and different alternatives can be selected for classes within the same inheritance hierarchy.

Design 1) can always be used regardless of the properties of the classes and their participation in the inheritance. If the subclasses have very few attributes, the result can be inefficient. Design 2) reduces cross-table references and removes the need for some integrity constraints. However, the table resulting from the merge can be very large. It can contain many unused fields, if the subclasses have many attributes, especially if the subclasses are disjoint. Design 3) can only be used if the superclass is abstract. The superclass’s attributes are duplicated into the subclasses’ tables, which can hinder maintainability. If the subclasses are overlapping there will be data redundancy as well.

The primary key for a table is formed either by selecting a group of its fields or by creating a new artificial identity field. Some groups of attributes suitable as the primary key are

5.2 Decision validity

If the designer chooses design 2) for classes Teacher and Researcher, they map to the same table (Staff). If Staff is re-
moved, the classes will be inheritance roots and a table will be created for each. Despite its simplicity, the transformation exhibits some counter-intuitive behavior. Removing an element from the source model can lead into adding an element into the target model. Similarly, adding an element to the source model may lead into removing element(s) from the target model.

A small local change in the source model can cause scattered changes in the target model. In this case, adding or removing a class has the potential to affect the interpretation of every class in several inheritance hierarchies. It can cause tables to be added or removed and columns moved from table to another. Evaluating the consequences of a change is no longer simple. By modeling the decision context the validity of decision can be determined automatically, saving manual work from the transformation engineer.

5.3 Transformation specification

We model the inheritance mapping decision as a per-class multiple-choice (1, 2 or 3) task. A task graph fragment is attached to each class that acts as a subclass and will determine how that class (branch of the inheritance hierarchy) is transformed. This model is more permissive than the designs above, where the same decision is applied to every subclass in an inheritance relationship.

Create column tasks are created for the attributes of the classes. One attribute can map to columns in more than one table, if design 3) is chosen for some classes. A create table task is added for each class at the top of an inheritance hierarchy and for each class for which option 1) or 3) is chosen. A select column task is attached to each table for choosing the columns for the primary key.

Three of the transformational patterns used in the transformation are shown in Figure 5. The cls-tbl mapping maps a class to a table. It is attached to each class in the class diagram. It only notes which table the class primarily maps to, and does not actually create a new table. This pattern does not represent any decisions.

The inherit mapping represents the inheritance transformation decision. The decision task is a multiple-choice decision with three options. When the designer has made the decision between the three options, it is remembered for as long as the subclass inherits the superclass.

The attr-col mapping in Figure 5b maps an attribute to a column. One or more of these is attached to each attribute. The cl task creates a new column within the table T. The new column will have the same name as the attribute. The attribute task A and the table task T form the decision context and the column task cl is be the decision.

The new_tbl mapping is used to create a new table based on a class. It is used for the classes at the top of the inheritance hierarchy, as well as for subclasses when they are separated into their own table (design 1) or when the hierarchy is split (design 3). Task T uses the name of the class bound to C as the name for the new table. Task C acts as the context.

The productions for the graph rewrite system are illustrated in Figure 6. The first production (at the top, marked #1) is applied to each class and it creates a cls-tbl pattern and binds the select class task to the class. An attrsFrom edge is added from the table task to the class task. Such edges are later used by rule #7 to track which classes produce attributes for a certain table. The negative application condition states that there must not already be a class task connected to the class. This ensures the production is only applied once for each class.

The second production is applied to each attribute. It creates a select attribute task and binds it to the attribute (the octagon). It also connects the attribute task to the corresponding class task, to make rule #7 a little simpler. The production does not create a new task for the class. Instead, it uses the class task created by rule #1.

The third production is applied to each subclass. It creates an inherit mapping, which represents the decision point for choosing between designs 1), 2) and 3) for this subclass/superclass relationship. Task C (bound to the subclass) is merged with the task created by rule #1 for the subclass, and similarly for the superclass. The merged tasks will preserve the bindings made by rule #1, and will therefore act as the decision context. The multi-choice task (labeled xor) and the three options are the decisions. Exactly one of the options can be performed (chosen) and at that time the multi-choice is marked as performed, too.

The fourth production is applied only if the design 2) is chosen for the subclass. The LHS requires that the single-choice task connected to the inherit pattern with a (ii) edge has been performed. Design 2) means the subclass will share the superclass’s table, and all its attributes will be mapped to that table. The production merges the two classes’ table tasks. It also adds a grp edge from the subclass task to the superclass task. Such edges are used in production #6 to track attributes that need to be duplicated.

Normally more than one node in the left-hand side could match the same concrete node in the graph. In this production a constraint is added that requires the two nodes to be distinct. In the figure it is represented by an inequality sign between the two nodes.
The fifth production is applied if design 3) is chosen. It adds a grp edge and an attrsFrom edge to be used by rules #6 and #7. The left-hand side states that there must not already be an attrsFrom edge between the two tasks. In design 1) the subclass has its own table and no attributes from the superclass are transformed into it. There is no need for a production for that case, unlike for designs 2) and 3).

The sixth production propagates the existing attrsFrom edges up the inheritance hierarchy. When design 3) is chosen for a subclass, it will have its own table. Columns are created in that table based on the subclass’s direct and indirect superclasses up to the first design 1) choice or the root of the inheritance. This production is a rather mechanical part of the GRS and does not involve any decisions. We skip a more detailed explanation.

The seventh production adds a attr-col mapping for each class-attribute-table triple that has been marked with an attrsFrom edge by rules 1, 5 or 6. The eighth production adds a new-tbl mapping for each class that is not a subclass or for which design 1) or 3) was chosen. The negative application condition for this production cannot be expressed in the visual notation.

6 Applying ER → DB transformation

6.1 Initial transformation

The transformation in Section 5 is applied to the class diagram in Figure 4. The transformation engineer decides to map classes Teacher and Researcher to the same table as their superclass Staff (design 2). He maps classes Athlete and ExchStudent to their superclasses as well (design 2). He decides that classes FootballPlr and BasketballPlr will have their own tables (design 1).

The inherit mapping is used for capturing those decisions. The values for the mapping is shown in the middle column in Table 3. There are no entries for Staff and Student, because they are not subclasses.

<table>
<thead>
<tr>
<th>Context</th>
<th>Initial decisions xor 1, 2, 3</th>
<th>Decisions after changes xor 1, 2, 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teacher, Staff</td>
<td>true, -, true, -</td>
<td>(reused)</td>
</tr>
<tr>
<td>Researcher, Staff</td>
<td>true, -, true, -</td>
<td></td>
</tr>
<tr>
<td>ExchStudent, Student</td>
<td>true, -, true, -</td>
<td></td>
</tr>
<tr>
<td>FootballPlr, Athlete</td>
<td>true, true, -, -</td>
<td></td>
</tr>
<tr>
<td>BasketballPlr, Athlete</td>
<td>true, true, -, -</td>
<td></td>
</tr>
<tr>
<td>Athlete, Student</td>
<td>true, -, true, -</td>
<td>(N/A)</td>
</tr>
<tr>
<td>Staff, Person</td>
<td>(N/A)</td>
<td></td>
</tr>
<tr>
<td>Student, Person</td>
<td>(N/A)</td>
<td></td>
</tr>
</tbody>
</table>
In a more complicated transformation the causalities are more difficult to grasp. If the user is not familiar with the system, domain, transformation, or some combination of those the task is even more difficult. This is often the case with the ER model evolution, where automatic processing are done in separate phases. However, there is no indication of traceability is lost at decision points. The decisions in the transformation, such as creating a column based on an attribute, require only a small effort. It may seem that the effort saved in creating a column is insignificant. However, it should be noted that the amount of decision tasks that have to be performed in the maintenance transformation depends on the changes to the source model, not the size of the model. If the changes were the same as above, but the initial transformation consisted of 100 decision tasks, there would still only be seven decision tasks to perform.

6.3 Model marking

In the model marking approach, esp. with UML models, decisions are recorded using of stereotypes. Just recording the decisions does not help determine which decisions need to be checked. Source model elements are manually marked with tags (stereotypes). (Sub)classes could be stereotyped with one of Des\(_1\), Des\(_2\) and Des\(_3\) to denote which design to use. An automatic transformation is executed and uses the stereotypes to perform the right operation. This is a semi-automatic transformation where user decisions and automatic processing are done in separate phases.

The stereotypes (decisions) are preserved across modifications to the model. However, there is no indication of which decisions are still valid. Informal instructions can be added to help identify the affected decisions, but that still requires manual effort. It is also not possible to say, after the fact, which source model elements affected the decision and thus traceability is lost at decision points.

In this small example, in the initial transformation there were 19 decisions for the transformation engineer. In the transformation carried out after the ER model evolution there are 20 decisions. Using the decision context models, 13 out of these 20 decisions can be associated with a decision from the initial transformation and thus reused. That leaves seven decisions, each related to the changed region of the model, e.g. the new class Person.

Many of the decisions in the transformation, such as creating a column based on an attribute, require only a small effort. It may seem that the effort saved in creating a column is insignificant. However, it should be noted that the amount of decision tasks that have to be performed in the maintenance transformation depends on the changes to the source model, not the size of the model. If the changes were the same as above, but the initial transformation consisted of 100 decision tasks, there would still only be seven decision tasks to perform.
case with maintenance activities. If the actual source model change and the changes it causes in the target model are small in relation to the models’ sizes, verifying valid decisions could take more time than making new decisions.

Modeling the decision context and automatically reusing decisions removes some of the unnecessary manual decision validity checks. This lightens the maintainer’s work load when propagating changes and preserves traceability.

7 Discussion

Model transformations play a key role in model-driven software development, reverse engineering, and software analysis activities. Depending on the scope and purpose of the transformations, the degree of automation varies. Automatic methods have been proposed and used in certain limited scope in model-driven software development applications, while in reverse engineering and software analysis the transformations can be partly or even totally manual. Semi-automated and interactive approaches seem to provide a more flexible and adaptable solution and in an optimal case combine the benefits of fully automatic and manual approaches. However, to achieve maintainability and reusability in later model transformation activities, decisions made by the user need to be preserved and saved. In this paper we have proposed a way to model user’s decisions and the decision context during interactive model transformations. The decision context includes the source model elements that affected the decisions.

As part of our future work, we aim to develop the mechanism and its implementation further. Decision recovery and reuse should be enabled across more than one model change. It might be possible to replace the current graph rewrite engine with a more efficient and robust third-party implementation. We are also interested in exploring how the evolution of transformations themselves could be facilitated simultaneously with the evolution of the system under construction or maintenance.

8 Acknowledgments

This research has been financially supported by Finnish Funding Agency for Technology and Innovation (TEKES), Nokia, TietoEnator, Plenware, John Deere, and Solita.

References