# Towards Model Round-Trip Engineering: An Abductive Approach

[1] School of Information Technology
Queensland University of Technology, Brisbane, Australia
t.hettel@student.qut.edu.au, k.raymond@qut.edu.au
[2] SAP Research, CEC Brisbane, Australia
t.hettel@sap.com
[3] The Australian E-Health Research Centre,
CSIRO ICT Centre, Brisbane, Australia
michael.lawley@csiro.au

**Abstract.** Providing support for reversible transformations as a basis for round-trip engineering is a significant challenge in model transformation research. While there are a number of current approaches, they require the underlying transformation to exhibit an injective behaviour when reversing changes. This however, does not serve all practical transformations well. In this paper, we present a novel approach to round-trip engineering that does not place restrictions on the nature of the underlying transformation. Based on abductive logic programming, it allows us to compute a set of legitimate source changes that equate to a given change to the target model. Encouraging results are derived from an initial prototype that supports most concepts of the Tefkat transformation language.

## 1 Introduction

In the vision of model-driven software development, models are the prime artefacts. They undergo a process of gradual refinement turning high level descriptions of a system into detailed models and finally code. As part of this process, models are translated to various modelling languages most appropriately expressing important concepts of particular abstraction layers and from certain perspectives. Once models have been translated, the results are subject to revision and there is no easy way to reflect changes back to their original source. However, propagating changes is indispensable in order to keep the interconnected mesh of models in a consistent state.

While there are many different approaches to model synchronisation, they place restrictions on the underlying transformation. Generally, transformations are required to exhibit some injective behaviour such that unique source models can be found for each and every change to the target model. However, there are many practical transformations where such an injective behaviour is not achievable (e.g., see Sec. 3.1) as important information is discarded in the transformation process and not encoded in the target model. In general there are

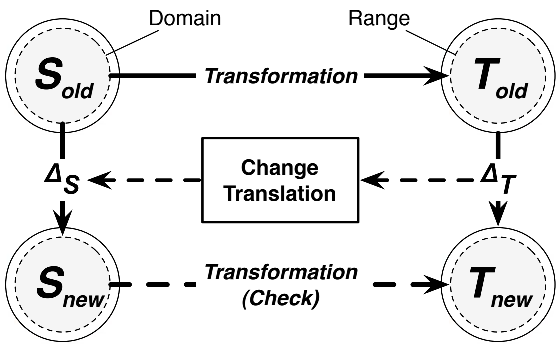

**Fig. 1.** Synchronisation through change translation: changes to the range in model $T$ are translated into corresponding changes to the domain in $S$

often many different ways to reflect changes to the target in terms of changes to the source model and no simple decision can be made to prefer one over another.

To cope with such scenarios, this paper presents an approach to model round-trip engineering (RTE) based on unidirectional, non-injective transformations. Borrowing from abductive reasoning, a number of different source changes can be computed that all equate to the desired target change. Building upon our previous work on the formal foundations of model synchronisation [1], changes performed on the target model are translated into changes to the source. Target changes can either be relevant, manipulating the range of the transformation or irrelevant, in which case the synchronised state is not impaired and no change translation is necessary. In the following (w.l.o.g.) we only consider relevant changes. Translating changes must ensure that applying the transformation to the changed source yields exactly the range (relevant part) of new target model (Fig. 1). No other changes, called side effects, are permissible. In this paper we present an implementation of this change translation function.

The remainder of this paper is structured as follows. Sec. 2 introduces the concept of abductive reasoning and outlines how it can be leveraged to solve the round-trip engineering problem. Illustrating our approach, Sec. 3 introduces the Tefkat model transformation language alongside a running example that is used throughout this paper. The main contribution, Sec. 4, details how the idea of abduction together with other techniques can be used to effectively reverse a unidirectional transformation based on Tefkat to synchronise two models. The presented ideas are then compared to related work in Sec. 5. Concluding, we summarise and discuss our findings and provide an outlook to future work.

## 2 Abduction and RTE

Abduction was introduced as an epistemological theory to scientific knowledge acquisition by C.S. Pierce [2]:

*"The surprising fact, C, is observed. But if A were true, C would be a matter of course. Hence, there is reason to suspect that A is true."*

Arriving at $A$ is the process of abduction also paraphrased as the "*inference to the best explanation*". To illustrate this, consider the abductive feat achieved by Johannes Kepler (1571–1630). He noticed that Mars' orbit around the sun did not comply to a circular trajectory that had been attributed to planetary motion. After years of studying planetary motion and generalising from Mars to all planets[1], he came up with his theory that planets follow an elliptical trajectory around the sun rather than a circular motion. Kepler's laws of planetary motion still hold today and correctly describe the orbits of planets and comets discovered long after Kepler's death.

Due to Pierce's broad definition of abduction, contrasting it against induction is difficult and largely depends on its concrete interpretation. Some philosophers regard abduction as a special case of induction. Others maintain it is more general and subsumes induction [3].

## 2.1 Abductive Logic Programming

In this paper we adopt the much narrower interpretation of abduction as pursued by the logic programming and artificial intelligence community [4,5]. The main difference is that an existing, possibly incomplete, but fixed background theory is required. By making certain assumptions, the abduction process can complete this theory so as to provide an explanation for an observed phenomenon. These incomplete parts of the theory, for which it is not known whether they hold true or not, are called *abducibles.*

Formally speaking, an abductive framework is a triple $(P, A, I)$, where

- $P$ is the program (or theory), a set of logic implications;
- $A$ the set of abducibles, predicates used in $P$ that can be assumed as required; and
- $I$ the set of integrity constraints over predicates in $A$.

In order to explain an observed phenomenon $Q$, the abductive query, a hypothesis $H \subset A$ is sought such that:

- $H \cup P \models Q$, the hypothesis applied to the program explains the observation;
- $H \cup P \models I$, the hypothesis and the program comply with the integrity constraints; and
- $H \cup P$ *is consistent*, i.e., the hypothesis does not contradict the program.

To arrive at an explanation, the abductive logic programming (ALP) proof procedure [6] can be applied. It leverages an algorithm similar to backwards chaining, which is for instance used in Prolog implementations. An abductive explanation for $Q$ is produced by unfolding it against the logic program or theory $P$. If the

[1] This generalisation is not obvious as the orbits of the other known planets are less eccentric and could crudely be approximated by a circle.

procedure encounters an abducible, it is assumed as required (i.e. such that $Q$ succeeds) and the integrity checking phase is entered to verify that the hypothesis does not violate the constraints. While doing so, other abducibles may be encountered, assumed and checked as well. Eventually, the proof procedure will terminate with a set of hypotheses that all constitute legitimate explanations for $Q$ with respect to the aforementioned criteria. However, it may also happen that no explanation can be found. In this case, the observation $Q$ cannot, under no legitimate assumptions, be explained by the theory $P$.

## 2.2 Reversing and RTE as an Abductive Problem

The idea of abduction can be applied to reverse model transformations. This is achieved by interpreting the new target model as the observed phenomenon $Q$, which should be explained in terms of the transformation that corresponds to the program $P$ by hypothesising about the existence of source model elements, which correspond to $H$ and $A$ respectively. Possible explanations are constraint by the source meta-model in terms of the defined type hierarchy and cardinality and nature of references (association vs. aggregation).

By extending the previous interpretation, also incremental RTE scenarios, which are our primary concern, can be covered as illustrated in Fig. 2. Therefore, we assume the old source and target models are accessible and only changes need to be propagated. Moreover, it is assumed that the old target model is the result of applying the transformation to the old source. In this case, the program $P$ corresponds to the old source and target models, the trace connecting both and the transformation producing the new target from the new source. Changes to the target are represented by the observation $Q$ and are explained in terms of source changes as part of the hypothesis $H$. As aforementioned, explanations have to comply with the integrity constraints $I$, which are mainly derived from the meta-model. This rather abstract interpretation of RTE as an abductive problem will be further refined and elaborated on in Sec. 4.

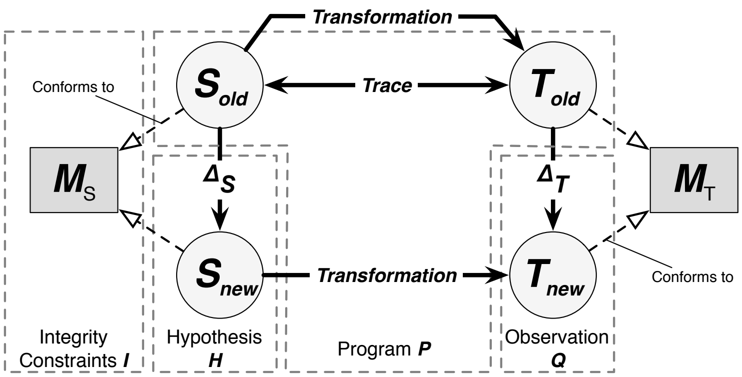

**Fig. 2.** Model round-trip engineering interpreted as an abductive problem. $M_S$ and $M_T$ represent the meta-models of source model $S$ and target model $T$ respectively.

## 3 Tefkat

To illustrate how abduction can be leveraged to facilitate model synchronisation based on a unidirectional, non-injective transformation between two models, the Tefkat transformation language [7] is used. It provides a rule-based and declarative way to specify transformations. Guaranteeing confluence of the transformation result, rules are automatically scheduled by the engine.

```
RULE   class2table
FORALL Class c
WHERE  c.persistent AND c.name = n
MAKE   Table t FROM t4c(c)
SET    t.name = append("tbl",n);

PATTERN hasAttribute(c,a)
FORALL  Class c2
WHERE   c.ownsAttr = a
OR      (c.super = c2
AND     c2.ownsAttr = a);
```

```
RULE   attr2col
FORALL Class c, Attribute a
WHERE  hasAttribute(c,a)
       AND c.persistent
       AND a.name = n
MAKE   Table t FROM t4c(c),
       Column col FROM col(c,a)
SET    t.cols = col,
       col.name = n;
```

**Fig. 3.** Model transformation rules given in Tefkat [7] for mapping UML class diagrams onto relational database schema

Facilitating reuse of transformation fragments, Tefkat offers rule inheritance and a concept called `PATTERN` for reusing pattern definitions. Rules consist of 3 parts (refer to Fig. 3 for examples). The source pattern (`FORALL` and `WHERE`), the target pattern (`MAKE` and `SET`) and the trace (`FROM`) connecting both. Elements that are matched as part of the source pattern can be used to uniquely identify target elements through a function[2]. Using the same function in different rules makes sure that a target object is only created once and can subsequently be reused in other rules. For instance in the example depicted in Fig. 3 the function `t4c(c)` in the `MAKE` statement of rule `class2table` creates one `Table` per `Class c`. The same function is reused in rule `attr2col` for adding `Columns` to `Tables`. In other words: `Class c` together with the function *t4c* uniquely identifies `Table t`. This mapping produced during the transformation process is stored in the trace, which can be queried for relations between source and target model.

### 3.1 Running Example

To illustrate the concepts introduced in the paper the following running example is used, which is based on the popular UML to relational-database-schema mapping. The transformation (see Fig. 3) is not injective as there are at least two different class diagrams that equate to the database schema depicted in Fig. 4. One is also depicted in Fig. 4, another can be derived by flattening the class hierarchy and effectively moving the *name* attribute to *Student* and *Staff*.

[2] This does not mean that Tefkat is restricted to injective transformations in any way. Rather the identity of target elements is restricted to one particular set of source elements to allow different rules to define different aspects of one target element.

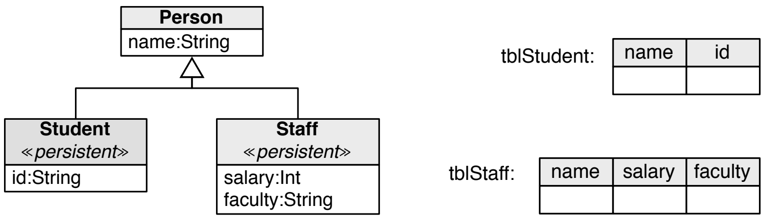

**Fig. 4.** A simple UML class diagram and the corresponding relational database schema with respect to the transformation depicted in Fig. 3

## 4 Reversing Transformations

There are a number of steps involved in reversing transformations. The following subsections elaborate on each of these steps.

### 4.1 Logic Programming Representation

Employing abductive logic programming to solve the RTE problem requires that models and transformation are represented in terms of first-order logic constructs. The source and target models are encoded using three different predicates for instances, attributes and references. One further predicates is needed to encode the trace:

- $inst(o, t)$, where $o$ is a unique object identifier and $t$ refers to a specific type;
- $attr(o, a, v)$, where $o$ identifies the object, $a$ the attribute and $v$ the value;
- $ref(o_1, r, o_2)$, where $o_1$ is the source object and $r$ the reference pointing to object $o_2$; and
- $trace(i, t, s)$ where $i$ is the name of the uniqueness function, $t$ the target element uniquely identified by $i$ and the source elements $s$.

The following predicates encode (parts of) the UML class diagram as depicted in Fig. 4.[3] In the following, `atoms` are denoted by `type writer font`, whereas *variables* are denoted by *italics*.

$$\begin{array}{ll}
inst(\texttt{person}, \texttt{class}), & ref(\texttt{person}, \texttt{attributes}, \texttt{name}), \\
attr(\texttt{person}, \texttt{name}, \text{‘}\texttt{Person}\text{’}), & inst(\texttt{student}, \texttt{class}), \\
inst(\texttt{name}, \texttt{attribute}), & ref(\texttt{student}, \texttt{super}, \texttt{person}), \\
attr(\texttt{name}, \texttt{name}, \text{‘}\texttt{name}\text{’}), & \ldots
\end{array}$$

Transformation rules can be interpreted as logic implications of the form $Tgt(Y)$, $Trace(X, Y) \leftarrow Src(X)$ where $Src$ is the source pattern and $X$ the elements matched by it. $Trace$ represents the trace part uniquely mapping source elements $X$ to target elements $Y$ and $Tgt$ the target pattern to be established.

[3] Please note that the object identifiers were chosen to improve readability, but can be completely arbitrary as far as abduction logic programming is concerned.

For instance, the `class2table` transformation rule can be represented as follows:

$$inst(t, \texttt{table}), attr(t, \texttt{name}, n), trace(\texttt{t4c}, t, c) \leftarrow inst(c, \texttt{class}), attr(c, \texttt{name}, n), attr(c, \texttt{persistent}, \texttt{true}).$$

Employing ALP requires that there is only one head predicate. Therefore, the target pattern is collapsed into a single predicate and each rule is split in two parts: the source part, which is amenable to abduction, and the target part, which is used to match target patterns. Trace is included in the source part so that changes to it can be abduced. Both parts are discussed in the following sections. For instance, the rule `class2table` can be represented as follows:

Source part: $class2table(t, n) \leftarrow inst(c, \texttt{class}), attr(c, \texttt{persistent}, \texttt{true}), attr(c, \texttt{name}, n), trace(\texttt{t4c}, t, c)$

Target part: $class2table(t, n) \leftarrow inst(t, \texttt{table}), attr(t, \texttt{name}, n).$

## 4.2 Matching Target Patterns

Transformation rules generally match source patterns and then create the corresponding target pattern. Considering only one application of one rule, say `attr2col`, results in one instance of the target pattern being created. Removing only parts of that pattern–only the column for instance–is not possible. There is no way, the aforementioned rule can produce an isolated column. Only removing (or creating) the whole target pattern constitutes a legitimate target change. Therefore, changes can only be propagated on a target pattern basis. Either the whole pattern is created or deleted.

Usually, target patterns overlap to produce an interconnected network for objects (see Fig. 5), rather than unconnected islands of target pattern instances. For example, consider the transformation in Fig. 3. Assume there is one persistent class with one attribute. Applying the transformation produces one table with the corresponding name through rule `class2table`. Rule `attr2col` (re-) creates the same table and adds a column to it. Through the overlapping of both target patterns, the table is now supported by both rules whereas the column is only supported by one.

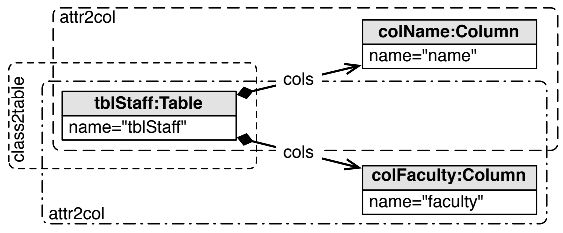

**Fig. 5.** Object diagram showing overlapping target patterns

This overlapping effectively allows the removal of parts of target patterns, but not arbitrary target elements. For instance, removing the column and retaining the table in the previous example is a legitimate change, even though only a part of the `attr2col` target pattern is removed. The other part, namely the table, is still supported by `class2table`. Changing the source model such that `attr2col` cannot be applied any more results in the deletion of the target instances that were solely supported by this particular rule application. Other target elements, supported by other rules, still remain.

Not only target patterns can be overlapping, but also source patterns. This is also the case in our previous example. The same class is matched by both rules. Therefore, solutions to deleting the column may also accidentally delete the table, which, however, should be retained. Consider a source change where the class is deleted, which results in the deletion of the column *and* the table. To prevent such side effects, overlapping patterns can be actively retained by "observing" their insertion. With respect to the example, deleting the column requires the whole `attr2col` target pattern to be deleted and the table to be retained by (re-)inserting `class2table`.

As changes can only be propagated on a target pattern basis, individual changes need to be coalesced. This is done by matching the target patterns against the old and new target model and requiring that with each pattern at least one change (deletion or insertion) is matched. Based on these matches it can be determined which patterns need to be deleted or inserted. For pattern matches where only some of the elements were subject to deletion, all other elements need to be supported by other rules and actively retained, as discussed before.

## 4.3 Formulating the Abductive Query

Based on the target patterns that need to be created or removed, the abductive query can be formulated. This step is straightforward for deletions in which case the corresponding source pattern can be looked up in the trace.

For insertions, a more complex process is necessary. Target patterns may be overlapping and even different rules can have the same target pattern. Therefore, it may be possible that not all source patterns can be created to support all target pattern matches. For instance, consider the following transformation:

```
RULE R1                RULE R2
FORALL X x             FORALL Y y
MAKE Z z FROM f(x)     MAKE Z z FROM g(y)
```

Assume a new instance of $Z$ was created in the otherwise empty target model and this change is now propagated. Both target patterns match this change. However, as they use different uniqueness functions ($f(x)$ in `R1` and $g(y)$ in `R2`) only one of the rules can support the new $Z$. There is an exclusive choice to be made. Even more complex scenarios are possible where there are two (or more) sets of rules covering the same target patterns. Again, not all of them can and have to support the target patterns.

Essentially, the abductive query is a conjunction of disjunctions of rules, where all variables are bound to target elements. Disjunctions of transformation rules represent the different alternatives. For propagating pattern deletions, rules are negated (e.g. $\neg attr2col(\texttt{tblStudent}, \texttt{colName}, \texttt{`name'})$) to require a set of source changes that explain the absence of this particular pattern match in the new target model. Positively mentioned (not negated) rules lead to explanations in terms of source changes as to the pattern's insertion in the new target model.

Consider our previous example where there was an alternative between rules `R1` and `R2` to support a new instance of `Z`, say `newZ`. The abductive query $Q$ can be formulated as follows:

$$Q_1 = R1(\texttt{newZ}) \vee R2(\texttt{newZ}).$$

In other words, an explanation is sought for the existence of `newZ` either through applying `R1` or `R2`.

Consider the running example (transformation Fig. 3, instances Fig. 4). Assume column `name` in table `tblStudent` was deleted. This results in the deletion of rule $attr2col(\texttt{tblStudent}, \texttt{colName}, \text{`}\texttt{name}\text{'})$. As discussed before, in order to retain table `tblStudent` and column `id` they have to be actively retained by re-inserting their supporting rules into the query:

$$\begin{aligned} Q_2 =& \neg attr2col(\texttt{tblStudent}, \texttt{colName}, \texttt{`name'}) \wedge \\ & attr2col(\texttt{tblStudent}, \texttt{colId}, \texttt{`id'}) \wedge class2table(\texttt{tblStudent}, \texttt{`Student'}) \end{aligned}$$

## 4.4 Abducing Source Changes

With the abductive query in place, this section now focuses on the computation of the corresponding source changes that provide an explanation to the query. Recall that an abductive framework is a triple $(P, A, I)$, consisting of the program or theory $P$, the set of abducibles $A$, and integrity constraints $I$. Moreover, there is an abductive query $Q$, which was discussed before and a set of hypotheses $H$ explaining $Q$.

As part of the program $P$, a representation of the original (or old) source model is required. As introduced in Sec. 4.1 three predicates are used to encode instances ($inst$), references ($ref$) and attributes ($attr$) as parts of the source model. A forth predicate ($trace$) represents the trace connecting source elements with their corresponding target elements based on the `FROM`-statements in the transformation. To distinguish between the old and new source model, the aforementioned predicates are prefixed with **old** and **new** respectively.

Since the new source model is not known, it is formulated in terms of changes (insertions **ins** and deletions **del**) to the old source model:

$$\begin{aligned} \textbf{new}\ inst(c,t) \leftarrow&\ \ \textbf{old}\ inst(c,t), \neg\textbf{del}\ inst(c,t) \\ \textbf{new}\ inst(c,t) \leftarrow& \neg\textbf{old}\ inst(c,t),\ \ \ \textbf{ins}\ inst(c,t) \end{aligned}$$

with similar rules for attributes, references and trace. In words: instances (attributes, references, or trace) are in the new model, if they are in the old and have not been deleted, or if they are not in the old model but have been inserted.

In terms of abductive logic programming, the changes to the source model is the part of the theory or program that is incomplete. It is not known whether a particular instance was inserted or deleted. However, the abductive proof procedure has the freedom to hypothesise about this in order to account for new elements in the target model. The old source model is given and must not be changed. Formally speaking, the predicates prefixed with **ins** or **del** in the above rules are the abducibles in $A$, which can be assumed as required.

To complete the abductive program $P$, the transformation has to be included. As introduced in Sec. 4.1 the transformation rules can be written as logic implications, formulated over the predicates that make up the *new* source model. Hence, all predicates referring to source elements have to be prefixed with **new**. As the query $Q$ is formulated over rules rather than individual target elements, only the source side and the trace of the transformation rules are of interest:

$$P = \begin{cases} \mathit{class2table}(t,n) \leftarrow \textbf{new}\ \mathit{inst}(c,\texttt{class}), \textbf{new}\ \mathit{attr}(c,\texttt{persistent},\texttt{true}), \\ \qquad \textbf{new}\ \mathit{attr}(c,\texttt{name},n), \textbf{new}\ \mathit{trace}(\texttt{t4c},t,c). \\ \mathit{attr2col}(t,\mathit{col},n) \leftarrow \textbf{new}\ \mathit{inst}(c,\texttt{class}), \textbf{new}\ \mathit{inst}(a,\texttt{attribute}), \\ \qquad \textbf{new}\ \mathit{attr}(c,\texttt{persistent},\texttt{true}), \mathit{hasAttribute}(c,a), \\ \qquad \textbf{new}\ \mathit{attr}(a,\texttt{name},n), \textbf{new}\ \mathit{trace}(\texttt{t4c},t,c), \\ \qquad \textbf{new}\ \mathit{trace}(\texttt{col4attr},\mathit{col},[c,a]). \\ \mathit{hasAttribute}(c,a) \leftarrow \textbf{new}\ \mathit{inst}(c,\texttt{class}), \textbf{new}\ \mathit{ref}(c,\texttt{ownsAttr},a). \\ \mathit{hasAttribute}(c,a) \leftarrow \textbf{new}\ \mathit{inst}(c,\texttt{class}), \textbf{new}\ \mathit{ref}(c,\texttt{super},\mathit{sc}), \\ \qquad \textbf{new}\ \mathit{ref}(\mathit{sc},\texttt{ownsAttr},a). \end{cases}$$

To complete the abductive framework, a set of integrity constraints has to be provided, which is necessary to get sensible answers from the system. These integrity constraints are made up of three parts.

– Constraints concerning the uniqueness of the trace, i.e., there must be exactly one set of source elements giving rise to one target element, regardless of the function used.
– Constraints on the usage of **ins** and **del**, i.e. elements cannot be inserted and deleted at the same time. Moreover, only existing elements can be deleted and only non-existing elements can be inserted.
– Constraints concerning the types of elements, cardinality of references, containment, as derived from the meta-model.

With this set of rules given, the abductive query can be evaluated. This happens in a fashion similar to backward chaining. However, abduction has the freedom to assume the abducibles (the source changes) as required to succeed the query. If the query cannot be succeeded, changes made to the target are not valid and no source changes exist such that the desired target model can be produced by applying the transformation. To avoid littering the source model with excess elements, only minimal source changes are sought.

To illustrate the abduction process, consider the deletion of `name` column in `tblStudent`. As per previous discussions, the corresponding query equates to

$$Q = \neg \mathit{attr2col}(\texttt{tblStudent},\texttt{colName},\texttt{'name'}) \wedge \\ \mathit{attr2col}(\texttt{tblStudent},\texttt{colId},\texttt{'id'}) \wedge \mathit{class2table}(\texttt{tblStudent},\texttt{'Student'})$$

The proof procedure now unfolds the query against $P$ (see previous page) and first tries to fail $attr2col$(`tblStudent`, `colName`, `'name'`). Therefore, it tries to fail **new** $attr$(`student`, `persistent`, `true`) by assuming the class was non-persistent (**del** $attr$(`student`, `persistent`, `true`)), which is an abducible. Now the integrity checking phase is entered, which declares this solution to be legitimate. However, advancing other parts of the query will rule out this solution as it cannot explain $class2table$(`tblStudent`, `'Student'`). There are more potential solutions that need to be explored. Another alternative is to fail $hasAttribute$(`student`, `name`) by assuming **del** $ref$(`student`, `super`, `person`). Again integrity check declares this assumption to be viable and indeed it sustains advancing all other parts of the query. The proof procedure will continue unfolding all parts of the query, explore all alternatives and arrive at a set of alternative source changes.

## 4.5 Compensating Side Effects

Abductive explanations (source changes) may have different qualities when transformed back to the target side. All will explain the observation, i.e., perform the desired target change. However, some will do more and inflict further changes on the target model. Assume that for some reason the `name` column in `tblStudent` is to be removed. The corresponding alternative source changes proposed by the abduction process are:

- $H_1$ = {delete attribute `name` in `Person`},
- $H_2$ = {delete `Person`},
- $H_3$ = {delete inheritance relationship between `Person` and `Student`}.

The first two changes have side effects and cause `tblStaff` to lose its `name` column. One solution seems to be to add additional constraints to $I$ that reject these solutions. However, this proves to be too short-sighted as possible solutions may be overlooked. By temporarily accepting these side effects, but requiring their compensation by insisting on reinserting the deleted elements, new solutions can be generated. These new solutions are super-sets of the previous solution. In a new abductive query an explanation is then sought for the observation that `tblStudent` does not have a `name` column any more but `tblStaff` still has.

With this extended query, the system comes up with the following suggestions:

- $H'_1$ = {delete inheritance relationship between `Person` and `Student`},
- $H'_2$ = {move attribute `name` from `Person` to `Staff`},
- $H'_3$ = {introduce a new, non-persistent class, move `name` there and make `Staff` a subclass of the new class}

All of these changes exactly perform the requested target change and therefore do not exhibit any side effects when transformed to the target model. Note that we have no basis for choosing any one of these solutions as "superior" but the user might have some criteria unknown to the tool.

### 4.6 Implementation

In our first attempt to implement the outlined procedure for synchronising models we experimented with ProLogICA [8], which is a direct implementation of the proof procedure in Prolog. It can essentially be applied to any Prolog program. While it was easy to use and the rules needed were essentially the rules outlined above, we quickly run into performance issues and answers took too long to compute, making it infeasible for even quite small transformations and models. Moreover, it required a set of "blank" object identifiers for use in creating new instances of types. This resulted in an combinatoric explosion of isomorphic solutions, where the only difference was the object identifier used to create a new instance of a type.

Our second and current attempt is based on constraint handling rules (CHR) as suggested by Abdennadher and Christiansen [9]. CHR implementations are readily available in most Prolog environments. Given a set of re-writing rules, constraints are rewritten until *false* was produced or no rules are applicable any more. In this case the set of remaining constraints constitute the answer. Models are encoded as aforementioned but in addition are "closed" by a constraint prohibiting the creation of new facts through rewriting. Moreover, modifications to the transformation rules were necessary as CHR does not support negation as failure. Instead explicit negation had to be used, which required splitting the transformation rules in two parts. One for inserting and one for deleting. Even though there was a larger number of rules involved to encode the RTE problem, solutions were produced much more quickly. Moreover, using CHR also allows us deal with attribute value manipulation, such as adding numbers, concatenating strings, etc. and comparisons of attribute values in source patterns. When reversing such rules, a number of constraints can be provided restricting possible values.

Even though the CHR-based approach seems to be far away from how the abductive proof procedure works, it is in fact not that different. Essentially, the same steps are executed but not necessarily in the same order. Queries are still unfolded against the program. This unfolding, however, can be advanced in a more breadth-first manner, building up constraints for explanations quickly, rather than traversing the search space in a strictly depth-first fashion. We believe that this together with the fact that CHR implementations are mature and directly translated, optimised and executed in Prolog, rather than executing an ALP-interpreter, account for the big difference in performance.

## 5 Related Work

There are a number of existing approaches to model synchronisation and round-trip engineering, imposing different restrictions on the underlying transformations. In general it is required that there is a one-to-one relationship between source and target changes. How this is achieved depends on the concrete approach.

There some approaches centred around a set primitive of injective functions that can be combined to produce more complex transformations. These are guaranteed to be injective and can be easily reversed [10,11]. Injective functions, however, are quite limited and not even arithmetic operations can be used. To lift restrictions on the transformation, Foster et al [12] present an approach based on so-called lenses; pairs of functions defining the forward and the reverse transformation. The forward function solely works on the source model and produces the target model. Conversely, the reverse uses the old source model and the new target to produce the new source model. Still, target changes together with the old source model have to uniquely identify the new target model.

Yet another approach [13] that considers three models for synchronisation is based on triple graph grammars [14], which govern the co-evolution of source and target models. Any relationship—even non-functional relations—can be specified, but not necessarily executed. Changes are propagated by identifying the matching pattern and then establishing or invalidating the corresponding pattern in the other model. When invalidating patterns, all elements are deleted that do not partake in another pattern match. Transformations are not required to be bijective on an element level in order to be usable for this synchronisation approach, as shown by Ehrig et al [15]. However, there must be a one-to-one relationship between source and target patterns.

Fewer restrictions on the nature of the transformation are imposed by the approach presented by Cicchetti et al [16]. It allows for non-injective partial transformations. The reverse transformation, specified by the user, may not be a function and hence there may be several source models for a given target model. However, there is no way to ensure that the provided reverse is reasonable in the sense that when transformed forward again all sources result in the changed target model. Moreover, round-trips without any changes produce all possible source models rather than just the original one.

Query/View/Transformation (QVT) [17] is a recent standard for model transformation, which allows the declarative definition of relationships between source and target models. Relations between models can be checked or enforced in both directions. There is no restriction on the nature of these relationships. They do not have to be one-to-one but can also be many-to-one or even many-to-many in one or the other direction. In other words, there may be more than one source model that corresponds to a given target model and vice versa. This is very similar to the problems discussed in this paper. In fact, QVT model transformation and RTE based on QVT can also be understood as an abductive problem, analogous to Sec. 2.2. We are certain that our technique can also be applied in the context of QVT to support more than just the one-to-one relationships between models.

Not directly related to model synchronisation, is an approach by Varró and Balogh [18] where they propose a model transformation approach based on an inductive learning system. By providing pairs of corresponding models, the inductive logic programming systems derives a set of rules that transform one model into another. As pointed out earlier, there is a close relationship between abduction and induction and therefore, there are also parallels between Varro and Balogh's approach on our approach. However, the premises are different. We consider the theory (transformation) as give and immutable, and derive new source models, as opposed to considering the models as immutable and trying to derive a theory. Self-evidently, both approaches could be combined to result in "synchronisation by example", where the system tries to derive rules based on pairs of source and target changes chosen by a user. This, however, presupposes that all information required to make a choice is contained in the models and the transformation. There may be scenarios where this is the case. As far as our running example is concerned, this does not hold. Whether to delete a class or to mark it non-persistent so as to delete a table is nothing that can be derived from any of the models or the transformation. It rather depends on the meaning a user assigns to the class in question and therefore is not amenable to logic programming.

## 6 Conclusion and Future Work

In this paper we presented a novel approach to model round-trip engineering based on abductive logic programming. Abductive reasoning, the inference to the best explanation, allows us to compute hypotheses that together with a theory explain an observed phenomenon. We showed how RTE can be interpreted as an abductive problem. Changes to a target model represent the phenomena for which a set of source changes is hypothesised that account for the target changes with respect to the transformation. This was operationalised by translating the transformation into first-order logic with rules of the form "*source pattern* implies *target pattern*". Models were represented using predicates for instances, attributes and references. Abductive reasoning can then be applied to this first-order-logic program to result in a set of source changes performing the desired target change. While performing the target change, the proposed source changes may inflict further changes, side effects, on the target model. In this case compensation can be applied to avoid side effects and arrive at more solutions. The presented techniques are implemented in Prolog using constraint handling rules (CHR), which also allow for dealing with attribute value comparisons in the sense, that solutions may contain a number of constraints restricting attribute values.

With the proposed techniques most of Tefkat's features can be reversed. This includes negation, `LINKS/LINKING`s and `PATTEN`s. Features that are not yet supported are recursive `PATTEN`s or reflection. The presented ideas, however are not limited to Tefkat. Also model synchronisation based on QVT or triple graph grammars, which both allow the specification of non-functional relations, can be interpreted as abductive problems.

Abduction was paraphrased as "*inference to the* ***best*** *explanation*" and therefore needs a way to assess the quality of the produced solution. Based on this a list of likely or recommended solutions could then be presented to user or picked automatically. We have investigated very simple heuristics based on the size of the proposed changes, which worked surprisingly well. However, further investigations are required. This heuristic could be directly integrated into the abduction mechanism such that each choice-point creates a backlog of solutions, while only the "best" solution is explored further. Such a merge of abduction and A* search is subject to future work to improve performance and find "good" solutions quickly. Furthermore, Hearnden et al's [19] approach to incrementally propagating source changes to the target model of the transformation, could prove beneficial for incrementally checking for side effects of proposed source changes.

## References

1. Hettel, T., Lawley, M., Raymond, K.: Model synchronisation: Definitions for round-trip engineering. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 31–45. Springer, Heidelberg (2008)
2. Pierce, C.S.: Collected Papers of Charles Sanders Peirce, vol. 2. Harvard University Press, Cambridge (1931-1958)
3. Aliseda, A.: Abductive Reasoning: Logical Investigations Into Discovery and Explanation. Springer, Heidelberg (2005)
4. Kakas, A., Denecker, M.: Abduction in Logic Programming. Computational Logic: Logic Programming and Beyond, 402–436 (2002)
5. Kakas, A., Kowalski, R., Toni, F.: Abductive Logic Programming. Journal of Logic and Computation 2(6), 719–770 (1993)
6. Kakas, A., Mancarella, P.: Generalized Stable Models: A Semantics for Abduction. In: Proceedings of the 9th European Conference on Artificial Intelligence, ECAI 1990, Stockholm, Sweden, pp. 385–391 (1990)
7. Lawley, M., Steel, J.: Practical Declarative Model Transformation with Tefkat. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)
8. Ray, O., Kakas, A.: ProLogICA: a practical system for Abductive Logic Programming. In: Proceedings of the 11th International Workshop on Non-monotonic Reasoning (2006)
9. Abdennadher, S., Christiansen, H.: An Experimental CLP Platform for Integrity Constraints and Abduction. In: Proceedings of FQAS 2000, Flexible Query Answering Systems: Advances in Soft Computing series, pp. 141–152 (2000)
10. Mu, S.C., Hu, Z., Takeichi, M.: An Injective Language for Reversible Computation. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 289–313. Springer, Heidelberg (2004)
11. Mu, S., Hu, Z., Takeichi, M.: An Algebraic Approach to Bi-directional Updating. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 2–20. Springer, Heidelberg (2004)
12. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem. ACM Transactions on Programming Languages and Systems (2007)
13. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)

14. Königs, A.: Model transformation with triple graph grammars. In: Proceedings of the Model Transformations in Practice Satellite Workshop of MODELS 2005 (2005)
15. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)
16. Cicchetti, A., Ruscio, D.D., Eramo, R.: Towards Propagation of Changes by Model Approximations. In: Proceedings of the 10th International Enterprise Distributed Object Computing Conference Workshops, p. 24. IEEE Computer Society, Los Alamitos (2006)
17. Object Management Group (OMG) formal/08-04-03: Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification Version 1.0 (November 2005)
18. Varró, D., Balogh, Z.: Automating model transformation by example using inductive logic programming. In: SAC 2007: Proceedings of the, ACM symposium on Applied computing, pp. 978–984. ACM, New York (2007)
19. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 321–335. Springer, Heidelberg (2006)