Constraint-based Specification of Model Transformations

K. Lano, S. Kolahdouz-Rahimi
Dept. of Informatics, King’s College London

Abstract. Model transformations are a central element of model-driven development (MDD) approaches. The correctness, modularity and flexibility of model transformations is critical to their effective use in practical software development. In this paper we describe an approach for the automated derivation of correct-by-construction transformation implementations from high-level specifications. We illustrate this approach on a range of model transformation case studies of different kinds (re-expression, refinement, quality improvement and abstraction transformations) and describe ways in which transformations can be composed and evolved using this approach.

1 Introduction

Model transformations are an essential part of development approaches such as Model-driven Architecture (MDA) [43] and Model-driven Development (MDD) [12]. Model-transformations are becoming large and complex and business-critical systems in their own right, and so require systematic development. In particular there is a need to verify the syntactic and semantic correctness of transformations, and to support evolution required by evolution of the languages which the model transformation operates on.

At present, a large number of different model transformation approaches exist, such as graph transformations (eg., Viatra [45]), declarative (QVT-Relations [41]), imperative (QVT-Operations, Kermeta [21]) and hybrid (ATL [20]) languages [7]. These are all primarily based around the concept of transformation rules, which define individual steps within a transformation process. The overall effect of a transformation is then derived from the implicit (QVT-Relations, ATL) or explicit (Kermeta, QVT-Operations, Viatra) combination of individual rule applications. These descriptions are closer to the level of designs, rather than specifications, and are also specific to particular languages, ie., they are PSMs (platform-specific models) in terms of the MDA.

However for effective verification and reuse of transformations a higher level of structuring and specification is required, to define the complete behaviour of a transformation as a (black-box) process, for example, by pre and postconditions. This level of specification also provides support for the correct external composition of transformations, such as by the sequential chaining of transformations.
In this paper we will describe the following components of a systematic approach for specifying and developing model transformations to address these problems:

- A general model-driven development process for model transformations (Appendix B) based upon a formal semantics for UML and transformations (Appendix A).
- Specification of model transformations using OCL constraints and UML class diagrams (Section 2).
- Automated design and implementation strategies for these specifications. We give proofs of correctness of these strategies (Section 3).
- Structuring and composition techniques for specifications, to enable the effective evolution of transformations in response to evolving metamodels or changes in requirements (Section 4).

To minimise development costs, we implement transformation specifications by the automated derivation of (correct by construction) designs and executable implementations. This process has been implemented as part of the UML-RSDS toolset for MDD using UML. It could also be applied to other model transformation approaches, such as QVT or ATL.

Section 5 evaluates the approach. Section 6 describes related work, and Section 7 summarises the paper.

1.1 Categories of model transformation

Model transformations can be classified in different ways [7, 39]. At a syntactic level, we can differentiate between those transformations where the source and target languages $S$ and $T$ are entirely disjoint, or where they overlap, or where one is a sub-language of the other. Transformations may be update-in-place, i.e., they operate on a single model and produce the target model by modifying elements of the source model, or separate models if the source and target are distinct. The latter are also termed input-preserving if they do not modify the source model. Endogenous transformations have the same source and target language, whilst exogenous transformations have distinct languages [7].

Semantically, a transformation can be classified in terms of its role in a development process:

**Refinement** A transformation that replaces source model elements by target elements or structures to map an abstract model (such as a PIM) to a more specific version (such as a PSM).

**Abstraction** A transformation that produces an abstraction of a model, e.g., by discarding some details of the source model data.

**Quality improvement/Restructuring** A transformation that produces a target model at the same abstraction level as the source, but that re-organises a model to achieve some quality goal (e.g., removing duplicated attributes from a class diagram). Usually these are update-in-place and endogenous.
Re-expression A transformation that maps a model in one language into its equivalent in another language at the same level of abstraction, e.g., migration transformations from one version of a language to another version.

We will consider the following examples of transformations from these categories:

- The well-known UML to relational database refinement transformation [42, 32], and the refinement example of [25].
- A quality improvement restructuring to remove duplicated attributes from a class diagram [23].
- Re-expression transformations to compute the transitive closure of a graph [33] and of an association (Section 4.3).

These will be used to illustrate the development process and transformation patterns.

2 Model transformation specification

In this section and the following section we describe how the general model transformation development process of [32] can be implemented in UML-RSDS. UML-RSDS is a model-driven development approach which has the following general principles:

- Systems are specified using declarative UML models and OCL constraints, at a CIM (computationally independent model) level, where possible.
- Designs and executable implementations are automatically derived by means of verified transformations, so that they are correct-by-construction with respect to the specification. Alternatively, developers can write explicit designs at a PIM level, similar to QVT-Operations or Kermeta in style.
- Capabilities for formal analysis are provided, for use by specialised users.

As an approach to transformation specification, this means that transformations are specified purely using UML notations, with no additional specialised syntax required.

Each transformation $\tau : S \rightarrow T$ from a source language $S$ to a target language $T$ is defined as a UML use case (Chapter 16 of [40]), with a set $\text{Asm}$ of assumptions, which are the preconditions of the use case (i.e., the preconditions of the $\text{BehavioralFeature}$ associated to the use case), and a set $\text{Cons}$ of postconditions. Additional conditions $\text{Ens}$ for the poststate should follow from $\text{Cons}$. Invariants $\text{Inv}$ may be defined for the use case. The predicates $\text{Asm}$, $\text{Cons}$, $\text{Inv}$ and $\text{Ens}$ may involve both languages $S$ and $T$.

Logically, the transformation is interpreted as achieving the conjunction of the postconditions, under the assumption that the conjunction of the preconditions holds at its initiation. Procedurally, the postcondition constraints $\text{Cn}$ can be interpreted as transformation rules or statements $\text{stat}(\text{Cn})$ which establish $\text{Cn}$ (Appendix C).
The precondition constraints $Asm$ define checks which should be carried out on the source model, whilst the postconditions $Cons$ also define consistency conditions that should hold (and be maintained) between the source and target models as a result of the transformation.

The structure and organisation of the constraints will be used to automatically derive the design and implementation of the transformation.

An example of such constraint-based specification is the well-known UML to relational database transformation. This transformation maps a data model expressed in UML class diagram notation into the more restricted data modelling language of relational database schemas. Modelling aspects such as inheritance and associations are removed from the source model and their semantics expressed instead using the language facilities (tables, primary keys and foreign keys) of relational databases.

Figure 1 shows the source (on the left hand side) and target (on the right) metamodels. $Asm$ consists of assertions that attribute names are unique within a class, that class and association names are unique within a package, and so forth [32].

The transformation is defined as a use case operating on instance models of these metamodels. The formal specification $Cons$ of the transformation as a global relation between the source and target languages can be defined by six postcondition constraints of the use case, of which the following three define the basic transformation of attributes to columns and classes to tables:
C1 “For each persistent attribute in the source model there is a unique column in the target model, of corresponding type”:

\[ \forall a: Attribute \cdot a.owner.kind = \text{“Persistent”} \implies \exists cl: Column \cdot cl.rdbId = a.umlId \text{ and } \\
cl.name = a.name \text{ and } cl.kind = a.kind \text{ and } \\
(a.type.name = \text{“Integer”} \implies cl.type = \text{“NUMBER”}) \text{ and } \\
(a.type.name = \text{“Boolean”} \implies cl.type = \text{“BOOLEAN”}) \text{ and } \\
(a.type.name \neq \text{“Integer”} \text{ and } a.type.name \neq \text{“Boolean”} \implies cl.type = \text{“VARCHAR”}) \]

C2 “For each persistent class in the source model, there is a unique table representing the class in the target model, with columns for each owned attribute”:

\[ \forall c: Class \cdot c.kind = \text{“Persistent”} \implies \exists t: Table \cdot t.rdbId = c.umlId \text{ and } t.name = c.name \text{ and } \\
t.kind = \text{“Persistent”} \text{ and } \\
Column[c.attribute.umlId] \subseteq t.column \]

\( E[v] \) denotes lookup of \( E \) elements by their primary key values: \( E[v] \) is the \( E \) instance with primary key value \( v \), if this is a single value, otherwise \( E[v] \) denotes the set of \( E \) instances with primary key values in \( v \).

C5 “If \( c \) is a subclass of \( d \), all columns of \( d \)’s table are included in \( c \)’s table”:

\[ \forall c, d: Class \cdot c.kind = \text{“Persistent”} \text{ and } d: c.general \implies \\
Table[d.umlId].column \subseteq Table[c.umlId].column \]

Constraints C1 and C2 are similar in structure to the corresponding transformation rules of the QVT-Relations specification of this problem [42]: individual elements of the source model are related to corresponding elements in the target model. However, in our specification, the constraints are defining postconditions of the completed transformation, i.e., what relation it should establish, at its termination, between the source and target model data—indeed independently of any design or implementation strategy to enforce this relation. In this respect the constraints are more abstract and declarative than the QVT rules.

Constraint C5 is different in character to the QVT rules, again it expresses a condition on the target model, relative to the source model, but this condition is recursive in style and needs a more complex implementation, using fixpoint iteration. Nevertheless, it can be used as a postcondition of the transformation at the specification level to reason about the effect of the transformation, independently of how it is implemented.

The constraints have a dual aspect: they express what conditions should be true at the completion of an entire transformation, but they can also be interpreted as the definitions of specific rules executed within the transformation. For example, C1 can be operationally interpreted as “For each \( a \) in \( Attribute \) which is owned by a persistent class, create an element \( cl \) in \( Column \), with corresponding identity, name, kind and type”.

Constraints C1 and C2 are similar in structure to the corresponding transformation rules of the QVT-Relations specification of this problem [42]: individual elements of the source model are related to corresponding elements in the target model. However, in our specification, the constraints are defining postconditions of the completed transformation, i.e., what relation it should establish, at its termination, between the source and target model data—indeed independently of any design or implementation strategy to enforce this relation. In this respect the constraints are more abstract and declarative than the QVT rules.

Constraint C5 is different in character to the QVT rules, again it expresses a condition on the target model, relative to the source model, but this condition is recursive in style and needs a more complex implementation, using fixpoint iteration. Nevertheless, it can be used as a postcondition of the transformation at the specification level to reason about the effect of the transformation, independently of how it is implemented.

The constraints have a dual aspect: they express what conditions should be true at the completion of an entire transformation, but they can also be interpreted as the definitions of specific rules executed within the transformation. For example, C1 can be operationally interpreted as “For each \( a \) in \( Attribute \) which is owned by a persistent class, create an element \( cl \) in \( Column \), with corresponding identity, name, kind and type”.

Constraints C1 and C2 are similar in structure to the corresponding transformation rules of the QVT-Relations specification of this problem [42]: individual elements of the source model are related to corresponding elements in the target model. However, in our specification, the constraints are defining postconditions of the completed transformation, i.e., what relation it should establish, at its termination, between the source and target model data—indeed independently of any design or implementation strategy to enforce this relation. In this respect the constraints are more abstract and declarative than the QVT rules.

Constraint C5 is different in character to the QVT rules, again it expresses a condition on the target model, relative to the source model, but this condition is recursive in style and needs a more complex implementation, using fixpoint iteration. Nevertheless, it can be used as a postcondition of the transformation at the specification level to reason about the effect of the transformation, independently of how it is implemented.

The constraints have a dual aspect: they express what conditions should be true at the completion of an entire transformation, but they can also be interpreted as the definitions of specific rules executed within the transformation. For example, C1 can be operationally interpreted as “For each \( a \) in \( Attribute \) which is owned by a persistent class, create an element \( cl \) in \( Column \), with corresponding identity, name, kind and type”.

Constraints C1 and C2 are similar in structure to the corresponding transformation rules of the QVT-Relations specification of this problem [42]: individual elements of the source model are related to corresponding elements in the target model. However, in our specification, the constraints are defining postconditions of the completed transformation, i.e., what relation it should establish, at its termination, between the source and target model data—indeed independently of any design or implementation strategy to enforce this relation. In this respect the constraints are more abstract and declarative than the QVT rules.

Constraint C5 is different in character to the QVT rules, again it expresses a condition on the target model, relative to the source model, but this condition is recursive in style and needs a more complex implementation, using fixpoint iteration. Nevertheless, it can be used as a postcondition of the transformation at the specification level to reason about the effect of the transformation, independently of how it is implemented.

The constraints have a dual aspect: they express what conditions should be true at the completion of an entire transformation, but they can also be interpreted as the definitions of specific rules executed within the transformation. For example, C1 can be operationally interpreted as “For each \( a \) in \( Attribute \) which is owned by a persistent class, create an element \( cl \) in \( Column \), with corresponding identity, name, kind and type”.

Constraints C1 and C2 are similar in structure to the corresponding transformation rules of the QVT-Relations specification of this problem [42]: individual elements of the source model are related to corresponding elements in the target model. However, in our specification, the constraints are defining postconditions of the completed transformation, i.e., what relation it should establish, at its termination, between the source and target model data—indeed independently of any design or implementation strategy to enforce this relation. In this respect the constraints are more abstract and declarative than the QVT rules.

Constraint C5 is different in character to the QVT rules, again it expresses a condition on the target model, relative to the source model, but this condition is recursive in style and needs a more complex implementation, using fixpoint iteration. Nevertheless, it can be used as a postcondition of the transformation at the specification level to reason about the effect of the transformation, independently of how it is implemented.

The constraints have a dual aspect: they express what conditions should be true at the completion of an entire transformation, but they can also be interpreted as the definitions of specific rules executed within the transformation. For example, C1 can be operationally interpreted as “For each \( a \) in \( Attribute \) which is owned by a persistent class, create an element \( cl \) in \( Column \), with corresponding identity, name, kind and type”.

Constraints C1 and C2 are similar in structure to the corresponding transformation rules of the QVT-Relations specification of this problem [42]: individual elements of the source model are related to corresponding elements in the target model. However, in our specification, the constraints are defining postconditions of the completed transformation, i.e., what relation it should establish, at its termination, between the source and target model data—indeed independently of any design or implementation strategy to enforce this relation. In this respect the constraints are more abstract and declarative than the QVT rules.

Constraint C5 is different in character to the QVT rules, again it expresses a condition on the target model, relative to the source model, but this condition is recursive in style and needs a more complex implementation, using fixpoint iteration. Nevertheless, it can be used as a postcondition of the transformation at the specification level to reason about the effect of the transformation, independently of how it is implemented.

The constraints have a dual aspect: they express what conditions should be true at the completion of an entire transformation, but they can also be interpreted as the definitions of specific rules executed within the transformation. For example, C1 can be operationally interpreted as “For each \( a \) in \( Attribute \) which is owned by a persistent class, create an element \( cl \) in \( Column \), with corresponding identity, name, kind and type”.

Constraints C1 and C2 are similar in structure to the corresponding transformation rules of the QVT-Relations specification of this problem [42]: individual elements of the source model are related to corresponding elements in the target model. However, in our specification, the constraints are defining postconditions of the completed transformation, i.e., what relation it should establish, at its termination, between the source and target model data—indeed independently of any design or implementation strategy to enforce this relation. In this respect the constraints are more abstract and declarative than the QVT rules.

Constraint C5 is different in character to the QVT rules, again it expresses a condition on the target model, relative to the source model, but this condition is recursive in style and needs a more complex implementation, using fixpoint iteration. Nevertheless, it can be used as a postcondition of the transformation at the specification level to reason about the effect of the transformation, independently of how it is implemented.

The constraints have a dual aspect: they express what conditions should be true at the completion of an entire transformation, but they can also be interpreted as the definitions of specific rules executed within the transformation. For example, C1 can be operationally interpreted as “For each \( a \) in \( Attribute \) which is owned by a persistent class, create an element \( cl \) in \( Column \), with corresponding identity, name, kind and type”.
The procedural interpretation of constraints is used to synthesise a correct-by-construction design and executable implementation of the transformation.

The procedural interpretation of constraint $C_n$ is a UML activity $\text{stat}(C_n)$ (in the activity language of Figure 7) which establishes the truth of $C_n$, given certain assumptions $Asm_{C_n}$:

$$Asm_{C_n} \Rightarrow [\text{stat}(C_n)]C_n$$

where $[act]P$ is the weakest precondition of $P$ with respect to act. $Asm_{C_n}$ includes conditions to ensure that expression evaluations in $C_n$ are well-defined.

The ordering of the constraints has no significance to their logical interpretation as a conjunction. However, the procedural interpretation uses this ordering to generate the design as a sequence of phases, and the ordering is also used to establish the design correctness (Section 3.2).

In general there may be several related use cases for a given transformation $\tau : S \rightarrow T$:

- A checking transformation, which checks that the source and target models satisfy $Asm$.
- The forward transformation $\tau$ that establishes $Cons$, assuming $Asm$.
- A reverse transformation $\tau^\sim$, derived from $\tau$, which takes a well-formed target language model $M_T$ and generates a source model $M_S$ which $\tau$ would map to $M_T$.

We recommend a constraint-based specification approach for transformations for several reasons:

1. Constraints have the key advantage that they are unambiguous, with a transparent semantics, and they can be understood without knowledge of the execution semantics of a particular tool. In contrast, even the most declarative style of transformation rule specification, in QVT-Relations or a graph transformation language, requires knowledge of the particular rule scheduling and selection strategies of the language. Simple specifications (such as replacing names of elements) can fail to terminate because of such strategies [38].

2. Constraints are usually more concise than transformation rules or executable versions of a transformation.

3. Analysis of transformation implementation properties such as termination and confluence (that the transformation result is independent of the order of execution of transformation steps) can be carried out at the specification level by analysing the constraints.

4. Constraints facilitate verification, since they are in a form close to that used by theorem-proving and analysis tools, such as B [26] and OCL checkers such as the Dresden OCL tools [8].

5. The inverse of a transformation can be directly computed from the constraints in many cases. Likewise change-propagation forms of a transformation can be derived from its constraints.
Visual specifications of rules can be more usable than textual constraints, particularly if expressed in terms of concrete rather than abstract syntax, however there is currently no widely-accepted visual equivalent of a rich constraint language such as OCL. Visual representations could be used to document and explain the formal constraints.

2.1 Specification analysis

The UML-RSDS tools analyse constraints and use cases using data-use and data-dependency analysis. This analysis is used to (i) identify possible flaws in the specification to the developer; (ii) check that syntactic conditions for confluence and non-interference hold; (iii) to determine the choice of design and implementation of the constraints and use case.

For each predicate \( P \) we define the write frame \( \text{wr}(P) \) of \( P \), and the read frame \( \text{rd}(P) \) (Appendix C). These are the sets of entities and features which \( \text{stat}(P) \) may update or access, respectively. For example, \( \text{wr}(C1) \) is \{Column, Column :: rdId, Column :: name, Column :: kind, Column :: type\}.

A dependency ordering \( Cn < Cm \) is defined between distinct constraints by

\[
\text{wr}(Cn) \cap \text{rd}(Cm) \neq \{\}
\]

A use case with postconditions \( C_1, \ldots, C_n \) should satisfy the syntactic non-interference conditions:

1. If \( C_i < C_j \) for \( i \neq j \), then \( i < j \).
2. if \( i \neq j \) then \( \text{wr}(C_i) \cap \text{wr}(C_j) = \{\} \).

Together, these conditions ensure that the activities \( \text{stat}(C_j) \) of subsequent constraints \( C_j \) cannot invalidate earlier constraints \( C_i, i < j \).

A use case satisfies semantic non-interference if for each \( i < j \):

\[
C_i \Rightarrow [\text{stat}(C_j)]C_i
\]

Syntactic non-interference ensures semantic non-interference, but not conversely. Our example satisfies the first condition of syntactic non-interference but not the second, since \( C2 \) and \( C5 \) both write to \( column \). Nonetheless, semantic non-interference holds true, because the constraints modify \( column \) in a mutually consistent manner: both add elements to this set.

Constraints \( C1 \) and \( C2 \) also satisfy the condition

\[
\text{wr}(C_1) \cap \text{rd}(C_1) = \{\}
\]

for a constraint \( C_i \). Such constraints are termed type 1 constraints. They have a particularly simple implementation as bounded iterations over the source model entity \( S_i \) of their \( \forall \) quantifier. There are also syntactic checks which ensure confluence of such constraints (Appendix C).

The UML-RSDS tools perform the following analyses on the \( Cons \) constraints of transformation specifications:
– calculation of $rd(C_n)$ and $wr(C_n)$ for each constraint, and identification of the constraint type (0, 1, 2 or 3 as defined in Section 2.5).
– checks for syntactic non-interference between constraints, and for their correct ordering in $Cons$. Groups of mutually data-dependent constraints are automatically identified and marked for treatment as a unit for the purposes of design synthesis.
– syntactic checks for confluence of type 1 constraints.
– automatic derivation of inverse constraints for type 1 constraints.
– checks that constraints are complete (eg., when an object of entity $E$ is created, all its features are set).
– checks that constraints are determinate and well-defined (no division by zero, or reference to other undefined elements, succedent of constraints cannot use indeterminate choice: any, or).

Figure 2 shows a screen shot of analysis of the UML to relational database specification.

![Constraint analysis results](image)

**Fig. 2.** Constraint analysis results

Any specification $Cons$ consisting of an ordered conjunction $C_1$ and ... and $C_n$ of constraints has an implementation

$$stat(C_1); \ldots; stat(C_n)$$
provided that each $C_i$ has an executable form $\text{stat}(C_i)$ as defined in Table 11. However, the condition of semantic non-interference is required to ensure that this implementation establishes $\text{Cons}$ (Section 3.2).

### 2.2 Computational model for transformations

Conceptually, a transformation execution can be considered to consist of a collection of individual transformation steps, which are applications of a transformation rule, relation or constraint to specific source elements that match the application conditions of the rule or constraint, to produce an incrementally modified target and/or source model.

The permitted transformation steps for a constraint $C$ defined as

$$\forall s_1:S_1;\ldots; s_n:S_n \cdot \text{SCond implies Succ}$$

are the actions $\text{stat}(\text{Succ})$ applied to any specific $x_1:S_1, \ldots, x_n:S_n$ which satisfy $\text{SCond}$, and for which $\text{Succ}$ does not hold.

The application conditions of $C$ are: $\text{SCond}$ (positive application condition) and $\text{Succ}$ (negative application condition).

Generally, a UML-RSDS transformation $\tau$ maps pairs $(m,n)$ of initial source and target models $m$ and $n$, which satisfy the assumptions $\text{Asm}$ of $\tau$, to pairs $(m',n')$ which satisfy the postcondition constraints $\text{Cons}$ of $\tau$. We denote this by

$$(m,n) \rightarrow_{\tau} (m',n')$$

For each starting pair $(m,n)$, the result $(m',n')$ is minimal such that $\text{Cons}$ holds: no strict subset of the transformation steps of this computation produces a pair $(m'',n'')$ which satisfy $\text{Cons}$.

This definition can be generalised to cases where more than two models are involved.

Update-in-place transformations only modify a single source model, so their computations can be represented as:

$$m \rightarrow_{\tau} m'$$

where $m'$ is the result of a minimal set of transformation steps applied to a starting model $m$ that satisfies $\text{Asm}$ to establish $\text{Cons}$ for $m'$.

Input-preserving separate models transformations such as refinements, abstractions and migrations usually have an initially empty target model: they map a source model $m$ and an empty target model $\emptyset$, that satisfy $\text{Asm}$, to a pair $(m,n)$ which satisfies the postcondition constraints $\text{Cons}$ of $\tau$. We denote this by

$$(m,\emptyset) \rightarrow_{\tau} (m,n)$$

If the steps of $\tau$ only enlarge the target model (ie., they create target elements or extend target element roles), then the minimality condition can be expressed by
saying that no strict submodel of \( n \) satisfies \( \text{Cons} \) with respect to \( m \). Logically the minimality can be expressed by including \( \neg\text{Cons} \) in the assumptions of the obligation (I) of Section B, i.e., as part of \( \text{Inv} \).

The UML to relational database transformation is of this form. If a source model had one persistent class with two attributes:

\[
c : \text{Class} \\
cname = "c" \\
ckind = "Persistent" \\
cumlld = "c" \\
t : \text{PrimitiveDataType} \\
tname = "Integer" \\
tumlld = "Integer" \\
a1 : \text{Attribute} \\
a1name = "a1" \\
a1type = t \\
a1umlld = "a1" \\
a2 : \text{Attribute} \\
a2name = "a2" \\
a2type = t \\
a2umlld = "a2"
\]

then the transformation computation will consist of three steps: (i) application of \( C1 \) to \( a1 \), (ii) application of \( C1 \) to \( a2 \), (iii) application of \( C2 \) to \( c \). Steps (i) and (ii) can occur in either order, step (iii) can occur after (i) and (ii) have completed. The resulting target model satisfies \( C1, C2 \) and \( C5 \) with respect to the unchanged source model, and has data:

\[
t1 : \text{Table} \\
t1name = "c" \\
t1rdbld = "c" \\
t1kind = "Persistent" \\
cl1 : \text{Column} \\
cl1name = "a1" \\
cl1type = "NUMBER" \\
c1l1rdbld = "a1" \\
cl2 : \text{Column} \\
cl2name = "a2" \\
cl2type = "NUMBER" \\
c1l2rdbld = "a2"
\]

This concept of transformation computation was referred to as \textit{transformation state} in [6]. A similar concept, in the domain of graph transformations, is used in [46]. For triple graph grammars, a transformation step consists of the application of a single rewrite rule to specific elements matching the application conditions of the rule [4]. For QVT-Relations, a transformation step could be considered
as a complete application of a top-level relation (including all its invoked non-
top-level relations) [4], or at a finer level of granularity, an application of any
relation, not including its invoked relations.

A specific implementation of a transformation can be defined as an activity
$I$ in the language of Figure 7. For each activity there is a definition of its per-
missible execution behaviour, and hence a transformation implementation will
define a restricted set of possible computations for the transformation, composed
from the basic transformation steps. Completed implementation computations
are denoted by

$$(m, n) \rightarrow^{I} (m', n')$$

The implementation is terminating if for each $(m, n)$ satisfying $\text{Asm}$ there is
a finite complete computation of the implementation starting from $(m, n)$, and
if every computation of the implementation starting from $(m, n)$ is finite. The
implementation is semantically correct if every complete computation starting
from each pair $(m, n)$ of models satisfying $\text{Asm}$ terminates in a pair $(m', n')$
satisfying $\text{Cons}$. An implementation of transformation $\tau$ is confluent if for each
pair of computations $(m, n) \rightarrow^{I}_{\tau} (m', n')$ and $(m, n) \rightarrow^{I}_{\tau} (m'', n'')$ of
the implementation, the result models are isomorphic: $m' \equiv m''$ and $n' \equiv n''$.
In other words, the transformation implementation always produces essentially
equivalent target models from the same source model.

For type 1 constraints (such as $C_1$ and $C_2$), confluence of their UML-RSDS
implementation can be demonstrated by showing that distinct transformation
steps of the constraint always write distinct data (in this case, features of dis-
tinct $\text{Column}$ and $\text{Table}$ objects), and that any shared data is written in an
order-independent manner (they independently add new elements to the sets of
columns and tables). Section C.3 gives the precise conditions for confluence. Ter-
mination always holds since a bounded iteration is used for the implementation.
Correctness follows by construction of the implementation.

For more complex forms of constraint (such as $C_5$), analysis of termination,
confluence and correctness is carried out by defining a variant function $Q : \mathbb{N}$
which is strictly decreased by each transformation step. Formal proof of these
properties can be performed using a representation of the transformation in the
formal B language [26]. This verification process is described in [29] and [36].

2.3 Transformation specification design patterns

A large number of model transformation design patterns have been defined [3,
5, 9, 1, 16, 18]. In the following sections we will specifically consider how two
particular model transformation specification patterns can be used to construct
declarative specifications of transformations:

Conjunctive-implicative form : the specification is written as an ordered
conjunction of constraints with the form

$$\forall s : S_i \cdot S\text{Cond implies } \exists t : T_j \cdot T\text{Cond and Post}$$
defining how source model elements $s$ map to one or more target model elements $t$. There are specialised subpatterns for cases where multiple target elements are created from one source element (entity splitting), for the case where multiple source elements are merged into one target element (entity merging), and for cases of self-referential data structures (map objects before links).

**Auxiliary metamodel** : a metamodel for auxiliary data is introduced, to assist in simplifying the transformation specification.

Table 1 summarises when specific patterns should be used. In this paper we describe the conjunctive-implicative form, recurrent constraints, entity splitting, map objects before links and auxiliary metamodel patterns in detail, other patterns are described in [35].

<table>
<thead>
<tr>
<th>Pattern</th>
<th>When used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conjunctive-implicative form</td>
<td>Postcondition constraints can be written with direct literal meaning as termination conditions for the transformation</td>
</tr>
<tr>
<td>Recurrent constraints</td>
<td>Data-dependency of a constraint upon itself: transformation computation is a fixpoint iteration of individual steps</td>
</tr>
<tr>
<td>Entity splitting</td>
<td>Refinement of source data elements into multiple linked target data elements</td>
</tr>
<tr>
<td>Entity merging</td>
<td>Abstraction of multiple source data elements into single target data element</td>
</tr>
<tr>
<td>Map objects before links</td>
<td>Cycles of entity dependencies in source model need to be mapped</td>
</tr>
<tr>
<td>Auxiliary metamodel</td>
<td>Simplify/optimise complex source to target mapping by recording auxiliary data during transformation, eg., traces or intermediate data structures</td>
</tr>
</tbody>
</table>

**Table 1.** Specification pattern choices

### 2.4 Conjunctive-implicative form

**Synopsis** To specify the effect of a transformation in a declarative manner, as a global pre/post predicate, consisting of a conjunction of constraints with a $\forall \Rightarrow \exists$ structure.

This pattern enables the specification to be consistently interpreted both as a logical expression of the complete effect of the transformation, and as a procedural definition of the detailed transformation steps.

**Forces** Useful whenever a platform-independent specification of a transformation is required, for example, for portability between different model transformation
tools, or for external composition of the transformation with other transformations.

The conjunctive-implicative form can be used to analyse the semantics of a transformation, and used as the basis for automatic synthesis of a design and implementation.

**Solution** For a transformation τ from one language S to another language T, one characteristic specification pattern is that Cons will have the general form of a conjunction of clauses, each of the form

$$\forall s : S_i \cdot S\text{Cond} \implies \exists t : T_j \cdot T\text{Cond} \text{ and } \text{Post}$$

where $S\text{Cond}$ is a predicate over the source model elements only, $S_1, ..., S_n$ are the entities of $S$ which are relevant to the transformation, $T_1, ..., T_m$ are the entities of $T$, $T\text{Cond}$ is a condition in $T$ elements only, eg., to specify explicit values for t’s attributes, and Post refers to both $t$ and $s$ to specify $t$’s attributes and possibly linked (dependent) objects in terms of $s$’s attributes and linked objects. Post can be expressed as a conjunction $L\text{Post}$ and $G\text{Post}$ to specify these two aspects (local versus global relationship between $s$ and $t$). $T\text{Cond}$ does not contain quantifiers, Post may contain $\exists$ quantifiers to specify creation/lookup of subordinate elements of $t$. If the $t$ should be unique for a given $s$, the $\exists_1$ quantifier may be alternatively used in the succedent. Additional $\forall$-quantifiers may be used at the outer level of the constraint, if quantification over groups of source model elements is necessary, instead of over single elements.

Figure 3 shows a schematic structure of this pattern.

![Fig. 3. Conjunctive-implicative form](image-url)
Introducing the pattern involves separating the creation of target entities which are at different levels of the entity hierarchy in $T$ into separate constraints, i.e., into separate phases of the transformation.

For example, a complex constraint of the form

$$\forall s : S_1 \cdot \exists t : T_1 \cdot t.id = s.id \text{ and } \forall s' : S_2 \cdot s' \in s.f \implies \exists t' : T_2 \cdot t'.id = s'.id \text{ and } t'.g$$

can be re-expressed as two simpler constraints which use the conjunctive-implicative form:

$$\forall s : S_2 \cdot \exists t : T_2 \cdot t.id = s.id$$
$$\forall s : S_1 \cdot \exists t : T_1 \cdot t.id = s.id \text{ and } t.g = T_2[s.f.id]$$

This improves the modularity and reusability of the specification, and also supports sequential decomposition of the transformation into phases.

The pattern typically applies when $S$ and $T$ are similar in structure, for example in the UML to relational database mapping the source structure of Package, Class, Attribute corresponds to the target structure of Schema, Table, Column. It can also be applied to update-in-place transformations, by using terms of the form $S_i@pre$, $f@pre$ to denote the original sets of objects of entities, and original values of features.

Consequences

The form of the constraints enables their ordering to be interpreted both as logical conjunction and as sequential composition of their effects: constraints $C_i$:

$$\forall s : S_i \cdot SCond \implies \exists t : T_j \cdot TCond \text{ and } Post$$

earlier in the sequence of constraints will generally not be invalidated by following constraints $C_k$ that create new target ($T_j$) elements, nor by following constraints that delete source ($S_i$) elements. In the first case $\text{wr}(C_i) \cap \text{wr}(C_k)$ is non-empty, but if the $T_j$ elements created by the first constraint are a disjoint set from those created and initialised by the second, then the effects are (semantically) non-interfering.

The conjunction of the $Cons$ constraints represents the termination condition of the transformation: $\tau$ continues executing transformation steps (derived from the constraints) until source and target models are reached which satisfy $Cons$.

If $Cons$ consists of type 1 constraints that satisfy internal syntactic non-interference (Appendix C), we can derive a reverse transformation $\tilde{\tau}$ with constraints $Cons^\sim$ computed from $Cons$, which expresses that elements of $T$ can only be created as a result of the application of one of the forward constraints: each reverse constraint has the form:

$$\forall t : T_j \cdot TCond \implies \exists s : S_i \cdot SCond \text{ and } Post^\sim$$

where $Post^\sim$ expresses the inverse of $Post$.

The form of the $Post$ predicates themselves will often be of one of the following kinds:
- \( t.f = e \) where \( f \) is some feature of \( T_j \) and \( e \) an expression using feature values of \( s \). If \( e \) is a direct feature reference \( s.g \) then \( \text{Post}^{-} \) is \( s.g = t.f \).
- \( \exists t_{sb} : \text{TSub} \cdot \text{Post}_{t_{sb}} \) and \( t.f = t_{sb} \) to define a subpart of \( t \). Such quantifiers are grouped at the outer scope of \( \text{Post} \), the inverse constraint then has corresponding \( \forall \) quantifiers at its outer scope.
- Conjunctions of implications

\[
(\text{Cond}_1 \text{ implies } P_1) \text{ and } ... \text{ and } (\text{Cond}_r \text{ implies } P_r)
\]

where the \( P_i \) are of the first two forms. Each conjunct is separated into a distinct constraint in order to form the inverse transformation.

A special case are equalities \( t.f = \text{TSub}[s.g.id1] \) which select a single element of \( \text{TSub} \) or a set of elements, with primary key value(s) \( \text{id2} \) equal to \( s.g.id1 \) (if this is a single value), or in \( s.g.id1 \) (if it is a collection). The reversed form \( \text{Post}^{-} \) in this case is \( s.g = \text{SSub}[t.f.id2] \), if source model entity \( \text{SSub} \) is in 1-1 correspondence with \( \text{TSub} \) via the identities.

**Implementation** Either by the change-propagation, phased construction or recursive descent implementation strategies (Section 3).

**Code examples** Examples of transformations which fit this pattern are the model migration transformation of UML 1.4 state machines to UML 2.2 activity diagrams [31], and the UML to relational database mapping, with persistent classes mapping to tables, packages to schemas, attributes to columns, etc [32]. Large scale specifications of software analysis systems such as slicing tools have also been defined using this approach [29].

The UML to relational database transformation has the additional constraints:

**C3** “For each root class in the source model there is a unique primary key in the target model”:

\[
\forall c : \text{Class} \cdot c.\text{kind} = \text{"Persistent" and } c.\text{general} = \{\} \text{ implies } \\
\exists k : \text{Key} \cdot k.\text{rdbId} = c.\text{umlId} + \text{"_Pk" and } k.\text{name} = c.\text{name} + \text{"_Pk" and } k.\text{owner} = \text{Table}[c.\text{umlId}] \text{ and } k.\text{kind} = \text{"PrimaryKey" and } \\
\exists cl : \text{Column} \cdot cl.\text{rdbId} = c.\text{umlId} + \text{"_Id" and } cl.\text{name} = c.\text{name} + \text{"_Id" and } cl.\text{type} = \text{"NUMBER" and } cl : k.\text{column and } cl.\text{kind} = \text{"PrimaryKey" and } \\
cl : k.\text{owner.column}
\]
For each association in the source model, there is a unique foreign key representing it in the target model:

\[
\forall \ a : \text{Association} \quad \Rightarrow \quad \exists \ f_k : \text{ForeignKey} \quad \text{such that} \quad f_k.rdbId = a.umlId + "Fk" \quad \text{and} \quad f_k.name = a.name + "Fk" \quad \text{and} \quad f_k.owner = \text{Table}[a.source.umlId] \quad \text{and} \quad f_k.kind = "association" \quad \text{and} \quad f_k.refersTo = \text{Table}[a.destination.umlId] \quad \text{and} \quad \exists \ c_l : \text{Column} \quad \text{such that} \quad c_l.rdbId = a.umlId + "Ref" \quad \text{and} \quad c_l.name = a.name + "Ref" \quad \text{and} \quad c_l.column and c_l.kind = "Foreign Key" \quad \text{and} \quad c_l.type = "NUMBER" \quad \text{and} \quad c_l : f_k.owner.column}
\]

For each package in the source model, there is a unique schema representing it in the target model:

\[
\forall \ p : \text{Package} \quad \Rightarrow \quad \exists \ s : \text{Schema} \quad \text{such that} \quad s.rdbId = p.umlId \quad \text{and} \quad s.name = p.name \quad \text{and} \quad s.kind = p.kind \quad \text{and} \quad s.tables = \text{Table}[p.elements.umlId]}
\]

C1, C2 and C5 can be considered a core specification of the transformation, with the other constraints expressing extensions of this mapping for further language elements (primary keys, associations and schemas, respectively). The dependency ordering of these constraints is:

\[
C1 < C2, C2 < C5
\]

The other constraints depend upon the core constraints:

\[
C2 < C3, C2 < C4, C2 < C6
\]

and

\[
C3 < C5, C4 < C5
\]

Notice that \( wr(C1) \) and \( wr(Cj) \) are not disjoint, for \( j = 3,4,5 \). The later constraints create new columns, and set the features of these new columns, however this does not invalidate C1 because of the \( \forall \exists \) form of C1, and because the columns created by C1 are not deleted or modified by the later constraints.

The \( wr(Cj) \) set for these constraints also intersects \( rd(C2) \), however because the columns created by the later constraints have distinct rdbid's from any derived from class attributes, the value of \( Column[c.attribute.umlId] \) is unaffected, and so C2 remains valid.

In general a constraint with right-hand side \( \exists t : T_j \cdot Succ_1 \) will not conflict with a constraint with rhs \( \exists t : T_j \cdot Succ_2 \) unless the two predicates could select the same instance of \( T_j \) and modify it in ways that invalidate each other.
A constraint \( Cn \) which creates \( T_j \) elements will not invalidate earlier constraints that refer to \( T_j[v] \) provided the id’s of the instances created by \( Cn \) are distinct from the values in \( v \).

2.5 Recurrent constraints

Synopsis A special case of conjunctive-implicative form, where a constraint reads and writes the same data, and therefore may require a fixpoint implementation strategy to ensure that it is established.

Solution A constraint \( Cn \) of form

\[
\forall s : S_i \cdot SCond \ implies \ \exists t : T_j \cdot TCond \ and \ Post
\]

is termed a type 2 constraint if

\[
w(\text{Cn}) \cap (\text{rd}(\text{Post}) \cup \text{rd}(\text{TCond}))
\]

is non-empty, but

\[
w(\text{Cn}) \cap (\text{rd}(\text{SCond}) \cup \{S_i\}) = \{}
\]

This means that the order of application of the constraint to instances may be significant, and that a single iteration through the source model elements may be insufficient to establish \( Post \) for all elements. A fixpoint computation may be necessary instead, with iterations repeated until no further change takes place.

A constraint is of type 3 if \( S_i \in w(\text{Cn}) \) or \( w(\text{Cn}) \cap \text{rd}(\text{SCond}) \neq \{\} \). Again in this case a fixpoint computation is usually required, with additional complexity because the set of source objects being considered by the constraint is itself dynamically changing.

Consequences A measure \( Q : \text{Models}_S \times \text{Models}_T \rightarrow \mathbb{N} \) on the source and target model data is necessary to establish the termination, confluence and correctness of the implementation of type 2 and type 3 constraints, and should be defined together with the constraint. \( Q \) should have the property that it is decreased by each application of the constraint, and \( Q = 0 \) at termination of the phase for the constraint.

Formally, \( Q \) is a variant function for applications of the constraint:

\[
\forall \nu : \mathbb{N} \cdot Q(\text{smodel}, \text{tmodel}) = \nu \land s \in S_i \land SCond \land \nu > 0 \Rightarrow \left[ \text{stat}(\text{Succ}) \right](Q(\text{smodel}, \text{tmodel}) < \nu)
\]

where \( S_i \) denotes the extension of \( S_i \), and

\[
Q(\text{smodel}, \text{tmodel}) = 0 \equiv \{ s \in S_i \mid SCond \land \lnot (\text{Succ}) \} = \{}
\]

\( \text{Succ} \) abbreviates the constraint rhs \( \exists t : T_j \cdot TCond \ and \ Post \).
Termination of the implementation of the constraint follows from the variant function property.

If for each particular starting state of the source and target models there is a unique (up to isomorphism) possible final state of the models (produced by applying the constraint until \( Cn \) holds true) in which \( Q = 0 \), then the implementation is confluent.

\( Q \) will be syntactically defined by some expression in the union language of \( S \) and \( T \).

In some cases the existence of such a \( Q \) can be deduced from the form of the constraint. A type 2 constraint \( Cn \) with \( wr(Cn) = \{ r \} \), for a many-valued association end \( r : F(T_i) \) of entity \( T_i \), and that updates \( r \) by addition of elements only (i.e., by formulae \( e \subseteq x.r, e : x.r \) or equivalent forms) must have a \( Q \) measure bounded by

\[
\Sigma_{T_i} \#(T_j - t.r)
\]

since the sets \( T_j \) and \( T_j \) are not modified by \( Cn. - \) denotes set subtraction.

Likewise, for cases where such an \( r \) is updated by removal of elements only (formulae \( e \subseteq x.r \rightarrow \text{excludesAll}(e) \) and equivalents), we have a bound:

\[
\Sigma_{T_i} \#(t.r - r)
\]

**Code example** Constraint \( C5 \) in the UML to relational database example has the type 2 form:

\[
\forall c, d : \text{Class} \cdot c.\text{kind} = \text{"Persistent" and } d : c.\text{general implies} \\
\text{Table}[d.\text{umlId}].\text{column} \subseteq \text{Table}[c.\text{umlId}].\text{column}
\]

Here, \( Table :: column \) is both read and written. \( Q \) can be defined for this example as the number of columns which need to be copied:

\[
\Sigma_{c:Class} \#(\bigcup_{d : c.\text{general}} \text{Table}[d.\text{umlId}].\text{column} - \text{Table}[c.\text{umlId}].\text{column})
\]

One way to implement the constraint is to successively propagate columns down from superclass tables to subclass tables, starting from root classes down the inheritance hierarchy. However, since \( Q = 0 \) for a unique minimal state reachable by the effect of the constraint (copying columns from an immediate superclass to its subclass) it can also be achieved by applying this action on classes in any order until no further changes occur.

An example of a type 3 constraint is the derivation of the transitive closure of a graph [33]:

\[
\forall e1, e2 : \text{Edge} \cdot e1 \neq e2 \text{ and } e1.\text{try} = e2.\text{src} \text{ implies} \\
\exists e3 : \text{Edge} \cdot e3.\text{src} = e1.\text{src} \text{ and } e3.\text{try} = e2.\text{try}
\]

For this, \( Q \) can be defined as the number of distinct chains of edges in the original graph without an edge linking the start of the chain to its end. This enables us to show the correctness, termination and confluence of the derived implementation (Section 3).
2.6 Entity splitting

Synopsis A special case of conjunctive-implicative form where for some entity $S_i$ in the source language, there may be several different $T_j$ entities on the right-hand side of constraints in $Cons$, even with the same $SCond$ conditions. This may occur because the features of $S_i$ are being split or distributed between different target language entities, or because complex and separate structures are being generated in the target model from the same source data. These typically occur in refinement transformations.

Forces A single constraint may define objects of several different target entities in its succedent, for example:

$$\forall s : S_i \cdot SCond \implies \exists t_1 : T_j \cdot TCond \text{ and } Post_1 \text{ and } \exists t_2 : T_k \cdot t_2.f = t_1 \text{ and } Post_2$$

This structure may suffer from **tangling** (the mixing of different functionalities in a single specification unit).

Solution It is possible to ‘untangle’ such constraints and split them into multiple constraints with the same antecedent, but distinct succedents for each separate target type, provided that identity attributes exist for the target entities, for example:

$$\forall s : S_i \cdot SCond \implies \exists t_1 : T_j \cdot TCond \text{ and } t_1.t_1.Id = s.sId \text{ and } Post_1$$

$$\forall s : S_i \cdot SCond \implies \exists t_2 : T_k \cdot t_2.f = T_j[s.sId] \text{ and } t_2.t_k.Id = s.sId \text{ and } Post_2$$

The value of the identity attribute of $S_i$ is used to implicitly link the two target objects. Since $T_k$ depends on $T_j$, the constraints are ordered with the constraint creating $T_j$ instances first.

Consequences In these cases it is important to define in the separate target entities some key attribute or stereotype which records the fact of the semantic link between them (that they represent separate parts of the same source entity). This allows the definition of a reverse transformation.

If the inverse of each constraint in an entity-splitting specification

$$\forall s : S_1 \cdot SCond_i \implies \exists t : T_1 \cdot TCond_i \text{ and } t.id = s.id \text{ and } Post_i$$

$$\vdots$$

$$\forall s : S_1 \cdot SCond_n \implies \exists t : T_n \cdot TCond_n \text{ and } t.id = s.id \text{ and } Post_n$$

can be formed, then the inverse transformation has constraints

$$\forall t : T_1 \cdot TCond_i \implies \exists s : S_i \cdot SCond_i \text{ and } s.id = t.id \text{ and } Post_i$$

$$\vdots$$

$$\forall t : T_n \cdot TCond_n \implies \exists s : S_i \cdot SCond_n \text{ and } s.id = t.id \text{ and } Post_n$$

and this transformation merges information from multiple objects $t_i : T_i$, $i = 1, \ldots, n$, with the same id values, into a single $s : S_1$ object.
Code examples: An example is given in [25], of source model objects of types such as `OpenElement` being transformed in a single rule into interconnected model, view and controller objects `Open`, `OpenView`, `OpenController` (Figure 4).

![Code examples](image)

Fig. 4. MVC introduction metamodels

The single rule could be written as:

\[
\forall e : \text{OpenElement} \cdot \\
\exists t : \text{OpenView} \cdot t.\text{id} = e.\text{id} \text{ and } \exists \text{op} : \text{Open} \cdot \text{op.id} = e.\text{id} \text{ and } \text{op.observer} = t \text{ and } \\
\exists \text{oc} : \text{OpenController} \cdot \text{oc.view} = t
\]

Since the entities `Open` and `OpenController` depend on (refer to) `OpenView`, we can split the rule into three separate constraints: the first creates `OpenView` objects, the second creates `Open` objects and links these to the `OpenView`, and the third does the same for `OpenController`:

\[\text{(C1)} : \forall e : \text{OpenElement} \cdot \exists t : \text{OpenView} \cdot t.\text{id} = e.\text{id}\]

\[\text{(C2)} : \forall e : \text{OpenElement} \cdot \exists t : \text{Open} \cdot t.\text{id} = e.\text{id} \text{ and } t.\text{observer} = \text{OpenView}[e.\text{id}]\]

\[\text{(C3)} : \forall e : \text{OpenElement} \cdot \exists t : \text{OpenController} \cdot t.\text{view} = \text{OpenView}[e.\text{id}]\]
In the UML to relational database example, constraints $C_3$ and $C_4$ are not examples of tangling, because the multiple elements they create (keys and columns) are elements of a single concept.

In Section 4 we describe higher-level structuring solutions for tangling problems.

The one $S_i$ to multiple $T_j$ case generally occurs in refinements, such as the generation of multiple J2EE elements (database tables, value objects, EJBs) from single UML classes. The pattern is related to the local source to global target transformation structures of [6].

2.7 Map objects before links

*Synopsis* To declaratively specify a transformation when there are cycles of entity dependencies in the source model.

*Forces* Useful whenever the source model contains self-associations on entities or longer cycles of entity dependencies, which need to be mapped by the transformation. A strictly hierarchical conjunctive-implicative form is not suitable in this case.

*Solution* Split the transformation specification into two phases, the first maps entities and their attributes (i.e., it carries out $LPost$) and any other data which is not involved in dependency cycles. The second phase links together the target objects, using identity attributes to look up the objects that should be linked (carrying out $GPost$).

*Consequences* The specification of $LPost$ and $GPost$ for source entities becomes split into separate constraints, an example of scattering of specification text.

*Code examples* Figure 5 shows a typical example of this situation, where in the source metamodel, entity $E$ depends upon itself via the subclass association role elements. The target metamodel has a similar recursive structure based on entity $F$.

Applying the conjunctive-implicative form pattern directly to this problem results in a specification of the form:

\[
\forall d : D \cdot \exists g : G \cdot LPostDG
\]

\[
\forall e : EBasic \cdot \exists f : FBasic \cdot LPostEBasicFBasic and f.gr = G[e.dr.id]
\]

\[
\forall e : EComp \cdot \exists f : FComp \cdot LPostECompFComp and
f.gr = G[e.dr.id] \text{ and } f.elements = F[e.elements.id]
\]

But in the final constraint, $F$ is both written and read, so the constraint is of type 2 and needs a fixpoint implementation. The problem is that not all of $e.elements$ may have been mapped to $F$ elements when the mapping of $e$ is attempted – so the mapping of $e$ would need to be redone whenever one of its elements is mapped. This approach is therefore quite inefficient.
An alternative is to separate the object mapping and link mapping:

\[ \forall d : D \cdot \exists g : G \cdot \text{PostDG} \]

\[ \forall e : EBasic \cdot \exists f : FBasic \cdot \text{LPostEBasicFBasic} \]

\[ \forall e : EComp \cdot \exists f : FComp \cdot \text{LPostECompFComp} \]

in a first phase, and

\[ \forall e : E \cdot F[e.id].gr = G[e.dr.id] \]

\[ \forall e : EComp \cdot FComp[e.id].elements = F[e.elements.id] \]

in the second.

Logically, the two versions of the specification have an identical meaning as postconditions of the transformation, because in the final state of the transformation computation it is assumed that all \( F \) elements corresponding to source \( E \) elements exist, but the second version permits a simpler implementation because all constraints are of type 1.

The inverse of a constraint

\[ \forall s : S_1 \cdot T_1[s.id1].f = T_2[s.g.id1] \]

is

\[ \forall t : T_1 \cdot S_1[t.id2].g = S_2[t.f.id2] \]

where \( S_1 \) corresponds 1-1 with \( T_1 \), and \( S_2 \) with \( T_2 \).

Another example of this situation is in [34]. The pattern is related to the separation of generate and refinement transformation phases in [6].

### 2.8 Auxiliary metamodel

**Synopsis** The introduction of a metamodel for auxiliary data, neither part of the source or target language, used in a model transformation.
Forces. Useful whenever auxiliary data needs to be introduced into a transformation, for example:

- To store intermediate results of a transformation, to improve efficiency or to simplify the transformation definition.
- To store tracing information and correspondences between the source and target models, to provide matching element lookup facilities and algorithm control.
- To permit decomposition of the transformation into a series of subtransformations.
- To define additional model structures for convenience in model processing, such as root elements of a model.

This pattern is referred to as intermediate structure in [7].

Solution. Define the auxiliary metamodel as a set of entities, features and generalisations extending the source and/or target metamodels. These elements may then be used in the Cons constraints, e.g., to derive the auxiliary data from source model data or to derive target model data from the auxiliary data.

Figure 6 shows a typical structure of this pattern. The auxiliary metamodel simplifies the mapping between source and target by factoring it into two steps.

\begin{center}
\begin{tikzpicture}
\node[draw] (S) at (0,0) {S};
\node[draw] (Tj) at (3,0) {Tj};
\node[draw] (Aux1) at (1.5,-2) {Aux1};
\node[draw] (Aux2) at (0.5,-4) {Aux2};
\node[draw] (Tsub1) at (2,-3) {Tsub1};
\node[draw] (Tsub2) at (3,-4) {Tsub2};
\draw[->] (S) -- (Aux1);
\draw[->] (Aux1) -- (Tj);
\draw[->] (S) -- (Aux2);
\draw[->] (Aux2) -- (Tj);
\draw[->] (S) -- (Tsub1);
\draw[->] (Tsub1) -- (Tj);
\draw[->] (S) -- (Tsub2);
\draw[->] (Tsub2) -- (Tj);
\end{tikzpicture}
\end{center}

**Fig. 6.** Auxiliary metamodel structure

Code example. An example of the use of auxiliary metamodel elements to store pre-computed data to facilitate and simplify a transformation is the state machine slicing transformation of [30], which introduces auxiliary additional associations recording dependency sets of variables in each state, and the reachability
relation between states. Auxiliary metamodels may be used to define intermediate languages, to factor a complex transformation into sub-steps, and to increase the flexibility of a transformation. In [15] the sequential decomposition of a complex transformation using an intermediate language auxiliary metamodel is described.

An auxiliary metamodel entity can be used to identify and retain groups of source model elements which together satisfy some matching condition, in order to simulate multiple-element matching rules by single-element matching [3].

Auxiliary metamodels are also used to implement tracing facilities, in transformation languages such as Viatra [45] and Kermeta [21], the auxiliary entities and associations record information such as a history of rules applied and connections between target model elements and the source model elements they were derived from. In Triple Graph Grammars, correspondence models are defined to record the connections established between source and target model elements.

Related patterns This pattern extends the conjunctive-implicative form pattern by allowing constraints to refer to data which is neither part of the source or target languages.

3 Implementation of constraint-based specifications

Implementation of a model transformation may be carried out by the use of a special-purpose model transformation language such as ATL [20] or Kermeta [21], or by production of code in a general purpose language such as Java [29]. In either case, the implementation needs to be organised and structured in a modular manner, and ideally it should be possible to directly relate the implementation to the specification, expressed in the Asm and Cons predicates.

We use a small procedural language including assignment, conditionals, operation calls and loops to allow platform-independent imperative definitions of behaviour for transformation implementations (Figure 7). This language corresponds to a subset of UML structured activities, and serves as an intermediate language, from which transformation implementations in different executable languages can be generated. It can also be mapped into the BPMN statement language [26], to support verification. There is a definition of weakest precondition \( [stat]P \) for each form of statement \( stat \) in the language, supporting the verification of implementations.

3.1 Implementation strategies

There are three alternative strategies for implementation of a model transformation specification defined by conjunctive-implicative form constraints:

- Change-propagation: the constraints are used directly to implement the transformation, by interpreting them in an operational form. If a change is made to the source model, any constraint whose application condition is made true
by the change is applied, and the effects defined by its conclusion $Succ$ are executed to modify the target model. This is particularly suitable for quality-improvement transformations which must maintain a certain property (such as no redundant inheritances or duplicated attributes).

- **Recursive descent**: the transformation is initiated by construction of instances of the topmost entities in the entity hierarchy of the target language $T$, such construction may require recursive construction of their components, etc.

- **Layered**: the base elements (instances of the entities lowest in the entity hierarchy of $T$) are constructed first, then elements that depend upon these are constructed, etc.

In this paper we will only consider the layered approach in detail, the change-propagation and recursive approaches are described in [35].

The following implementation patterns can be used with these strategies:

**Phased creation**: before creating an instance of an entity $T_1$, create all instances of entities $T_2$ hierarchically below $T_1$, which are mapped to by the transformation. In the case of mutually dependent entities, create all objects of the entities before setting the links between them.

**Object indexing**: to efficiently compute collections $T_j[ids]$ of $T_j$ elements with a given set $ids$ of primary key values, maintain a map from primary keys to $T_j$ elements.

**Unique instantiation**: before creating an instance to satisfy $\exists t : T_j \cdot Pred$, search to see if one already exists, and use this if so. In addition, for a consequent $\exists t : T_j \cdot t.id = v$ and $Pred$, lookup $T_j[v]$, and if this exists, update its features using $Pred$, otherwise create a new $t$ satisfying the consequent.
This pattern is related to the Singleton pattern [11] and may use Object indexing.

**Construction and cleanup**: separate construction of new elements from the deletion of elements, placing these processes in successive phases.

**Recursive descent**: recursively map subcomponents of a source model element as part of the mapping operation of the element, passing down the target object(s) in order to link subordinate target elements to it/them.

The UML-RSDS tools automatically apply the phased creation, object indexing and unique instantiation patterns to specifications written using the conjunctive-implicative form. Developers can implement the construction and cleanup pattern by decomposing a transformation into two sequential sub-transformations, using the *include* composition mechanism (Section 4). The recursive descent pattern can be applied at the design level by defining recursive operations of the source language metaclasses. We describe phased creation here, the other patterns are described in [35].

### 3.2 Phased creation

**Synopsis** Construct target model elements in phases, ‘bottom-up’ from individual objects to composite structures, based upon a structural dependency ordering of the target language entities.

**Forces** Used to enable the modular decomposition of the transformation, usually as a sequential composition (chaining) of sub-transformations.

**Solution** Decompose the transformation into phases, based upon the *Cons* constraints. These constraints should be ordered so that data read in one constraint is not written by a subsequent constraint, in particular, phase $p_1$ must precede phase $p_2$ if it creates instances of an entity $T_1$ which are read in $p_2$.

Figure 8 shows a schematic structure of this pattern.

![Fig. 8. Phased creation structure](image-url)
**Consequences** The stepwise construction of the target model leads to a transformation implementation as a sequence of phases: earlier phases construct elements that are used in later phases. Some mechanism is required to look up target elements from earlier phases, such as by key-based search or by trace lookup.

For a phased implementation, the form of the specification can be used to define the individual transformation rules. If $\text{Cons}$ is a conjunction $C_1 \text{ and } ... \text{ and } C_n$ of constraints $C_k$ of the form

$$\forall s : S_i \cdot SCond \text{ implies } \exists t : T_j \cdot TCond \text{ and } \text{Post}$$

then each constraint may be mapped individually into a phase $P_k$ which implements it.

Each phase $P_i$ is defined as $\text{stat}(C_i)$, so will establish $C_i$ at its termination, under certain assumptions $\text{Asm}_i$ of syntactic correctness of the model being operated on. By induction, we can prove that the sequence of phases $P_1; ...; P_n$ establishes $\text{Cons}$, under the assumption that each $P_k$ preserves $C_l$ for $l < k$:

$$C_1 \land ... \land C_{i-1} \Rightarrow [\text{stat}(C_i)][(C_1 \land ... \land C_{i-1})]$$

for each $i : 2..n$ (generalised semantic non-interference).

Let $\text{Asm}_0$ be the assertion that the source model is syntactically correct. Each phase must preserve this property. In addition, each phase $P_i$ will establish intermediate assertions $\text{Asm}_i$ which can be used by the following phases. These may be assertions that parts of the target model are uninhabited (ie., that $T_l = \{\}$ for all concrete target model entities $T_l$ which are not in $\text{wr}(C_k)$ for $k \leq i$).

Therefore we can assert that (1):

$$\text{Asm}_0 \land \bigwedge_{k<i} C_k \Rightarrow [P_i](C_i \land \text{Asm}_i)$$

The correctness of the composition $P_1; ...; P_n$ follows from this by induction. For $n = 1$ we have

$$\text{Asm}_0 \Rightarrow [P_1](C_1 \land \text{Asm}_1)$$

and if

$$\text{Asm}_0 \Rightarrow [P_1; ...; P_i](C_1 \land ... \land C_i)$$

for some $i < n$, then also

$$\text{Asm}_0 \Rightarrow [P_1; ...; P_i](C_1 \land ... \land C_i \land [P_{i+1}](C_{i+1} \land \text{Asm}_{i+1}))$$

by (1), and

$$C_1 \land ... \land C_i \Rightarrow [P_{i+1}](C_1 \land ... \land C_i)$$

by the semantic non-interference property, so

$$\text{Asm}_0 \Rightarrow [P_1; ...; P_{i+1}](C_1 \land ... \land C_{i+1} \land \text{Asm}_{i+1})$$

as required.

If a group of constraints are mutually data dependent, then they must be implemented by a single phase.
Implementation Individual constraints \( Cn \):

\[ \forall s : S_i \cdot SCond \text{ implies } \exists t : T_j \cdot TCond \text{ and Post} \]

are examined to identify which implementation approach can be used to derive their design. This depends upon the features and objects read and written within the constraint (Table 2).

<table>
<thead>
<tr>
<th>Constraint properties</th>
<th>Implementation choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 0 constraint</td>
<td>Constraint defining update on single object</td>
</tr>
<tr>
<td>Type 1 constraint</td>
<td>No interference between different applications of constraint, and no change to ( S_i ) or ( rd(SCond) ): ( wr(Cn) \cap rd(Cn) = { } )</td>
</tr>
<tr>
<td>Type 2 constraint</td>
<td>Interference between different applications of constraint: ( wr(Cn) \cap rd(Cn) \neq { } ) but no update of ( S_i ) or ( rd(SCond) ) within constraint: ( S_i \notin wr(Cn) ), ( wr(Cn) \cap rd(SCond) = { } )</td>
</tr>
<tr>
<td>Type 3 constraint</td>
<td>Update of ( S_i ) or ( rd(SCond) ) within constraint: ( S_i \in wr(Cn) ), or ( wr(Cn) \cap rd(SCond) \neq { } )</td>
</tr>
</tbody>
</table>

Table 2. Design choices for constraints

In the simple case where a constraint \( Cn \) satisfies the type 1 condition:

\[ wr(Cn) \cap rd(Cn) = \{ \} \]

the constraint can be implemented by a loop

\( \text{for } s : S_i \text{ do } s.\text{op()} \)

where in \( S_i \) we include an operation of the form:

\( \text{op}_i() \)

\text{post:}

\( SCond[self/s] \text{ implies } \exists t : T_j \cdot TCond \text{ and Post}\text{[self/s]} \)

We refer to this as constraint implementation approach 1. The proof of correctness of this approach uses the property that the inference rule: from

\( v : s \Rightarrow [\text{acts}(v)]P(v) \)
derive
\[
[\text{for } v : s \text{ do } \text{acts}(v)](\forall v : s @ \text{pre} \cdot P(v))
\]
is valid for such iterations, provided that one execution of \text{acts} does not affect another: the precondition of each \text{acts}(v) has the same value at the start of \text{acts}(v) as at the start of the loop, and if \text{acts}(v) establishes \( P(v) \) at its termination, \( P(v) \) remains true at the end of the loop [29].

Specifically,
\[
s \in \overline{S_i} \Rightarrow [s.\text{op}_i()](\text{SCond} \Rightarrow \text{Succ})
\]
by definition of \( \text{op}_i() \), where \( \text{Succ} \) is \( \exists t : T_j \cdot \text{TCond and Post} \), so by the above inference, \( \text{stat}(Cn) \) defined as
\[
\text{for } s : S_i \text{ do } s.\text{op}_i()
\]
establishes \( Cn \). Confluence also follows if the updates of written data in different executions of the loop body are independent of the order of the executions.

The time complexity of the implementation is linear in the size \( \#\overline{S_i} \) of the domain. More precisely the worst case complexity is linear in
\[
\#\overline{S_i} \cdot (\text{cost}_{\text{eval}}(\text{SCond}) + \text{cost}_{\text{act}}(\text{Succ}))
\]
where \( \text{cost}_{\text{eval}}(e) \) is the time required to evaluate \( e \), and \( \text{cost}_{\text{act}}(e) \) the time required to execute \( \text{stat}(e) \). If additional domains need to be iterated over, the cost is also multiplied by their size. This shows that multiple element matching or complex expressions in the constraint should be avoided for efficiency reasons.

A more complex implementation approach is required if the non-interference condition does not hold. Consider a constraint \( Cn \):

\[
\forall s : S_i \cdot \text{SCond implies } \exists t : T_j \cdot \text{TCond and Post}
\]

In the case where
\[
\text{wr}(Cn) \cap (\text{rd}(\text{Post}) \cup \text{rd}(\text{TCond}))
\]
is non-empty but the other conditions of non-interference still hold (ie., a type 2 constraint), an iteration of the form:

\[
\begin{align*}
\text{running} & := \text{true}; \\
\text{while } (\text{running}) \text{ do} \\
& (\text{running} := \text{false}; \\
& \text{for } s : S_i \text{ do} \\
& \quad \text{if } \text{SCond then} \\
& \quad \quad \text{if } \text{Succ then skip} \\
& \quad \quad \text{else } (s.\text{op}(); \text{running} := \text{true}) \\
& \quad \text{else skip})
\end{align*}
\]
can be used, where \( \text{Succ} \) is \( \exists t : T_j \cdot T\text{Cond} \) and \( \text{Post} \) and \( \text{op}() \) is defined as:

\[
\text{op}()
\]

\[
\text{post}:
\]

\[
\text{Succ}[\text{self/s}]
\]

In the conditional test \( \text{Succ} \) is evaluated in a non side-effecting manner.

We refer to this approach as constraint implementation approach number 2. The conditional test of \( \text{Succ} \) can be omitted if it is known that \( S\text{Cond} \Rightarrow \neg (\text{Succ}) \). The UML-RSDS tools perform algebraic simplification to check if \( S\text{Cond} \) contradicts \( \text{Succ} \).

A measure \( Q : \mathbb{N} \) over the source and target models can be used to prove termination, as in Section 2.5.

Termination holds, if \( Q \) is a variant for the while loop (2):

\[
\forall \nu : \mathbb{N} \cdot Q = \nu \land \text{running} = \text{true} \land \nu > 0 \Rightarrow [\text{body}](Q < \nu)
\]

where \( \text{body} \) is the body of the while loop.

\( Q \) is also necessary to prove correctness: while there remain \( s : S_i \) with \( S\text{Cond} \) true but \( \text{Succ} \) false, then \( Q > 0 \) and the iteration will apply \( \text{op} \) to such an \( s \). At termination, \( \text{running} = \text{false} \), which can only occur if there are no \( s : S_i \) with \( S\text{Cond} \) true but \( \text{Succ} \) false, so the constraint therefore holds true, and \( Q = 0 \). Confluence also follows if \( Q = 0 \) is only possible in one unique (up to isomorphism) terminal state of the source and target models which can be reached from the initial state by applying the constraint: this will be the state at termination regardless of the order in which elements were transformed.

The time complexity of the implementation depends on the value of \( Q \) on the starting models \( smodel, tmodel \), and on the size \( \#S_i \) of the domain. The worst case complexity is of the order

\[
Q(\text{smodel, tmodel}) \ast \#S_i \ast (\text{cost}_{\text{eval}}(S\text{Cond}) + \text{cost}_{\text{eval}}(\text{Succ}) + \text{cost}_{\text{act}}(\text{Succ}))
\]

since the inner loop may be performed \( Q \) times. Optimisation by omitting the successor test reduces the complexity by removing the term \( \text{cost}_{\text{eval}}(\text{Succ}) \).

If the other conditions of non-interference fail (a type 3 constraint), then the application of a constraint to one element may change the elements to which the constraint may subsequently be applied to, so that a fixed for-loop iteration over these elements cannot be used. Instead, a schematic iteration of the form:

\[
\text{while some source element } s \text{ satisfies a constraint lhs do select such an } s \text{ and apply the constraint}
\]

can be used. This can be explicitly coded as:

\[
\text{running} := \text{true};
\]

\[
\text{while running do select such an } s \text{ and apply the constraint}
\]

\[
\text{running} := \text{search()}
\]

where:
search() : Boolean
(for s : $S_i$ do
  if $SCond$ then
    if Succ then skip
    else ($s$.op(); return true);
  return false)

and where op has postcondition Succ[self/s]. We call this approach 3, iteration
of a search-and-return loop. The conditional test of Succ can be omitted if it is
known that $SCond \Rightarrow \neg$Succ.

As in approach 2, a Q measure is needed to prove termination and correctness.
Termination follows if Q is a variant of the while loop: applying op() to
some $s : S_i$ with $SCond$ and $\neg$Succ decreases Q, even if new elements of
$S_i$ are generated, as in the graph transitive closure computation. Correctness
holds since search returns false exactly when $Q = 0$, ie, when no $s : S_i$ falsifying
the constraint remain. Again, confluence can be deduced from uniqueness of the
termination state.

The worst case complexity is of the order

$$Q(smodel, tmodel) \ast \max S \ast (\text{cost}_{eval}(SCond) + \text{cost}_{eval}(Succ) + \text{cost}_{act}(Succ))$$

where maxS is the maximum size of $\#S_i$ reached during the computation. Again,
optimisation can remove the cost_{eval}(Succ) term.

If a group of constraints are mutually data dependent, then they must be
implemented by a single phase. In the case that there is a group of two mutually
dependent constraints (ie, $C_k < C_l$ and $C_l < C_k$), both with outer quantifier
$\forall s : S_i$, approach 2 has the form

running := true;
while (running) do
  (running := false;
   for s : $S_i$ do loop1;
   for s : $S_i$ do loop2)

where loop1 is the code:

if $SCond_1$ then
  if Succ1 then skip
  else ($s$.op1(); running := true)

and loop2 is:

if $SCond_2$ then
  if Succ2 then skip
  else ($s$.op2(); running := true)

op1 implements the succedent of the first constraint, and op2 that of the second.
The order of the constraints is assumed to indicate their relative priority, so that
the first is executed as many times as possible before the second is attempted, and so forth.

Similar extensions can be made for approach 3, and for approach 2 and 3 with distinct source entities for the outer quantifier.

For approach 3, the search operation becomes:

```plaintext
search() : Boolean
(for s : S_i do
  if SCond1 then
    if Succ1 then skip
    else (s.op1(); return true));
(for s : S_i do
  if SCond2 then
    if Succ2 then skip
    else (s.op2(); return true));
return false
```

The same pattern is used for constraint groups of size 3 or more: all possible applications of the first constraint are attempted first, followed by all possible applications of the second constraint, and so forth.

By construction therefore, we can establish termination of the phased implementation of UML-RSDS transformations, and completeness and correctness of this implementation approach in the sense of [46]: Cons can be established by the approach, and any computation that satisfies the approach establishes Cons. This also proves semantic correctness of the implementation as defined in the correctness obligation (II) of Section B.

**Code examples** Examples are the model migration [31] and UML to relational database [32] case studies.

**Related patterns** Object indexing can be used to find the target elements constructed for particular source elements in earlier phases.

## 4 Composition of model transformations

Current model transformation languages support a wide range of composition mechanisms, classified as *internal* or *external* compositions. Table 3 summarises the internal composition mechanisms of some well-known transformation languages.

Rule inheritance is used in ATL to extend existing rules with additional functionality. This facilitates the evolution of a transformation to deal with extended source metamodels (e.g., if a source entity $S_j$ inherits a source entity $S_i$, and extends it by additional attributes, the transformation rule that maps $S_j$ can inherit the original rule for $S_i$, and extend it with mappings for the new attributes). However, as [25] explain, this form of adaption is not very flexible
because the inheritance mechanism requires explicit named reference to the rules being extended.

For inter-rule invocation, we can distinguish *implicit* invocation, where the caller does not explicitly name or identify the called rule, and *explicit* invocation, where the called rule is explicitly identified by the caller.

In ATL, implicit invocation is used when a rule needs to convert source model elements of entity $S_2$ subordinate to one of its source elements $s : S_1$ to corresponding elements of a target entity $T_2$, in order to assemble the subelements of the image $t : T_1$ of $s$.

This is in contrast to explicit invocation, for example in QVT-Relations, where the conversion is carried out by calling an explicitly named rule.

The implicit approach has the advantage that changes to the way that subordinate elements are mapped does not affect the caller, since the caller is unaware of what rules are used to carry out these mappings. Therefore the part of the transformation that manages the $S_2$ to $T_2$ mapping can be modified in its structure and details without affecting the syntax of the $S_1$ to $T_1$ mapping.

There may be a small loss of efficiency in implicit calls, due to the need to resolve the actual target of the call. In reasoning about the rules, we need to analyse all possible cases of rules that may be invoked by the implicit call.

ETL has a similar mechanism, which enables a rule to refer to $s.ssub equivalents()$ to recover the $T_2$ elements that have been mapped from the $s.ssub$ elements, or to execute rules to create the $T_2$ elements if they do not already exist [24].

Other transformation languages also support implicit calls or equivalent mechanisms. In UML-RSDS, a constraint mapping an element $s : S_1$ with a feature $s.ssub : Set(S2)$ to an element $t : T1$ will look up the $T2$ elements corresponding to $s.ssub$, and this lookup is independent of the construction process used.

---

### Table 3. Internal composition mechanisms

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Application</th>
<th>Supported by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule inheritance</td>
<td>Reuse by extension</td>
<td>ATL, ETL</td>
</tr>
<tr>
<td>Inter-rule</td>
<td>Factorisation, eg.</td>
<td>All</td>
</tr>
<tr>
<td>invocation</td>
<td>by source model</td>
<td>languages</td>
</tr>
<tr>
<td>Factorisation, eg.</td>
<td>All</td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>Decomposition</td>
<td>ATL, Viatra,</td>
</tr>
<tr>
<td>combination of</td>
<td>of transformation</td>
<td>Kermet, ETL,</td>
</tr>
<tr>
<td>rules</td>
<td>process</td>
<td>UML-RSDS</td>
</tr>
<tr>
<td>Fixpoint</td>
<td>Required for</td>
<td>QVT-Relations,</td>
</tr>
<tr>
<td>iteration</td>
<td>update-in-place</td>
<td>Viatra, ATL, ETL,</td>
</tr>
<tr>
<td>of rules</td>
<td>transformations</td>
<td>UML-RSDS</td>
</tr>
<tr>
<td>Module</td>
<td>Reuse, adaption</td>
<td>ATL, ETL,</td>
</tr>
<tr>
<td>superposition</td>
<td>adaptation</td>
<td>QVT-Relations</td>
</tr>
</tbody>
</table>

---

- ATL
- ETL
- Viatra
- Kermet
- UML-RSDS
- QVT-Relations
to derive these elements:

\[ \forall s : S1 \cdot \exists t : T1 \cdot t.tsub = T2[s.ssub.id] \]

This however does not create the subordinate elements if they do not already exist, instead the constraints must be ordered so that all constraints that create \(T2\) elements precede constraints that use them.

Tracing mechanisms can be used to achieve a similar effect, in Kermeta and Viatra.

Explicit rule invocation is available in each of the transformation languages. The imperative and hybrid languages (ATL, Kermeta, Viatra, ETL and UML-RSDS) also support other procedural combinations of rules: conditional choice, bounded and unbounded iteration. Such mechanisms provide powerful facilities for defining complex transformations, however they can also lead to transformation definitions that are difficult to modify or adapt, because of intricate control and data dependencies. The use of declarative features where possible is therefore preferred for hybrid languages.

Fixpoint iteration is used to iterate a rule repeatedly until it cannot be applied any further. ATL matched rules and QVT-Relations top-level rules are executed using fixpoint iteration. In UML-RSDS constraints are analysed to identify if fixpoint iteration is potentially required: this is the case if (the procedural interpretation of) the constraint can write data which it also reads. A fixpoint implementation is then used for such constraints.

Module superposition overrides elements (rules, auxiliary operations) of a transformation module with a module containing new versions of these elements, and possibly additional elements [56]. Elements of the original module that have redefinitions in the new module are replaced by these redefinitions, other elements are retained. Other elements of the new module are also retained. This allows adaption and reuse of the original module, especially for situations where only part of a module needs to be updated. A similar facility can be defined for ETL. QVT-Relations has a related concept of extension transformations [13]. In UML-RSDS and Kermeta the usual concept of inheritance on classes can be used to achieve this effect, Kermeta also has an aspect-oriented mechanism of class merging which corresponds to a form of superposition. Use-case inheritance in UML-RSDS replaces named constraints in the parent use case by same-named constraints in the subclass and otherwise takes the union of the set of constraints. In order that this specialisation is substitutable, new precondition assumptions cannot be added.

Proposals for the union and fork combination of transformation rules are made in [49]. Union can be applied to two constraints of the form

\[ \forall s : S_i \cdot SCond_i \text{ implies } Post_i \]

\[ \forall s : S_i \cdot SCond_2 \text{ implies } Post_2 \]
to produce a (logically equivalent) merged constraint

\[ \forall s : S_i \cdot (SCond_1 \text{ or } SCond_2) \implies ((SCond_1 \implies Post_1) \text{ and } (SCond_2 \implies Post_2)) \]

It can be regarded as an abstraction operator on transformations: the implementation of the first version consists of all steps of the first constraint followed by all steps of the second, whilst the merged version permits arbitrary interleaving of such steps.

External composition differs from internal composition in that the composition combines transformations which are themselves complete transformation units, and which may be written using different transformation languages. In addition, the specifications of the component transformations may not be available and the transformations generally cannot be modified, but must be reused in their existing form.

Sequential chaining of transformations is the most common form of external composition. It permits two separate transformations

\[ \tau_1 : S \rightarrow R \]
\[ \tau_2 : R \rightarrow T \]

to be combined in series \( \tau_1; \tau_2 \) to achieve the effects of both, provided that:

- The assumptions of \( \tau_2 \) are established by \( \tau_1 \)
- \( \tau_2 \) does not overwrite any effects of \( \tau_1 \) which are required for the overall composed transformation.

Conditional execution of a transformation enables its application to be skipped if a necessary condition does not hold.

Fixpoint iteration of a transformation applies the transformation (which must be endogenous) repeatedly until no change in the model occurs. As in the case of fixpoint iteration of a rule, confluence and termination need to be established for the iteration.

Invocation as a subroutine permits a transformation to be invoked within a larger transformation process. As with sequential composition, the assumptions of the invoked transformation should be true at the point of call.

Conjunction is a particular mechanism supported in UML-RSDS, and is expressed by one transformation \( \tau_1 \) (defined as a UML use case) having an extend dependency to another, \( \tau_2 \). The conjunction of \( \tau_1 \) and \( \tau_2 \) achieves the postconditions of both transformations, assuming that all the preconditions of both transformations hold. Unlike use-case inheritance, constraints of \( \tau_1 \) are not replaced, and new precondition constraints can be added.

Transformations may be specified with data parameters, such as integers or booleans, permitting different options for the transformation effect to be externally controlled. These are in addition to the usual source and target model parameters. This facility allows a limited degree of flexibility.
Higher-order parameterisation enables transformations to be supplied as parameters to other transformations. This permits very flexible reuse and combination of transformations. In UML-RSDS this facility is implemented in a declarative manner by defining *predicate parameters* which can be instantiated with any logical predicate, this predicate is then substituted in place of the formal parameter throughout the text of the main transformation. The correctness of the result is then analysed using the composed text. Higher-order transformations can be defined in ATL [54] to provide a higher-order parameterisation facility.

Generic transformations enable a general-purpose transformation to be specified independently of specific metamodels, using minimal metamodels which are sufficient to express the transformation, and then to be instantiated as required with particular language elements for reuse within a transformation process, to embed the generic transformation and its metamodels within more general processes and metamodels. This concept is directly supported by Viatra [53] and UML-RSDS, and higher-order transformations can be used to achieve such generic instantiation in QVT and ATL.

The union and fork mechanisms of [49] can also be considered as external combinators of transformations, union can be achieved by conditional combination, and fork corresponds to conjunction.

### 4.1 Transformation composition in UML-RSDS

In UML-RSDS we use logical composition mechanisms such as conjunction and parameterisation in order to flexibly compose components: these mechanisms have the advantage that the components have minimal knowledge about each other and hence can be modified independently of each other.

UML use cases can be composed by means of **extend** and **include** relations between use cases:

**Use case e extends use case m** means that *e* carries out some additional functionality based upon *m*. *e* may apply at particular extension points within *m* to extend specific parts of *m*’s functionality.

**Use case m includes use case f** means that *f* carries out some element of the functionality of *m*, that is, some part of *m*’s functionality is delegated to *f*.

Use cases can also be composed by inheritance.

We can use these composition relationships to structure and compose transformations. In particular, extension can be logically interpreted as a conjunction of the base and extended use cases (the component preconditions $Asm_e$ and $Asm_m$ are conjoined to form the composed precondition $Asm$, and likewise for the postconditions $Cons$ and $Ens$), and used as a means for separating out different areas of concern into separate transformations. Procedurally, extension can be interpreted as the weaving of the extension into the base by inserting constraints from the extension, in order, within the ordering of base constraints,
so that the syntactic non-interference properties between constraints are established for the merged use case.

We described in Section 2.6 how the ‘entity splitting’ pattern can remove tangling problems within constraints. By separating tangled aspects into separate transformations related by extend we can go further in defining independent specifications of such aspects.

The example of [25] can be decomposed so that the main use case sourceToViews includes the constraints defining the mapping of elements to view objects, this is extended by a use case sourceToModels containing the constraints creating model objects, linked to the views, and also extended by a use case sourceToControllers that contains the constraints defining controllers based on the views (Figure 9).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure9.png}
\caption{Decomposition using extend}
\end{figure}

The sourceToViews use case could be:

\begin{align*}
\forall e : \text{OpenElement} \cdot \\
\exists t : \text{OpenView} \cdot t.\text{obsId} = e.\text{id} \\
\forall e : \text{MultipleChoiceElement} \cdot \\
\exists t : \text{MultipleChoiceView} \cdot t.\text{obsId} = e.\text{id}
\end{align*}

The sourceToModels use case could be:

\begin{align*}
\forall e : \text{OpenElement} \cdot \\
\exists t : \text{Open} \cdot t.\text{modId} = e.\text{id} \text{ and } t.\text{observer} = \text{OpenView}[e.\text{id}] \\
\forall e : \text{MultipleChoiceElement} \cdot \\
\exists t : \text{MultipleChoice} \cdot t.\text{modId} = e.\text{id} \text{ and } t.\text{observer} = \text{MultipleChoiceView}[e.\text{id}]
\end{align*}

The sourceToControllers use case could be:

\begin{align*}
\forall e : \text{OpenElement} \cdot \\
\exists t : \text{OpenController} \cdot t.\text{view} = \text{OpenView}[e.\text{id}] \\
\forall e : \text{MultipleChoiceElement} \cdot \\
\exists t : \text{MultipleChoiceController} \cdot t.\text{view} = \text{MultipleChoiceView}[e.\text{id}]
\end{align*}
The inverse relationship controller of view is set implicitly by the assignments to view, in the implementation generated by the UML-RSDS tools.

This decomposition can also be used in the case of transformations which affect multiple connected models, such as UML class diagrams, OCL constraints and state machines.

The decomposition of a transformation $\tau$ into two sub-transformations $\tau_1$ and $\tau_2$ composed into $\tau$ by extend is possible if: wr($\tau$) can be partitioned into two disjoint non-empty sets $V_1$, $V_2$, such that the set slice($\tau$, $V_1$) of constraints $C$ of $\tau$ with wr($C$) $\subseteq$ $V_1$ is non-empty and disjoint from slice($\tau$, $V_2$), which is also non-empty, and then for $\tau_1 = \text{slice}(\tau, V_1)$ and $\tau_2 = \text{slice}(\tau, V_2)$, $V_1$ is disjoint from rd($\tau_2$) and $V_2$ is disjoint from rd($\tau_1$). If $V_2 \cap \text{rd}(\tau_1) = \emptyset$ holds, but $V_1 \cap \text{rd}(\tau_2) \neq \emptyset$, then $\tau$ can instead be sequentially decomposed into $\tau_1; \tau_2$ using include.

In the UML-to-relational database example, it can be seen from the structure of the transformation and the target metamodel that the Key and ForeignKey entities both depend on Table and Column, but are not depended upon by any entity, and so the creation of keys and foreign keys could be performed as extensions of the main transformation use case. The extension point in this case is between C2 and C5.

Such decomposition may make it easier to evolve the transformation, as a consequence of metamodel evolution: the core parts of a metamodel may remain unchanged by an evolution, and so the corresponding components of the transformation will remain unaffected. Extension use cases may be modified, and additional or replacement extension use cases introduced.

The include mechanism can be used to sequentially chain together a series of use cases, or to compose these using a workflow activity. For example, removal of redundant and multiple inheritance followed by the UML to relational database transformation. The included transformations are relatively independent of each other, although a preceding use case must ensure the assumptions of their successor. Included use cases can be shared by several different including use cases.

In the general case an including use case can have an activity which invokes the included use cases according to a specific algorithm or workflow.

Inheritance of use cases can be used if there are two or more alternative constraints for a particular step (eg., event slicing versus data slicing, for the state machine slicing algorithms of [30]).

4.2 Transformation parameterisation

Composition of UML-RSDS transformations can be achieved by the use of parameterised transformations, as proposed in [57]. Parameters can be predicates, so permitting the separation of some internal parts of constraints from other parts, eg., LPost from GPost. In particular, updates that are common to several constraint succedents [25] can be factored out into a single predicate which is supplied as a parameter and can be changed independently of the main use case. As an example, consider the creation of tracing information by rules. Rules in
the primary use case can have the form

\[ \forall s : S_i \cdot S\text{Cond} \implies \exists t : T_j \cdot T\text{Cond} \text{ and Post and GenTrace} \]

where GenTrace is the predicate parameter, assumed to have variables \( s \) and \( t \).

The usual actual predicate parameter supplied as the argument GenTrace would be:

\[ \exists tr : \text{Trace} \cdot \text{tr.source} = s \text{ and tr.target} = t \]

To switch off tracing, an actual parameter \( \text{true} \) can be supplied instead.

UML-RSDS supports general parameterisation of transformations, both by basic values such as integers, booleans and strings, and by expressions and predicates. The instantiated specification is formed by textual substitution of the actual parameters for the formal parameters in the transformation constraints. Analysis is performed on the substituted result.

### 4.3 Generic transformations

Some transformation problems, such as the computation of the transitive closure of an association [53], computing the flattening of a composition structure, or the dual of a node-edge network [16], can be considered to be generic transformations, independent of specific metamodels, and only assuming certain characteristic structures of entities and features. This permits reuse of the transformation for many different metamodels. The generic transformation reads and modifies only particular structures within a model, leaving other parts of the model unchanged.

One way to specify a generic transformation is to define it as a transformation \( \tau : S_0 \rightarrow T_0 \) between the minimal metamodel structures that it concerns, and then to instantiate it by embedding these metamodels within specific source and target languages \( S \) and \( T \).

For example, a generic transformation \( \tau \) to produce a transitive closure \( \text{ancestor} \) of a many-many relation \( \text{parent} \) on an entity \( E \) has \( S_0 \) consisting of \( E \) and \( \text{parent} \), and \( T_0 \) consisting of \( E \), \( \text{parent} \) and \( \text{ancestor} \). Its Cons constraints are:

\[ \forall e : \text{E} \cdot e.\text{parent} \subseteq e.\text{ancestor} \]
\[ \forall e : \text{E} \cdot e.\text{parent.ancestor} \subseteq e.\text{ancestor} \]

Its Inv constraint is: \( \forall e : \text{E} \cdot e.\text{ancestor} \subseteq e.\text{parent} \cup e.\text{parent.ancestor} \)

In UML-RSDS we implement generic transformations by the general parameterisation mechanism used for predicate parameterisation. To instantiate a generic transformation \( \tau : S_0 \rightarrow T_0 \) to \( S \) and \( T \), the developer must define consistent embeddings

\[ I : S_0 \rightarrow S \]
\[ J : T_0 \rightarrow T \]
of the languages, such that $I$ and $J$ map entities to entities, and features to expressions, and:

\[
\begin{align*}
I(x) &= J(x) \quad \text{for } x \in S_0 \cap T_0 \\
\text{rd}(I(x)) \cap \text{rd}(J(y)) &= \{\} \quad \text{for } x \neq y \\
\text{if } f : E \rightarrow \text{Typ} \text{ then } I(f) : I(E) \rightarrow \text{Typ} \\
\text{if } f : E_1 \rightarrow E_2 \text{ then } I(f) : I(E_1) \rightarrow I(E_2) \\
\text{if } f : E_1 \rightarrow \mathbb{F}(E_2) \text{ then } I(f) : I(E_1) \rightarrow \mathbb{F}(I(E_2)) \\
\text{if } f : E_1 \rightarrow \text{seq}(E_2) \text{ then } I(f) : I(E_1) \rightarrow \text{seq}(I(E_2))
\end{align*}
\]

and likewise for $J$. The instantiations must satisfy any assumptions in $\text{Asm}$ of $\tau$. If $E \in \text{wr}(\tau)$ is a concrete entity then $E$ can only be instantiated by a single concrete entity. If $f \in \text{wr}(\tau)$, then $f$ must be instantiated by a single writable feature.

The substitution $\tau[I(x)/x, J(y)/y]$ is then used as a transformation from $S$ to $T$. Its assumptions are the substituted forms $\text{Asm}[I(x)/x, J(y)/y]$ of the generic assumptions, likewise for its $\text{Ens}$, $\text{Inv}$ and $\text{Cons}$ constraints.

For the transitive closure example, if we have $S$ containing entity $\text{Ent}$ and self-association $r$ on $\text{Ent}$, and we want to produce a transitive closure association $rr$ on $\text{Ent}$ in $T$, the generic $\tau$ is instantiated with

\[
\begin{align*}
I(E) &= \text{Ent} \\
I(\text{parent}) &= r \\
J(E) &= \text{Ent} \\
J(\text{ancestor}) &= rr
\end{align*}
\]

which satisfies the conditions for consistent instantiation. The resulting instantiated $\text{Cons}$ constraints are:

\[
\begin{align*}
\forall e : \text{Ent} \cdot e.r \subseteq e.rr \\
\forall e : \text{Ent} \cdot e.r.rr \subseteq e.rr
\end{align*}
\]

It is possible also to instantiate $\text{parent}$ by a composition $r_1.r_2$ of two associations, or by longer chains of compositions which combine to form a set-valued self association on an entity.

Generic transformations should be provided with general proofs of their correctness, termination and confluence properties. These proofs can be used to assure the correct behaviour of the transformation when it is reused in a new context [36].

5 Evaluation

We have carried out a wide variety of transformations using UML-RSDS:

- Refinement transformations, including the UML to relational database transformation [32] and UML to J2EE and to Java (incorporated into the UML-RSDS tools).
– Re-expression transformations, including a mapping from state machines to activity diagrams [31] and other migration examples [33, 34].
– Abstraction transformations, including state machine slicing algorithms [29], which have also been incorporated within the UML-RSDS toolset.
– Quality-improvement transformations, such as class diagram rationalisation by the removal of multiple inheritance and the removal of duplicated attributes [23].

The largest metamodels considered were those for the state machine to activity diagram mapping (31 source entities and features, 35 target entities and features). This is an actual industrial problem, as is the GMF migration example of [34], which involves a highly complex restructuring and a combination of update-in-place and exogenous transformation mechanisms.

A formal evaluation of UML-RSDS, ATL, GrGen, Kermeta and QVT-Relations on the class diagram rationalisation problem was carried out in [23]. The evaluation was based on the software quality standard ISO/IEC 9126-1 [17], and on comparison of the six software quality characteristics of functionality, reliability, usability, efficiency, maintainability and portability for these transformation approaches. For each characteristic, relevant subcharacteristics were identified for transformation tools and transformation specifications/designs, and then measurable attributes were assigned to the subcharacteristics.

Some examples of this decomposition are:

**Functionality** Subcharacteristics:
- **Suitability** transformation abstraction level; transformation size; transformation complexity; effectiveness; development effort; execution time.
- **Accuracy** transformation correctness; transformation completeness.
- **Interoperability** tool is interoperable with Eclipse and other model-driven engineering environments; transformations can be externally composed.

**Usability** Subcharacteristics:
- **Understandability** Measured by survey results.
- **Learnability** Measured by survey results.
- **Attractiveness** Measured by survey results.

**Efficiency** Subcharacteristics:
- **Time behaviour** execution time; maximum capability.

Specific measures include number of lines of code/specification for transformation size, and expression complexity + structural complexity for transformation complexity. Development effort is measured in minutes required to write the specification (not including time taken to understand the problem).

Table 4 shows the values of these measures for the class diagram rationalisation solutions.

In each case, development was carried out by an expert in the individual language. The ATL and QVT-Relations solutions only addressed two of the three main requirements of the case study, due to the difficulty of expressing update-in-place transformations in these languages. The solutions can be tested at [51].
Table 4. Suitability measures of transformation solutions

<table>
<thead>
<tr>
<th>Solution</th>
<th>Size (LOC)</th>
<th>Complexity</th>
<th>Developer effort (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML-RSDS</td>
<td>24</td>
<td>132</td>
<td>100</td>
</tr>
<tr>
<td>GrGen</td>
<td>102</td>
<td>102</td>
<td>100</td>
</tr>
<tr>
<td>Kermeta</td>
<td>653</td>
<td>318</td>
<td>440</td>
</tr>
<tr>
<td>ATL</td>
<td>69</td>
<td>121</td>
<td>190</td>
</tr>
<tr>
<td>QVT-Relations</td>
<td>83</td>
<td>190</td>
<td>280</td>
</tr>
</tbody>
</table>

Usability, learnability and attractiveness were measured based on the responses to an online survey (at: http://www.inf.kcl.ac.uk/pg/kolahdou/survey.html). The survey was designed based on previous experience of surveys of program comprehension. There are five questions, one asks the participant for their existing level of knowledge of the transformation language (ATL, GrGen, Kermeta, QVT-Relations or UML-RSDS), other questions ask the participant to identify where in the transformation code a particular requirement is implemented, and ask for their estimates of the understandability, modularity and attractiveness of the specification. The participants in the survey were ten UK and European informatics professionals, ranging from PhD students to senior academics.

Table 5 summarises the results of the survey, measures are in the range 0 to 4.

Table 5. Usability summaries

<table>
<thead>
<tr>
<th>Approach</th>
<th>Understandability</th>
<th>Attractiveness</th>
<th>Learnability</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML-RSDS</td>
<td>1.4</td>
<td>2.2</td>
<td>1.3</td>
</tr>
<tr>
<td>GrGen</td>
<td>2.3</td>
<td>2</td>
<td>1.9</td>
</tr>
<tr>
<td>Kermeta</td>
<td>1.9</td>
<td>0.9</td>
<td>1.8</td>
</tr>
<tr>
<td>ATL</td>
<td>1.8</td>
<td>2.6</td>
<td>0.6</td>
</tr>
<tr>
<td>QVT-R</td>
<td>0.7</td>
<td>1.8</td>
<td>0.4</td>
</tr>
</tbody>
</table>

The overall usability ranking of the approaches is therefore: GrGen (6.2); ATL (5.0); UML-RSDS (4.9); Kermeta (4.6); QVT-R (2.9).

This ordering is consistent with the orderings of UML-RSDS, GrGen and Kermeta with regard to complexity (Figure 10), modularity, and developer effort.

The correlation between empirical measures of usability and measures expected to be predictors of usability gives some assurance of the correctness of the usability measures.

The overall result of the evaluation placed GrGen first with 15 good or high values for the 24 subcharacteristics of software quality considered, and UML-RSDS second with 14. Kermeta (with 11) was third [23].

Together, these case studies have demonstrated that the constraint-based approach is versatile and applicable to a wide range of transformation problems.

The efficiency of the generated code has also been evaluated. We carried out the tests described in [2] on our version of the UML to relational database...
mapping (rules $C_1$ and $C_2$, since the test cases of [2] do not include inheritance). Table 6 shows the resulting execution times, on a Java 1.6 platform, on a Windows 7 laptop with Intel i3 processor, 2.53GHz clock and 4GB of RAM.

<table>
<thead>
<tr>
<th>Test case</th>
<th>Number of classes</th>
<th>Number of attributes per class</th>
<th>Execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>100</td>
<td>0.031</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>100</td>
<td>0.062</td>
</tr>
<tr>
<td>3</td>
<td>300</td>
<td>100</td>
<td>0.109</td>
</tr>
<tr>
<td>4</td>
<td>400</td>
<td>100</td>
<td>0.125</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>100</td>
<td>0.156</td>
</tr>
</tbody>
</table>

Table 6. Test case results

All classes and attributes were set as ‘Persistent’ in the source model, so that these tests correspond to the ‘worst case’ tests (Figure 1(b)) in [2]. As expected, the execution time increases linearly with respect to the input model size: rule $C_1$ is executed $100 \times cs$ times, where $cs$ is the number of classes, and it creates this number of columns. Rule $C_2$ is executed $cs$ times, creating $cs$ tables, involving $100 \times cs$ lookups of columns by their primary keys.

Although our hardware specification is below that of [2], the execution times obtained from our tests are substantially lower than the corresponding times for ATL (25 seconds for test case 3, over 60 seconds for test case 5) and the Medini implementation of QVT-Relations (which took 100 seconds even for test case 1) [2].
Maximum capability was measured in [23] by identifying the maximum model size which the transformation implementations could process without stack overflow or other memory error. Table 7 gives results for the case study of [23]. In general, the approaches (GrGen and UML-RSDS) which compile transformation specifications into executable code, are significantly more efficient and have higher capabilities than interpreted transformation approaches.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Maximum model size (number of elements)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GrGen</td>
<td>50,000</td>
</tr>
<tr>
<td>UML-RSDS</td>
<td>100,000</td>
</tr>
<tr>
<td>Kermeta</td>
<td>10,000</td>
</tr>
<tr>
<td>QVT-Relations</td>
<td>5,000</td>
</tr>
</tbody>
</table>

Table 7. Maximum capability results

Remaining deficiencies of UML-RSDS are the lack of an interface to Eclipse, and the lack of graphical specifications of transformations, e.g., by means of concrete or abstract syntax diagrams as in QVT-Relations. Transformation verification is not self-contained, but requires access to a toolkit for the B formalism.

The UML-RSDS tools, together with examples of transformation specification and implementation, are available at: http://www.dcs.kcl.ac.uk/staff/kcl.uml2web/.

6 Related Work

Model transformation approaches have been characterised as declarative: focussed upon the definition of relations between the source and target models; imperative: using algorithms to compute the target model from the source; or hybrid: combining the two techniques. Declarative approaches include QVT-Relations and Triple graph grammars [50]. Imperative approaches include Kermeta and QVT-Operations. Hybrid approaches include Viatra and ATL.

Table 8 summarises the differences between these approaches.

UML-RSDS is primarily intended for use in a declarative manner: transformations should be defined as use cases, specified by sets of assumption (precondition) and postcondition constraints. From these, executable code satisfying the constraints can be automatically synthesised. However, developers have the freedom to define their own design-level descriptions of transformations (e.g., to define a recursive descent strategy) in terms of operations of metaclasses, in a similar manner to Kermeta. Code can be generated automatically from such designs.

The constraint-based approach of UML-RSDS is at a higher level of abstraction than current model transformation approaches, because it describes the global behaviour of a transformation, independent of specific transformation rules. Such a high-level form of specification for transformations was recommended by [6] in order to provide effective transformation reuse and extension.
Relative to QVT and ATL, the constraint-based approach is more abstract because it does not use implicit or explicit invocations between rules. In this respect it is similar to graph transformation approaches and to triple-graph grammars [50]: if a UML-RSDS transformation $\tau$ has an inverse $\tau^-$, then these two transformations together can be used to derive TGG-style rules: an incremental modification to the source model (such as by creating a new element) may result in the antecedents of one or more constraints of $\tau$ becoming true, and hence the consequent changes to the target model derived from the succedents of these constraints can be determined.

Likewise, deletion of a source model element may make the antecedent of the contrapositive of a constraint of $\tau^-$ true, with a consequent change to the target model being specified by the succedent of the contrapositive.

For example, the constraint $C6$ of the UML to relational database example:

$$\forall p: \text{Package} \cdot$$
$$\exists s: \text{Schema} \cdot s.rdbId = p.umlId \land$$
$$s.name = p.name \land s.kind = p.kind \land$$
$$s.tables = Table[p.elements.umlId]$$

has inverse constraint

$$\forall s: \text{Schema} \cdot$$
$$\exists p: \text{Package} \cdot p.umlId = s.rdbId \land$$
$$p.name = s.name \land p.kind = s.kind \land$$
$$p.elements = \text{PackageElement}[s.tables.rdbId]$$

and the contrapositive of this is:

$$\forall s: \text{Schema} \cdot$$
$$\not(\exists p: \text{Package} \cdot p.umlId = s.rdbId \land$$
$$p.name = s.name \land p.kind = s.kind \land$$
$$p.elements = \text{PackageElement}[s.tables.rdbId]) \Rightarrow$$
$$s.isDeleted()$$
UML-RSDS provides more structure to the transformation computation than a pure graph-transformation approach, by organising the execution into phases. This assists in verification and comprehension, at a possible cost in efficiency.

In [4], specifications of the conjunctive-implicative form are derived from model transformation implementations in triple graph grammars and QVT-Relations, in order to analyse properties of the transformations, such as definedness and determinacy. This form of specification is therefore implicitly present in QVT-Relations and other transformation languages, and we consider that it is preferable for such logical specifications to be defined prior to detailed coding of the transformation rules, in order to identify possible errors at an earlier development stage.

In [46], a process for deriving graph transformation rules from TGG-style graph pattern specifications is defined, and the resulting transformation is proved to be sound and complete with respect to the specification. Instead of our variant functions to establish termination, [46] use additional negative application conditions to prevent repeated application of rules. This approach is however more restricted in the scope of its specifications: only input-preserving transformations are considered, and a restricted expression language is used, whereas we consider specifications using the full OCL standard library. Confluence is not established by the approach of [46], instead the graph transformation rules can generate any result model that satisfies the specification patterns. Efficiency of the graph transformation is not evaluated, and the potentially large number of generated negative application conditions for rules may result in low efficiency.

In [15], a general method transML for model transformation development is described, using multiple levels of description (requirements, specification, high-level design and low-level design). Our specification predicates Asm, Cons, Inv and Ens play the same role as the pattern-based specification language of [15]. In comparison to transML, our approach is more lightweight, utilising only UML and OCL notations, and avoiding the explicit construction of designs. Instead, designs and implementations are generated from specifications, which are made the focus of transformation development activities.

Patterns for model transformations have been proposed by several researchers [3, 5, 9, 1, 16, 18]. The patterns include techniques to optimise transformation specifications, particularly to make the evaluation of conditions on models more efficient [5], to enhance the model-processing capabilities of transformations [3, 1], to organise the processing of transformations [16], and to organise the structure of transformations for improved modularity and reuse [9, 6, 16, 18].

In [47, 48] the conjunctive-implicative form is used as the basis of model transformation specifications in constructive type theory, with the $\exists x.P$ quantifier interpreted as an obligation to construct a witness element $x$ satisfying $P$. A partial ordering of entities is used to successively construct such witnesses. In this paper we show how this approach may be carried out using first order logic, with systematic strategies used to derive transformation implementations from transformation specifications, based on the detailed structure of the specifications. The correctness of the strategies are already established and do not
need to be re-proved for each transformation, although side conditions (such as data-dependency conditions and existence of variants) need to be established.

In [3], a transformation specification pattern is introduced, Transformation parameters, to represent the case where some auxiliary information is needed to configure a transformation. This could be considered as a special case of the auxiliary metamodel pattern. A pattern Multiple matching is also defined, to simulate rules with multiple element matching on their antecedent side, using single element matching. Auxiliary metamodel can be used to implement this pattern by associating together matching groups of objects. In [5], specific patterns for the optimization of rule executions are defined: short-circuit boolean expressions evaluation; determining an opposite relationship; collections; usage of iterators; finding constant expressions. Of these, the second and fifth can be considered as special cases of the auxiliary metamodel pattern. The patterns are relevant to UML-RSDS and could be used to make the generated code of transformations more efficient. In [9] three general styles for transformations are described: source-driven transformations, target-driven transformations, and aspect-driven transformations. The first two correspond to entity-splitting and entity-merging in our terminology, the third corresponds to parallel composition of transformations using extend, where different aspects of the target model are constructed using separate parallel transformations. Patterns specific to graph transformation languages are defined in [1]: leaf collector pattern; map-using-link pattern; transitive closure pattern; proxy generator idiom. Of these, map-using-link is a special case of the auxiliary metamodel pattern. Computation of a transitive closure can be considered as a generic transformation, as in Viatra [53]. In [16] five model transformation design patterns are described: Mapping, Refinement, Abstraction, which can be considered as general categories of model transformation (the cases of 1-to-1, 1-to-many, and many-to-1 relations between the source and target model elements, respectively), Duality, for computing the edge-node dual of a software engineering model, and Flattening, for contracting a source model object hierarchy. The latter two are also candidates for formalisation as generic transformations. In [18] four patterns are defined: Element mapping, Element mapping with variability, which are special cases of entity splitting, and Attribute mapping and Link mapping, which are elementary aspects of transformations specified with the conjunctive-implicative form.

Our work extends these previous works on model transformation patterns by combining patterns into an overall process for developing model transformation designs and implementations from their specifications, and by automating the introduction of implementation patterns.

7 Summary

The approach described here provides a systematic specification and development approach for model transformations, based upon declarative specifications of transformations using constraints. We have described techniques for the structuring of such transformation specifications (conjunctive-implicative form, aux-
iliary metamodel), and described how executable implementations of these specifications can be automatically derived, so that the implementations are correct with respect to the specifications.

The novelty of this work consists of the higher level of abstraction provided for transformations (with transformations considered as single operations or use cases, defined by pre and post conditions) compared to other approaches, which focus on individual rules within a transformation. We also differ from most other approaches in that we try to utilise standard UML notations as far as possible, to improve the reusability of model transformations and minimise the amount of training required to use the approach. The extensive automation provided supports rapid and agile development of transformations for use in a wide range of application scenarios.

Acknowledgement

The work presented here was carried out in the EPSRC HoRTMoDA project at King’s College London.

References

11. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.


A Model transformation semantics

The four-level metamodelling framework of UML is used as the context for UML-RSDS transformations [44]. This framework consists of levels M0: models which consist of runtime instances of M1 models, which are user models such as class diagrams, which are in turn instances of metamodels M2, such as the definition of the class diagram language itself. Level M3 contains the EMOF and CMOF languages for defining metamodels. Other languages such as EMF Ecore could alternatively be used at level M3 [10].

Most transformations operate upon M1 level models, so the M2 level is termed the language level (the models at this level define the languages which the transformation...
relates) and the M1 level is termed the model level (the models which the transformation operates upon).

For each model M at levels M2 and M3, there is (i) a logical language \( \mathcal{L}_M \) that corresponds to \( M \), and (ii) a logical theory \( \Gamma_M \) in \( \mathcal{L}_M \), which defines the semantic meaning of \( M \), including any internal constraints of \( M \) [32]. The set-theory based axiomatic semantics of UML described in Chapter 6 of [27] is used to define \( \Gamma_M \) and \( \mathcal{L}_M \). If \( M \) at level M1 is itself a UML-based model to which a semantics \( \Gamma_M \) can be assigned, then also \( \Gamma_M \) and \( \mathcal{L}_M \) will be defined for \( M \).

\( \mathcal{L}_M \) consists of type symbols for each type defined in \( M \), including primitive types such as integers, reals, booleans and strings which are normally included in models, and types \( E \) for each entity type \( E \) defined in \( M \). The boolean operators and, implies, forAll, exists, one of OCL are semantically interpreted by the logical connectives \( \land, \lor, \exists, \forall \) respectively. For each entity \( C \) of \( M \) there are logical attribute symbols \( f(c: C): \text{Typ}' \) for each data feature \( f \) of type \( \text{Typ} \) in the feature set of \( C \), where \( \text{Typ}' \) is the semantic type corresponding to \( \text{Typ} \), and action symbols \( op(c: C, p: P) \) for each operation \( op(p: P) \) in the features of \( C \).

Attributes and single-valued associations \( att : \text{Typ} \) of \( C \) are essentially represented as functions \( att : C \to \text{Typ}' \), set-valued associations \( f \) are represented as functions \( f : C \to \mathcal{F}(D) \) where \( D \) is the target entity of \( f \), and sequence-valued associations \( f \) are represented as functions \( f : C \to \text{seq}(D) \) where \( D \) is the target entity of \( f \). There are attributes \( \overline{C} \) to denote the set of instances of each entity \( C \) (corresponding to \( C::allInstances() \) in OCL). \( \overline{C} \) represents the extension of \( C \).

Table 9 shows some examples of semantic interpretations of collection types and operators.

<table>
<thead>
<tr>
<th>OCL Term e</th>
<th>Condition</th>
<th>Semantics e'</th>
</tr>
</thead>
<tbody>
<tr>
<td>s-&gt;size()</td>
<td>set or sequence s</td>
<td>cardinality ( #s' )</td>
</tr>
<tr>
<td>s-&gt;size()</td>
<td>bag s</td>
<td>sum of ( s'(x) ) for ( x \in \text{dom}(s') )</td>
</tr>
<tr>
<td>s-&gt;includes(x)</td>
<td>set s</td>
<td>( x' \in s' )</td>
</tr>
<tr>
<td>s-&gt;excludes(x)</td>
<td>set s</td>
<td>( x' \notin s' )</td>
</tr>
<tr>
<td>s-&gt;includes(x)</td>
<td>sequence s</td>
<td>( x' \in \text{ran}(s') )</td>
</tr>
<tr>
<td>s-&gt;excludes(x)</td>
<td>sequence s</td>
<td>( x' \notin \text{ran}(s') )</td>
</tr>
<tr>
<td>s-&gt;asSet()</td>
<td>bag s</td>
<td>( x' \in \text{dom}(s') )</td>
</tr>
<tr>
<td>s-&gt;asSet()</td>
<td>s set</td>
<td>( s' )</td>
</tr>
<tr>
<td>s-&gt;asSet()</td>
<td>s sequence</td>
<td>( \text{ran}(s') )</td>
</tr>
<tr>
<td>s-&gt;excludesAll(t)</td>
<td>sets s and t</td>
<td>( t' \subseteq s' )</td>
</tr>
<tr>
<td>s-&gt;excludesAll(t)</td>
<td>sequences s and t</td>
<td>( \text{ran}(t') \subseteq \text{ran}(s') )</td>
</tr>
<tr>
<td>s-&gt;excludesAll(t)</td>
<td>sets s and t</td>
<td>( s' \cap t' = { } )</td>
</tr>
<tr>
<td>s-&gt;excludesAll(t)</td>
<td>sequences s and t</td>
<td>( \text{ran}(s') \cap \text{ran}(t') = { } )</td>
</tr>
<tr>
<td>s-&gt;sum()</td>
<td>set s</td>
<td>sum of elements of ( s' )</td>
</tr>
<tr>
<td>s-&gt;sum()</td>
<td>sequence s</td>
<td>( #s' = n(s'(1)) + \ldots + s'(n) )</td>
</tr>
</tbody>
</table>

Table 9. Semantic mapping for collection operations

\( \Gamma_M \) includes axioms expressing the multiplicities of association ends, the mutual inverse property of opposite association ends, deletion propagation through composite
aggregations, the existence of generalisation relations, and the logical semantics of any explicit constraints in $M$. Constraints of $M$ are expressed as axioms in $\Gamma_M$.

For a sentence $\varphi$ in $\mathcal{L}_M$, there is the usual notion of logical consequence:

$$\Gamma_M \vdash \varphi$$

means the sentence is provable from the theory of $M$, and so holds in $M$. Inference within a particular language $L$ is specified by the notation $\vdash_L$, if $L$ is not clear from the context.

If $M$ is at the M1 level and is an instance of a language $L$ at the M2 level, then it satisfies all the properties of $\Gamma_L$, although these may not be expressible within $\mathcal{L}_M$ itself. The notation $M \models \varphi$ denotes satisfaction of an $\mathcal{L}_L$ sentence $\varphi$ in $M$.

The collection of M1 level models of an M2 model (language) $L$ is $\text{Models}_L$. We write $m : L$ to mean $m \in \text{Models}_L$. In concrete terms a model $m$ can be considered to be a tuple

$$(E_1, \ldots, E_k, f_1, \ldots, f_l)$$

of the sets $E_i$ of existing instances in $m$ of each entity $E_i$ of $L$, and of the maps $f_j : E_i \rightarrow \text{Typ}$ representing the values in $m$ of data features (attributes and associations) of these instances.

Models are considered isomorphic if they cannot be distinguished on the basis of feature values. Two models $m = ((E_i), (f_j))$ and $m' = ((E_i'), (f'_j))$ of the same language $L$ are isomorphic, $m \equiv m'$, if there is a family $h_i : E_i \rightarrow E_i'$ of bijections such that:

1. $h_i(x) = x' \Rightarrow f_j(x) = f'_j(x')$ for each attribute feature $f_j : \text{Typ}$ of $E_i$, $x \in E_i$
2. $h_i(x) = x' \Rightarrow h_k(f_j(x)) = f'_j(x')$ for each single-valued role feature $f_j : E_k$ of $E_i$, $x \in E_i$
3. $h_i(x) = x' \Rightarrow h_k(f_j(x)) = f'_j(x')$ for each set-valued role feature $f_j$ with element type $E_k$ of $E_i$, $x \in E_i$
4. $h_i(x) = x' \Rightarrow f_j(x) ; h_k = f'_j(x')$ for each sequence-valued role feature $f_j$ with element type $E_k$ of $E_i$, $x \in E_i$.

A model transformation $\tau$ is invertible if there is an inverse transformation $\tau^-$ from $T$ to $S$ such that $\tau$ followed by $\tau^-$ is effectively the identity relation on $\text{Models}_S$.

For separate model input-preserving transformations $\tau$ this means that if

$$(m, \emptyset) \longrightarrow_{\tau} (m, n)$$

and

$$(n, \emptyset) \longrightarrow_{\tau^-} (n, m')$$

then

$$m \equiv m'$$

This is a desirable property since it provides a means to view a model of $T$ as a model of $S$, and therefore to check that the semantic meaning of the source model has been preserved by the transformation.
B A model-driven development process for model transformations

In this section we outline a general model-driven development process for model transformations specified as constraints and operations in UML. We assume that the source and target metamodels of a transformation are specified as MOF class diagrams [44], $S$ and $T$, respectively, possibly with OCL constraints defining restrictions of these languages.

For a transformation $\tau$ from $S$ to $T$, there are four separate predicates which characterise its global properties, and which need to be considered in its specification and design [32]:

1. $\text{Asm}$ – assumptions, expressed in the union language $L_{S\cup T}$ of $S$ and $T$, which can be assumed to be true before the transformation is applied. These may be assertions that the source model is syntactically correct, that the target model is empty, or more specialised assumptions necessary for $\tau$ to be well-defined. $\text{Asm0}$ denotes the assumptions that refer only to entities and features of $S$.

2. $\text{Ens}$ – properties, usually expressed in $L_T$, which the transformation should ensure about the target model at termination of the transformation. These properties usually include the constraints of $T$. For update-in-place transformations, $\text{Ens}$ may refer to the pre-state versions $E@\text{pre}$, $f@\text{pre}$ of model entities $E$ and features $f$.

3. $\text{Inv}$ – invariant properties, usually expressed in $L_{S\cup T}$, which the transformation implementation should maintain during its execution.

4. $\text{Cons}$ – constraints, expressed in $L_{S\cup T}$, which define the transformation as a relationship between the elements of the source and target models, which should hold at termination of the transformation. Update-in-place transformations can be specified as relations between the final and initial states of the single model by using a syntactically distinct copy of the source language, for example by postfixing all its entity and feature names by $@\text{pre}$. Transformations between multiple models can distinguish entities in these models by prefixing them with the model name and $\$: $m1E$, $m2E$, etc.

These predicates can be expressed using OCL notation, this corresponds directly to a fully formal version in the axiomatic UML semantics (Appendix A). Together these predicates give a global and declarative definition of the transformation and its requirements, so that the correctness of a transformation may be analysed at the specification level, independently of how it is implemented.

$\text{Ens}$ is defined separately from $\text{Cons}$ because the same transformation may be used with different variants of the target language, ie., $\text{Ens}$ can vary while $\text{Cons}$ is fixed. In addition, $\text{Cons}$ must consist of constraints which have a procedural interpretation via $\text{stat}()$, whilst $\text{Ens}$ can have a general form.

The following result (I: syntactic correctness) should be provable for input-preserving transformations:

$$\text{Asm0}, \text{Cons}, \text{Inv}, \Gamma_S \vdash_{\mathcal{L}_{S\cup T}} \text{Ens}$$

where $\Gamma_S$ is the source language theory. A checking transformation should be used to verify that $\text{Asm0} \land \Gamma_S$ holds for the source model.

The property (I) is an internal consistency property of the specification: it asserts that any correct implementation of $\tau$ also establishes $\text{Ens}$. 
Development of the transformation then involves the construction of a design which ensures that the relationship Cons holds between the source and target models. This may involve decomposing the transformation into phases or sub-transformations, each with their own specifications. Different phases may be implemented using different model transformation languages, appropriate for the particular task of the phase.

By reasoning using the weakest-precondition operator \([\square]\) the composition of phases in a particular design/implementation should be shown to achieve Cons (II: semantic correctness):

\[
\Gamma_S \vdash_{S \cup T} \text{Asm} \Rightarrow [\text{activity}]\text{Cons}
\]

where activity is the algorithm of the transformation design.

Inv should be true initially, and should be preserved by any step \(\delta\) of activity:

\[
\text{Asm}_0, \Gamma_S \vdash_{S \cup T} \text{Inv}
\]

and

\[
\Gamma_S \vdash_{S \cup T} \text{Inv} \Rightarrow [\delta]\text{Inv}
\]

The invariant predicate is useful in establishing correctness of transformations [36].

C  Constraint analysis

C.1  Write frames and read frames

The write frame \(\text{wr}(P)\) of a predicate \(P\) is the set of features and classes that it modifies, when interpreted as an action (an action \(\text{stat}(P)\) to establish \(P\)). This includes object creation. The read frame \(\text{rd}(P)\) is the set of classes and features read in \(P\). The read and write frames can help to distinguish different implementation strategies for conjunctive-implicative constraints. In some cases, a more precise analysis is necessary, where \(\text{wr}^*(P)\) and \(\text{rd}^*(P)\), which include the sets of objects written and written in \(P\), are used instead.

Table 10 gives the definition of some cases of these sets.

If an association end \(\text{role}_2\) has a named opposite end \(\text{role}_1\), then \(\text{role}_1\) depends on \(\text{role}_2\) and vice-versa. Deleting an instance of class \(E\) may affect any superclass of \(E\) and any association end incident with \(E\) or with any superclass of \(E\). The read frame of an operation invocation \(e\cdot\text{op}(\text{pars})\) is the read frame of \(e\) together with that of the postcondition of \(\text{op}\) (excluding formal parameters of \(\text{op}\)), and those of the actual parameters \(\text{pars}\).

The write frame \(\text{wr}(G)\) of a set \(G\) of constraints is the union of the write frames of each constraint, likewise for \(\text{rd}(G)\). The write frame \(\text{wr}(\tau)\) of a transformation \(\tau\) is \(\text{wr}(\text{Cons}_\tau)\), likewise for \(\text{rd}(\tau)\).

C.2  Definition of \(\text{stat}(P)\)

The design-level activity \(\text{stat}(P)\) associated with a specification predicate \(P\) can be defined systematically based on the structure of \(P\). Table 11 shows some of the main cases of this definition. An expression is assignable if it is (i) the result variable of an operation; (ii) a reference \(x.f\) to a writable feature \(f\) of a single object \(x\).

There are special cases for \(P_1\) implies \(P_2\) where \(P_1\) may contain variables which denote implicit additional universal quantifiers or let expression definitions. \(\text{stat}(\forall x : E \cdot P_1)\) is defined for type 2 and 3 quantified formulae as for transformations (Section 3.2).
C.3 Confluence checks for type 1 constraints

The UML-RSDS tools check confluence of type 1 constraints \( C_n \)

\[ \forall s : S_i . \text{SCond implies } \exists t : T_j . \text{TCond and Post} \]

by checking if all effects of the succedent are order-independent.

The following conditions (internal syntactic non-interference) ensure that applications of \( C_n \) on distinct \( s_1, s_2 : S_i, s_1 \neq s_2 \), cannot interfere with each other’s effects:

The only source data read by \( C_n \) should be source data navigable from \( s \). There should be no reference to any primary key of \( T_j \) in the succedent of \( C_n \), except in an assignment of the primary key value of \( s \) to that of \( t : t.tsd = s.sid \). Updates in \( T\text{Cond} \) and \( \text{Post} \) should be local to \( t \) or \( s \): only direct features of \( t \) or \( s \) should be updated. Updates \( t.f = e, e : t.f \) or \( e \subseteq t.f \) to direct features \( f \) of \( t \) are permitted, in addition \( t \) can be added to a set or sequence-valued expression \( e \) which does not depend on \( t \) or \( s : t.e \). Likewise for \( s \).

These conditions can be generalised slightly to allow 1-1 mappings of \( S_i \) identities to \( T_j \) identities.

Notice that \( S_i \) is not equal to \( T_j \) or to any ancestor of \( T_j \), and that no feature is both read and written in \( C_n \) (by the type 1 property).

Taken together, these conditions prevent one application of \( C_n \) from overwriting the effect of another application, because the sets \( wr^n \) of write frames of the two applications are disjoint, except for collection-valued shared data items (such as \( T_j \) itself), and these are written in a consistent manner (both applications add elements) by the distinct applications.

Therefore, the execution of the individual constraint applications in any order, or concurrently (provided updates to shared data are sequentialised) will achieve the required logical condition \( C_n \) once they have all completed. Semantic correctness therefore holds for internally syntactically non-interfering type 1 constraints.
<table>
<thead>
<tr>
<th>$P$</th>
<th>$\text{stat}(P)$</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = e$</td>
<td>$x := e$</td>
<td>$x$ is assignable</td>
</tr>
<tr>
<td>$\text{objs}.f = e$</td>
<td>for $x : \text{objs}$ do $x.f := e$</td>
<td>Writable feature $f$ collection $\text{objs}$</td>
</tr>
<tr>
<td>$e : x$</td>
<td>$x := x \cup {e}$</td>
<td>$x$ is assignable</td>
</tr>
<tr>
<td>$x \rightarrow \text{includes}(e)$</td>
<td>$x := x \setminus {e}$</td>
<td>$x$ is assignable</td>
</tr>
<tr>
<td>$e / : x$</td>
<td>$x := x \cup e$</td>
<td>$x$ is assignable</td>
</tr>
<tr>
<td>$x \rightarrow \text{excludes}(e)$</td>
<td>$x := x \setminus e$</td>
<td>$x$ is assignable</td>
</tr>
<tr>
<td>$e / &lt; : x$</td>
<td>$x := x$</td>
<td>$x$ is assignable</td>
</tr>
<tr>
<td>$x \rightarrow \text{excludesAll}(e)$</td>
<td>$E := E \setminus {x}$</td>
<td>Each entity $E$ containing $x$</td>
</tr>
<tr>
<td>$\text{obj}.op(e)$</td>
<td>$\text{obj}.op(e)$</td>
<td>Single object $\text{obj}$</td>
</tr>
<tr>
<td>$\text{objs}.op(e)$</td>
<td>for $x : \text{objs}$ do $x.op(e)$</td>
<td>Collection $\text{objs}$</td>
</tr>
<tr>
<td>$\exists x : E \cdot x.id = v$ and $P_1$</td>
<td>if $v \in E.id$ then $x : E := E[v]; \text{stat}(P_1)$ else $x : E := \text{new}(E); \text{stat}(x.id = v$ and $P_1)$</td>
<td>$E$ is a concrete entity, with primary key $id$ $E$ is a concrete entity, $P_1$ not of form $x.id = v$ and $P_2$</td>
</tr>
<tr>
<td>$\exists x : E \cdot P_1$</td>
<td>$x : E := \text{new}(E); \text{stat}(P_1)$</td>
<td>$E$ is a concrete entity, with primary key $id$ $E$ is a concrete entity, $P_1$ not of form $x.id = v$ and $P_2$</td>
</tr>
<tr>
<td>$\exists x : e \cdot x.id = v$ and $P_1$</td>
<td>if $E[v] \in e$ then $(x : E := E[v]; \text{stat}(P_1))$ else $(x : E := \text{new}(E); \text{stat}(P_1))$</td>
<td>Non-writable expression $e$ with element type $E$ Non-writable expression $e$ with element type $E$</td>
</tr>
<tr>
<td>$\exists x : e \cdot P_1$</td>
<td>if $e \rightarrow \text{notEmpty}()$ then $(x : E := e \rightarrow \text{any}(); \text{stat}(P_1))$</td>
<td>Any form of $X$</td>
</tr>
<tr>
<td>$\exists_1 x : X \cdot P_1$</td>
<td>if $X \rightarrow \exists(x \mid P_1)$ then skip else stat$(\exists x : X \cdot P_1)$</td>
<td>Any form of $X$</td>
</tr>
<tr>
<td>$P_1$ and $P_2$</td>
<td>$\text{stat}(P_1); \text{stat}(P_2)$</td>
<td>$P$ is of type 1</td>
</tr>
<tr>
<td>$\forall x : E \cdot P_1$</td>
<td>for $x : E$ do $\text{stat}(P_1)$</td>
<td>$P$ is of type 1</td>
</tr>
<tr>
<td>$P_1$ implies $P_2$</td>
<td>if $P_1$ then $\text{stat}(P_2)$</td>
<td></td>
</tr>
</tbody>
</table>

Table 11. Definition of $\text{stat}(P)$
For confluence, we need the further conditions that the $Cn$ applications are determinate, and that additions of $t$ or $s$ to a sequence are not permitted.

A transformation $\tau$ consisting of constraints of this form is invertible if $\tau$ is fully representative of $S$: all data features of all entities of $S$ are used to compute entities and data features of $T$, and the values of associations $f$ in $T$ are set by $\tau$ using lookup by primary keys, i.e., by definitions of the form: $t.f = TSub[e.id]$.

In addition, each $SCond$ and $Post$ should have a procedural interpretation.