Verification of Model Transformations

K. Lano, S. Kolahdouz-Rahimi, T. Clark

Dept. of Informatics, King's College London

Abstract. Model transformations are a central element of model-driven development (MDD) approaches such as the model-driven architecture (MDA). The correctness of model transformations is critical to their effective use in practical software development, since users must be able to rely upon the transformations correctly preserving the semantics of models. In this paper we define a formal semantics for model transformations, and provide techniques for proving the termination, confluence and correctness of model transformations.

1 Introduction

Model transformations are an essential part of development approaches such as Model-driven Architecture (MDA) [43] and Model-driven Development (MDD). 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.

At present, a large number of different model transformation approaches exist, such as graph transformations (eg., Viatra [45]), declarative (QVT-Relations [41]), imperative (Kermeta [20]) and hybrid (ATL [19]) languages [7]. These are all primarily based around the concept of *transformation rules*, which define one step within a transformation process. The overall effect of a transformation is then derived from the implicit (QVT-Relations, ATL) or explicit (Kermeta, 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 a higher level of structuring and specification is required, to define the complete behaviour of a transformation as a 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.

Model transformation verification remains a poorly developed field, with relatively few research publications prior to 2010 (for example, [25, 52, 26, 46, 5]) and no dedicated conference prior to 2012. Current research into model transformation verification has investigated several different formalisms for model transformation specification and verification, such as graph patterns in [16], type theory [54] and OCL/first order logic in [32]. None of these approaches has yet provided

detailed techniques and practical tools to carry out transformation verification, however.

In this paper we will describe the following components of a systematic approach for specifying and verifying model transformations, together with tool-supported techniques for specification analysis and verification:

- A formal semantics for UML and transformations (Section 2).
- A constraint-based style of model transformation specification, using specification patterns and the UML-RSDS approach to model-driven development (Sections 3, 4).
- A formal computational model for transformation execution (Section 5) and correctness conditions for transformation specifications (Section 6).
- Verification techniques for model transformation specifications, using syntactic analysis (Section 7) and the B formal method (Section 8).
- Automated design and implementation strategies for these specifications. We give proofs of correctness of these strategies (Section 9).
- Verification techniques for internal and external compositions of transformations (Section 10).

Section 11 evaluates the approach. Section 12 describes related work, and Section 13 summarises the paper.

2 Model transformation semantics

We use the four-level metamodelling framework (MOF) of UML as the context for transformations [44]. This framework consists of levels M0: models which consist of run-time 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 we refer to the M2 level as the *language level* (the models at this level define the languages which the transformation relates) and to the M1 level as the *model level* (the models which the transformation operates upon). Transformations themselves are specified by constraints at the M2 level, and they will be implemented by activities and operations at this level also.

It is often the case that a transformation which operates on a class diagram or state machine model will also need to operate on the internal constraints of the model, to re-express them in the target model in a way that correctly captures their original meaning. This is referred to as *model-level semantic preservation*.

For each model M at levels M2 and M3, we can define (i) a first-order logical language $\mathcal{L}_M$ that is the formal syntactic representation of the type structure of M, and (ii) a logical theory Γ_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 Γ_M and $\mathcal{L}_M$.

If M at level M1 is itself a UML-based model to which a semantics Γ_M can be assigned, then also Γ_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 C for each entity type C defined in M. The boolean operators *and*, *implies*, *forAll*, *exists*, *one* of OCL are semantically interpreted by the logical connectives $\wedge$, $\Rightarrow$, $\forall$, $\exists$, $\exists_1$. For each entity C of M there are logical attribute symbols $f(c : C) : Typ'$ for each data feature f of type Typ in the feature set of C, where Typ' is the semantic type corresponding to 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 $att : Typ$ of C are essentially represented as functions $att : C \rightarrow Typ'$, set-valued associations f are represented as functions $f : C \rightarrow \mathbb{F}(D)$ where D is the target entity of f, and sequence-valued associations f are represented as functions $f : C \rightarrow \text{seq}(D)$ 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). These represent the *extension* of C. The collection and primitive types of the OCL standard library [40] and the operations of these types can be represented in $\mathcal{L}_M$.

A *structure* for $\mathcal{L}_M$ is a tuple

$$(\overline{E_1}, \ldots, \overline{E_n}, f_1, ..., f_m)$$

of sets $\overline{E_i}$ interpreting the extension of each entity E_i of $\mathcal{L}_M$, and of maps $f_j : \overline{E_i} \rightarrow Typ$ representing the values of data features (attributes and associations) of these entities.

This semantic representation is used as the basis of the translation to B notation.

Tables 1 and 2 show some examples of semantic interpretations of OCL collection types and operators.

The theory Γ_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 Γ_M. Some variation points of UML semantics can be expressed by the variation of some axioms of Γ_M, for example, we assume that distinct subclasses of a class have disjoint sets of instances, and therefore include the axiom:

$$\overline{B} \cap \overline{C} = \{\}$$

in Γ_M if B and C are distinct direct subclasses of a class A. Likewise for distinct root classes.

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

$$\Gamma_M \vdash \varphi$$

OCL Term e	*Condition*	*Semantics* e'
$s{\rightarrow}size()$	set or sequence s	cardinality $\#s'$
$s{\rightarrow}size()$	bag s	sum of $s'(x)$ for $x \in \mathrm{dom}(s')$
$s{\rightarrow}includes(x)$	set s	$x' \in s'$
$s{\rightarrow}excludes(x)$	set s	$x' \notin s'$
$s{\rightarrow}includes(x)$	sequence s	$x' \in \mathrm{ran}(s')$
$s{\rightarrow}excludes(x)$	sequence s	$x' \notin \mathrm{ran}(s')$
$s{\rightarrow}includes(x)$	bag s	$x' \in \mathrm{dom}(s')$
$s{\rightarrow}excludes(x)$	bag s	$x' \notin \mathrm{dom}(s')$
$s{\rightarrow}asSet()$	s set	s'
$s{\rightarrow}asSet()$	s sequence	$\mathrm{ran}(s')$
$s{\rightarrow}asSet()$	s bag	$\mathrm{dom}(s')$
$s{\rightarrow}includesAll(t)$	sets s and t	$t' \subseteq s'$
$s{\rightarrow}includesAll(t)$	sequences s and t	$\mathrm{ran}(t') \subseteq \mathrm{ran}(s')$
$s{\rightarrow}excludesAll(t)$	sets s and t	$s' \cap t' = \{\}$
$s{\rightarrow}excludesAll(t)$	sequences s and t	$\mathrm{ran}(s') \cap \mathrm{ran}(t') = \{\}$
$s{\rightarrow}sum()$	set s	sum of elements of s'
$s{\rightarrow}sum()$	sequence s, $\#s' = n$	$s'(1) + ... + s'(n)$

Table 1. Semantic mapping for OCL collection operations

OCL Term e	*Condition*	*Semantics* e'
$objs{\rightarrow}select(P)$	set $objs$	$\{x \mid x \in objs' \wedge x.P'\}$
$objs{\rightarrow}collect(e)$	set $objs$	map c with domain $\{x.e' \mid x \in objs'\}$ and $c(y) = \#(\{x \mid x \in objs' \wedge x.e' = y\})$
$objs{\rightarrow}collect(e)$	sequence $objs$	$\{i \mapsto objs'(i).e' \mid i \in \mathrm{dom}(objs')\}$
$s{\rightarrow}subSequence(i,j)$	sequence s	$(s'{\uparrow}j'){\downarrow}(i'-1)$
$s{\rightarrow}count(x)$	sequence s	$\#(s'^{\frown}[\{x'\}])$
$s{\rightarrow}indexOf(x)$	sequence s	$min(s'^{\frown}[\{x'\}])$

Table 2. Semantic mapping for selection expressions

means the sentence is provable from the theory of M, and so holds in M. We may specify inference within a particular language L 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 Γ_L, although these may not be expressible within $\mathcal{L}_m$ itself. We use the notation $m \models \varphi$ to express satisfaction of an $\mathcal{L}_L$ sentence φ in m.

A structure m for language $\mathcal{L}_M$ is said to be a (semantic) model of M if $m \models \varphi$ for each $\varphi \in \Gamma_M$.

A pair (m, n) of models of languages $L1$ and $L2$ can also satisfy predicates φ in the union language $L1 \cup L2$. This is denoted by $(m, n) \models \varphi$.

The collection of M1 level models of an M2 model L is $Models_L$. We may simply write $m : L$ to mean $m \in Models_L$.

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

1. $h_i(x) = x' \;\Rightarrow\; f_j(x) = f_j'(x')$ for each attribute feature $f_j : Typ$ of E_i, $x \in \overline{E_i}$
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, $x \in \overline{E_i}$
3. $h_i(x) = x' \;\Rightarrow\; h_k(\llparenthesis\, f_j(x)\, \rrparenthesis) = f_j'(x')$ for each set-valued role feature f_j with element type E_k of E_i, $x \in \overline{E_i}$.
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, $x \in \overline{E_i}$.

A model transformation τ from a language S to a language T can be characterised by a domain of (pairs of) models to which it can be applied:

$$Dom_\tau \subseteq Models_S \times Models_T$$

and a relation defining which models should correspond under the transformation:

$$Rel_\tau : Models_S \leftrightarrow Models_T$$

A *computation* of τ maps a pair $(m, n) \in Dom_\tau$ to a pair $(m', n') \in Rel_\tau$.

Transformations are termed *input-preserving* if they do not modify the source model. Such transformations usually have computations of the form

$$(m, \varnothing) \longrightarrow_\tau (m, n)$$

Transformations that operate on a single model are termed *update in place* transformations, their computations have the form

$$m \longrightarrow_\tau m'$$

A model transformation τ is *invertible* if there is an inverse transformation $\tau^{\sim}$ from T to S such that τ followed by $\tau^{\sim}$ is effectively the identity relation on $Models_S$.

That is, if there are computations

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

of an input-preserving transformation τ and

$$(n, \varnothing) \longrightarrow_{\tau^{\sim}} (n, m')$$

of its inverse, then

$$m \equiv m'$$

(ie., $\tau^{\sim}$ is a right inverse of τ).

Operationally, $\tau^{\sim}$ takes a model n of T and an empty model of S, and reconstructs a minimal model m of S that would map to n (or an isomorphic copy of n) under τ.

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.

This property is directly related to the existence of an *interpretation* χ from S to T, a mapping of languages which defines a representation $\chi(\overline{E})$ in T for each entity E of S, and a representation $\chi(f)$ in T for each feature f of S [35].

3 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 semantic properties of these languages.

For a transformation τ from language S to language T, we can express its requirements in a graphical form such as the SysML requirements diagrams used in [16], or as equivalent structured text.

The formal specification of τ can be expressed in OCL or a similar first order logic by four separate predicates which characterise the global properties of τ, and which need to be considered in its specification and design [32]:

1. Asm – assumptions, expressed in the union language $\mathcal{L}_{S \cup T}$ 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 τ to be well-defined. $Asm0$ denotes the assumptions which refer only to S. Asm is the syntactic equivalent of the Dom relation.

2. *Ens* – properties, usually expressed in $\mathcal{L}_T$, which the transformation should ensure about the target model at termination of the transformation. These properties usually include the constraints of T, in order that syntactic correctness holds. For update-in-place transformations *Ens* may refer to the pre-state versions $E@pre$, $f@pre$ of model entities E and features f.
   Ens is considered separately from *Cons* because it may contain predicates which cannot be interpreted operationally, but which nonetheless should be established for the target model.
3. *Pres* – properties, usually expressed in $\mathcal{L}_S$, which the transformation should preserve from the source model to the target model, under a language-level interpretation χ. This could include language-level semantic preservation. Properties at the model level may also be specified for preservation via a model-level interpretation ζ.
4. *Cons* – constraints, expressed in $\mathcal{L}_{S \cup T}$, which define the transformation as a relationship between the elements of the source and target models, which should hold at termination of the transformation. These constraints should have an operational interpretation using the *stat*() operator (Table 9).
   If τ is input-preserving, *Cons* is also the relation between the initial state of the source model and the final state of the target model. For update-in-place transformations *Cons* can refer to the initial state of modified source model features and entities by postfixing such entity and feature names by $@pre$.
   Transformations between multiple models can distinguish entities in these models by prefixing them with the model name and \$: $m1\$E$, $m2\$E$, etc.
   Cons is the syntactic specification of the semantic relation Rel_τ.

We can express these predicates using OCL notation, this corresponds directly to a fully formal version in the axiomatic UML semantics given above, and also to a formalisation in the B notation.

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.

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

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

where Γ_S is the source language theory. A checking transformation should be used to verify that Γ_S holds for the source model.

Likewise, *Cons* should prove that *Pres* is preserved, via a suitable interpretation χ from the source language to the target language. (II) semantic preservation:

$$Asm0, Cons, Pres, \Gamma_S \vdash_{\mathcal{L}_{S \cup T}} \chi(Pres)$$

These are internal consistency properties of the specification: that any implementation which establishes *Cons* at its termination, starting from a model satisfying $Asm0$, will also establish *Ens* and preserve *Pres*. By using internal

consistency proof in B, we can establish (I) and (II) for a wide range of possible implementations, in particular for the phased implementations produced by the UML-RSDS synthesis process.

Development of the transformation then involves the construction of a design which ensures that the relationship *Cons* holds between the source and target models at termination of the transformation. 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 [ ] the composition of phases should be shown to achieve *Cons*. (III) semantic correctness:

$$\Gamma_S \vdash_{\mathcal{L}_{S \cup T}} Asm \Rightarrow [activity]Cons$$

where *activity* is the algorithm of the transformation design.

3.1 Case studies

We will consider three small running examples of transformations: (1) a version of the UML to relational database transformation [42]; (2) the computation of the non-reflexive transitive closure of an association; (3) computation of the GCD of two integer attributes of a class.

(1) maps attributes to columns, and classes to tables, with all subclasses of a given root class being merged into a single table, which has columns for each attribute of any of the subclasses, together with a primary key for the table (Figure 1).

The requirements are:

1. The objective is to produce a relational data model able to represent the data of the UML class diagram.
   (a) The source model should not be modified by the transformation.
   (b) The result is given by instances of *RDBElement*.
   (c) Each root class of the source model is represented by a table in the target model, with the same name, and with a primary key column.
   (d) Each attribute of a class in the source model is represented by a column with the same name, in the table of the root superclass of the class.

Requirements (c) and (d) are formalised as *Cons* constraints:

$$\forall\, c : Entity \cdot c.parent = \{\} \ implies \\ \exists\, t : Table;\ k : Column \cdot t.rdbname = c.name \ and \\ k.rdbname = c.name + \text{“\_}Key\text{”} \ and \ k : t.column$$

and

$$\forall\, c : Entity;\ a : c.ownedAttribute \cdot \\ \exists\, cl : Column \cdot cl.rdbname = a.name \ and \ cl : Table[c.rootClass().name].column$$

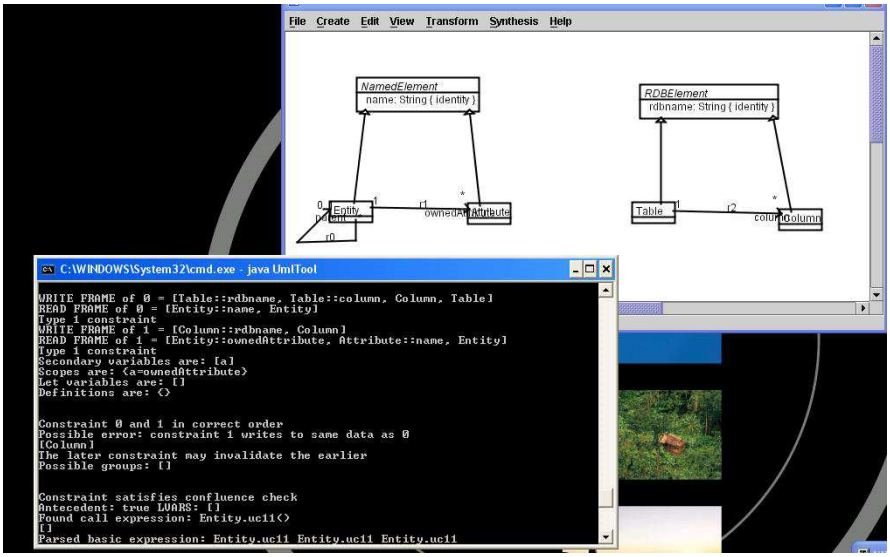

Fig. 1. UML to relational mapping transformation

The notation $E[v]$ 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 it denotes the set of E instances with primary key values in v.

The assumptions Asm are that the source model is well-formed (*name* is a primary key in the model, and association multiplicities are satisfied, also that *parent* has no cycles) and that the target model is empty:

$$RDBElement = \{\}$$

and that _ is not used in names:

$$\forall\, a : NamedElement \cdot \text{“\_”} \not/ : a.name$$

An *Ens* property is that every table has at least one column:

$$\forall\, t : Table \cdot t.column.size > 0$$

A possible *Pres* property is that all attributes have a non-empty name:

$$\forall\, a : Attribute \cdot a.name.size > 0$$

The interpretation χ in this case is only a partial interpretation, since *ownedAttribute* has no interpretation in the target language (information about which attributes belong to which classes has been lost by the transformation, ie., it is an abstraction in this respect). However, *Entity* is interpreted by *Table*, *Attribute* by *Column*, and *name* by *rdbname*, so $\chi(Pres)$ is:

$$\forall\, a : Column \cdot a.rdbname.size > 0$$

(2) is a generic transformation which computes the transitive closure of a many-many self association *parent* on an entity E, as a derived many-many association *ancestor* on E (Figure 2).

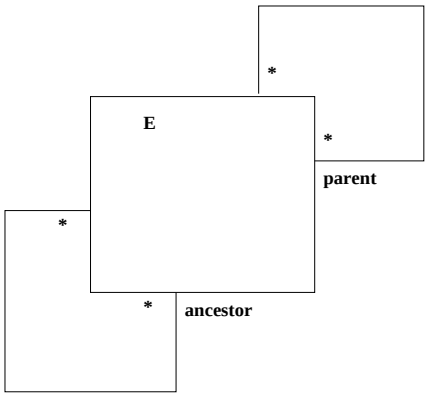

Fig. 2. Transitive closure metamodel

The requirements are:

1. The objective is to compute the non-reflexive transitive closure of an association.
   (a) The original association *parent* should not be modified by the transformation.
   (b) The result is given by a new association *ancestor* which represents $parent^+$ at termination of the transformation.

There are two *Cons* constraints:

$\forall\, s : E \cdot s.parent \subseteq s.ancestor$
$\forall\, s : E \cdot s.parent.ancestor \subseteq s.ancestor$

Asm asserts that *ancestor* is initially empty:

$\forall\, s : E \cdot s.ancestor = \{\}$

Ens asserts that *ancestor* is always a subset of the non-reflexive transitive closure:

$\forall\, s : E \cdot s.ancestor \subseteq s.parent^+$

(3) computation of the GCD of two positive integer-valued attributes of a class (Figure 3).

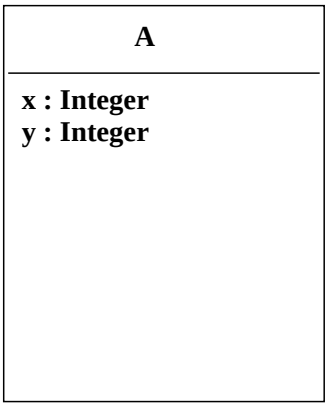

Fig. 3. GCD metamodel

The assumption Asm is:

$$\forall\, a : A \cdot a.x > 0 \;\; and \;\; a.y > 0$$

This is also the *Ens* predicate.

The *Cons* constraints are:

$$\forall\, a : A \cdot a.x < a.y \;\; implies \;\; a.y = a.y - a.x$$
$$\forall\, a : A \cdot a.y < a.x \;\; implies \;\; a.x = a.x - a.y$$

4 Model transformation specification

In this section and the following section we describe how the general development framework described above 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.
- 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 is defined as a UML use case (Chapter 16 of [39]), these have a set Asm of assumptions, which are the preconditions of the use case (ie, the *precondition* of the *BehavioralFeature* associated to the use case), and a set

Cons of postconditions. 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 Cn can be interpreted as transformation rules or statements $stat(Cn)$ which establish Cn.

The precondition constraints define checks which should be carried out on the source model, whilst the postconditions also define consistency conditions that should hold between the source and target models as a result of the transformation, and which should be maintained by a change-propagation implementation of the transformation.

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

There are two primary forms of *Cons* specification structure: (i) *conjunctive-implicative form*, consisting of an ordered series C_1, ..., C_n of constraints of the general form

$$\forall\, s : S_i \cdot SCond \ implies \ Succ$$

where the primary quantification is over a source language entity S_i. Logically *Cons* is the conjunction of the C_i.

Recursive form (ii): an ordered disjunction of clauses

$$\exists\, s : S_i \cdot SCond \ and \ Succ$$

In the first case, the ordering of constraints represents an implicit sequential ordering of the transformation implementation, and a potential decomposition of the transformation into sequenced *phases* or sub-transformations. In the second case, which is used when the constraints do not have a literal interpretation as postconditions of the transformation, but only as definitions of incremental steps in the transformation, the ordering represents a relative priority in the application of the steps.

For conjunctive-implicative specifications, the constraints define the postconditions of the entire transformation, ie, what relation it should establish, at its termination, between the source and target model data – independently of any design or implementation strategy to enforce this relation. In this respect the constraints are more abstract and declarative than QVT-Relations or TGG rules.

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.

That is, the individual constraints, and the entire use case, have both a logical and a procedural interpretation. Logically, they can be used as a platform-independent specification to reason about the correctness of the transformation. Procedurally, they can be used to generate a correct-by-construction design and executable implementation of the transformation.

The procedural interpretation of conjunctive-implicative constraint Cn is a UML activity $stat(Cn)$ (in the activity language of Figure 5) which establishes

the truth of Cn, given certain assumptions Asm_{Cn}:

$$Asm_{Cn} \;\Rightarrow\; [stat(Cn)]\,Cn$$

Asm_{Cn} includes conditions $def(Cn)$ to ensure that expression evaluations in Cn are well-defined. Table 9 defines $stat(P)$ for different kinds of predicate P.

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.

In general there may be several related use cases for a given transformation:

- A checking transformation, which checks that the source and target models satisfy Asm.
- The forward transformation τ that establishes $Cons$, assuming Asm.
- A reverse transformation $\tau^{\sim}$, derived from τ, which takes a well-formed target language model M_T and generates a source model M_S which τ 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 [37].
2. Constraints are usually more concise than transformation rules or executable versions of a transformation.
3. Constraints can be analysed at the specification level, to identify if the transformation implementation synthesised from the constraints is terminating, confluent, complete, etc.
4. Constraints facilitate verification, since they are in a form close to that used by theorem-proving and analysis tools, such as B [24] and OCL checkers such as the Dresden OCL tools [9].
5. The inverse of a transformation can be directly computed from the constraints in many cases, as can a change-propagation version of the transformation.

5 Computational model for transformations

A transformation implementation typically executes as a collection of individual *transformation steps*, which are applications of a transformation rule 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 *application conditions* of a conjunctive-implicative constraint

$$\forall\, s : S_i \cdot SCond \; implies \; Succ$$

are the positive application conditions *SCond* and the negative application conditions $not(Succ)$.

A computation of a general transformation τ with assumptions Asm and postconditions *Cons* maps a pair (m, n) of a source model m and an initial target model n, which satisfy Asm, to a pair (m', n') which satisfies *Cons*.

We denote this by

$$(m, n) \longrightarrow_\tau (m', n')$$

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

Such general transformations arise as model migrations where the source and target metamodels are not disjoint (for example, [34]). The definition can be directly generalised to cases of more than 2 involved models.

Many restructuring transformations only modify a single source model, ie., they are update in place. Their computations can be represented as:

$$m \longrightarrow_\tau m'$$

where m' is the result of a minimal set of transformation steps applied to an m which satisfies Asm, to establish *Cons* for m'.

Input-preserving transformations such as refinements, abstractions and migrations between disjoint languages usually have an initially empty target model: they map a source model m and an empty target model $\varnothing$, satisfying Asm, to a pair (m, n) which satisfies the postcondition constraints *Cons* of τ. We denote this by

$$(m, \varnothing) \longrightarrow_\tau (m, n)$$

If the steps of τ only create target elements and extend target element roles, then the minimality condition can be expressed by saying that no strict submodel of n satisfies *Cons* with respect to m.

The minimality can be logically expressed by including $Cons^\sim$ in the assumptions of obligations of (I) and (II) in Section 3. $Cons^\sim$ can also be included as an *Ens* property, for B consistency proof (Section 8).

This concept of transformation computation was referred to as *transformation state* in [8]. 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 [5]. 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), or at a finer level of granularity, an application of any relation, not including its invoked relations.

Typically, the computations of constraint-based specifications will be divided into computations of individual phases [8]: all the steps for the first phase, ie., for the first constraint, C_1, will execute first, then, once C_1 is established, the steps for the second phase will commence, and so forth.

6 Model transformation correctness

By using the above semantics and computational model for transformations, we can precisely define correctness criteria for a model transformation τ from a language S to a language T [21].

We will use the concept of a *transformation model* from [5]: the theory Γ_τ formed from the union of Γ_S and Γ_T together with constraints representing the semantics of the transformation itself. In order to carry out formal verification, this theory will be expressed in the B language as one or more B components.

The following notions of transformation correctness at the level of individual constraints or rules have been defined [52, 5]:

Applicability A constraint Ci is *forward applicable* if there is a pair (m, n) of models, where $m \in Models_S$ and n is a structure for T, such that Ci's application condition holds in (m, n).
Failure of this property would indicate an ill-defined rule that can never be applied, perhaps because of an over-restrictive application condition.

Executability A constraint Ci is *forward executable* if for every model pair (m, n) with $m \in Models_S$ and n a structure for T, such that Ci is applicable in (m, n), there is an extension n' of n such that $(m, n') \models Ci$ and $n' \in Models_T$.
Failure of this property indicates that Ci is inconsistent with Γ_T, that is, the effects specified in its postcondition contradict the constraints of T.

Completeness For individual transformation rules, completeness means that all elements and settings in the target model intended to be established by the rule are explicitly defined in the rule, unless these can be deduced from the explicit definitions of the rule.

Definedness The expressions within the constraint or rule are well-defined, ie., division by zero or other invalid expression evaluations cannot occur.

Determinacy The rule cannot produce more than one possible modified model, when applied to a particular model and specific elements in that model, so that all its required input values are fixed.

At the level of complete transformations $\tau : S \rightarrow T$ there are corresponding global properties on implementations[1] I of input-preserving transformations,

[1] Implementations are defined by activities/algorithms in terms of basic transformation steps. Eg., apply the first transitive closure constraint to all instances of E concurrently, then sequentially iterate applicable instances of the second constraint. It is assumed that only finitely many applications can occur in a finite time interval.

using $(m, n) \longrightarrow_{\tau}^{I} (m', n')$ to denote that there is a completed computation of the implementation I of τ from (m, n) to (m', n'):

Syntactic correctness For each model which conforms to (is a model of the language) S, and to which the transformation implementation can be applied, the transformed model conforms to T:

$$\forall\, m : S;\ n \cdot m \models Asm0 \wedge (m, \varnothing) \longrightarrow_{\tau}^{I} (m, n)\ \Rightarrow\ n : T$$

If *Ens* includes Γ_T, then this is ensured by property (I) of Section 3.

Semantic correctness Each target model that can be produced by the transformation implementation satisfies the required transformation postcondition *Cons*:

$$\forall\, m : S;\ n \cdot m \models Asm0 \wedge (m, \varnothing) \longrightarrow_{\tau}^{I} (m, n)\ \Rightarrow\ (m, n) \models Cons$$

For input-preserving transformations where *Asm* requires an initially empty target model, this property is established by property (III) of Section 3.

Semantic preservation At the language level this means that for each property of the source model which should be preserved (correctness properties), the target model satisfies the property, under a fixed interpretation χ of the source language into the target language.
There may also be model-level properties which should be preserved via a model-level interpretation ζ.

1. (Language-level semantic preservation): each language-level property $\varphi : \mathcal{L}_S$ satisfied by a source model m is also satisfied, under an interpretation χ on language-level expressions, in the target model n:

   $$m \models (Asm0\ and\ \varphi)\ \wedge$$
   $$(m, \varnothing) \longrightarrow_{\tau}^{I} (m, n)\ \Rightarrow\ n \models \chi(\varphi)$$

   This shows that T is at least as expressive as S, and that users of n can view it as a model of S, because the entities and features of S can be expressed in terms of those of T. In particular, no information about m is lost by τ.
   For φ as *Pres*, this is established by property (II) of Section 3.
   χ can be expressed in B by means of B machine invariants and verified by internal consistency proof (Section 8).
2. (Model-level semantic preservation): each model-level property φ in $\mathcal{L}_m$ of a source model m is also true, under an interpretation ζ on model-level expressions, in a target model n:

   $$m \models Asm0\ \wedge\ \Gamma_m \vdash \varphi\ \wedge$$
   $$(m, \varnothing) \longrightarrow_{\tau}^{I} (m, n)\ \Rightarrow\ \Gamma_n \vdash \zeta(\varphi)$$

   This means that internal constraints of m remain valid in interpreted form in n, for example, that subclassing in a UML model is mapped to subsetting of sets of table primary key values, in a transformation from UML to relational databases [42].

Model-level semantic preservation means that the diagram of Figure 4 commutes: each formula $\varphi \in \Gamma_m$ has

$$\Gamma_n \vdash \zeta(\varphi)$$

where Γ_m is the semantics of m under $Sem1$, and Γ_n is the semantics of n under $Sem2$.

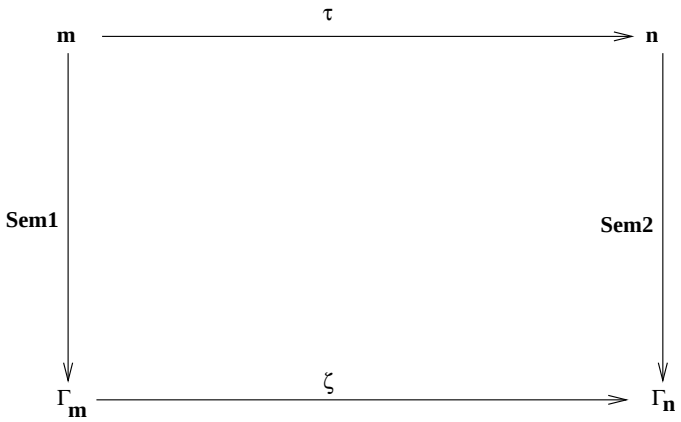

Fig. 4. Model-level semantic preservation

Notice that if τ and σ are semantically preserving, so is their composition $\tau;\ \sigma$, if the composition of the interpretations is used.

More generally, we have the properties:

Uniqueness The transformation implementation should produce a unique result (up to isomorphism) from a given starting model. This is also termed *confluence*: for each pair (m, n) of starting models satisfying Asm, if $(m, n) \longrightarrow^I_\tau (m', n')$ and $(m, n) \longrightarrow^I_\tau (m'', n'')$ then $m' \equiv m''$ and $n' \equiv n''$. In other words, the transformation implementation always produces essentially equivalent target models from the same source model.

Implementations of transformations τ should normally satisfy the stronger condition of being *isomorphism-preserving*:

$$(m, n) \longrightarrow^I_\tau (p, q)$$

and

$$(m', n') \longrightarrow^I_\tau (p', q')$$

where $m \equiv m'$ and $n \equiv n'$, should imply that $p \equiv p'$ and $q \equiv q'$.

Termination An implementation of transformation τ is *terminating* if every set of computation steps starting from (m, n) satisfying Asm, permitted by the implementation, is finite.

Model-level semantic preservation has not been fully addressed by any support tool for model transformations, despite its importance for ensuring the integrity of a development using transformations. The paper [52] defines a model-checking technique for testing that a transformation preserves selected properties, on a model-per-model basis, but does not provide a means to verify transformations on a global basis, for arbitrary models of the source language.

Some transformations may be inherently non-functional, for example, the transformation (translation) between the abstract syntax trees representing texts in different natural languages. However, as [5] points out, non-determinism can also result from specification errors such as under-specification and omission of intended predicates.

7 Verification using syntactic analysis

We can perform some analysis of constraints and use cases using data-dependency analysis. This analysis is used to (i) identify possible flaws in the specification to the developer, and (ii) to determine the choice of design and implementation of the constraints and use case.

For each predicate P we define the *write frame* $wr(P)$ of P, and the *read frame* $rd(P)$ (Table 3). These are the sets of entities and features which $stat(P)$ may update or access, respectively.

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

If an association end *role*2 has a named opposite end *role*1, then *role*1 depends on *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 $e.op(pars)$ is the read frame of e and *pars* together with that of the postcondition $Post_{op}$ of op, excluding the formal parameters v of op. Its write frame is that of $Post_{op}$, excluding v. $wr(G)$ of a set G of constraints is the union of the constraint write frames, likewise for $rd(G)$.

A dependency ordering $Cn < Cm$ is defined between distinct *Cons* constraints by

$$wr(Cn) \cap rd(Cm) \neq \{\}$$

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

P	$rd(P)$	$wr(P)$	$rd^*(P)$	$wr^*(P)$
Basic expression e without quantifiers, logical operators or $=, \in, \subseteq, E[]$	Set of features and entities used in P + dependents	$\{\}$	Set of objects $\times$ features and entities referred to in P + dependents	$\{\}$
$e1 \in e2.r$ r multiple-valued	$rd(e1) \cup rd(e2)$	$\{r\}$	$rd^*(e1) \cup rd^*(e2)$	$rd^*(e2) \times \{r\}$
$e1.f = e2$	$rd(e1) \cup rd(e2)$	$\{f\}$	$rd^*(e1) \cup rd^*(e2)$	$rd^*(e1) \times \{f\}$
$e1 \subseteq e2.r$ r multiple-valued	$rd(e1) \cup rd(e2)$	$\{r\}$	$rd^*(e1) \cup rd^*(e2)$	$rd^*(e2) \times \{r\}$
$E[e1]$	$rd(e1) \cup \{E\}$	$\{\}$	$rd^*(e1) \cup \{E\}$	$\{\}$
$\exists\, x : E \cdot Q$ (in succedent)	$rd(Q)$	$wr(Q) \cup \{E\}$	$rd^*(Q)$	$wr^*(Q) \cup \{E\}$
$\forall\, x : E \cdot Q$ (at outer level)	$rd(Q) \cup \{E\}$	$wr(Q)$	$rd^*(Q) \cup \{E\}$	$wr^*(Q)$
C *implies* Q	$rd(C) \cup rd(Q)$	$wr(Q)$	$rd^*(C) \cup rd^*(Q)$	$wr^*(Q)$
Q *and* R	$rd(Q) \cup rd(R)$	$wr(Q) \cup wr(R)$	$rd^*(Q) \cup rd^*(R)$	$wr^*(Q) \cup wr^*(R)$

Table 3. Definition of read and write frames

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

Together, these conditions ensure that the activities $stat(C_j)$ of subsequent constraints C_j cannot invalidate earlier constraints C_i, for $i < j$.

A use case satisfies *semantic non-interference* if for $i < j$:

$$C_i \Rightarrow [stat(C_j)]C_i$$

Syntactic non-interference implies semantic non-interference, but not conversely.

Constraints C_i may also satisfy the condition

$$wr(C_i) \cap rd(C_i) = \{\}$$

We refer to such constraints as *type 1* constraints. Subject to further restrictions, they have an implementation as bounded iterations over the source model entity S_i of their outermost $\forall$ quantifier.

The general form of a conjunctive-implicative constraint is an implication:

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

where *SCond* 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_j is some entity of T, *TCond* 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 *LPost and GPost* to specify these two aspects

(local versus global relationship between s and t). *TCond* 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 of clauses. Additional $\forall$-quantifiers may be used at the outer level of the constraint, if quantification over multiple source model elements is necessary, instead of over single elements. Each source entity type S_i which is $\forall$-quantified over at the outer level is referred to as a *source domain* of the constraint. The T_j are *target domains* of the constraint.

We can classify transformation constraints into several types, of increasing complexity:

- Type 0 constraints: no quantification over source elements, instead only updates to specific objects are specified. For example:

  $$Account[\text{“33665”}].balance = 0$$

  to set the balance of a specific identified account.
- Type 1 constraints with 1-1 mapping of identities (structure preserving constraints): these have disjoint *wr* and *rd* frames. For example, the first constraint of the transitive closure computation:

  $$\forall\, s : E \cdot s.parent \subseteq s.ancestor$$

- Type 1 constraints with merging of source entity instances into target instances, ie, with a many-1 mapping of identities.
- Type 2 constraints:

  $$wr(Cn) \cap (rd(Post) \cup rd(TCond))$$

  is non-empty, but

  $$wr(Cn) \cap (rd(SCond) \cup \{S_i\}) = \{\}$$

  These constraints usually need to be implemented by a fixpoint iteration: the basic transformation step $stat(Succ)$ is iterated over applicable source elements until no applicable source element remains.
  An example is the second constraint in the transitive closure computation:

  $$\forall\, s : E \cdot s.parent.ancestor \subseteq s.ancestor$$

- Type 3 constraints: these have

  $$wr(Cn) \cap (rd(SCond) \cup \{S_i\}) \neq \{\}$$

  These constraints need to be implemented by a fixpoint iteration: each transformation step may modify the sets of applicable source objects for subsequent steps.

– Recursive form constraints: constraints that are not intended to hold literally in the post-state of the transformation, but which describe instead incremental transformation steps. Type 2 or type 3 in terms of data-dependencies, they typically contain a succedent which is false when interpreted as a logical assertion, as in the GCD computation:

$$\forall\, a : A \cdot a.x < a.y \ \mathit{implies}\ a.y = a.y - a.x$$

Alternatively, the succedent may contradict the antecedent, for example:

$$\forall\, e : E \cdot \mathit{SCond}\ \mathit{implies}\ e{\rightarrow}\mathit{isDeleted}()$$

which removes from the model all instances of E that satisfy SCond.

Some forms of analysis apply regardless of the constraint type. Each constraint should use correct syntax and should be type-correct in the context of the source and target metamodels. *Internal completeness* checks that for each instance t of an entity E that is created in the constraint, all defined data features of E are set for t. Syntactic checks can be applied to ensure that constraints are determinate and well-defined (they contain no division by zero, or reference to other undefined expressions, and the succedent of constraints cannot use indeterminate choice: *any*, *or*). For each constraint, a definedness condition is produced, this condition is a necessary assumption which should hold before the constraint is applied, in order that its evaluation is well-defined. Likewise with determinacy.

Examples of the clauses for definedness are given in Table 4.

Examples of the clauses for determinacy are given in Table 5.

7.1 Analysis of type 1 constraints

For type 1 constraints, many of the verification properties (such as confluence, semantic correctness and termination) can be established by syntactic checks on the constraint to ensure that distinct applications of the constraint cannot semantically interfere.

Given a type 1 constraint Cn:

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

the following conditions (*internal syntactic non-interference*) ensure that applications of Cn on distinct $s1, s2 : S_i$, $s1 \neq s2$, cannot interfere with each other's effects:

> Cn should only read source data navigable from s. There should be no reference to any primary key of T_j in the succedent of Cn, except in an assignment to it of the primary key value of s: $t.tid = s.sid$. Updates in TCond and 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

Constraint expression E	*Definedness condition def(E)*
a/b	$b \neq 0$
$s[ind]$ sequence, string s	$ind > 0$ *and* $ind \leq s.size$
$E[v]$ entity E with primary key id, v single-valued	$v \in E.id$
$s \rightarrow last()$ $s \rightarrow first()$ $s \rightarrow max()$ $s \rightarrow min()$ $s \rightarrow any()$	$s.size > 0$
$v.sqrt$	$v \geq 0$
$v.log$	$v > 0$
A *and* B	$def(A)$ *and* $def(B)$
A *implies* B	$def(A)$ *and* $def(B)$
$\exists\, x \cdot A$	$\forall\, x \cdot def(A)$
$\forall\, x \cdot A$	$\forall\, x \cdot def(A)$

Table 4. Definedness conditions for constraints

Constraint expression E	*Determinacy condition det(E)*
$s \rightarrow any()$	$s.size = 1$
P *or* Q	*false*
Case-conjunction ($E1$ *implies* $P1$) *and* ... (En *implies* Pn)	Conjunction of $not(Ei$ *and* $Ej)$ for $i \neq j$, and each $det(Pi)$
A *and* B	$det(A)$ *and* $det(B)$
A *implies* B	$det(A)$ *and* $det(B)$
$\exists\, x \cdot A$	$\forall\, x \cdot det(A)$
$\forall\, x \cdot A$	$\forall\, x \cdot det(A)$

Table 5. Determinacy conditions for constraints

f of t are permitted, in addition t can be added to a set or sequence-valued expression e which does not depend on s or t: $t : e$. 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 Cn (by the type 1 property).

Taken together, these conditions prevent one application of Cn from overwriting the effect of another application, because the sets wr^* 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.

The standard implementation of type 1 constraints is a fixed for-loop iteration over their source domains (Section 9). The execution of the individual constraint applications in any sequential order by this implementation will achieve the required logical condition Cn once all applications have completed. Semantic correctness and termination therefore hold for the standard implementation of internally syntactically non-interfering type 1 constraints.

For confluence of this implementation, we need the further conditions that the Cn are determinate, and that additions of t or s to a sequence are not permitted.

Theorem 1 If a type 1 constraint Cn is syntactically restricted as described above, then its standard implementation is confluent.

Proof By determinacy, each individual application act of Cn has a unique (up to isomorphism) result from a specific starting state.

Two applications $act1$ and $act2$ of Cn for distinct $s1$, $s2$ in S_i have disjoint wr^* frames, except for collection-valued shared data items (such as T_j itself), because these are based on distinct T_j objects $t1$ and $t2$ or on the distinct $s1$ and $s2$. Hence the effects of $act1$ and $act2$ are independent on these write frames. If a set-valued expression e is written in $Post$ by a formula $t : e$ or $e \rightarrow includes(t)$, then e is not read in Cn, so its value cannot affect applications of Cn. The order of addition of $t1$ and $t2$ to e does not make any difference to its resulting value, so such updates are order independent. Likewise with additions of s to a set. □

Counter-examples to confluence when the conditions do not hold can easily be constructed. If elements of S_i are simply added to a global sequence:

$$\forall\, s : S_i \cdot s : Root.instance.slist$$

for a singleton class $Root$, with an ordered association end $slist : \mathrm{seq}(S_i)$, then two different executions of the transformation could produce two different orderings of $slist$.

Likewise, if the mapping of identities is not 1-1, then the same T_j instance could be updated by two different source objects, with only the second update

being retained:

$$\forall\, s : S_1 \cdot \exists\, t : T_1 \cdot t.id = s.id/2 \ \mathit{and}\ t.y = s.x$$

where all attributes are integer-valued. This is also a counter-example to semantic correctness of the standard implementation.

Update of objects other than t and s can violate confluence and semantic correctness in a similar way, for example:

$$\forall\, s : S_1 \cdot \exists\, t : T_1 \cdot t.r = T_0[\text{“1”}] \ \mathit{and}\ t.r.att = s.x$$

Analysis of syntactic correctness and semantic preservation for type 1 constraints can be achieved by internal consistency proof in B (Section 8).

7.2 Analysis of type 1 entity and instance merging constraints

Instance and entity merging occurs when two or more source objects, possibly of two or more source language entities, are merged into a single object of a target language entity.

The constraints are written in conjunctive-implicative form, but the same target entity T_j may appear on the right-hand side of two or more constraints. In the first constraint to be applied, instances of T_j are created, in subsequent constraint applications that identify existing T_j instances, these are looked-up and their data supplemented by the effect of the applications.

An example are the constraints for the UML to relational database transformation, in which the same table object may be written by different constraint applications.

Internal syntactic non-interference needs to be considered carefully for such transformations. In addition to the restrictions on type 1 constraints defined above, we add the following:

> Updates to the same target entity instance $t : T_j$ in different constraint applications must: (i) not change the primary key value of t after its initialisation; (ii) modify disjoint sets of attributes and single-valued associations of t; (iii) modify collection-valued associations r of t in a consistent way, if r is written by different constraint applications: these updates must all be additions; (iv) mixtures of additions and removals of t to collections e are forbidden. Only additions are permitted.

For confluence, additions to sequence-valued roles r in case (iii) or sequences e in case (iv) are forbidden. Individual applications of the constraint must be determinate.

Semantic correctness of the standard implementation follows from the disjointness of wr^* frames for distinct constraint applications: if the objects being updated are the same, then the features are different, except in the case of non-interfering updates to collection-valued roles.

Confluence follows since the updates to a specific t by multiple constraint applications are order-independent by the above restrictions.

The UML to relational database example satisfies these conditions, since $t.column$ is added to by each distinct constraint application that affects $t : Table$.

A simple counter-example for semantic correctness and confluence in this case is the constraint:

$$\forall\, s : S_1 \cdot \exists\, t : T_1 \cdot t.id = s.id/2 \ and \\ (s.x > 0 \ implies \ t : e) \ and \ (s.x \leq 0 \ implies \ t \ / : e)$$

where all attributes are integer-valued, and e is a set-valued collection independent of s and t. Depending on the order in which S_1 elements are processed, a specific t may or may not be in e at termination.

Type 1 constraints that fail the internal non-interference restrictions can be analysed using the techniques for type 2 and 3 constraints in the following sections.

7.3 Analysis of type 2 and type 3 constraints

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

$$wr(Cn) \cap (rd(Post) \cup rd(TCond))$$

is non-empty, but

$$wr(Cn) \cap (rd(SCond) \cup \{S_i\}) = \{\}$$

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 least-fixed point computation may be necessary instead, with iterations repeated until no further change takes place.

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

A measure $Q : Models_S \times Models_T \to \mathbb{N}$ on the source and target model data is used to establish the termination, confluence and correctness 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:

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

and

$$Q(smodel, tmodel) = 0 \ \equiv \ \{s \in \overline{S_i} \mid SCond \wedge \neg \, (Succ)\} = \{\}$$

Succ abbreviates the constraint rhs $\exists \, t : T_j \cdot TCond \ and \ Post$.

Q will be syntactically defined as an expression in the union language $S \cup T$.

The variant function property of Q establishes termination of the fixpoint implementation of Cn: each application of Cn strictly reduces Q, and $Q \geq 0$, so there can only be finitely many such applications. Semantic correctness also follows, since when $Q = 0$, there are no remaining instances of S_i which violate the constraint, ie, Cn holds true.

Confluence requires that the $Q = 0$ state is unique:

Theorem 2 If for each particular starting state of the source and target models there is a unique (up to isomorphism) possible terminal state of the models (produced by applying the constraint until it cannot be applied further) in which $Q = 0$, then the fixpoint implementation of the type 2 or 3 constraint is confluent.

Proof The terminal states of the transformation are characterised by the condition

$$\{s \in \overline{S_i} \mid SCond \wedge \neg \, (Succ)\} = \{\}$$

But in such states we also have

$$Q(smodel, tmodel) = 0$$

Therefore, there is a unique termination state. □

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

$$\Sigma_{t:T_i} \#(T_j - t.r)$$

since the sets T_i 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 $x.r \rightarrow excludesAll(e)$ and equivalents), we have an upper bound:

$$\Sigma_{t:T_i} \# t.r$$

For example, a Q measure for the second transitive closure transformation constraint is:

$$\Sigma_{s:E} \#(s.parent^+ - s.ancestor)$$

where $parent^+$ is the non-reflexive transitive closure of *parent*.

Verification of the variant function and unique 0 state properties of Q require refinement proof in B, syntactic correctness and semantic preservation also require proof in B (Section 8).

7.4 Recursive form constraints

These constraints are interpreted as defining clauses of a recursive function

$$\tau : Models_S \times Models_T \rightarrow Models_S \times Models_T$$

with $\tau(m, n) = \tau(m', n')$ where (m', n') are m and n modified by the updates specified by the constraint.

For example, for the GCD computation, the constraints correspond to the following definition of τ:

$$\begin{array}{l} \exists\, ax \in \overline{A} \cdot x(ax) < y(ax) \wedge \tau(\overline{A}, x, y) = \tau(\overline{A}, x, y \oplus \{ax \mapsto y(ax) - x(ax)\}) \ \vee \\ \exists\, ax \in \overline{A} \cdot y(ax) < x(ax) \wedge \tau(\overline{A}, x, y) = \tau(\overline{A}, x \oplus \{ax \mapsto x(ax) - y(ax)\}, y) \ \vee \\ \forall\, ax \in \overline{A} \cdot x(ax) = y(ax) \wedge \tau(\overline{A}, x, y) = (\overline{A}, x, y) \end{array}$$

The definition of τ therefore is a recursive form transformation, the poststate of the transformation is given by $\tau(m, n)$ where m and n are the initial models.

The treatment of these constraints is similar to type 2 and 3 constraints, with correctness, termination and confluence established by means of suitable Q variant functions.

7.5 Summary

Table 6 summarises the different types of transformation constaint, based on the internal data-dependencies of the constraint, and how different properties of the constraints can be established.

The UML-RSDS tools perform the following analyses on transformation specifications:

- calculation of $rd(Cn)$ and $wr(Cn)$ for each constraint, and identification of the constraint type (0, 1, 2 or 3 as defined above).
- 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 termination, semantic correctness and confluence of type 1 constraints.
- automatic derivation of inverse constraints for invertible type 1 constraints.
- checks that constraints are complete (when an object of entity E is created, all its features are set, and that the disjunction of *SCond* conditions for each given source entity S_i is *true*).
- checks that constraints are determinate and well-defined (no division by zero, or reference to other undefined expressions, the succedents of constraints cannot use indeterminate choice: *any*, *or*).

	Constraint properties	*Termination*	*Confluence*	*Syntactic correctness*
Type 0 constraint	No outer $\forall$ quantifier: single application.	Syntactic check $def(Cn)$	Always true	Rules for []
Type 1 constraint	No interference between different applications of constraint, and no change to S_i or $rd(SCond)$: $wr(Cn) \cap rd(Cn) = \{\}$.	Syntactic checks	Syntactic checks	Rules for [] Internal consistency proof in B
Type 2 constraint	Interference between different applications of constraint, but no update of S_i or $rd(SCond)$ within constraint: $S_i \notin wr(Cn)$, $wr(Cn) \cap rd(SCond) = \{\}$	Q measure: variant function	Q measure: uniqueness of $Q = 0$ state	Internal consistency proof in B
Type 3 constraint	Update of S_i or $rd(SCond)$ within constraint. $S_i \in wr(Cn)$, or $wr(Cn) \cap rd(SCond) \neq \{\}$	Q measure: variant function	Q measure: uniqueness of $Q = 0$ state	Internal consistency proof in B

Table 6. Types of transformation constraints and verification techniques

8 Verification using B AMN

B abstract machine notation is an established formal method, with a large number of users and with successful large-scale application to safety-critical systems. Its mathematical foundation is first-order logic and set theory, which makes it a natural formalism for the treatment of UML and OCL, which have the same semantic basis [27, 49].

A B specification consists of a linked collection of modules, termed *machines*. Each machine encapsulates data and operations on that data. Entities and metamodels can naturally be represented in machines, using the set-theoretic semantics of [27].

The general form of a B machine M_τ representing the transformational model of a transformation τ with source language S and target language T is:

```
MACHINE Mt SEES SystemTypes
VARIABLES
  /* variables for each entity and feature of S */
  /* variables for each entity and feature of T */
INVARIANT
  /* type definitions for each entity and feature of S and T */
  Asm0 & Pres & Ens & Pres'
INITIALISATION
  /* var := {} for each variable */
OPERATIONS
  /* creation operations for entities of S, restricted by Asm, Pres */
```

```
  /* update operations for features of S, restricted by Asm, Pres */
  /* operations representing transformation steps */
END
```

where $Pres'$ is $\chi(Pres)$, $Asm0$ is the sub-part of Asm which concerns only S. If $Pres$ and $Asm0$ are conjunctions of universally quantified formulae $\forall\, s : S_i \cdot \psi$, then the appropriate instantiated formulae $\psi[sx/s, vx/v]$ are used as the preconditions of operations creating $sx : S_i$ (or subclasses of S_i) or modifying its features. All these operations will include preconditions that the target model is empty (for transformations which start with an empty target model).

$Pres$ and $Pres'$ are only included in M_τ when proof of correctness property (II) of Section 3 is being carried out.

As an example, the entity E and its associations in the transitive closure computation transformation can be defined by the following partial machine:

```
MACHINE E SEES SystemTypes
VARIABLES es, parent, ancestor
INVARIANT
  es <: E_OBJ &
  parent : es --> FIN(es) &
  ancestor : es --> FIN(es)
INITIALISATION
  es, parent, ancestor := {}, {}, {}
END
```

Using these machines we can verify syntactic correctness and semantic preservation properties of a model transformation, by means of *internal consistency* proof of a B machine representing the transformation and its metamodels. Internal consistency of a B machine consists of the following logical conditions:

- That the state space of the machine is non-empty: $\exists\, v.Inv$ where v is the tuple of variables of the machine, and Inv its invariant.
- That the initialisation establishes the invariant:

  $$[Init]Inv$$

- That each operation maintains the invariant:

  $$Pre \wedge Inv \;\Rightarrow\; [Code]Inv$$

  where Pre is the precondition of the operation, and $Code$ its effect.

Therefore, if the Ens and $\chi(Pres)$ constraints of a transformation τ are included in the invariant of the machine M_τ which has data representations of the source and target metamodels, operations to add elements to the source model, and operations for each transformation step of τ, internal consistency proof of M_τ proves that Ens and $\chi(Pres)$ are valid for any (partial or complete) target model constructed by the UML-RSDS (phased and sequential) implementation of the transformation, relative to the source model. Occurrences $E@pre$

or $f@pre$ in *Ens* are interpreted by additional variables es_pre, f_pre which have the same typing as es and f. They are modified in parallel with es and f by the source model creation and modification operations, and are unchanged by transformation steps.

M_τ expresses the transformation model theory Γ_τ in B notation.

The correctness properties (I) and (II) of Section 3 are established for the UML-RSDS implementations of transformations by internal consistency proof in B, because M_τ expresses $\mathcal{L}_{S \cup T}$ in its data, and Γ_S, $Asm0$ in its invariants. The phased implementation of τ is expressed by the operation definitions of transformation steps: each step is for a specific constraint C_j of *Cons*, and its preconditions include the assumptions that all preceding constraints C_i, $i < j$ are true.

Internal consistency proof establishes that the *Ens* property holds for each individual target instance of a particular target entity as it is created, so when all applicable steps have been applied, *Cons* holds and so does *Ens* for all target instances. Likewise for $\chi(Pres)$.

For the transitive closure computation, the resulting complete B machine is:

```
MACHINE E SEES SystemTypes
VARIABLES es, parent, ancestor
INVARIANT
  es <: E_OBJ &
  parent : es --> FIN(es) &
  ancestor : es --> FIN(es) &
  /* Ens: */ !ex.(ex : es  =>  ancestor(ex) <: (closure1(rel(parent)))[{ex}])
INITIALISATION
  es, parent, ancestor := {}, {}, {}
OPERATIONS
  create_E() =
    PRE es /= E_OBJ & !ex.(ex : es => ancestor(ex) = {})
    THEN
        ANY ex WHERE ex : E_OBJ - es
        THEN
            es := es \/ { ex } ||
            parent(ex) := {} ||
            ancestor(ex) := {}
        END
    END;

  addparent(ex,parentx) =
    PRE ex : es & parentx : es & !ex.(ex : es => ancestor(ex) = {})
    THEN
      parent(ex) := parent(ex) \/ { parentx }
    END;

  uc11(ex) =
    PRE ex : es
    THEN
      ancestor(ex) := ancestor(ex) \/ parent(ex)
```

```
    END;

  uc12(ex) =
    PRE ex : es & not(union(ancestor[parent(ex)]) <: ancestor(ex)) &
      !ex.( ex : es  =>  parent(ex) <: ancestor(ex) )
    THEN
      ancestor(ex) := ancestor(ex) \/ union(ancestor[parent(ex)])
    END
END
```

The final invariant expresses the *Ens* property of the transformation, $rel(parent)$ is the relation $es \leftrightarrow es$ that corresponds to *parent*. $uc11$ defines the transformation step of the first constraint, $uc12$ defines the transformation step of the second constraint (and assumes that the first constraint has been established). The machine is generated automatically by the UML-RSDS tools from the UML specification of the transformation.

Internal consistency proof then establishes that *Ens* is a provable invariant, the key deduction is that

```
  !ex.(ex : es  =>  ancestor(ex) <: (closure1(rel(parent)))[{ex}]) &
  e : es  =>
    union(ancestor[parent(e)])  <:  (closure1(rel(parent)))[{e}]
```

Using Atelier B version 4.0, 17 proof obligations for internal consistency of the above machine are generated, of which 11 are automatically proved, and the remainder can be interactively proved using the provided proof assistant tool.

Similarly, if $\chi(Pres)$ has a universally quantified form, with these quantifications over target model entities, then internal consistency proof will include the statement that $\chi(Pres)$ holds for each target element produced by applying the transformation constraints.

For the UML to relational database example, the B machine has the form:

```
MACHINE UMLTORDB SEES SystemTypes, String_TYPE
VARIABLES entitys, namedelements, attributes,
  name, ownedAttributes, parent,
  rdbelements, tables, columns, rdbname, column
INVARIANT
  namedelements <: NamedElement_OBJ &
  attributes <: namedelements & entitys <: namedelements &
  attributes /\ entitys = {} &
  name : namedelements >-> STRING &
  ownedAttributes : entitys --> FIN(attributes) &
  parent : entitys --> FIN(entitys) &
  !entityx.(entityx : entities => card(parent(entityx)) <= 1) &
  rdbelements < RDBElement_OBJ &
  tables <: rdbelements & columns <: rdbelements &
  tables /\ columns = {} &
  rdbname : rdbelements >-> STRING &
  column : tables --> FIN(columns) &
```

```
  /* Asm0: */ !a.(a : namedelements  =>  "_" /: ran(name(a))) &
  /* Ens: */ !t.(t : tables  =>  card(column(t)) > 0) &
  /* Pres: */ !a.(a : attributes  =>  card(name(a)) > 0) &
  /* Pres': */ !a.(a : columns  =>  card(rdbname(a)) > 0)
INITIALISATION
  entitys := {} || ... || column := {}
OPERATIONS
  create_Entity(namex) =
    PRE namex : STRING & namedelements /= NamedElement_OBJ &
        card(namex) > 0 & "_" /: ran(namex) & rdbelements = {}
    THEN
        ANY entityx WHERE entityx : NamedElement_OBJ - namedelements
        THEN
            entitys := entitys \/ { entityx } ||
            namedelements := namedelements \/ { entityx } ||
            name(entityx) := namex ||
            ownedAttributes(entityx) := {} ||
            parent(entityx) := {}
        END
    END;

  addparent(entityx,parentx) =
    PRE entityx : entitys & parentx : entitys &
      card(parent(entityx) \/ { parentx }) <= 1 & rdbelements = {}
    THEN
      parent(entityx) := parent(entityx) \/ { parentx }
    END;

  /* Other creation and update operations for source model */

  uc11(cx) =
    PRE cx : entitys & parent(cx) = {}
    THEN
      ANY tt, kk WHERE tt : RDBElement_OBJ - rdbelements &
        kk : RDBElement_OBJ - rdbelements & kk /= tt
      THEN
        tables := tables \/ { tt } ||
        rdbname(tt) := name(cx) ||
        columns := columns \/ { kk } ||
        rdbelements := rdbelements \/ { tt, kk } ||
        rdbname(kk) := name(cx) ^ "_Key" ||
        column(tt) := column(tt) \/ { kk }
      END
    END;

  uc12(cx,ax) =
    PRE cx : entitys & ax : ownedAttribute(cx) &
      !cc.(cc : entitys & parent(cc) = {} =>
        #tt.(tt : tables & rdbname(tt) = name(cc) & ... ) )
    THEN
```

```
      ANY cl WHERE cl : RDBElement_OBJ - rdbelements
      THEN
        columns := columns \/ { cl } ||
        rdbelements := rdbelements \/ { cl } ||
        rdbname(cl) := name(ax) ||
        column(rdbname~(name(rootClass(cx)))) :=
            column(rdbname~(name(rootClass(cx)))) \/ { cl }
    END
END
```

The preconditions of the step operation $uc12$ for the second constraint include the logical assertion that the first constraint already holds. *Ens* follows from the definition of $uc11$, whilst $\chi(Pres)$ follows from *Pres* and the definitions of $uc11$ and $uc12$.

In order to prove that a postulated Q measure is actually a variant function, refinement proof in B can be carried out, with an abstraction of the transformation machine M_τ defined as $M0_\tau$:

```
MACHINE M0t SEES SystemTypes
VARIABLES /* variables for source model entities */, q
INVARIANT
  /* typing of entity sets */ &
  q : NAT
INITIALISATION
  es, q := {}, 0
OPERATIONS
  /* creation and update operations for source
      model: these may set q arbitrarily in NAT */

  step() =
    PRE q > 0
    THEN
      q :: 0..q-1
    END
END
```

step represents a transformation step of the constraint for which q is the postulated variant. In the usual case of input-preserving transformations, this step does not modify any of the source model entities. Each constraint C_i may have a corresponding variant q_i, the operation $step_i$ for transformation steps of C_i then has the form:

```
  stepi() =
    PRE qi > 0 & qk = 0 /* for k < i */
    THEN
      qi :: 0..qi-1 || qj :: NAT /* for j > i */
    END
```

For the transitive closure computation, the machine is:

```
MACHINE E0 SEES SystemTypes
```

```
VARIABLES es, q1, q2
INVARIANT
  es <: E_OBJ &
  q1 : NAT & q2 : NAT
INITIALISATION
  es, q1, q2 := {}, 0, 0
OPERATIONS
  create_E() =
    PRE es /= E_OBJ
    THEN
        ANY ex WHERE ex : E_OBJ - es
        THEN
            es := es \/ { ex }
        END
    END;

  addparent(ex,parentx) =
    PRE ex : es & parentx : es
    THEN
      q1 :: NAT || q2 :: NAT
    END;

  step1() =
    PRE q1 > 0
    THEN
      q1 :: 0..q1-1 || q2 :: NAT
    END;

  step2() =
    PRE q1 = 0 & q2 > 0
    THEN
      q2 :: 0..q2-1
    END
END
```

The operations adding elements to the source model set $q1$ and $q2$ arbitrarily, whilst the transformation step operations strictly reduce some q.

The original M_τ machine is then used to define a refinement of $M0_\tau$, with the refinement relation giving an explicit definition of the q variants:

```
REFINEMENT E0_REF
REFINES E0 SEES SystemTypes
VARIABLES es, parent, ancestor
INVARIANT
  es <: E_OBJ &
  parent : es --> FIN(es) &
  ancestor : es --> FIN(es) &
  !ex.(ex : es  =>  ancestor(ex) <: (closure1(rel(parent)))[{ex}]) &
  q1 = card({ exx | exx : es & not(parent(eex) <: ancestor(eex)) }) &
  q2 = SIGMA(exx).( exx : es | card((closure1(rel(parent)))[{exx}] - ancestor(exx)))
```

```
INITIALISATION
  es, parent, ancestor := {}, {}, {}
OPERATIONS
  create_E() =
    PRE es /= E_OBJ
    THEN
        ANY ex WHERE ex : E_OBJ - es
        THEN
            es := es \/ { ex } ||
            parent(ex) := {} ||
            ancestor(ex) := {}
        END
    END;

  addparent(ex,parentx) =
    PRE ex : es & parentx : es
    THEN
      parent(ex) := parent(ex) \/ { parentx }
    END;

  step1() =
    PRE #ex.( ex : es & not(parent(ex) <: ancestor(ex)))
    THEN
        ANY ex WHERE ex : es & not(parent(ex) <: ancestor(ex))
        THEN
          ancestor(ex) := ancestor(ex) \/ parent(ex)
        END
    END;

  step2() =
    PRE !ex( ex : es  =>  parent(ex) <: ancestor(ex) )  &
      #ex.( ex : es & not(union(ancestor[parent(ex)]) <: ancestor(ex)))
    THEN
        ANY ex WHERE ex : es & not(union(ancestor[parent(ex)]) <: ancestor(ex))
        THEN
          ancestor(ex) := ancestor(ex) \/ union(ancestor[parent(ex)])
        END
    END
END
```

*step*2 represents an arbitrary transformation step of the second constraint of the transformation, the ANY statement chooses one element ex to which the second constraint can be applied, and applies it.

The refinement proof then attempts to verify that the explicit definition of $q2$ obeys the abstract specification, ie, that it is strictly decreased by every execution of *step*2. The refinement proof also establishes that $q1$ and es are not changed by *step*2.

The refinement obligations in B are [24]:

– The joint invariants $Inv_A \wedge Inv_R$ of the abstract and refined machines are satisfiable together.
– The refined initialisation establishes the invariants:

$$[Init_R]\neg\,[Init_A]\neg\,(Inv_A \wedge Inv_R)$$

– Each refined operation satisfies the pre-post relation of its abstract version:

$$Inv_A \wedge Inv_R \wedge Pre_A \;\Rightarrow\; Pre_R \wedge [Def_R]\neg\,[Def_A]\neg\,(Inv_A \wedge Inv_R)$$

In the transitive closure example, there is clearly a joint state which satisfies the combined invariants (eg., the empty model with $q1 = 0$ and $q2 = 0$). For $step2$, $q2 > 0$ means that some $ex : es$ has

$$\neg\,(union(ancestor[parent(ex)]) \subseteq ancestor(ex))$$

because otherwise $ancestor(ex) = (closure1(rel(parent)))[\{ex\}]$ for all ex. Also, $q1 = 0$ implies that

$$parent(ex) \subseteq ancestor(ex)$$

for all $ex : es$.

Therefore the abstract precondition implies the refined precondition. In addition, each $ex : es$ which satisfies the WHERE condition in the refined operation has that

$$union(ancestor[parent(ex)]) \subseteq ancestor(ex)$$

after the operation, so the operation adds at least one new element of $(closure1(rel(parent)))[\{ex\}]$ to $ancestor(ex)$. Therefore the term $card((closure1(rel(parent)))[\{ex\}] - ancestor(ex))$ in the sum defining $q2$ has been reduced, and no other term in the sum has been modified, and so $q2$ has been strictly decreased, satisfying the abstract operation.

To verify confluence, we prove that there is essentially a unique state where $q2 = 0$. This can be verified in the refinement by adding an *ASSERTIONS* clause:

```
ASSERTIONS
  q2 = 0  =>
      !ex.(ex : es  =>  ancestor(ex) = (closure1(rel(parent)))[{ex}])
```

This clause expresses that the predicate of the clause logically follows from the invariant and other contextual information for the data of the component.

The B tools will produce proof obligations for this assertion, and will act as a proof assistant in structuring the proof and carrying out routine proof steps automatically. A further consequence of the proof of the assertion is semantic correctness of the standard implementation for the transformation: that the desired semantics of target model elements relative to source model elements also holds at termination.

For recursive form specifications, the transformation steps can be expressed as B operations in the same way as for conjunctive-implicative form specifications. Internal consistency proof can be used to prove some inductive assertions about the intermediate and final transformation state (eg., $x(ax) > 0$ in the gcd example). Refinement proof can be used to establish that a postulated Q function is a variant, proving termination, and that other expressions have invariant values over transformation steps, and therefore are not modified by the transformation. Confluence and semantic correctness are formalised as properties which should follow from $Q = 0$, as for the conjunctive-implicative case.

To verify that a transformation satisfies a postulated language-level interpretation χ, invariants can be added to M_τ to express that χ holds between corresponding source and target model elements. For example, if source entity S_i is interpreted by target entity T_j, and attribute *satt* of S_i has interpretation $\chi(satt)$ some function $f(tatt)$ of T_j, we can verify this interpretation for τ by adding the invariant

```
!(s,t).(s : sis & t : tjs & idS(s) = idT(t)   =>  satt(s) = f(tatt(t)))
```

to M_τ, where s and t correspond if they have equal identity attribute values. Similarly for validation of association interpretations.

The above procedures can be used as a general process for proving *Ens* and *Pres* properties, for verifying interpretations, and for proving termination, semantic correctness and confluence, by means of suitable postulated Q variant functions. They can also be used to show that the values of particular expressions are not changed by transformation steps.

The techniques can be generalised to any transformation implementation whose transformation steps can be *serialised*, that is, each execution of the implementation is equivalent to one in which the steps occur in a strict sequential order. This assumption is made implicitly in the B model.

Representation and verification of other declarative model transformation languages, such as TGG or QVT-R could be carried out by means of translations of these languages to OCL constraints, following the approach of [5], and hence to B.

Proof transcripts can be produced by the Atelier B proof assistant, these could then be checked by an independent proof checker in order to achieve certification requirements of standards such as DO178C [12].

B is not suitable for establishing satisfiability properties asserting the existence of models of certain kinds, and tools such as UMLtoCSP [4] and USE [14] are more appropriate for these.

9 Implementation of constraint-based specifications

At the design and implementation level we need to verify that:

– The design and implementation of the transformation satisfy its specification.

– The design and implementation satisfy termination and confluence properties.

As far as possible, we minimise the amount of verification required by synthesising designs and implementations according to a process which produces correct-by-construction designs and code from specifications.

The procedural interpretation $stat(Cn)$ of constraints Cn can be used as the basis for designs and implementations of transformations.

$stat(Cn)$ is expressed in a small procedural language including assignment, conditionals, operation calls and loops (Figure 5). 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 implementation level of the B AMN statement language [24], to support detailed verification of designs.

For each statement construct in the language there is a definition of weakest precondition $[\,]P$, supporting the verification of implementations. This is defined as for B generalised substitutions [24].

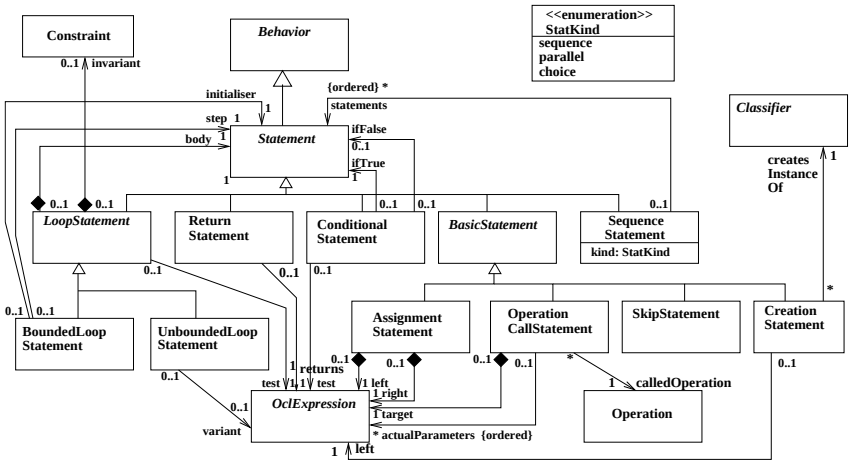

Fig. 5. Statement metamodel

$stat(P)$ is defined so that it is the minimal statement in the activity language which ensures that P holds:

$$[stat(P)]P$$

For individual constraints, therefore, correctness is ensured by the definition of $stat$. In cases of fixpoint computations (type 2 and 3 constraints), termination and confluence need to be proven separately, using a suitable Q measure for the constraint.

Individual constraints Cn:

$$\forall\, s : S_i \cdot SCond \ implies \ \exists\, t : T_j \cdot TCond \ and \ Post$$

are examined to identify which implementation strategy can be used to derive their design. This depends upon the features and objects read and written within the constraint (Table 7).

	Constraint properties	*Implementation choice*
Type 0 constraint	Update of single object	$stat(Cn)$
Type 1 constraint	No interference between different applications of constraint, and no change to S_i or $rd(SCond)$: $wr(Cn) \cap rd(Cn) = \{\}$.	Approach 1: single for loop `for` $s : S_i$ `do` $s.op()$
Type 2 constraint	Interference between different applications of constraint, but no update of S_i or $rd(SCond)$ within constraint: $S_i \notin wr(Cn)$, $wr(Cn) \cap rd(SCond) = \{\}$	Approach 2: while iteration of for loop. Q measure needed for termination and correctness proof
Type 3 constraint	Update of S_i or $rd(SCond)$ within constraint. $S_i \in wr(Cn)$, or $wr(Cn) \cap rd(SCond) \neq \{\}$	Approach 3: while iteration of search-and-return for loop Q measure needed for termination and correctness proof

Table 7. 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

`for` $s \ : \ S_i$ `do` $s.op_i()$

where in S_i we include an operation of the form:

$op_i()$
`post:`
$SCond[self/s] \quad implies \quad \exists \ t \ : \ T_j \ \cdot \ TCond \ and \ Post[self/s]$

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

$$v : s \ \Rightarrow \ [acts(v)]P(v)$$

derive

$$[for\ v : s\ do\ acts(v)](\forall\, v : s@pre \cdot P(v))$$

is valid for such iterations, provided that one execution of *acts* does not affect another: the precondition of each $acts(v)$ has the same value at the start of $acts(v)$ as at the start of the loop, and if $acts(v)$ establishes $P(v)$ at its termination, $P(v)$ remains true at the end of the loop [29].

Specifically,

$$s \in \overline{S_i} \ \Rightarrow\ [s.op_i()](SCond \ \Rightarrow\ Succ)$$

by definition of $op_i()$, where *Succ* is $\exists\, t : T_j \cdot TCond\ and\ Post$, so by the above inference, $stat(Cn)$ defined as

$$for\ s : S_i\ do\ s.op_i()$$

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 source domain. More precisely the worst case complexity is linear in

$$\#\overline{S_i} * (cost_{eval}(SCond) + cost_{act}(Succ))$$

where $cost_{eval}(e)$ is the time required to evaluate e, and $cost_{act}(e)$ the time required to execute $stat(e)$. If additional source 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 strategy is required for type 2 and type 3 constraints. Consider a constraint *Cn*:

$$\forall\, s : S_i \cdot SCond\ \ implies\ \ \exists\, t : T_j \cdot TCond\ and\ Post$$

In the case where

$$wr(Cn) \cap (rd(Post) \cup rd(TCond))$$

is non-empty but the other conditions of non-interference still hold (ie., a type 2 constraint), a fixpoint iteration of the form:

```
running := true;
while (running) do
  (running := false;
   for s : S_i do
     if SCond then
       if Succ then skip
       else (s.op(); running := true)
     else skip)
```

can be used, where *Succ* is $\exists\, t : T_j \cdot TCond \ and \ Post$ and $op()$ is defined as:

$op()$
`post:`
$Succ[self/s]$

In the conditional test *Succ* is evaluated in a non side-effecting manner.

We refer to this strategy as constraint implementation approach number 2. The conditional test of *Succ* can be omitted if it is known that $SCond \Rightarrow not(Succ)$. The UML-RSDS tools perform algebraic simplification to check if *SCond* contradicts *Succ*.

If the updates to the written data are monotone and bounded (eg, as in the case of the inclusion operator $\subseteq$ for the transitive closure computation example), then the iteration terminates and computes the least fixed point. A measure $Q : \mathbb{N}$ over the source and target models can be used to prove termination, as in Section 7.3.

A suitable measure in this case is the sum $\Sigma_{x:E} \#(x.parent^+ \ - \ x.ancestor)$. Q is decreased by each application of *op* and is never increased.

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

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

where *body* is the body of the while loop.

If some $s : S_i$ has $SCond \ and \ not(Succ)$ true, at the start of an iteration of the while loop, then $s.op()$ is executed for such an s, and this execution decreases Q. Q is never increased, so condition (2) holds.

Q is also necessary to prove correctness: while there remain $s : S_i$ with *SCond* true but *Succ* false, then $Q > 0$ and the iteration will apply *op* to such an s. At termination, $running = false$, which can only occur if there are no $s : S_i$ with *SCond* true but *Succ* false, so the constraint therefore holds true, and $Q = 0$. Confluence also follows if $Q = 0$ is only possible in one unique 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.

For the transitive closure computation, the inner loop iterates:

`if` $x.parent.ancestor \ \subseteq \ x.ancestor$
`then skip`
`else (`$t.op();$ $running \ := \ true$`)`

The time complexity of the implementation depends on the value of Q on the starting models, and on the size $\#\overline{S_i}$ of the domain. The worst case complexity is of the order

$$Q(smodel, tmodel) * \#\overline{S_i} * (cost_{eval}(SCond) + cost_{eval}(Succ) + cost_{act}(Succ))$$

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

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:

`while` *some source element s satisfies a constraint lhs* `do`
select such an s and apply the constraint

can be used. This can be explicitly coded as:

running := *true*;
`while` *running* `do`
  running := *search*()

where:

```
search() : Boolean
  (for s : Si 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 $SCond \Rightarrow not(Succ)$.

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 $not(Succ)$ decreases Q, even if new elements of S_i are generated.

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

$$Q(smodel, tmodel) * maxS * (cost_{eval}(SCond) + cost_{eval}(Succ) + cost_{act}(Succ))$$

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

A similar implementation strategy to approach 3 can be used for recursive constraints.

To link the implementation of one constraint to the implementation of another, the implementation of sequentially later constraints needs to identify which target elements of specific target entity types have already been created, and from which source elements.

The UML-RSDS tools efficiently implement searches for the set of elements $TSub[sexp.id1]$ of a target type *TSub* with primary key values in $sexp.id1$ by maintaining a map from the primary keys of *TSub* objects to the objects – this is termed the *object indexing* pattern.

To implement conjunctive-implicative specifications which satisfy semantic non-interference, we 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.

This approach enables the modular decomposition of the transformation, usually as a sequential composition (chaining) of sub-transformations.

The transformation design is decomposed 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 $p1$ must precede phase $p2$ if it creates instances of an entity $T1$ which is read in $p2$.

Figure 6 shows the schematic structure of this approach.

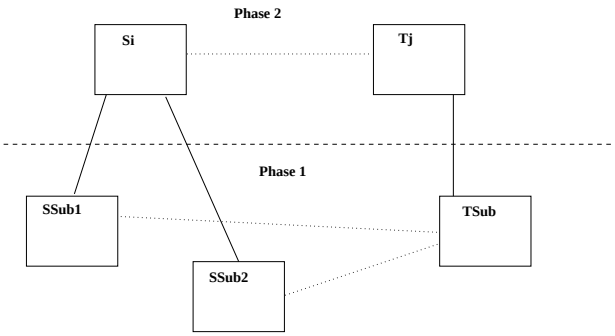

Fig. 6. Phased creation structure

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 *Cons* is a conjunction of constraints of the form

$$\forall\, s : S_i \cdot SCond \ implies \ \exists\, t : T_j \cdot TCond \ and \ Post$$

then each constraint may be mapped individually into a phase P_i which implements it, provided that there are no circularities in the data dependency relationship $C_k < C_l$ between constraints. This relationship should imply that $k < l$.

Each phase P_i is defined as $stat(C_i)$, so will establish C_i at its termination, under certain assumptions 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 *Cons*, under the assumption that each P_k preserves C_l for $l < k$:

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

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

Let 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 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 $wr(C_k)$ for $k \leq i$).

Therefore we can assert that (1):

$$Asm_0 \wedge \bigwedge_{k<i} C_k \;\Rightarrow\; [P_i](C_i \wedge Asm_i)$$

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

$$Asm_0 \;\Rightarrow\; [P_1](C_1 \wedge Asm_1)$$

and if

$$Asm_0 \;\Rightarrow\; [P_1;\ ...;\ P_i](C_1 \wedge ... \wedge C_i)$$

for some $i < n$, then also

$$Asm_0 \;\Rightarrow\; [P_1;\ ...;\ P_i](C_1 \wedge ... \wedge C_i \wedge [P_{i+1}](C_{i+1} \wedge Asm_{i+1}))$$

by (1), and

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

by the semantic non-interference property, so

$$Asm_0 \;\Rightarrow\; [P_1;\ ...;\ P_{i+1}](C_1 \wedge ... \wedge C_{i+1} \wedge Asm_{i+1})$$

as required.

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 $C_l < C_k$) with outer quantifier $\forall\, s : S_i$, approach 2 has the form

```
running := true;
while (running) do
  (running := false;
   for s : Si do loop1;
   for s : Si do loop2)
```

where *loop*1 is the code:

```
if SCond1 then
  if Succ1 then skip
  else (s.op1(); running := true)
```

and *loop*2 is:

```
if SCond2 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:

```
search() : Boolean
  (for s : Si do
     if SCond1 then
       if Succ1 then skip
       else (s.op1(); return true));
  (for s : Si 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 the *completeness* and *correctness* of this implementation strategy in the sense of [46]: *Cons* can be established by the strategy, and any computation that satisfies the strategy establishes *Cons*. This also proves semantic correctness as defined in the third correctness obligation (III) of Section 3.

10 Verification of model transformation composition

In the preceeding sections we have considered *basic* transformations, transformations defined without reference or dependence upon other transformations. In practice, transformations need to be combined together as parts of a software system, and need to be adapted to work in modified environments, such as for extended metamodels.

The following forms of internal and external compositions of transformations are used in model transformation languages, and are supported by UML-RSDS:

- External composition of transformations using sequencing, conditional execution, iteration, including fixpoint iteration.
- Conjunction and parallel composition.
- Inheritance and superposition.

– Parameterisation, including higher-order parameterisation (parameterisation of transformations by transformations).
– Instantiation of generic transformations.

The simplest and most widely-used composition of transformations is sequencing or chaining of two or more transformations. For example, a quality-improvement transformation on class diagrams could consist of three successive subtransformations (i) to remove redundant inheritances; (ii) to remove multiple inheritance; (iii) to make concrete non-leaf classes abstract.

If transformations $\tau_1 : S \rightarrow T$ and $\tau_2 : T \rightarrow R$ satisfy the correctness conditions of Section 3 for their individual specifications Asm_i, Ens_i, $Pres_i$, $Cons_i$, χ_i, $i = 1, 2$ and designs act_1, act_2, then $\tau_1 ; \tau_2$ satisfies the specification Asm_1, Ens_2, $Pres_1$, $Cons_1$ *and* $Cons_2$, with design $act_1 ; act_2$ and $\chi = \chi_1 ; \chi_2$, provided that:

1. τ_1 establishes the preconditions of τ_2:

$$\vdash_{\mathcal{L}_{S \cup T \cup R}} Cons_1 \Rightarrow Asm_2$$

2. τ_2 does not invalidate $Cons_1$:

$$\vdash_{\mathcal{L}_{S \cup T \cup R}} Cons_1 \Rightarrow [act_2]Cons_1$$

3. Ens_1 includes Γ_T
4. $\chi_1(Pres_1)$ is a preserved property of τ_2:

$$Asm0_2, Cons_2, \chi_1(Pres_1), \Gamma_T \vdash_{\mathcal{L}_{T \cup R}} \chi_2(\chi_1(Pres_1))$$

If τ_1 is terminating and semantically correct under the assumption of Asm_1, and τ_2 is terminating and semantically correct under the assumption of Asm_2, and τ_1 establishes Asm_2 under the assumption of Asm_1, and τ_2 is semantically non-interfering with τ_1, then $\tau_1 ; \tau_2$ is terminating and semantically correct under the assumption of Asm_1. Under these assumptions, if τ_1 and τ_2 are confluent and τ_2 is isomorphism-preserving, then $\tau_1 ; \tau_2$ is confluent. Thus a sequential combination of basic transformations can be combined under the above conditions into a single basic transformation, whose *Cons* sequence is $Cons_1$ followed by $Cons_2$.

Similar reasoning can be used to establish properties for the conditional execution *if* E *then* τ and bounded iteration *for* $x : s$ *do* τ of transformations. For unbounded iteration *while* E *do* τ, a variant function $Q : \mathbb{N}$ is required to ensure termination of the iteration: this must be decreased by each execution of the design *act* of τ:

$$\forall \nu : \mathbb{N} \cdot Q = \nu \wedge E \wedge \nu > 0 \Rightarrow [act](Q < \nu)$$

In UML-RSDS such procedural composition of transformations is implemented by means of the UML composition mechanisms for use cases. These mechanisms also provide a form of parallel composition or conjunction of transformations. UML use cases can be composed by means of *extend* and *include* relations between use cases (Figure 7):

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 externally compose transformations. In particular, extension can be logically interpreted as a conjunction of the base and extended use cases (the component preconditions are conjoined to form the composed precondition, and likewise for the postconditions), and used as a means for separating out different aspects 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.

Logically, the composed semantics of a basic transformation b extended by a basic transformation e is the transformation specified by Asm as $Asm_b \wedge Asm_e$, Ens as $Ens_b \wedge Ens_e$, and $Cons$ as $Cons_b \wedge Cons_e$. Procedurally, the order of the $Cons$ constraints (ie, the extension points to insert the postcondition constraints of e into those of b) is determined by the data dependency relations between them.

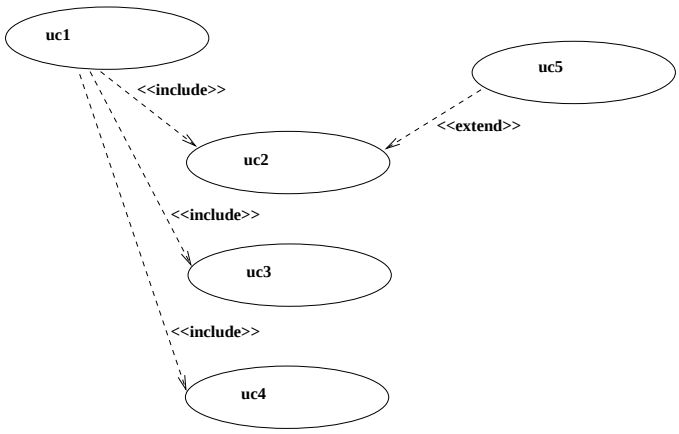

Fig. 7. Decomposition using *extend* and *include*

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 *include* mechanism can be used to sequentially chain together a series of use cases. The included transformations are relatively independent of each other, although a preceding use case must ensure the assumptions of its successor. Included use cases can be shared by several different including use cases. An including use case can have (via its associated *classifierBehavior*) an activity which invokes the included use cases according to a specific algorithm or workflow.

The decomposition of a transformation τ into two concurrent sub-transformations τ_1 and τ_2 composed into τ by *extend* is possible if: the write frame $wr(\tau)$ of τ can be divided into two disjoint non-empty sets V_1, V_2, such that the set $slice(\tau, V_1)$ of constraints C of τ with $wr(C) \subseteq V_1$ is non-empty and disjoint from $slice(\tau, V_2)$, which is also non-empty, and then for $\tau_1 = slice(\tau, V_1)$ and $\tau_2 = slice(\tau, V_2)$, V_1 is disjoint from $rd(\tau_2)$ and V_2 is disjoint from $rd(\tau_1)$. If only the last condition $V_2 \cap rd(\tau_1) = \{\}$ holds then τ can instead be sequentially decomposed into τ_1; τ_2 using *include*.

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]). If $uc2$ inherits $uc1$ and both represent basic transformations from S to T with assumptions $Asm1$, $Asm2$ and postconditions $Cons1$ and $Cons2$ respectively, then:

- $Asm2$ cannot be logically stronger than $Asm1$:

  $$\vdash_{S \cup T} Asm1 \Rightarrow Asm2$$

  because $uc2$ should be applicable whenever $uc1$ is.
- $Cons2$ should be at least as logically strong as $Cons1$:

  $$\vdash_{S \cup T} Cons2 \Rightarrow Cons1$$

  because $uc2$ should satisfy the postcondition of $uc1$. New postcondition constraints can be introduced in $Cons2$, and named constraints of $Cons1$ can be replaced by logically stronger constraints in $Cons2$, but otherwise the ordering of constraints in $Cons1$ cannot be changed.

This is a more formal version of the concept of *superposition* of transformations [53]. Some of the verification work used for $uc1$ can be reused for $uc2$: proofs that Q functions are variants for constraints that occur in both use cases can be retained, for example.

Internal composition of transformations can be achieved by the use of *parameterised* transformations, as proposed in [54]. 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 can be factored out into a single predicate which is supplied as a parameter and can be changed independently of the main use case.

For example, consider the creation of tracing information by rules. Rules in the primary use case can have the form

$$\forall\, s : S_i \cdot SCond \ implies \ \exists\, t : T_j \cdot TCond \ and \ Post \ and \ GenTrace$$

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:

$$\exists\, tr : Trace \cdot tr.source = s \ \& \ tr.target = t$$

To switch off tracing, an actual parameter *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. Assumptions on parameters can be included in Asm. The instantiated specification is formed by textual substitution of the actual parameters for the formal parameters in the transformation constraints: transformation $\tau(v)$ with formal parameters $v_1, ..., v_n$ and assumptions Asm and postconditions $Cons$ is instantiated by actual parameters $a_1, ..., a_n$ to form a transformation $\tau(a)$ with assumptions $Asm[a/v]$ and postconditions $Cons[a/v]$.

For parameters of simple types (booleans, strings, numerics), B analysis can be carried out using the formal parameters, which become parameters of the machine representing the transformation. This enables proofs of syntactic correctness, termination, semantic preservation and other properties to be carried out independently of the parameters. For general parameters, analysis of the transformation is performed after instantiation.

Parameterisation provides a means to reuse and instantiate *generic* transformations, such as the computation of transitive closures.

One way to specify a generic transformation is to define it as a transformation $\tau : S0 \rightarrow T0$ between the minimal metamodel structures that it concerns, and then to instantiate it to specific source and target languages S and T.

In UML-RSDS we implement generic transformations by parameterisation: the formal parameters are the entities and features involved in the transformation, and these are instantiated by suitable set-valued expressions and expressions denoting features or compositions of features, respectively.

To instantiate a generic transformation $\tau : S0 \rightarrow T0$ to languages S and T, the developer must define consistent embeddings

$$I : S0 \rightarrow S$$
$$J : T0 \rightarrow T$$

of the languages, such that I and J map entities to entities, and features to expressions, they are injective, and satisfy:

$$I(x) = J(x) \ \ for \ \ x \in S0 \cap T0$$
$$I(x) \neq J(y) \ \ for \ \ x \neq y$$
$$if \ \ f : E \rightarrow Typ \ \ then \ \ I(f) : I(E) \rightarrow Typ$$
$$if \ \ f : E_1 \rightarrow E_2 \ \ then \ \ I(f) : I(E_1) \rightarrow I(E_2)$$
$$if \ \ f : E_1 \rightarrow \mathbb{F}(E_2) \ \ then \ \ I(f) : I(E_1) \rightarrow \mathbb{F}(I(E_2))$$
$$if \ \ f : E_1 \rightarrow \text{seq}(E_2) \ \ then \ \ I(f) : I(E_1) \rightarrow \text{seq}(I(E_2))$$

and likewise for J. The instantiations must satisfy any assumptions in Asm. If $E \in wr(\tau)$ is a concrete entity then it can only be instantiated by a single

concrete entity. If $f \in wr(\tau)$, then it 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 $Asm[I(x)/x, J(y)/y]$ of the generic assumptions, likewise for its *Cons* constraints.

Proofs in B of correctness, termination and confluence properties for the generic transformation will be preserved by the instantiation, since these proofs make use only of the representation of entities as sets of instances, and as features as maps of certain forms. However, type 1 constraints in the generic transformation may become type 2 or 3 constraints in the instantiated version, so that verification needs to be carried out taking into account the most general instantiation of the constraints.

11 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, including the removal of multiple inheritance and removal of duplicated attributes [22].

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 input-preserving transformation mechanisms. The class diagram rationalisation problem [22] also involves complex restructuring of models. A comparison of the UML-RSDS solution with Kermeta and QVT-R solutions identified that our solution is simpler and more modular (Table 8) than these solutions.

Solution	*Lines of specification*	*Number of inter-rule invocations*
UML-RSDS	25	0
Kermeta	202	8
QVT-R	189	16

Table 8. Class diagram rationalisation solutions

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

The increasing use of model-driven engineering for critical software systems has prompted the formulation of standards and standards supplements for the use of model-based development. In particular, the DO-178C avionics standard [12] updates the earlier DO-178B standard by including an annex for model-based development and verification.

Regarding transformations and transformation tools, the requirements of DO-178C imply that:

- Transformations that derive one model from another, and code text from models, should be able to provide detailed tracing information to identify how elements of the target model are derived from elements of the source model. This is necessary to ensure that all intended functionality and properties of the source model are correctly implemented/represented in the target, and that no unintended functionalities or properties are implemented.
- When using tools to analyse transformations or models, any mapping to the representation used by the tool must be *conservative*, ie., if a property can be derived from the representation then it is true for the original system.

These requirements are satisfied by the UML-RSDS approach: tracing is provided by means of element primary keys (the target model elements derived from a source model element will usually be assigned the same key value as that element). Other tracing schemes can also be defined to achieve detailed correlation between models. The mapping to B is conservative.

The UML-RSDS tools, together with examples of transformation specification and implementation, are available at: *http* : *//www.dcs.kcl.ac.uk/staff/kcl/uml*2*web/*.

12 Related Work

In summary, we have provided verification techniques for the following properties of model transformations:

- Completeness of rules: syntactic checks on data written by the rule.
- Definedness of rules: syntactic checks on subexpressions within the rule.
- Syntactic correctness of rules and transformations: by internal consistency proof in B.
- Termination of transformations: by syntactic checks for type 1 constraints, by refinement proof in B for other constraints (proof that a specified Q function is a variant).
- Confluence of transformations: syntactic checks for type 1 constraints, by proof (manual or proof-tool assisted, such as B assertion proof) of uniqueness of terminal $Q = 0$ states for other constraints.
- Semantic correctness of transformations: by standard inferences for type 1 constraints (correctness by construction for the designs and implementations), by proof tools for other constraints, as derived consequences of $Q = 0$.
- Semantic preservation properties: by internal consistency proof in B.

Validation is also supported, both by animation of the B machine derived from the transformation, and by testing of the synthesised Java code.

Previous work in the verification of model transformations has been hindered by the general lack of high-level transformation specifications, with most transformation development focussing upon the implementation level [16].

In [5], 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. The derived constraints can be quite complex, and we consider that it is preferable to write transformation specifications – as a starting point for transformation development – in terms of constraints in order to facilitate direct verification, without the need for constraint extraction.

In [47] the concept of the conjunctive-implicative form was introduced to support the automated derivation of correct-by-construction transformation implementations from specifications written in a constructive type theory. A single undecomposed constraint of the $\forall \Rightarrow \exists$ form defines the entire transformation, and the constructive proof of this formula produces a function which implements the transformation. In [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 in the context of first order logic, with systematic strategies and patterns used to derive transformation implementations from transformation specifications, based on the detailed structure of the specifications. The correctness of the strategies and patterns are already established and do not need to be re-proved for each transformation, although side conditions (such as the data-dependency conditions and existence of a suitable Q measure) need to be established.

With correct-by-construction approaches, the verification effort is focussed upon syntactic correctness (proof of *Ens*) and semantic preservation (proof of *Pres*). Specific work on syntactic correctness includes formalisation and proof of target model constraints in MAUDE [11], and the graph-theory approach of [2]. For semantic preservation, an approach based on graph morphisms has been used for TGG [17]. Our approach using B has the advantage that syntactic correctness corresponds directly to internal consistency in B, and that the set-theoretic formalisation of metamodels, constraints and transformations in B can be directly related to standard set-theoretic semantics for UML and OCL [27, 49].

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 [16], 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*, *Ens* and *Pres* play the same role as the pattern-based specification language of [16]. 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.

We have adopted B for formal verification of transformation properties, as it has strong tool support, and a relatively simple mathematical basis which is appropriate for software verification. It can support verification of most forms of transformation verification properties, however it lacks support for analysis of satisfiability properties, such as the rule applicability and executability conditions, or for the instantiation of metamodels with counter-examples to properties, and other tools, such as the USE environment [14] or UMLtoCSP [4] could be applied instead to carry out such analysis.

13 Conclusions

The approach described here provides a systematic specification, development and verification 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, recursive form), and described how executable implementations of these specifications can be automatically derived, so that the implementations are correct with respect to the specifications.

We have described verification techniques to check properties of termination, syntactic and semantic correctness, semantic preservation, and confluence.

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. Our constraint-based definitions of transformations can be both logically interpreted as high-level specifications, and procedurally interpreted as templates for designs and implementations that satisfy the specifications.

Acknowledgement

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

References

1. M. van Amstel, S. Bosems, I. Kurtev, L. F. Pires, *Performance in Model Transformations: experiments with ATL and QVT*, ICMT 2011, LNCS 6707, pp. 198–212, 2011.
2. T. Baar, S. Markovic, *A graphical approach to prove the semantic preservation of UML/OCL refactoring rules*, Perspectives of Systems Informatics, LNCS vol. 4378, pp. 70–83, Springer-Verlag, 2007.
3. J. Bezivin, F. Jouault, J. Palies, *Towards Model Transformation Design Patterns*, ATLAS group, University of Nantes, 2003.
4. J. Cabot, R. Clariso, D. Riera, *UMLtoCSP: a tool for the verification of UML/OCL models using constraint programming*, Automated Software Engineering '07, pp. 547–548, ACM Press, 2007.
5. J. Cabot, R. Clariso, E. Guerra, J. De Lara, *Verification and Validation of Declarative Model-to-Model Transformations Through Invariants*, Journal of Systems and Software, 2010.
6. K. Czarnecki, S. Helsen, *Classification of Model Transformation Approaches*, OOPSLA 03 workshop on Generative Techniques in the context of Model-Driven Architecture, OOPSLA 2003.
7. K. Czarnecki, S. Helsen, *Feature-based survey of model transformation approaches*, IBM Systems Journal, vol. 45, no. 3, 2006, pp. 621–645.
8. J. Cuadrado, J. Molina, *Modularisation of model transformations through a phasing mechanism*, Software Systems Modelling, Vol. 8, No. 3, pp. 325–345, 2009.
9. Dresden OCL toolset, http://www.dresden-ocl.org, 2011.
10. Eclipse organisation, *EMF Ecore specification*, http://www.eclipse.org/emf, 2011.
11. M. Egea, V. Rusu, *Formal executable semantics for conformance in the MDE framework*, Innovations in System Software Engineering, 6 (1-2), pp. 73–81, 2010.
12. FAA, DO-178C, Software considerations in airborne systems and equipment certification, January 2012.
13. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
14. M. Gogolla, J. Bohling, M. Richters, *Validating UML and OCL models in USE by automatic snapshot generation*, Software and Systems Modeling, vol. 4, no. 4, pp. 386–398, 2005.
15. P. Van Gorp, S. Mazanek, A. Rensink, *Live Challenge Problem*, TTC 2010, Malaga, July 2010.
16. E. Guerra, J. de Lara, D. Kolovos, R. Paige, O. Marchi dos Santos, *transML: A family of languages to model model transformations*, MODELS 2010, LNCS vol. 6394, Springer-Verlag, 2010.
17. F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, Y. Xiong, *Correctness of model synchronisation based on Triple Graph Grammars*, MODELS 2011, LNCS vol. 6981, pp. 748–752, Springer-Verlag, 2011.
18. F. Jouault, F. Allilaire, J. Bezivin, I. Kurtev, *ATL: A model transformation tool*, Science of Computer Programming, 72, pp. 31–39, 2008.
19. F. Jouault, I. Kurtev, *Transforming Models with ATL*, in MoDELS 2005, LNCS Vol. 3844, pp. 128–138, Springer-Verlag, 2006.
20. Kermeta, http://www.kermeta.org, 2010.
21. S. Kolahdouz-Rahimi, K. Lano, *A Model-based Development Approach for Model Transformations*, FSEN 2011, Iran, 2011.

22. S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya, P. Van Gorp, *Goal-oriented measurement of model transformation methods*, submitted to Science of Computer Programming, 2012.
23. I. Kurtev, K. Van den Berg, F. Joualt, *Rule-based modularisation in model transformation languages illustrated with ATL*, Proceedings 2006 ACM Symposium on Applied Computing (SAC 06), ACM Press, pp. 1202–1209, 2006.
24. K. Lano, *The B Language and Method*, Springer-Verlag, 1996.
25. K. Lano, J. Bicarregui, *Semantics and Transformations for UML Models*, UML 98, Mulhouse, France, June 1998, Springer-Verlag LNCS vol. 1618, 1998, pp. 107–119.
26. K. Lano, D. Clark, *Model transformation specification and verification*, QSIC '08, pp. 45–54, 2008.
27. K. Lano (ed.), *UML 2 Semantics and Applications*, Wiley, New York, 400 pages, 2009.
28. K. Lano, *A Compositional Semantics of UML-RSDS*, SoSyM, 8(1): 85–116, 2009.
29. K. Lano, S. Kolahdouz-Rahimi, *Specification and Verification of Model Transformations using UML-RSDS*, IFM 2010, LNCS vol. 6396, pp. 199–214, 2010.
30. K. Lano, S. Kolahdouz-Rahimi, *Slicing of UML models using Model Transformations*, MODELS 2010, LNCS vol. 6395, pp. 228–242, 2010.
31. K. Lano, S. Kolahdouz-Rahimi, *Migration case study using UML-RSDS*, TTC 2010, Malaga, Spain, July 2010.
32. K. Lano, S. Kolahdouz-Rahimi, *Model-driven development of model transformations*, ICMT 2011, June 2011.
33. K. Lano, S. Kolahdouz-Rahimi, *Specification of the "Hello World" case study*, TTC 2011.
34. K. Lano, S. Kolahdouz-Rahimi, *Specification of the GMF migration case study*, TTC 2011.
35. K. Lano, S. Kolahdouz-Rahimi, J. Terrell, I. Poernomo, S. Zschaler, *Patterns for Model-transformation specification and implementation*, internal report, Dept. of Informatics, King's College London, 2011. `http://www.dcs.kcl.ac.uk/staff/kcl/mtpat2.pdf`.
36. S. Markovic, T. Baar, *Semantics of OCL Specified with QVT*, Software and Systems Modelling, vol. 7, no. 4, October 2008.
37. S. Markovic, T. Baar, *Refactoring OCL Annotated Class Diagrams*, MoDELS 2005, Springer-Verlag LNCS vol. 3713, Springer-Verlag, 2005.
38. T. Mens, K. Czarnecki, P. Van Gorp, *A Taxonomy of Model Transformations*, Dagstuhl Seminar Proceedings 04101, 2005.
39. OMG, *UML superstructure, version 2.3*, OMG document formal/2010-05-05, 2009.
40. OMG, *OCL 2.0 Specification*, Formal 06-05-01, 2006.
41. OMG, *Query/View/Transformation Specification*, ptc/05-11-01, 2005.
42. OMG, *Query/View/Transformation Specification*, annex A, 2010.
43. OMG, *Model-Driven Architecture*, http://www.omg.org/mda/, 2004.
44. OMG, *Meta Object Facility (MOF) Core Specification*, OMG document formal/06-01-01, 2006.
45. OptXware, *The Viatra-I Model Transformation Framework Users Guide*, 2010.
46. F. Orejas, E. Guerra, J de Lara, H. Ehrig, *Correctness, completeness and termination of pattern-based model-to-model transformation*, CALCO 2009, pp. 383–397, 2009.
47. I. Poernomo, *Proofs as model transformations*, ICMT 2008.
48. I. Poernomo, J. Terrell, *Correct-by-construction Model Transformations from Spanning tree specifications in Coq*, ICFEM 2010.

49. M. Richters, M. Gogolla, *On formalising the UML object constraint language OCL*, Proc. 17th Int. Conf. Conceptual Modelling (ER '98), Springer LNCS, 1998.
50. A. Schurr, *Specification of graph translators with triple graph grammars*, WG '94, LNCS vol. 903, Springer, 1994, pp. 151–163.
51. P. Stevens, *Bidirectional model transformations in QVT*, SoSyM vol. 9, no. 1, 2010.
52. D. Varro, A. Pataricza, *Automated Formal Verification of Model Transformations*, CSDUML 2003 Workshop, 2003.
53. D. Wagelaar, *Composition techniques for rule-based model transformation languages*, ICMT 2008.
54. S. Zschaler, I. Poernomo, J. Terrell, *Towards using constructive type theory for verifiable modular transformations*, proceedings of FREECO 2011.

A Semantic definitions

A.1 Definition of $stat(P)$

The design-level activity associated with a specification predicate P can be defined systematically based on the structure of P. Table 9 shows some of the main cases of this definition.

There are special cases for $P1$ *implies* $P2$ where $P1$ may contain variables which denote implicit additional universal quantifiers or let expression definitions. There are also special cases for $\exists$ and $\exists_1$ when the unique instantiation pattern applies. $stat(\forall\, x : E \cdot P1)$ is defined for type 2 and 3 quantified formulae as for transformations (Section 9).

A.2 Construction of inverse transformations

If all the *Cons* constraints of an input-preserving transformation τ satisfy internal syntactic non-interference and are of type 1, and *Cons* satisfies syntactic non-interference, we can derive a reverse transformation $\tau^{\sim}$ 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. Given a forward constraint Cn of the form

$$\forall\, s1 : S_{i1};\ ...;\ sn : S_{in} \cdot SCond\ implies\ \exists\, t1 : T_{j1};\ ...;\ tm : T_{jm} \cdot TCond\ and\ Post$$

where $SCond$ is a predicate in the si, and $TCond$ is a predicate in the tj only, the reverse constraint $Cn^{\sim}$ is:

$$\forall\, t1 : T_{j1};\ ...;\ tm : T_{jm} \cdot TCond\ implies\ \exists\, s1 : S_{i1};\ ...;\ sn : S_{in} \cdot SCond\ and\ Post^{\sim}$$

where $Post^{\sim}$ expresses the inverse of *Post*.

To form the inverse of *Post* predicates in the scope of quantifiers $\forall\, s : S_i ... \exists\, t : T_j$ we consider several cases of such predicates:

- $t.f = e$ where f is some feature of T_j and e is an expression $s.g$ which is a direct feature value of s. $Post^{\sim}$ is $s.g = t.f$.
- Evaluations $t.f = e.func$ where *func* is a reversible function such as *reverse*, *exp*, *log*, *sqrt*, $pow(k)$. $Post^{\sim}$ is $e = t.f.rfunc$ where *rfunc* is the reverse of *func* (Table 10).

P	$stat(P)$	*Condition*
$x = e$	$x := e$	x is assignable
$objs.f = e$	for $x : objs$ do $x.f := e$	Writable feature f collection $objs$
$e : x$ $x \rightarrow includes(e)$	$x := x \cup \{e\}$	x is assignable
$e \ / : x$ $x \rightarrow excludes(e)$	$x := x - \{e\}$	x is assignable
$e <: x$ $x \rightarrow includesAll(e)$	$x := x \cup e$	x is assignable
$e \ / <: x$ $x \rightarrow excludesAll(e)$	$x := x - e$	x is assignable
$x \rightarrow isDeleted()$	$E := E - \{x\}$	Each entity E containing x
$obj.op(e)$ $objs.op(e)$	$obj.op(e)$ for $x : objs$ do $x.op(e)$	Single object obj Collection $objs$
$P1$ *and* $P2$	$stat(P1);\ stat(P2)$	
$\exists\, x : E \cdot x.id = v$ *and* $P1$	if $v \in E.id$ then $x : E := E[v];\ stat(P1)$ else $x : E := new(E);\ stat(x.id = v \text{ and } P1)$	E is a concrete entity
$\exists\, x : E \cdot P1$	$x : E := new(E);\ stat(P1)$	E is a concrete entity $P1$ not of form $x.id = v$ *and* $P2$
$\exists\, x : E \cdot P1$	if $E \rightarrow exists(x \mid P1)$ then skip else $stat(\exists\, x : E \cdot P1)$	E is a concrete entity
$\exists\, x : e \cdot P1$	if $e \rightarrow exists(x \mid P1)$ then skip else $(x : E := e \rightarrow any();$ $stat(P1))$	Non-writable expression e with element type E
$\forall\, x : E \cdot P1$	for $x : E$ do $stat(P1)$	P type 1
$P1$ *implies* $P2$	if $P1$ then $stat(P2)$	

Table 9. Definition of $stat(P)$

– Equalities $t.f = TSub[s.g.id1]$ which select a single element of $TSub$ or a set of elements, with primary key value(s) $id2$ equal to $s.g.id1$ (if this is a single value), or in $s.g.id1$ (if it is a collection). The reversed form $Post^{\sim}$ in this case is $s.g = SSub[t.f.id2]$, if source model entity $SSub$ is in 1-1 correspondence with $TSub$ via the identities.
– Additions of t to a set e independent of s or t: $t : e$. The reversed constraint includes this as a condition on t in its $SCond$ clause in the antecedent.
– Conjunctions of implications

  $$(Cond_1 \ implies \ P_1) \ and \ ... \ and \ (Cond_r \ implies \ P_r)$$

  where the $Cond_l$ do not involve t. Each conjunct is re-written as a separate constraint

  $$\forall\, s : S_i \cdot SCond \ and \ Cond_l \ implies \ \exists\, t : T_j \cdot TCond \ and \ P_l$$

  which can then be reversed.
– For succedents of the form

  $$(\exists\, t : T_j \cdot TCond1 \ and \ P1) \ or \ (\exists\, t : T_k \cdot TCond2 \ and \ P2)$$

  for disjoint entities T_j, T_k, the reverse constraints have the form

  $$\forall\, t : T_j \cdot TCond1 \ implies \ \exists\, s : S_i \cdot SCond \ and \ P1^{\sim}$$
  $$\forall\, t : T_k \cdot TCond2 \ implies \ \exists\, s : S_i \cdot SCond \ and \ P2^{\sim}$$

$Post$	$Post^{\sim}$
$t.g = e.sqrt$	$e = t.g.sqr$
$t.g = e.exp$	$e = t.g.log$
$t.g = e.log$	$e = t.g.exp$
$t.g = e.pow(r)$	$e = t.g.pow(1.0/r)$
$t.g = e.reverse$	$e = t.g.reverse$
$t.g = e + K$	$e = t.g - K$
Numeric constant K	
$t.g = e - K$	$e = t.g + K$
Numeric constant K	
$t.g = e * K$	$e = t.g/K$
Numeric constant $K \neq 0$	
$t.g = e + SK$	$e = t.g.subString(1, t.g.size - SK.size)$
String constant SK	

Table 10. Inverse of basic assignments

This definition of $\tau^{\sim}$ satisfies the invertibility conditions of Section 2 provided that τ 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 in T are set by τ using lookup by primary keys: $t.f = TSub[e.id]$.

In addition, $SCond \ and \ Post^{\sim}$ should have a procedural interpretation.

Then, if there are computations

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

of τ and

$$(n, \varnothing) \longrightarrow_{\tau^{\sim}} (n, m')$$

of its inverse, then

$$m \equiv m'$$

because:

- For each entity S_i of S there is some constraint of *Cons* which maps it 1-1 to an entity T_j of T, and this is mapped 1-1 back to S_i by the inverse constraint of $\tau^{\sim}$. Thus, if $\overline{S_i}$ is the extent of S_i in m, there is a bijection $h_i : \overline{S_i} \to \overline{S_i}'$ between this and the corresponding extent of S_i in m'. The attribute values of $s : \overline{S_i}$ and of $h_i(s)$ are the same by the definition of $Post^{\sim}$ given above.
- The values of associations r of s and $h_i(s)$ have the same set of primary key values, by definition of $Post^{\sim}$. But any isomorphism of the entity *SSub* that is the target type of r must preserve identities also, and is characterised by equality of identities (if *ssub* and *ssub*$'$ have the same *SSub* primary key value, they correspond under the isomorphism), so the conditions of isomorphism hold also for associations.

The inverse of a sequential composition $\tau_1;\ \tau_2$ of transformations is the composition $\tau_2^{\sim};\ \tau_1^{\sim}$ of their inverses, provided that $\tau_2^{\sim}$ is isomorphism-preserving.

The inverse of $\tau^{\sim}$ is τ, provided that all constraints of *Cons* are determinate and each $Post^{\sim}$ is invertible.

A.3 Construction of change-propagation transformations

A change-propagation version of a transformation τ is a transformation τ^{Δ} which maintains the postcondition *Cons* of τ between a pair (m, n) of models which already satisfy it.

Cons must be maintained for changes to m of the following kinds:

1. Creation of a new element s of a source type S_i and the setting of its feature values.
2. Modification of a feature value of an existing source element.
3. Deletion of a source element $s : S_i$.

Only modifications which preserve *Asm* are considered.

For a constraint *Cn*:

$$\forall\, s : S_i \cdot SCond\ implies\ \exists\, t : T_j \cdot TCond\ and\ Post$$

Cn can itself be used to propagate creation and modification changes from m to n.

The dual version:

$$\forall\, t : T_j \cdot TCond\ and\ not(\exists\, s : S_i \cdot SCond\ and\ Post)\ implies\ t{\rightarrow}isDeleted()$$

is used to propagate modification and deletion changes from m to n.

The postcondition $Cons^{\Delta}$ of τ^{Δ} then consists of all the dual versions of constraints of *Cons*, in reversed order, followed by the *Cons* constraints.