A framework for model transformation verification

LANO, Kevin, CLARK, T. <http://orcid.org/0000-0003-3167-0739> and KOLAHDOUZ-RAHIMI, S.

A framework for verification of model transformations

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

Dept. of Informatics, King's College London; Dept. of Informatics, Middlesex University

Abstract. A model transformation verification task may involve a number of different transformations, from one or more of a wide range of different model transformation languages, each transformation may have a particular transformation style, and there are a number of different verification properties which can be verified for each language and style of transformation. Transformations may operate upon many different modelling languages. This diversity of languages and properties indicates the need for a suitably generic framework for model transformation verification, independent of particular model transformation languages, and able to provide support for systematic procedures for verification across a range of languages, and for a range of properties.

In this paper we describe the elements of such a framework, and apply this framework to a range of transformation verification problems. Our particular contributions are (i) language-independent verification techniques based around the concepts of transformation *invariants* and *variants*; (ii) language-independent representation metamodels for transformation specifications and implementations, (iii) mappings from these representations to the B and Z3 formalisms, (iv) the use of transformation patterns to facilitate verification.

The paper is novel in covering a wide range of different verification techniques for a wide range of MT languages, within an integrated framework.

Keywords: Model transformation verification; model transformation specification; model transformation engineering.

1 Introduction

Model transformation (MT) verification is a relatively new field, in which most work has so far been specific to particular model transformation languages, or to particular verification properties. Such approaches lead to problems in scenarios where systems consisting of multiple transformations, possibly defined using different languages, need to be verified. In addition, the reuse of verification techniques for different transformation languages is hindered by the language-specific nature of these techniques.

The development of model transformations has often been unsystematic, and focussed upon the implementation level, disregarding specifications. Common styles of transformation definition, utilising recursion and implicit operation calls, also hinder verification.

In this paper we define a general language-independent framework for transformation verification, and a range of language-independent verification techniques. The framework can be applied to different transformation languages, and can employ different verification technologies appropriate for establishing particular properties. The framework provides a systematic organisation for the process of transformation verification, and defines a uniform semantic basis for verification.

The elements of the framework are:

- Metamodels to represent modelling languages, transformation specifications and transformation implementations (Section 2).
- Transformation verification properties formalised in terms of this framework (Section 3).
- Language-independent verification techniques incorporated into the framework (Sections 4, 5, 7, 8).

In sections 9, 10 and 11 we illustrate the concepts using extracts from three verification case studies, of (i) a refinement transformation (code generation of Java from UML) using UML-RSDS [49]; (ii) a re-expression transformation using ETL [28]; (iii) a refactoring transformation (removal of attribute clones from a class diagram) using GrGen.NET [26]. Section 12 gives an evaluation and section 13 compares related work.

2 Metamodels for model transformations

In this section we define metamodels for modelling languages, transformation specifications and implementations, and show how languages, specifications and implementations for transformations can be given formal mathematical interpretations, in order to support transformation verification.

2.1 Representation of languages

Transformations manipulate models or texts which conform to some metamodel or syntax definition. There may be several input (source) models used by a transformation, and possibly several output (target) models. A transformation is termed *update-in-place* if a model is both an input and output, otherwise it is a *separate-models* transformation.

The key concepts for describing the effect of transformations at a high level are therefore *languages* and instances of languages (ie., models of the languages). Languages can be specified in many different ways, eg., by UML class diagrams, by BNF syntax definitions, etc. Here we will assume that UML class diagrams are used, together with OCL constraints. These form the *concrete syntax* of language descriptions. For example, Figure 1 shows a simple language with two entity types A and B, integer-valued attribute features of each, and a bidirectional association between them. There is an explicit constraint that $a.x > 0$ for each $a : A$, and an implicit constraint that the association roles ar and br are mutually inverse:

$$A{\rightarrow}\mathit{forAll}(a \mid a.x > 0)$$

and

$$A{\rightarrow}\mathit{forAll}(a \mid B{\rightarrow}\mathit{forAll}(b \mid a.br{\rightarrow}\mathit{includes}(b)\ \mathit{equiv}\ b.ar{\rightarrow}\mathit{includes}(a)))$$

For convenience, in the following we will use the notation $x : s$ to abbreviate the OCL expression $s{\rightarrow}\mathit{includes}(x)$.

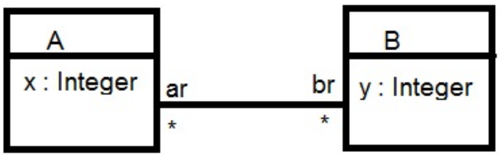

Fig. 1. Example language

In addition to such descriptions, we also need an *abstract syntax* for languages, ie., a metamodel for languages, and a mathematical equivalent of the languages, in order to define concepts of proof and satisfaction relative to a language.

Figure 2 shows a generic metamodel (termed *M4*) which can serve directly or indirectly as a metamodel for a wide range of modelling languages. The metamodel is also self-representative. Constraints on M4 are that (i) *general* is non-cyclic, (ii) $\mathit{mult1upper} = -1\ \mathit{or}\ \mathit{mult1lower} \leq \mathit{mult1upper}$ and likewise for $\mathit{mult2}$, (iii) $\mathit{type} : \mathit{PrimitiveDataType}\ \mathit{implies}\ \mathit{mult2upper} = 1\ \mathit{and}\ \mathit{mult2lower} = 1$ (no multi-valued attributes), (iv) names of entity types are unique, (v) names of data features owned by the same entity type are unique. Further restrictions on modelling languages could be specified, eg., no multiple inheritance. *composite* ends are interpreted as enforcing deletion propagation from objects of the composite to the contained objects. An association end is *mandatory* if it has lower bound ≥ 1.

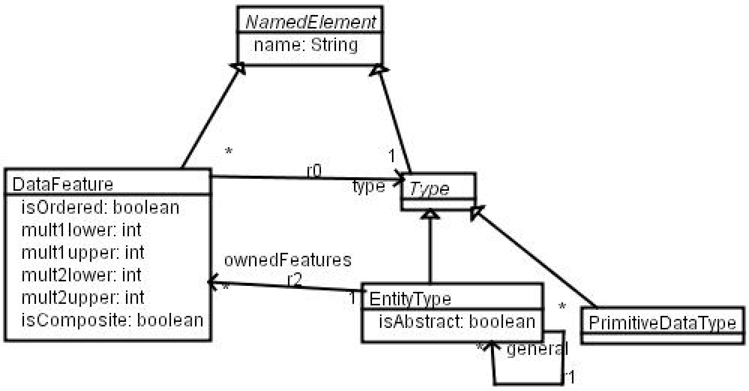

Fig. 2. Minimal metamodel (M4) for modelling languages

Our example language can be represented as elements

$A : EntityType \quad B : EntityType$
$x : DataFeature \quad y : DataFeature \quad ar : DataFeature \quad br : DataFeature$
$Integer : PrimitiveDataType$
$A.ownedFeatures = \{x, br\}$
$B.ownedFeatures = \{y, ar\}$
$x.type = Integer \quad y.type = Integer$
$ar.type = A \quad br.type = B$

of an instance of M4, together with instances $c1, c2$ of a constraint metatype to represent the constraints.

The metamodel for languages, based on M4, is shown in Figure 3.

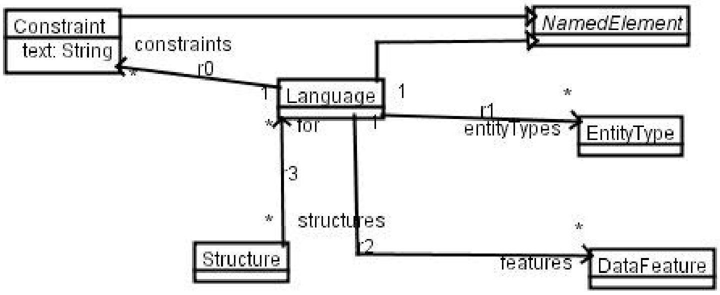

Fig. 3. Language metamodel

Many variations on constraint languages could be used, Figure 4 shows one possible metamodel for constraint expressions, which is used in UML-RSDS [49]. *BinaryExpression*, *BasicExpression*, *UnaryExpression* and *CollectionExpression* all inherit from *Expression*.

To support verification, we need to be able to define a proof theory and (logical) model theory wrt given languages, hence we need to associate a formal first order language and logic to each instance of M4. Table 1 defines how a formal first-order set theory (FOL) language $\mathcal{L}_L$ can be associated to instances L of M4. The table defines how different forms of elements in the M4 metamodel correspond to conceptual modelling elements, and how they are formalised in first-order set theory.

If $mult1upper = 1$ and $mult1lower = 1$ or $mult1lower = 0$ (ie, multiplicity 0..1 at the source end of the feature) for an attribute, it is an identity (unique) attribute, ie., a primary key in relational

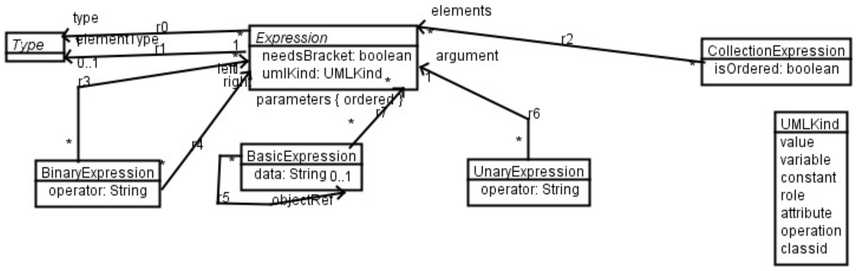

Fig. 4. Example constraint language

Modelling concept	*M4 representation*	*Formal semantics*
Entity type E	Element of *EntityType*	Type symbol E, the set of instances of E
Concrete entity type E	Element of *EntityType* with $isAbstract = false$	
Abstract entity type E	Element of *EntityType* with $isAbstract = true$	$E = E1 \cup ... \cup El$ where $E1$, ..., El are all direct subtypes of E
Primitive type T	Element of *PrimitiveDataType*	Mathematical type, $\mathbb{Z}$, $\mathbb{B}$, $\mathbb{R}$, $\mathbb{S}$, etc.
Single-valued attribute $att : Typ$ of E	*DataFeature* element with $mult2upper = 1$, $mult2lower = 1$ and $type \in PrimitiveDataType$	Function symbol $att : E \rightarrow Typ'$ Typ' represents Typ
Single-valued role r of E with target entity type $E1$	*DataFeature* element with $mult2upper = 1$, $mult2lower = 1$ and $type \in EntityType$	Function symbol $r : E \rightarrow E1$
Unordered many-valued role r of E with target entity type $E1$	*DataFeature* element with $mult2upper \neq 1$ or $mult2lower \neq 1$ and $isOrdered = false$ and $type \in EntityType$	Function symbol $r : E \rightarrow \mathbb{F}(E1)$
Ordered many-valued role r of E with target entity type $E1$	*DataFeature* element with $mult2upper \neq 1$ or $mult2lower \neq 1$ and $isOrdered = true$ and $type \in EntityType$	Function symbol $r : E \rightarrow \mathrm{seq}(E1)$
Supertype $E1$ of E	Element of $E.general$	Type $E1$ with $E \subseteq E1$

Table 1. Correspondence of M4 and first-order logics

data terminology. A -1 multiplicity value represents $*$ multiplicity. An expression is either single-valued (denoting a single value of a primitive data type, or a single instance of an entity type), or many-valued, denoting a collection. Collections according to this metamodel are either ordered (sequence-valued) or unordered (set-valued). Bags and ordered sets can be defined in terms of sequences. Notice that an optional association end (with $mult2lower = 0$ and $mult2upper = 1$) is formally represented as a set (or sequence) of size 0 or 1. Thus null elements are in this case represented as empty collections, as in the OCL standard [44]. A denumerable type $Object\_OBJ$ is included in each $\mathcal{L}_L$ to represent the set of all possible object references[1]. A finite set $objects \subseteq Object\_OBJ$ represents the set of all existing objects at any point in time. Each entity type E has $E \subseteq objects$. Additional axioms, eg., that distinct entity types unrelated by subtyping have disjoint extents, could be assumed if required.

For simplicity, we will not consider the cases of OCL *invalid* or *null* expression values in this paper, and we will operate entirely within two-valued logic both for OCL and its logical representations. As argued in [29], this is a pragmatic choice which greatly simplifies analysis of specifications, and is adequate in most cases. Transformation specifiers should separately ensure that invalid expression evaluations cannot occur in their transformations, using the definedness conditions of expressions (Section 5). Instead of defining operators such as select or collect by iterators, we will give these a conventional mathematical definition as set-comprehension expressions. Ultimately all the standard numeric, string and collection function symbols and predicate symbols can be derived from $=$ and set membership ($:$ or $\in$).

We use some minor extensions of OCL. For entity types E with one attribute id designated as an identity attribute (primary key), the abbreviation $E[v]$ is introduced to represent $E.allInstances()\rightarrow select(id = v)\rightarrow any()$, if v is single-valued, and to represent $E.allInstances()\rightarrow select(id : v)$ if v is collection-valued. $E.id$ denotes $E.allInstances()\rightarrow collect(id)$. The finite range of numbers from n to m inclusive is denoted $Integer.subrange(n, m)$. The operator $v\rightarrow subcollections()$ gives the set of finite subsets of a set v. The operator $v\rightarrow isDeleted()$ expresses that the object or object collection v are removed from the model, ie., that $objects\rightarrow excludes(v)$ for an object v, it is inverse to *oclIsNew* [44]. Operators $s\rightarrow intersectAll(e)$ and $s\rightarrow unionAll(e)$ form distributed intersections and unions, and transitive closure of a many-valued self-association role end $r : E \rightarrow Set(E)$ is defined[2] as $x.r\rightarrow closure() = Integer.subrange(0, E.size)\rightarrow unionAll(n \mid x.r^n)$.

Constraints in this version of OCL are mathematically represented as first-order set theory axioms in $\mathcal{L}_L$, for each language L. Tables 2 and 3 show some examples of semantic interpretations of OCL logical operators, collection types and operators. In total, we have defined a mathematical semantics for 33 OCL operators on non-collection types, and for 36 OCL collection type operators [49].

Here we adopt the notation for set theory used by the B AMN formalism [31]. OCL sets are represented as mathematical sets, OCL sequences s are represented as maps from $1..s.size$ to their set of elements. Strings are also represented as sequences, of integers representing characters, or as a primitive type.

$E[e/x]$ denotes the substitution of e for free occurrences of x in E (avoiding free variable capture). $sq\uparrow i$ for sequence sq is the initial subsequence of sq with domain $1..i$, $sq\downarrow i$ is the subsequence of the elements with index $> i$.

For our example language of Figure 1 we have a corresponding formal first-order set theory language $\mathcal{L}_L$, which has sort symbols (A, B), function symbols (x, y, ar, br), and axioms

$$\forall\, a : A \cdot x(a) > 0$$
$$\forall\, a : A;\ b : B \cdot b \in br(a) \ \equiv\ a \in ar(b)$$

Therefore we can operate simultaneously with three corresponding views of a language: (i) as a set of entity types and their data features and specialisation relations, defined by a class diagram, with OCL constraints defining restrictions on these elements; (ii) as instances of M4, together with abstract syntax representations of the constraints; (iii) as mathematical languages in first-order set

[1] As in [11], we could include a *nullid* reference in this type to represent OCL *null*.

[2] The undefinability of finite transitive closure in pure predicate calculus [5] does not apply to set theory. $x.r^0 = \{x\}$, $x.r^1 = x.r$, $x.r^2 = x.r.r$, etc.

OCL Term e	*Condition*	*Semantics* e'
P *and* Q		$P' \wedge Q'$
P *or* Q		$P' \vee Q'$
P *implies* Q		$P' \Rightarrow Q'$
$e \rightarrow forAll(x \mid P)$		$\forall\, x : e' \cdot P'$
$e \rightarrow exists(x \mid P)$		$\exists\, x : e' \cdot P'$
$e \rightarrow exists1(x \mid P)$		$\exists_1\, x : e' \cdot P'$
$Integer.subrange(a, b)$	$a, b : Integer$	$a'..b'$
$Set\{e1, ..., en\}$		$\{e1', ..., en'\}$
$Sequence\{e1, ..., en\}$		$[e1', ..., en']$
$s \rightarrow size()$	set or sequence s	cardinality $card(s')$
$s \rightarrow subcollections()$	set s	$\mathbb{F}(s')$
$s \rightarrow at(i)$	sequence s	$s'(i')$
$s \rightarrow includes(x)$	set s	$x' \in s'$
$s \rightarrow excludes(x)$	set s	$x' \notin s'$
$s \rightarrow intersection(t)$	sets s, t	$s' \cap t'$
$s \rightarrow union(t)$	sets s, t	$s' \cup t'$
$s \rightarrow includes(x)$	sequence s	$x' \in \mathrm{ran}(s')$
$s \rightarrow excludes(x)$	sequence s	$x' \notin \mathrm{ran}(s')$
$s \rightarrow including(x)$	set s	$s' \cup \{x'\}$
$s \rightarrow excluding(x)$	set s	$s' - \{x'\}$
$s \rightarrow asSet()$	s set	s'
$s \rightarrow asSet()$	s sequence	$\mathrm{ran}(s')$
$s \rightarrow includesAll(t)$	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, $card(s') = n$	$s'(1) + ... + s'(n)$
$s \rightarrow isUnique(x \mid e)$	set s	$card(s') = card(\{e'[v/x] \mid v \in s'\})$

Table 2. Semantic mapping for OCL logic and collection expressions

OCL Term e	*Condition*	*Semantics* e'
$objs \rightarrow select(x \mid P)$	set $objs$	$\{v \mid v \in objs' \wedge P'[v/x]\}$
$objs \rightarrow reject(x \mid P)$	set $objs$	$\{v \mid v \in objs' \wedge \neg\, P'[v/x]\}$
$objs \rightarrow collect(x \mid e)$	set $objs$	bag $\{e'[v/x] \mid v \in objs'\}$
$objs \rightarrow collect(x \mid e)$	sequence $objs$	$\{i \mapsto e'[objs'(i)/x] \mid i \in \mathrm{dom}(objs')\}$
$objs \rightarrow intersectAll(x \mid e)$	set-valued e	$\bigcap(objs \rightarrow collect(x \mid e))'$
$objs \rightarrow unionAll(x \mid e)$	set-valued e	$\bigcup(objs \rightarrow collect(x \mid e))'$
$s \rightarrow subSequence(i, j)$	sequence s	$(s' \uparrow j') \downarrow (i' - 1)$
$s \rightarrow insertAt(i, x)$	sequence s	$(s' \uparrow (i' - 1)) \frown [x'] \frown (s' \downarrow (i' - 1))$
$s \rightarrow count(x)$	sequence s	$card(s'^{\sim} (\!\mid \{x'\} \mid\!))$
$s \rightarrow indexOf(x)$	sequence s	$min(s'^{\sim} (\!\mid \{x'\} \mid\!))$

Table 3. Semantic mapping for selection expressions

theory, together with axioms in these languages representing the mathematical semantics of the constraints.

Assuming the use of an OCL-like language as a constraint language, the sets of syntactically valid expressions $\mathit{Exp}(L)$ and sentences $\mathit{Sen}(L)$ in the constraint language based upon a language L can be determined. $\mathit{Exp}(L)$ in this paper denotes the set of all expressions in Figure 4 over L. It can also be regarded as a set of expressions and formulae of $\mathcal{L}_L$, via the above semantic interpretation of expressions[3]. The terms occurring in elements of $\mathit{Exp}(L)$ and $\mathit{Sen}(L)$ are always based on finite collections, ie., for each expression $e{\rightarrow}\mathit{select}(x \mid P)$, $e{\rightarrow}\mathit{forAll}(x \mid P)$, etc., e is finite.

We will denote a language L by a metamodel Σ_L which is an instance of $M4$, and a set of (OCL or FOL) sentences Γ_L over Σ_L defining restrictions on this metamodel: $L = (\Sigma_L, \Gamma_L)$, where $\Gamma_L \subseteq \mathit{Sen}(L)$.

For convenience, we write a language metamodel Σ_L textually as a *signature*: a definition of a tuple $(E_1, ..., E_k)$ of entity types, and of a tuple $(f_1, ..., f_l)$ of data features (attributes and association role ends) on these entity types, together with language constraints on these elements, which are included in Γ_L. We usually assume the inclusion of the standard primitive types *Integer*, *Boolean*, *String* and *Real* in each language, so these will not be listed in Σ_L.

The metaclass *Structure* in Figure 3 represents the concept of a structure or interpretation of a language: such structures will have concrete representations as sets and maps for each entity type and feature of their language. Model transformations will operate on such structures, modifying them in-place or producing new structures from their data.

Structures for a language L can be represented as tuples $m = ((E_1^m, ..., E_k^m), (f_1^m, ..., f_l^m))$ which give interpretations of each element of Σ_L:

1. For each entity type E_i of L, E_i^m is a finite set (of atomic, unspecified, object identities) representing the *extent* $E_i.\mathit{allInstances}()$ of E_i. For abstract E_i, E_i^m is the union of the F^m extents of the direct subtypes F of E_i.
2. For an attribute $f_j : \mathit{Typ}$ of E_i, f_j^m is a map of type $E_i^m \rightarrow \mathit{Typ}'$ from its source entity type interpretation to a domain of values for its target type.
3. A single-valued (1-multiplicity) association end $f_j : F$ of E_i is interpreted by a map $f_j^m : E_i^m \rightarrow F^m$.
4. An unordered many-valued (ie., not 1-multiplicity) association end $f_j : \mathit{Set}(F)$ of E_i is interpreted by a map $f_j^m : E_i^m \rightarrow \mathbb{F}(F^m)$.
5. An ordered many-valued association end $f_j : \mathit{Sequence}(F)$ of E_i is interpreted by a map $f_j^m : E_i^m \rightarrow \mathrm{seq}(F^m)$.
6. If E is a subtype of F then $E^m \subseteq F^m$.

There is a notion of structure *isomorphism* (Appendix A): $m \simeq n$ means that the structures are semantically indistinguishable by sentences of L.

A structure m of L can be considered to be an instance of Σ_L, so we write $m : L$ to say that m is a structure for L. $\mathit{Mod}(L)$ denotes the set of structures for L. This can be considered also as the set of (logical) interpretation structures for the mathematical language $\mathcal{L}_L$.

A structure m of L is termed a (semantic) *model* of L if it satisfies all axioms in Γ_L (ie., all the implicit and explicit language constraints of L):

$$\forall \varphi \cdot \; \varphi \in \Gamma_L \;\Rightarrow\; m \models \varphi$$

where $\models \;\subseteq\; \mathit{Mod}(L) \times \mathbb{F}(\mathit{Sen}(L))$ is the satisfaction relation of L (equivalently of $\mathcal{L}_L$). If $L = S \cup T$ for disjoint languages S and T, satisfaction over pairs (m, n) of structures m of S and n of T is defined based on interpreting S language elements in m and T language elements in n: $(m, n) \models \varphi$ for $\varphi \in \mathit{Sen}(L)$. Likewise for tuples of structures and languages.

A deduction relation $\vdash \;\subseteq\; \mathbb{F}(\mathit{Sen}(L)) \times \mathit{Sen}(L)$ is given by the usual deduction rules for first-order logic with equality for $\mathcal{L}_L$. This is related to $\models$ by the property of *soundness*:

$$\begin{aligned}&\Delta \vdash \varphi \;\;\Rightarrow\\ &\qquad\qquad (\forall\, m : \mathit{Mod}(L) \cdot m \models \Delta \;\Rightarrow\; m \models \varphi)\end{aligned}$$

[3] In a first-order language *terms*, built from function symbols, +, $\rightarrow$*select*, etc., are distinguished from *formulae* or *predicates*, which have an outermost operator as a logical operator or predicate symbol, $\in$, $=$, $<$, etc. A sentence is a formula without free variables.

for $\Delta \subseteq Sen(L)$, $\varphi \in Sen(L)$.

The converse relationship of *completeness* will usually hold:

$$(\forall\, m : Mod(L) \cdot m \models \Delta \;\Rightarrow\; m \models \varphi) \quad\Rightarrow\quad \Delta \vdash \varphi$$

for finite $\Delta \subseteq Sen(L)$, $\varphi \in Sen(L)$.

We discuss this issue in Appendix B.

2.2 Representation of transformations

At the specification level, the effect of a transformation can be characterised by a collection of mapping specifications, which relate model elements of one or more models involved in the transformation to each other [25]. These mapping specifications define the intended relationships which the transformation should establish between the input (source) and output (target) structures of the transformation, at its termination. That is, they define the postconditions *Post* of the transformation.

In the case of in-place transformations, the initial values of entity types and features can be notated as E@pre, f@pre in postconditions to distinguish them from their post-state values.

For example, an in-place transformation τ on the language L of Figure 1 could be specified by a mapping specification constraint R on a single (input and output) model $m : L$. R specifies that there are B objects corresponding to each initial A object:

$$A@pre \rightarrow forAll(a \mid B \rightarrow exists(b \mid b.y = a.x@pre * a.x@pre \;\; and \;\; a.br \rightarrow includes(b)))$$

This is a postcondition of the transformation: a predicate which should hold at its termination. Since neither A or x are updated by τ (ie., they are not in the write frame $wr(R)$ of R, Section 5), the @pre suffix can be omitted, and R written more simply as:

$$A \rightarrow forAll(a \mid B \rightarrow exists(b \mid b.y = a.x * a.x \;\; and \;\; a.br \rightarrow includes(b)))$$

A precondition could also be expressed, eg., that B has initially no instances:

$$B = Set\{\}$$

We adapt the mapping metamodel of [25] to represent such transformation specifications (Figure 5).

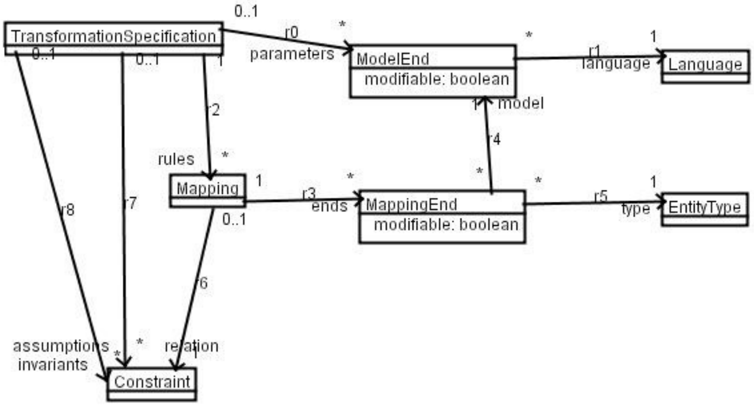

Fig. 5. Transformation specification metamodel

In this figure, *Mapping*, *TransformationSpecification*, *ModelEnd*, and *MappingEnd* are subclasses of *NamedElement*. Transformation specifications in declarative transformation languages can be expressed in this metamodel, using abstraction techniques such as those defined for TGG (triple graph grammars) and QVT-R in [14] and for ATL in [13]: these techniques express the intended effect of transformations by a metamodel plus OCL constraints describing the poststate of the transformation. Transformations in hybrid and imperative languages should also be given pre and postcondition specifications, to provide a basis for their verification. In turn formal representations of transformation specifications can be generated from representations in this metamodel, in a range of formalisms such as B [36], Z3 [51] or Alloy [2], to support semantic analysis. The metamodel could be extended to include generalisation relations between mappings, as in ETL [28].

The *rules.relation* constraints express the postconditions *Post* of the transformation: all of these constraints should be true at termination of the transformation. Typically each mapping relation constraint $Cn \in Post$ has the form of an implication

$$SCond \ implies \ Succ$$

forall-quantified over elements (the source mapping ends $s : S_i$) of the source models. The *application conditions* $ACond$ of the mapping are then $SCond$ *and* $not(Succ)$, ie., the mapping is applicable when its assumptions are true and when it is not already established. Analysis of the *determinacy* and *definedness* of the mappings can be carried out by syntactic analysis of their *relation* constraints, as described in Section 5.

The *assumptions* express the preconditions *Asm* of the transformation. The *invariants* define properties *Inv* which should be true initially, and which should be preserved by each *computation step* of the transformation: they serve to restrict the possible implementations of the transformation to those which do maintain *Inv*. The postconditions, preconditions and invariants can be expressed in the disjoint union of the languages *parameters.language* involved in the transformation, with pre-state versions of language elements also being used in the case of parameters which are both inputs and outputs.

A transformation τ preserves structures $p \in \tau.parameters$ which have $p.modifiable = false$, otherwise τ may change the data of an actual structure supplied as the value for p. If a mapping end is modifiable, so is its structure:

$$modifiable = true \ implies \ model.modifiable = true$$

is an invariant of *MappingEnd*. The entity type of a mapping end must also belong to the language of the model end of the mapping end:

$$type : model.language.entityTypes$$

is an invariant of *MappingEnd*. Target models have $modifiable = true$, source models are only modifiable if they are also target models.

Systems of transformations can be represented by UML activity diagrams, in which the transformations are executable activity nodes and the structures or models they operate on are object (data) nodes. Eg., Figure 6 shows a sequential composition of two transformations. An object flow from τ to $n : T$ indicates that n is a modifiable parameter of τ.

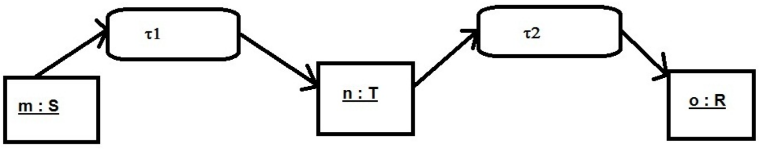

Fig. 6. Sequential composition of transformations

Transformation implementations are defined by a *behavior*, such as a UML *Activity*, in which *RuleImplementation* instances are the executable activity nodes. Each rule implementation is for a specific mapping, and itself has an activity or other behaviour defining its actions (Figure 7). A key concept for both graph-transformation and model-transformation languages is the idea of a *computation step* or *transformation step*: the application of a specific rewrite rule or transformation rule to a specific matching location in a graph or to specific matching element(s) in a model. This is also modelled as a behaviour in Figure 7: each rule implementation will usually be based upon some iteration of the step for the mapping that it implements.

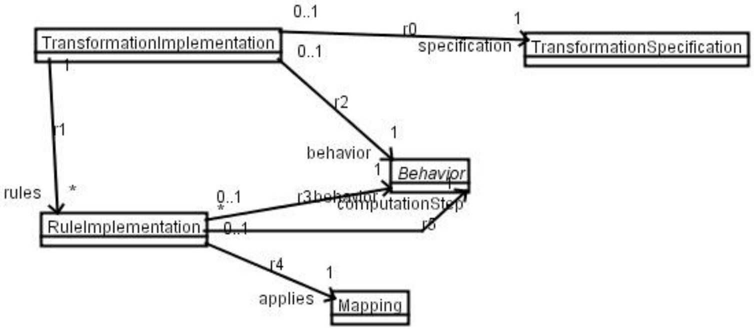

Fig. 7. Transformation implementation metamodel

This metamodel can be used to represent rules in hybrid or imperative languages, eg., GrGen [26], ATL [24], ETL [25] or Kermeta [17]. Instead of creating a metamodel for each different transformation language (ETL, ATL, GrGen, etc), we use the common implementation metamodel to express and reason about implementation-level properties, independently of languages. Mappings need to be defined between these languages and a suitable general behaviour representation, we consider that this is feasible because model transformation languages have many common aspects, such as lookup mechanisms using traces. We illustrate how such mappings can be defined in Sections 10 and 11.

The behavior of an implementation I of a transformation τ will determine the order in which the $\delta \in \mathit{RuleImplementation}$ will be executed. Each δ in turn has an internal behaviour, usually defined in terms of computation steps $\delta_r(ss)$ which attempt to establish the specification mapping $r = \delta.\mathit{applies}$ for particular elements ss in the source domains of the mapping. Thus $\delta.\mathit{computationStep} = \delta_r$.

For example, the constraint R would have individual computation steps $\delta_R(a)$ for $a \in A$, each step creates a new $b \in B$ and sets $b.y = a.x * a.x$ and adds b to $a.br$ (and implicitly adds a to $b.ar$), so establishing the quantified formula of R for a.

Such constraint applications $\delta_r(ss)$ are the computation steps both of the implementations δ of a specific $Cn \in \mathit{Post}$ and of the overall implementation I of a transformation τ. A *partial computation* of I for τ is a finite sequence sq of computation steps of I, such that sq has a form permitted by the behaviour of I, and such that Inv holds in the initial structures $(m_1, ..., m_q)$ supplied as input parameters of τ, and Inv holds in the final state of sq and after every initial subsequence of sq.

A *completed computation* of I from initial structures $(m_1, ..., m_q)$ supplied as the parameter values of τ, to terminal structure values $(n_1, ..., n_q)$ for these parameters, will be denoted

$$(m_1, ..., m_q) \longrightarrow_{\tau,I} (n_1, ..., n_q)$$

If parameter i is non-modifiable, $n_i = m_i$.

A completed computation from $(m_1, ..., m_q)$ to $(n_1, ..., n_q)$ for $\tau \in \mathit{TransformationSpecification}$ where $I.\mathit{specification} = \tau$, consists of a finite sequence sq of computation steps of I, such that sq has a form permitted by the implementation algorithm of I, Inv is true initially and at all intermediate states (ie., after any initial subsequence of sq), and where no further steps are permitted by I from $(n_1, ..., n_q)$, and such that no shorter initial subsequence of sq has this property.

For our example transformation, a completed computation according to the standard bounded loop implementation of R will be a sequence $[\delta_R(a1), ..., \delta_R(ap)]$ consisting of all the computation steps for the distinct $ai \in A$ in the source model, each ai processed exactly once, in an arbitrary order. A partial computation would be any initial subsequence of such a sequence.

For existing transformations, the computation steps and rule implementations may be defined using any means available in the transformation language (as in case studies 2 and 3 below). Alternatively, for new transformations, the steps, rule implementations and the overall implementation can be derived systematically from the transformation specification (Section 8).

The notation

$$(m_1, ..., m_q) \longrightarrow_\tau (n_1, ..., n_q)$$

means that there is a completed computation $(m_1, ..., m_q) \longrightarrow_{\tau,I} (n_1, ..., n_q)$ for some implementation I of τ.

We use the following program-like activity language for the language-independent behaviour description of hybrid and imperative transformations.

The concrete syntax BNF of the language is as follows:

$$\begin{array}{ll}
< \mathit{statement} > & ::= < \mathit{loop\_statement} > \mid < \mathit{creation\_statement} > \mid \\
& \quad < \mathit{conditional\_statement} > \mid < \mathit{sequence\_statement} > \mid \\
& \quad < \mathit{basic\_statement} > \\
< \mathit{loop\_statement} > & ::= \text{“while”} < \mathit{expression} > \text{“do”} < \mathit{statement} > \\
& \quad \text{“invariant”} < \mathit{expression} > \text{“variant”} < \mathit{expression} > \mid \\
& \quad \text{“for”} < \mathit{expression} > \text{“do”} < \mathit{statement} > \\
< \mathit{conditional\_statement} > & ::= \text{“if”} < \mathit{expression} > \text{“then”} < \mathit{statement} > \\
& \quad \text{“else”} < \mathit{basic\_statement} > \\
< \mathit{sequence\_statement} > & ::= < \mathit{statement} > \text{“;”} < \mathit{statement} > \\
< \mathit{creation\_statement} > & ::= < \mathit{identifier} > \text{“:”} < \mathit{type\ name} > \\
< \mathit{basic\_statement} > & ::= < \mathit{basic\_expression} > \text{“:=”} < \mathit{expression} > \mid \text{“skip”} \mid \\
& \quad \text{“return”} < \mathit{expression} > \mid \text{“(”} < \mathit{statement} > \text{“)”} \mid \\
& \quad < \mathit{call\_expression} >
\end{array}$$

This defines a subtype *Statement*(L) of *Behavior*, when based on the expressions *Exp*(L) of language L. The semantics of these statements is given by the weakest-precondition operator $[\ ] : \mathit{Statement}(L) \times \mathit{Exp}(L) \rightarrow \mathit{Exp}(L)$, using standard definitions, as for the B Generalised Substitution language [31]. For example (using standard mathematical notation for expressions on the RHS):

$$\begin{array}{l}
[\mathit{obj}.f \ := \ e]P \ \equiv \ P[(f \oplus \{\mathit{obj} \mapsto e\})/f] \ \ \mathit{for\ single\ object\ obj},\ \mathit{writable\ feature\ f} \\
[\mathit{objs}.f \ := \ e]P \ \equiv \ P[(f \oplus (\mathit{objs} \times \{e\}))/f] \ \ \mathit{for\ set\ objs}, \mathit{writable\ f} \\
[x \ := \ e]P \ \equiv \ P[e/x] \ \ \mathit{for\ variable\ or\ entity\ type}\ x \\
[(x \ : \ E;\ S)]P \ \equiv \\
\qquad \forall x \cdot x : \mathit{Object\_OBJ} - \mathit{objects} \ \Rightarrow \ [E := E \cup \{x\}; \\
\qquad\qquad\qquad \mathit{objects} := \mathit{objects} \cup \{x\};\ \ S]P \ \ \mathit{variable}\ x,\ \mathit{concrete\ entity\ type}\ E \\
[\mathit{if}\ E\ \mathit{then}\ S1\ \mathit{else}\ S2]P \ \equiv \ (E \Rightarrow [S1]P) \wedge (\neg\, E \Rightarrow [S2]P) \\
[S1;\ S2]P \ \equiv \ [S1]([S2]P) \\
[\mathit{for}\ x : e\ \mathit{do}\ S(x)]P \ \equiv \ [S(e1)]...[S(en)]P\ \mathit{if}\ e = \mathit{Sequence}\{e1, ..., en\} \\
[\mathit{for}\ x : e\ \mathit{do}\ S(x)]P \ \equiv \ \bigwedge_{sq \in serial(e)}[\mathit{for}\ x : sq\ \mathit{do}\ S(x)]P\ \mathit{otherwise}
\end{array}$$

Object_OBJ is a denumerable type of all possible object references, *objects* maintains the set of object references of all existing objects. In the clause for creation of $x : E$, updates $F \ := \ F \cup \{x\}$ for each supertype F of E will also be included.

In the last clause, a conjunction is taken over all the possible serialisations of e at the start of the loop:

$$\mathit{serial}(s) = \{sq : 1..s.\mathit{size} \rightarrow s \mid \mathit{ran}(sq) = s\}$$

for a set s.

In practice, a bounded loop of this kind can be analysed by showing that the individual $S(x)$ are order-independent in their execution for distinct x, and hence only one serialisation needs to be analysed, since all are semantically equivalent (Section 8).

For unbounded loops, we use the inference:

$$\begin{array}{l}(I \wedge E \;\Rightarrow\; [S]I) \;\wedge \\ (I \wedge \neg\, E \;\Rightarrow\; P) \;\wedge \\ (I \wedge E \;\Rightarrow\; v \in \mathbb{N}) \;\wedge \\ (\forall\, \gamma : \mathbb{N} \cdot I \wedge E \wedge v = \gamma \;\Rightarrow\; [S](v < \gamma)) \;\Rightarrow \\ \qquad\qquad [\mathit{while}\ E\ \mathit{do}\ S\ \mathit{invariant}\ I\ \mathit{variant}\ v]P\end{array}$$

For operation calls, *call-by-value-result* semantics is used.

In the remainder of the paper we will consider transformations which are either separate-models transformations with one source and one target: $\tau : S \rightarrow T$, or update-in-place transformations on a single model: $\tau : S \rightarrow S$. The verification techniques we define can be used in the same way for transformations with multiple input and output languages.

For separate-models transformations, in order to relate properties of a source model to those expressible in a target model, we use the concept of a *language morphism* or interpretation. This is a mapping $\chi : S \rightarrow T$ from the source language S to a target language T, consisting of a signature morphism $\chi : \Sigma_S \rightarrow \mathit{Exp}(T)$ of entity types of S to set-valued expressions of T, and features of S to features or expressions denoting maps of the same arity and of corresponding types of T, and induced morphisms $\mathit{Sen}(\chi) : \mathit{Sen}(S) \rightarrow \mathit{Sen}(T)$ and $\mathit{Mod}(\chi) : \mathit{Mod}(T) \rightarrow \mathit{Mod}(S)$ which use χ to interpret sentences of S as sentences of T, and to interpret structures of T as structures for S. $\mathit{Sen}(\chi)(\varphi)$ is written as $\chi(\varphi)$ in the following. There is a category $\mathcal{LANG}$ of languages and language morphisms, and a category $\mathcal{PLANG}$ of languages and partial language morphisms (where $\chi : \Sigma_S \nrightarrow \mathit{Exp}(T)$).

3 Transformation verification properties

A large number of verification properties have been proposed for model transformations, eg., [14, 36]. In this section we formalise some key properties using the framework of Section 2, and in the following sections we identify techniques and technologies to establish these properties.

For separate-models transformations τ with one modifiable target parameter $n : T$ and one preserved source $m : S$, completed computations of τ will be of the form

$$(m, n0) \longrightarrow_{\tau} (m, n)$$

where $n0$ is a default initial model which is usually the empty T structure $\varnothing$[4]. We say that n can be produced from $(m, n0)$ by τ if there is such a completed computation.

Correctness properties can either be considered for one specific implementation I of τ, or for all possible implementations which maintain the invariants Inv.

Syntactic correctness of τ can be formalised as:

$$(m, n0) \models \mathit{Asm} \cup \Gamma_S \;\Rightarrow\; n \models \Gamma_T$$

for each structure m of S, where $n : T$ can be produced from $(m, n0)$ by (any implementation of) τ, and Asm are the assumptions of τ. Ie., if m is a model of S, any structure n produced by τ from m should be a model of T. If restricted to a specific implementation I, syntactic correctness means that structures produced from models of S by I should be models of T.

τ is said to be *semantically preserving* relative to a language interpretation $\chi : S \rightarrow T$ [36] if: $m \models \varphi \;\Rightarrow\; n \models \chi(\varphi)$ for $n : T$ produced from $m : S$, $n0 : T$ by (any implementation of) τ, and for $\varphi \in \mathit{Sen}(S)$. Preservation may only be required for properties in a subset Pres of $\mathit{Sen}(S)$, and/or for $(m, n0) \models \mathit{Asm}$. Semantic preservation by a specific implementation I of τ is formulated similarly.

τ is a *semantic equivalence* if $m \models \varphi \;\equiv\; n \models \chi(\varphi)$ for $n \in \mathit{Mod}(T)$ produced from $m \in \mathit{Mod}(S)$, $n0$ by τ, and for $\varphi \in \mathit{Sen}(S)$.[5]

[4] The structure where each entity type of T is interpreted by the empty set.

[5] The morphism χ will be an *institution morphism* [21] if τ has $\mathit{Mod}(\chi)$ as a right inverse, and τ is a semantic equivalence.

Semantic correctness of an implementation I of τ means that the implementation establishes the specified postconditions $Post$ of τ: $(m, n0) \models Asm \;\Rightarrow\; (m, n) \models Post$ for $n : T$ produced from $m : S$ and $n0 : T$ by I.

Model-level semantic preservation means that the internal semantics of source models is preserved, possibly under some interpretation ζ, by τ. Let $sem_L : Mod(L) \rightarrow Sem(L)$ be the semantics-assigning functions for models of languages $L = S$, $L = T$, and $\zeta : Sem(S) \rightarrow Sem(T)$, then model-level semantics preservation means that:

$$\zeta(sem_S(m)) \approx sem_T(n)$$

for some relation $\approx$ of equivalence on the semantic domain, and where $(m, n0) \models Asm$ and $(m, n0) \longrightarrow_{\tau} (m, n)$ or $(m, n0) \longrightarrow_{\tau,I} (m, n)$. This can be alternatively expressed as a commuting-diagram property [39].

In many cases the semantics can be formalised within $\mathcal{L}_S$ and $\mathcal{L}_T$, so that model-level semantic preservation can be reduced to language-level semantic preservation relative to a suitable χ interpretation, or to invariant preservation of a formula expressing the commuting diagram property. For example, if the models are state machines, their sets of input event traces can be formalised in FOL as sets of sequences and preservation of such sets by τ can be expressed as preservation of a suitable invariant formula φ. Section 11 gives an example of such reasoning.

A transformation τ is *confluent*, if for every $m : S$, and n, n' which can be produced by a completed computation of τ from $(m, n0)$, $n \simeq n'$. Likewise for specific implementations I of τ.

τ (or an implementation I of τ) is *terminating* if for each model m of S, $(m, n0) \models Asm$, every partial computation of τ (I) from $(m, n0)$ has a completed computation extending it.

Similar formalisations can be given for update-in-place transformations. A transformation τ operating on a single model of language S can be considered to have computations

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

where we implicitly retain the initial model $m : S$ in order to express the effect of the transformation by predicates relating the initial values of entity type extents and features (denoted by E@pre and f@pre in the transformation mapping relations) to their final values (denoted by E and f).

Syntactic correctness of such τ can be formalised as:

$$m \models Asm \cup \Gamma_S \;\Rightarrow\; n \models \Gamma_S$$

for each structure m of S, where $n : S$ can be produced from m by τ, and Asm are the assumptions of τ.

Semantic correctness of an implementation I of τ means that I establishes the specified postconditions $Post$ of τ: $m \models Asm \;\Rightarrow\; (m, n) \models Post$ for $n : S$ produced from $m : S$ by I, where pre-state terms in $Post$ are evaluated in m.

The definitions of model-level semantic preservation, termination and confluence are as for the separate-models case.

4 Transformation verification techniques

Four main distinct approaches to model transformation verification have been used or proposed:

1. Static analysis by syntactic analysis of the transformation source text, eg., to identify data-dependency relations between variables or rules [36].
2. Constructing counter-examples to properties, using model checkers or satisfaction checkers [2, 14, 12, 13].
3. Proving properties using automated or interactive theorem-provers [36, 20].
4. Correct-by-construction synthesis of transformations from specifications [45, 38].

Both 2 and 3 may involve mapping the transformation text to a formalism which supports the semantic analysis: the semantic model is termed a *verification model* or *transformation model* [36]. The semantic mapping should itself be a semantics-preserving or semantic equivalence transformation, in the sense of institution co-morphisms [41].

Our verification framework is based upon using the process of Figure 8 to express a range of different transformation languages in the transformation specification and implementation metamodels of Section 2 and then mapping these representations to verification formalisms. This approach means that only one semantic mapping needs to be defined and verified for each target formalism, rather than semantic maps for each different transformation language and target formalism.

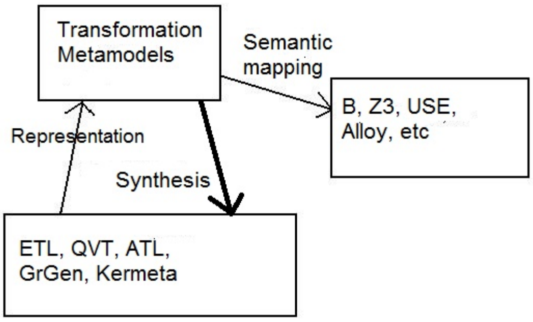

Fig. 8. Verification process

We can distinguish between single-state formalisations, which only define a formal semantic model of one state of a transformation, usually a terminal state [13], and multi-state approaches which formalise the possible sequences of execution of the transformation [20]. The former can be applied to analyse syntactic correctness and other end-state properties, whilst the latter can be used to prove invariance of properties, termination and confluence.

Syntactic analysis is usually the least resource-expensive of the approaches, but may not be able to establish all properties. Counter-example analysis is a specification-based version of testing, and can detect flaws but not establish properties for all cases. Proof approaches in contrast can establish, in principle, correctness of a transformation for all possible valid input models, but such approaches require substantial effort. Finally, correctness-by-construction approaches can ensure correctness for transformations which are specified using restricted forms of transformation rules. They may also require proof effort to establish that necessary specification properties hold.

Manual or tool-supported proof of verification properties requires a clear organisation of verification steps and allocation of these steps to appropriate tools. We have found the concept of a *transformation invariant* predicate *Inv* to be of key importance in connecting the properties of transformation computations to required properties of transformation specifications. For update-in-place transformations such as refactorings, the transformation invariant relates intermediate states produced during the transformation to the initial state, and supports proof, by induction over transformation steps, that certain properties are preserved. Termination proof for such transformations usually requires some *transformation variant* function Q to be defined: a non-negative integer-valued expression whose value is strictly decreased by each transformation step. Confluence follows if the condition $Q = 0$ is only possible in a unique (up to isomorphism) terminal state of the transformation reachable from each given initial state. For transformations with separate source and target models, the transformation invariant and postcondition can support proof of syntactic correctness and semantic preservation by relating target elements to the source elements they are derived from, and this correspondence of elements, together with the source language theory Γ_S, enables the deduction of target model properties from source model properties. The invariant, together with assumptions about the initial state of the target model, is particularly useful to show conservativeness of a transformation: that no extraneous elements or properties have been created in the target model. To show semantic correctness of a particular implementation I, we derive the postcondition *Post* of the transformation from the combination of *Inv* and the fact that in the terminal state of the transformation, no further computation step is applicable according to I's behaviour (and $Q = 0$ for update-in-place transformations).

Table 4 summarises how particular properties can be derived using manual or tool-supported proof, using transformation invariants and variants. $Asm0$ denotes the Asm constraints which are predicates of S only. For update-in-place transformations both *Inv* and *Post* may relate the pre-state values $f@pre$, $E@pre$ of features and entity type extents to their post-state values f, E.

	Separate models $\tau : S \to T$	*Update-in-place* $\tau : S \to S$
Semantic preservation of $\varphi \in Pres$	Asm0, Inv, Γ_S, Post, Pres $\vdash \chi(\varphi)$	Asm@pre, Inv, Γ_S, Post, Q = 0, Pres@pre $\vdash \chi(\varphi)$
Syntactic correctness	Asm0, Inv, Γ_S, Post $\vdash \varphi$ for $\varphi \in \Gamma_T$	Γ_S is invariant
Semantic correctness	Asm0, Inv, Γ_S, no step is applicable $\vdash$ *Post*	Asm@pre, Inv, Γ_S, Q = 0, no step is applicable $\vdash$ *Post*
Termination	Implementation uses bounded loops	Q is a variant
Confluence	Order-independence of rule applications	Unique Q = 0 state
Invariance of *Inv*	Asm0, Inv, $\Gamma_S \vdash$ $\forall s : S_i \cdot ECond \Rightarrow [\delta_i(s)]Inv$ Each computation step $\delta_i(s : S_i)$	Asm@pre, Inv, $\Gamma_S \vdash$ $\forall s : S_i \cdot ECond \Rightarrow [\delta_i(s)]Inv$ Each computation step $\delta_i(s : S_i)$

Table 4. Proof techniques for verification properties

In the last case, *ECond* is an execution condition expressing restrictions on when the computation step can be executed in the transformation implementation I of τ, eg., it can express that the relation of some preceding mapping is true. *Inv* must also hold in the initial state, based on $Asm0$ and Γ_S for separate-models transformations, and $Inv[v/v@pre]$ must hold based on Asm and Γ_S for update-in-place transformations.

An alternative approach for proving confluence is direct reasoning that any two distinct transformation steps commute with each other, ie., $[\alpha;\ \beta]P \equiv [\beta;\ \alpha]P$ for any predicate P.

These proof techniques can be related to the model-based formulations of the verification properties of Section 3 via the soundness condition linking $\vdash$ and $\models$. For example, for syntactic correctness, if the proof of line 2 of Table 4 for a separate-models transformation τ holds, and $(m, n0) \models Asm \cup \Gamma_S$, and

$$(m, n0) \longrightarrow_\tau (m, n)$$

then $(m, n) \models Asm0 \cup Inv \cup \Gamma_S \cup Post$ by invariance of *Inv*, semantic correctness, and the preservation of the data of S, so, by Table 4, $(m, n) \models \Gamma_T$ and $n \models \Gamma_T$, as required.

Proof that *Inv* is invariant is carried out using inductive reasoning that each computation step of the transformation preserves *Inv*, if executed according to the transformation implementation algorithm. Additionally, *Inv* must be true initially. Likewise, the variant property of Q can be proved by cases over computation steps.

The general scheme of proof-based verification is therefore:

1. Prove invariance of *Inv*, and (if necessary) the variant property of Q.
2. Use these to deduce semantic preservation, syntactic correctness and semantic correctness.
3. Use syntactic analysis or properties of Q to show termination and confluence.

We describe syntactic analysis techniques for our framework in Section 5, counter-example techniques in Section 6, proof techniques in Section 7, and correctness-by-construction techniques in Section 8.

5 Syntactic analysis of transformation specifications and implementations

Syntactic analysis uses the specification or implementation of a transformation, expressed in the metamodels of Figures 5 and 7, to statically identify properties of the transformation, such as the definedness and determinacy conditions of the transformation rules, and issues concerning their semantics, such as the need for a fixed-point implementation in the case of rules which potentially both write and read the same entity types or features in a model. Some performance bounds can also be estimated by static syntactic analysis [38].

Syntactic analysis has the advantage that no mapping to an additional formalism (such as B, Z3, etc) is necessary, hence there is no reliance on the correctness of such a mapping. In this section we will consider transformations represented in the metamodels of Figures 5 and 7. Regardless of the form of a transformation τ (ie., whether it is an update-in-place transformation or not), each of its mapping constraints should satisfy the following properties of *definedness* and *determinacy.*

For each postcondition, precondition and invariant constraint of a transformation, the definedness condition is a necessary assumption which should hold before the constraint is applied or evaluated, in order that its evaluation is well-defined. Postcondition constraints should normally also satisfy the condition of determinacy.

Examples of the clauses for the definedness function $\mathit{def} : \mathit{Exp}(L) \rightarrow \mathit{Exp}(L)$ are given in Table 5.

Constraint expression e	*Definedness condition def(e)*
a/b	$b \neq 0$ *and* $\mathit{def}(a)$ *and* $\mathit{def}(b)$
$s{\rightarrow}\mathit{at}(\mathit{ind})$ sequence, string s	$\mathit{ind} > 0$ *and* $\mathit{ind} \leq s.\mathit{size}$ *and* $\mathit{def}(s)$ *and* $\mathit{def}(\mathit{ind})$
$E[v]$ entity type E with identity attribute id, v single-valued	$E.\mathit{id}{\rightarrow}\mathit{includes}(v)$ *and* $\mathit{def}(v)$
$s{\rightarrow}\mathit{last}()$ $s{\rightarrow}\mathit{first}()$ $s{\rightarrow}\mathit{max}()$ $s{\rightarrow}\mathit{min}()$ $s{\rightarrow}\mathit{any}()$	$s.\mathit{size} > 0$ *and* $\mathit{def}(s)$
$v.\mathit{sqrt}$	$v \geq 0$ *and* $\mathit{def}(v)$
$v.\mathit{log}$	$v > 0$ *and* $\mathit{def}(v)$
A *and* B A *or* B A *implies* B	$\mathit{def}(A)$ *and* $\mathit{def}(B)$ $\mathit{def}(A)$ *and* $\mathit{def}(B)$ $\mathit{def}(A)$ *and* $(A$ *implies* $\mathit{def}(B))$
$E{\rightarrow}\mathit{exists}(x \mid A)$ $E{\rightarrow}\mathit{forAll}(x \mid A)$	$\mathit{def}(E)$ *and* $E{\rightarrow}\mathit{forAll}(x \mid \mathit{def}(A))$ $\mathit{def}(E)$ *and* $E{\rightarrow}\mathit{forAll}(x \mid \mathit{def}(A))$

Table 5. Definedness conditions for expressions

Examples of the clauses for the determinacy function $\mathit{det} : \mathit{Exp}(L) \rightarrow \mathit{Exp}(L)$ are given in Table 6.

More sophisticated syntactic analysis of a postcondition constraint can be carried out by considering the data-dependency relationships between the language-elements that it relates.

The *write frame* $\mathit{wr}(P)$ of a formula $P \in \mathit{Exp}(L)$ is the set of features and entity types (ie., entity type extents) that it modifies, when interpreted as an action (an action $\mathit{stat}(P)$ to establish P, Section 8). This includes object creation. The *read frame* $\mathit{rd}(P)$ is the set of entity types and features read in P. The frames $\mathit{wr}^*(P)$ and $\mathit{rd}^*(P)$ give further precision by recording the sets of objects (expressions denoting instances of entity types) whose features are written or read in P. Table 7 gives some cases of the definitions of these frames.

In computing $\mathit{wr}(P)$ we also take account of the features and entity types which depend upon the explicitly updated features and entity types of Cn, such as inverse association ends.

If there is a constraint $\varphi \in \mathit{\Gamma}_L$ which implicitly defines a feature g in terms of feature f, ie: $f \in \mathit{rd}(\varphi)$ and $g \in \mathit{wr}(\varphi)$, then g depends on f. In particular, if an association end $\mathit{role}2$ has a named opposite end $\mathit{role}1$, then $\mathit{role}1$ depends on $\mathit{role}2$ and vice-versa.

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

Table 6. Determinacy conditions for expressions

P	$rd(P)$	$wr(P)$	$rd^*(P)$	$wr^*(P)$
Basic expression e without quantifiers, logical operators or $=$, $:$, $E[]$, $\rightarrow includes$, $\rightarrow includesAll$, $\rightarrow excludesAll$, $\rightarrow excludes$, $\rightarrow isDeleted$	Set of features and entity type names used in P	$\{\}$	Set of pairs (obj, f) of objects and features, $obj.f$, in P, plus entity type names in P	$\{\}$
$e1 : e2.r$ $e2.r \rightarrow includes(e1)$ r many-valued $e1$, $e2$ single-valued	$rd(e1) \cup rd(e2)$	$\{r\}$	$rd^*(e1) \cup rd^*(e2)$	$\{(e2, r)\}$
$e2.r \rightarrow excludes(e1)$ r many-valued $e1$, $e2$ single-valued	$rd(e1) \cup rd(e2)$	$\{r\}$	$rd^*(e1) \cup rd^*(e2)$	$\{(e2, r)\}$
$e1.f = e2$ $e1$ single-valued	$rd(e1) \cup rd(e2)$	$\{f\}$	$rd^*(e1) \cup rd^*(e2)$	$\{(e1, f)\}$
$e2.r \rightarrow includesAll(e1)$ r, $e1$ many-valued $e2$ single-valued	$rd(e1) \cup rd(e2)$	$\{r\}$	$rd^*(e1) \cup rd^*(e2)$	$\{(e2, r)\}$
$e2.r \rightarrow excludesAll(e1)$ r, $e1$ many-valued $e2$ single-valued	$rd(e1) \cup rd(e2)$	$\{r\}$	$rd^*(e1) \cup rd^*(e2)$	$\{(e2, r)\}$
$E[e1]$	$rd(e1) \cup \{E\}$	$\{\}$	$rd^*(e1) \cup \{E\}$	$\{\}$
$E \rightarrow exists(x \mid Q)$ (E concrete entity type)	$rd(Q)$	$wr(Q) \cup \{E\}$	$rd^*(Q)$	$wr^*(Q) \cup \{E\}$
$E \rightarrow exists1(x \mid Q)$ (E concrete entity type)	$rd(Q)$	$wr(Q) \cup \{E\}$	$rd^*(Q)$	$wr^*(Q) \cup \{E\}$
$E \rightarrow forAll(x \mid Q)$	$rd(Q) \cup \{E\}$	$wr(Q)$	$rd^*(Q) \cup \{E\}$	$wr^*(Q)$
$x \rightarrow isDeleted()$ x single-valued, of entity type E	$rd(x)$	$\{E\}$	$rd^*(x)$	$\{E\}$
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 7. Definition of read and write frames

Creating an instance x of a concrete entity type E also adds x to each supertype F of E, and these supertypes are also included in the write frames of $E \rightarrow \mathit{exists}(x \mid Q)$ and $E \rightarrow \mathit{exists}1(x \mid Q)$ in the above table.

Deleting an instance x of entity type E by $x \rightarrow \mathit{isDeleted}()$ may affect any supertype of E and any association end owned by E or its supertypes, and any association end incident with E or with any supertype of E. Additionally, if entity types E and F are related by an association which is a composition at the E end, or by an association with a mandatory multiplicity at the E end, ie., a multiplicity with lower bound 1 or more, then deletion of E instances will affect F and its features and supertypes and incident associations, recursively.

The read frame of an operation invocation $e.\mathit{op}(\mathit{pars})$ is the read frame of e and of the pars corresponding to the input parameters of op together with the read frame of the postcondition $\mathit{Post}_{\mathit{op}}$ of op, excluding the formal parameters v of op. Its write frame is that of the actual parameters corresponding to the outputs of op, and $\mathit{wr}(\mathit{Post}_{\mathit{op}}) - v$. $\mathit{wr}(G)$ of a set G of constraints is the union of the constraint write frames, likewise for $\mathit{rd}(G)$, $\mathit{wr}^*(G)$, $\mathit{rd}^*(G)$.

An example of these definitions is

$$\begin{aligned} \mathit{rd}(R) &= \{A, x\} \\ \mathit{wr}(R) &= \{B, y, \mathit{br}, \mathit{ar}\} \end{aligned}$$

for the postcondition constraint R of the example of Figure 1, and

$$\begin{aligned} \mathit{rd}^*(R) &= \{A, (a, x)\} \\ \mathit{wr}^*(R) &= \{B, (b, y), (a, \mathit{br}), (b, \mathit{ar})\} \end{aligned}$$

Using the definitions of read and write frames we can perform some analysis of constraints and transformation specifications 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 transformation for correctness-by-construction implementation, in Section 8.

An important property of a postcondition constraint Cn is that its read and write frames are disjoint:

$$\mathit{wr}(\mathit{Cn}) \cap \mathit{rd}(\mathit{Cn}) = \{\}$$

We refer to such constraints as *type 1* constraints. Subject to further restrictions, they have an implementation as bounded iterations over their source model entity types. Our example constraint R satisfies this property, even though the transformation itself is an update-in-place transformation.

For transformations $\tau : S \rightarrow T$, where S and T may be the same language, the general form of transformation specification postcondition constraints Cn we consider are implications:

$$S_i \rightarrow \mathit{forAll}(s \mid \mathit{SCond} \;\; \mathit{implies} \;\; T_j \rightarrow \mathit{exists}(t \mid \mathit{TCond} \;\mathit{and}\; \mathit{Pred}))$$

or

$$S_i \rightarrow \mathit{forAll}(s \mid \mathit{SCond} \;\; \mathit{implies} \;\; \mathit{Succ}0)$$

where SCond is a predicate over the source language S elements only, S_1, ..., S_n are the entity types of S which are relevant to the transformation, T_j is some entity type of the target language T, TCond is a condition in T elements only, eg., to specify explicit values for t's attributes, and Pred 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. TCond does not contain quantifiers, Pred may contain further *exists* or *forAll* quantifiers to specify creation/lookup of subordinate elements of t. If the t should be unique for a given s, an $\mathit{exists}1$ (*one*) quantifier may be alternatively used in the succedent of the constraint. In the second form $\mathit{Succ}0$ does not have an *exists* quantifier, and does not create target elements but may look up previously created elements and modify them.

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 target language entity types T_j are the *target domains* of the constraint.

A restricted form of constraint in which no *forAll* quantifier can appear in *Pred* or $Succ0$ is called a *conjunctive-implicative form* constraint in [38], it has advantages in terms of comprehensibility and analysability compared to general constraints (eg., which may have nested alternating quantifiers in *Pred*).

The standard implementation (Section 8) of a type 1 postcondition *Cn* is a bounded loop *for* $s : S_i$ *do* $\delta_i(s)$ iterating a computation step $\delta_i(s : S_i)$ defined as

$$if\ SCond\ then\ stat(Succ)$$

where *Succ* is the succedent $Succ0$ or $T_j \rightarrow exists(t \mid TCond\ and\ Pred)$ of *Cn*, and *stat* is as defined in Table 10. We assume that rule implementations using these δ_i steps are used to implement the *Cn* in the following analysis.

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

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

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

  to set the balance of a specific identified account. These are directly implemented by a *RuleImplementation* whose behaviour is $stat(Cn)$.
- Type 1 constraints with 1-1 mapping of identities (*structure-preserving* constraints).
- Type 1 constraints with merging of multiple source instances into single target instances, ie, with a many-1 mapping of identities.
- Type 2 constraints:

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

  is non-empty, but

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

  These constraints usually need to be implemented by a fixpoint iteration (instead of a bounded loop): the basic transformation step $\delta_i(s)$ implementing one application of *Cn* is iterated over any applicable source domain elements s until no applicable source element remains.
- Type 3 constraints: these have

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

  These constraints also need to be implemented by a fixpoint iteration, and each transformation step $\delta_i(s)$ may modify the sets of applicable source objects for subsequent steps, making proof of termination and confluence potentially more difficult than for type 2 constraints.

5.1 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 implementation cannot semantically interfere.

Given a type 1 constraint *Cn*:

$$S_i \rightarrow forAll(s \mid SCond\ implies\ T_j \rightarrow exists(t \mid TCond\ and\ Pred))$$

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

> The only source data read in Cn should be data navigable from s. There should be no reference to any identity attribute of T_j in the succedent $Succ$ of Cn, except in an assignment to it of the identity attribute value of s: $t.tid = s.sid$. Updates in $TCond$ and $Pred$ 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 $t.f \rightarrow includesAll(e)$ to direct features f of t are permitted, in addition t can be added to a set or sequence-valued expression e which does not depend on s or t: $t : e$. Likewise for s. Deletions or removals of elements from collections are not permitted.
> 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). Similar conditions apply for constraints with the $Succ0$ form succedent.

These conditions prevent one application $\delta_i(s1)$ of Cn from invalidating the effect of a preceding $\delta_i(s2)$, $s1 \neq s2$, because the sets wr^* of write frames of the two applications (ie., of *SCond implies Succ* for $s1$ and $s2$) are disjoint[6] 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. Such constraints are termed *localised* type 1 constraints.

The standard implementation of type 1 constraints is a fixed for-loop iteration $for\ s : S_i\ do\ \delta_i(s)$ of their rule implementations over the source domains (Section 8). The above conditions ensure that the execution of the individual $\delta_i(s)$ 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 localised type 1 constraints.

For confluence of this implementation, we need the further conditions that the Cn are determinate, ie., $det(Cn)$ is *true*, and that additions of elements to any 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 $\delta_i(s)$ of Cn has a unique (up to isomorphism) result from a specific starting state.

Two applications $\delta_i(s1)$ and $\delta_i(s2)$ 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 $\delta_i(s1)$ and $\delta_i(s2)$ are independent on these write frames. If a set-valued expression e is written in $Pred$ by a formula $t : e$ or $e \rightarrow includes(t)$, then the written (outermost) feature of 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 $s1$ or $s2$ to a set. □

The example of the postcondition constraint R illustrates this case: the wr^* frames are disjoint for executions of R's computation step to distinct $a1, a2 : A$, except for the shared add-only collection B.

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:

$$S_i \rightarrow forAll(s \mid s : Root.instance.slist)$$

for a singleton entity type $Root$, with an ordered association end $slist : \mathrm{seq}(S_i)$, then two different executions of the transformation could produce two different orderings of *slist*. This is a localised type 1 constraint, and the standard for-loop implementation is semantically correct, but not confluent.

Likewise, if the mapping of identity attribute values is not 1-1, then the same T_j instance could be updated by values derived from two different source objects, with only the second update being retained:

$$S_1 \rightarrow forAll(s \mid T_1 \rightarrow exists(t \mid t.id = s.id/2\ and\ t.y = s.x))$$

[6] Although (t, f) for features f of T_j may occur in both wr^* frames, t denotes distinct objects $t1$, $t2$ for the two applications, because of the condition on tid and sid.

where all attributes are integer-valued.

In this example a T_1 object $t1$ can be created for $s1 : S_1$ with $s1.id = t1.id = 0$, but is then selected (because of the 'check before enforce' principle, Section 8) as the target object matching $s2$ with $s2.id = 1$. Only the value of $s2.x$ is recorded in $t1.y$ at termination.

This constraint is not localised, and is a counter-example to semantic correctness of the standard (bounded loop) implementation of type 1 constraints: completed computations of this implementation cannot satisfy the constraint if S_i elements with the same $s.id/2$ values have different $s.x$ values. A fixpoint iteration could be used instead, but then termination could not be proved in such cases.

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

$$S_1 \rightarrow \mathit{forAll}(s \mid T_1 \rightarrow \mathit{exists}(t \mid t.r = T_0[\text{“1”}] \ \mathit{and} \ t.r.\mathit{att} = s.x))$$

This type 1 constraint is not localised, since $(T_0[\text{“1”}], \mathit{att})$ is in the wr^* frames of different applications, and att is not modified by addition of elements, but by assignment. Again, there is no semantically correct implementation of this constraint, in general, and in practice type 1 non-localised constraints should be avoided in transformation postconditions.

The above results apply directly to conjunctive-implicative form constraints, where the locality properties of the constraint can be easily checked. For example, if source entity types S_1 and S_2 are mapped to corresponding target entity types T_1 and T_2, where there are one-many associations $r1 : S_1 \rightarrow \mathit{Set}(S_2)$ and $r2 : T_1 \rightarrow \mathit{Set}(T_2)$ and unique $\mathit{name} : \mathit{String}$ attributes of each entity type:

$$S_2 \rightarrow \mathit{forAll}(s \mid T_2 \rightarrow \mathit{exists}(t \mid t.\mathit{name} = s.\mathit{name}))$$
$$S_1 \rightarrow \mathit{forAll}(s \mid T_1 \rightarrow \mathit{exists}(t \mid t.\mathit{name} = s.\mathit{name} \ \mathit{and} \ t.r2 = T_2[s.r1.\mathit{name}]))$$

In the second constraint, the expression $T_2[s.r1.\mathit{name}]$ returns all the existing T_2 instances with a name value in $s.r1.\mathit{name}$. Provided that all applications of the computation step of the first constraint are completed before any application of the computation step of the second constraint is attempted, this is a semantically correct transformation, and the standard implementation of each constraint is confluent, terminating and semantically correct according to Theorem 1.

However, a more usual style of transformation specification is the 'recursive descent' form, where subordinate parts of a source model element are transformed together with the element (eg., the use of the *where* clause in QVT-R, or the example of Section 10). For this example, this style would lead to a specification of the form:

$$\begin{aligned} &S_1 \rightarrow \mathit{forAll}(s \mid T_1 \rightarrow \mathit{exists}(t \mid t.\mathit{name} = s.\mathit{name} \ \mathit{and} \\ &\qquad s.r1 \rightarrow \mathit{forAll}(s2 \mid T_2 \rightarrow \mathit{exists}(t2 \mid t2.\mathit{name} = s2.\mathit{name} \ \mathit{and} \ t2 : t.r2)))) \end{aligned}$$

This alternative form also satisfies the condition of disjoint wr^* frames, even though there are non-local assignment updates (eg., to $t2.\mathit{name}$), because it is impossible, due to the association multiplicities, for two distinct t, t' to both contain the same $t2$ instance in their $r2$ sets, therefore applications of the implementation of the above single rule to distinct s, s' will update disjoint parts of the target model and will be non-interfering. We recommend that such specifications are rewritten into conjunctive-implicative form in order to improve the comprehensibility and verifiability of the specification. The efficiency of the implementation may also be higher [38].

A similar analysis can identify sufficient conditions for the confluence, termination and semantic correctness of type 1 entity and instance merging constraints [39].

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.

Analysis of syntactic correctness and semantic preservation for type 1 constraints can be achieved by internal consistency proof in B (Section 7). For these purposes it is useful to formulate an invariant *Inv* for the constraint. Usually the inverse constraint $Cn^{\sim}$:

$$T_j \rightarrow \mathit{forAll}(t \mid \mathit{TCond} \ \mathit{implies} \ S_i \rightarrow \mathit{exists}(s \mid \mathit{SCond} \ \mathit{and} \ \mathit{Pred}))$$

of a conjunctive-implicative Cn will be an invariant for computations of $\mathit{stat}(Cn)$, assuming an initially empty target model.

5.2 Analysis of type 2 and type 3 constraints

A constraint Cn of form

$$S_i{\rightarrow}forAll(s \mid SCond\ implies\ T_j{\rightarrow}exists(t \mid TCond\ and\ Pred))$$

is termed a *type 2* constraint if

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

is non-empty, but

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

This means that the order of application of the computation steps $\delta_i(s : S_i)$ of the constraint to instances $s : S_i$ may be significant, and that a single iteration through the initial set of S_i elements in the source model may be insufficient to establish Cn. A fixpoint computation may be necessary instead, with iterations of δ_i repeated until Cn is established.

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 variant function $Q : S \times T \rightarrow \mathbb{N}$ on the source and target model data can be 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 computation step δ_i of the constraint, and $Q = 0$ when the constraint is established.

Formally, Q is a *variant function* for Cn if:

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

and

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

Succ abbreviates the constraint rhs $T_j{\rightarrow}exists(t \mid TCond\ and\ Pred)$, *smodel* are expressions in the source model data, *tmodel* are expressions in the target model data. The proof of the variant property can assume that the invariants Γ_S and *Inv* of the transformation hold. Q will be syntactically defined as an expression in the union language $S \cup T$. For example, a fixpoint implementation of the constraint R would have a variant

$$A{\rightarrow}select(a \mid not(B{\rightarrow}exists(b \mid b.y = a.x * a.x\ and\ b : a.br))){\rightarrow}size()$$

ie., in mathematical notation:

$$card(\{a \in A \mid \neg\,(\exists\, b : B \cdot y(b) = x(a) * x(a) \wedge b \in br(a))\})$$

This decreases from $card(A)$ at the start of the transformation to 0 at the end.

The general fixpoint implementation of Cn has the form:

```
$while\ not(Cn)$
$do\ \delta_i(S_i{\rightarrow}select(s \mid SCond\ and\ not(Succ)){\rightarrow}any())$
$variant\ Q$
```

The variant function property of Q establishes termination of the fixpoint implementation of Cn: each application of δ_i 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 computation step δ_i to instances of S_i until the application conditions $SCond\ and\ not(Succ)$ are not true for any $s \in S_i$) 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

$$\{s \in S_i \mid SCond \land \neg\ (Succ)\} = \{\}$$

But in such states we also have $Q(smodel, tmodel) = 0$. Therefore, there is a unique termination state. □

Verification of the variant function and unique 0 state properties of Q require proof, eg., by refinement proof in B, syntactic correctness and semantic preservation also require proof (Section 7).

The optimisation patterns 'Replace recursion by iteration' and 'Omit negative application conditions' can also assist in verification of type 2 and type 3 constraints Cn [38]. In the first case bounded iteration can be used instead of fixpoint iteration, if each transformation step of Cn strictly reduces the set of model elements that can match Cn's application condition. Thus termination holds directly, and semantic correctness holds if the steps are non-interfering (localised). In the second case, the fixpoint iteration can be simplified by removing the test for $not(Succ)$, if $SCond$ is inconsistent with $Succ$. Semantic correctness holds directly for such implementations. An example of this is the case study of Section 11.

6 Counter-example analysis of transformation specifications

Together with syntactic analysis, model-checking or satisfaction-checking of transformation specifications can identify errors in specifications before more resource-intensive forms of proof analysis are attempted.

In particular, in the case of a separate models transformation $\tau : S \to T$, a theory Γ_τ formalising the axioms of $\Gamma_S \cup Asm0 \cup Inv \cup Post$ in the union logical language $\mathcal{L}_{S \cup T}$ of S and T expresses the conditions which should hold at termination of the transformation.

This theory should be satisfiable, otherwise the transformation is infeasible. In addition, each axiom $\varphi \in \Gamma_T$ of T should be consistent with Γ_τ. Counter-examples will identify explicit cases where the transformation could fail to establish Γ_T, and hence identify specific errors in the transformation specification.

A number of formalisms and tools can be used to support such analysis, here we will use the Z3 SMT checker [51] for first-order logic. This has an additional use as a theorem-prover: if

$$\Gamma_\tau \cup \{\neg\ \varphi\}$$

is unsatisfiable, then φ follows from Γ_τ, so that syntactic correctness (and likewise semantic preservation) can be proved, in principle, using Z3.

For an update-in-place transformation $\tau : S \to S$, Γ_τ formalises $\Gamma_S \cup Asm@pre \cup Inv \cup Post$, with pre-state data names $g@pre$ being represented by new constants g_pre.

Table 8 shows examples of the mapping of OCL expressions to Z3.

The most significant step in this translation is the expression of existential quantifiers by skolem functions: in a formula

$$E \to forAll(x \mid Cond\ implies\ F \to exists(y \mid Pred))$$

the existential quantifier is replaced by a new function $fnew : E \to F$ not occurring in any other part of the OCL or Z3 theories, and the formula is then translated as:

$$(forall\ ((x\ E))\ (\Rightarrow\ \ Cond'\ Pred'[fnew(x)/y]))$$

OCL expression/operator e	*Z3 expression/operator e'*
Integer	Int
Boolean	Bool
Real	Real
Entity type E	Sort E
Attribute $att : Typ$ owned by E	function $att : E \rightarrow Typ'$
Single-valued role r to F owned by E	function $r : E \rightarrow F$
Collection-valued role r to F owned by E	function $r : E \rightarrow List(F)$
implies	$\Rightarrow$
forAll	forall
exists	Expressed by skolemisation
first	head
prepend	insert
includes	memberE (for each entity type E)
Set{ }	nil
Set{ x1, ..., xn }	(insert x1' (... (insert xn' nil) ...))

Table 8. Mapping of UML and OCL to Z3

In general, an *exists*-quantifier in the scope of several *forAll*-quantifiers is replaced by a new function depending on all the *forAll*-quantified types.

Sets and sequences are both modelled as Z3 lists. Operators for OCL select and collect are not in-built in Z3 and need to be defined using auxiliary functions. In addition, operators to check membership in a list and to obtain a list element by its index need to be added as auxiliary functions. An alternative to using lists to model OCL collections would be to use bitsets [47], however this involves a highly complex encoding and requires size bounds to be placed on entity type extents and collection sizes.

For separate-models transformations, or update-in-place transformations where the read and updated sets of entity types and features are disjoint, the mapping from the metamodel of Figure 5 to a metamodel for Z3 performs the following translations:

- Each source and target language L (or language part) is represented by sorts E for each entity type E of the language, and maps of the form $f : E \rightarrow Typ$ for each owned feature f of E, together with Z3 encodings of the constraints of Γ_L for source languages L.
- The assumptions $Asm0$ on unmodified data are encoded and included.
- Each mapping $rule.relation$ of the form

  $$E{\rightarrow}forAll(x \mid SCond\ implies\ F{\rightarrow}exists(y \mid PCond))$$

  is encoded as a function $taurule : E \rightarrow F$ and a predicate

  $$\forall\, x : E \cdot SCond' \ \Rightarrow\ PCond'[taurule(x)/y]$$

- The invariant expressing the inverse of $rule$ is encoded by a function $sigmarule : F \rightarrow E$ and a predicate

  $$\forall\, y : F \cdot SCond'[sigmarule(y)/x]\ and\ PCond'[sigmarule(y)/x]$$

For data items g which are both read and updated, in update-in-place transformations, both the pre-state value g_pre and the post-state value g are represented, and the pre-state predicates of $Asm@pre$ are included.

For separate-models transformations, the consistency of the resulting theory Γ_τ with individual constraints $\varphi \in \Gamma_T$ can be checked by adding φ' to the theory, counter-examples can be searched for by instead adding $\neg\, \varphi'$.

Syntactic correctness can be shown by adding the negation of $\bigwedge \Gamma_T$ for the target language T, and establishing that the resulting theory is unsatisfiable. Semantic preservation of φ can then be shown by adding Γ_T, φ and the negation of $\chi(\varphi)$ and showing unsatisfiability. Unlike FOL or B, Z3 does not contain proof techniques for unbounded induction, and so may fail to establish true inferences which could be proved within FOL or B. In addition it provides decision procedures for only some of the decidable subsets of first order logic. There may therefore be situations where Z3 may neither demonstrate counter-examples to a theory, nor establish its validity, and for these cases analysis using proof in B or another theorem-prover will be necessary.

The mapping from UML and OCL to Z3 has been automated in the UML-RSDS tools [49].

As in first-order set theory, Z3 has no notion of undefined values, unlike in standard OCL. Division in Z3 is a total function, but its value is unspecified for division by 0. A transformation specifier should ensure that all expressions within a transformation invariant, assumption or rule relation Cn have a defined value, ie., $def(Cn)$ is true, whenever the predicate may be evaluated. When reasoning about transformations using Z3, the results are only meaningful for transformation implementations which satisfy this condition, ie., where no expression evaluation to OCL *invalid* ever occurs.

There is no inbuilt Z3 representation for strings: these can be modelled as an unspecified sort or as lists of integers. There is no direct representation for subtyping.

An example of mapping OCL to Z3 is our transformation example from Figure 1. The corresponding Z3 theory is:

```
(declare-sort A)
(declare-sort B)
(declare-fun x (A) Int)
(declare-fun y (B) Int)
(declare-fun br (A) (List B))
(declare-fun ar (B) (List A))
(assert (forall ((a A)) (> (x a) 0)))
(assert (forall ((a A)) (forall ((b B)) (= (memberB b (br a)) (memberA a (ar b))))))
```

This expresses the theory of the class diagram. For the transformation postcondition a new skolem function is introduced, and an axiom for the skolemised postcondition:

```
(declare-fun tau1 (A) B)
(assert (forall ((a A))
   (and (= (y (tau1 a)) (* (x a) (x a))) (memberB (tau1 a) (br a)))))
```

Note that this is not within a decidable subset of first-order logic, since it involves non-linear arithmetic.

The invariant is expressed by:

```
(declare-fun sigma1 (B) A)
(assert (forall ((b B))
    (and (= (y b) (* (x (sigma1 b)) (x (sigma1 b)))) (memberB b (br (sigma1 b))))))
```

This combined theory expresses the state at termination of the transformation, and also defines precisely how the terminal state of the system should relate to the starting state, by relating the modified entity types and features $\{B, br, ar, y\}$ to the unchanged values of $\{A, x\}$.

Consistency analysis shows that this combined theory is indeed consistent. If we add the additional constraint

```
(not (forall ((b B)) (> (y b) 2)))
```

a counter-example to $B \rightarrow forAll(b \mid b.y > 2)$ is found, where b is derived from $a1 : A$ with $a1.x = 1$.

For a language L, soundness of deductions about L in Z3 holds, relative to the FOL proof theory of $\mathcal{L}_L$, and using the above mapping: Z3 correctly represents the mathematical data types Integer, Real of OCL, and Boolean, and likewise for operators upon these types, and for those OCL collections and collection operators which can be expressed in Z3. If Z3 can represent all the entity and data types, features and constraints of L, in a Z3 theory L', then derivation of (the translation φ' of) a sentence $\varphi \in Sen(L)$ in Z3 from L' implies that φ is deducable in $\mathcal{L}_L$:

$$\vdash_{Z3,L'} \varphi' \;\Rightarrow\; \vdash_{\mathcal{L}_L} \varphi$$

The reverse implication fails, because Z3 is incomplete, even for decidable subsets of first-order set theory.

Provided that the constraints of a transformation τ are within a decidable subset of first order logic, we can ensure that the transformation specification of τ is in a decidable form if we restrict it to the *stratified sorts* subset of first-order logic. This means that the entity types used in the transformation can be assigned rankings in $\mathbb{N}$ so that for every association r used in the transformation specification, its target entity type is strictly lower in the ranking than its source. Such a restriction however rules out transformations which utilise self-associations and bi-directional associations: only navigations down strict composition hierarchies in the source and target models are permitted.

7 Proof-based verification

Proof-based techniques for verifying transformation correctness properties have two main advantages: (i) they can prove the properties for all cases of a transformation, that is, for arbitrary input models and for a range of different implementations; (ii) a record of the proof can be produced, and subjected to further checking, if certification is required. However, proof techniques invariably involve substantial human expertise and resources, due to the interactive nature of the most general forms of proof techniques, and the necessity to work both in the notation of the proof tool and in the transformation notation.

We have selected B AMN as a suitable formalism for proof-based verification, B is a mature formalism, with good tool support, which automates the majority of simple proof obligations. Table 9 gives a comparison of B with other verification tools. We provide an automated mapping from transformation specifications into B [49], and this mapping is designed to facilitate the comprehension of the B AMN proof obligations in terms of the transformation being verified.

The entity types and features of the languages involved in a transformation τ are mapped into B according to Table 1. OCL expressions are systematically mapped into set-theory expressions, Tables 2 and 3 illustrate this mapping.

A B AMN specification consists of a linked collection of modules, termed *machines*. Each machine encapsulates data and operations on that data. Each transformation is represented in a single main B machine, together with an auxiliary *SystemTypes* machine containing type definitions.

The mapping from the metamodel of Figure 5 to a metamodel for B performs the following translations:

- Each source and target language L is represented by sets es for each entity type E of the language, with $es \subseteq objects$, and maps $f : es \rightarrow Typ$ for each feature f of E, together with B encodings of the constraints of Γ_L for unmodified L. In cases where a language entity type or feature g is both read and written by the transformation, a syntactically distinct copy g_pre is used to represent the initial value of g at the start of the transformation. A supertype F of entity type E has B invariant $es \subseteq fs$. Abstract entity types E have a B invariant $es = f1s \cup ... \cup fls$ where the Fi are the direct subtypes of E.
  For each concrete entity type E of a source language, there is an operation $create\_E$ which creates a new instance of E and adds this to es. For each data feature f of an entity type E there is an operation $setf(ex, fx)$ which sets $f(ex)$ to fx.
- The assumptions Asm of the transformation can be included in the machine invariant (using g_pre in place of g for data which is written by the transformation). Asm is also included in the preconditions of the source language operations $create\_E$ and $setf$.
- Each mapping *rule* is encoded as an operation with input parameters the names of the non-modifiable ends of *rule*, and with its effect derived from *rule.relation* or from the *behavior* of an implementation of *rule*. The operation represents transformation computation steps δ_i of the *rule* implementation.
- Orderings of the steps for particular implementations can be encoded by preconditions of the operations, expressing that $rule'.relation$ for one or more other mapping $rule'$ has been already established for all applicable elements.
- Invariant predicates Inv are added as B invariants, using g_pre to express pre-state values g@pre.

In contrast to the mapping to Z3 described in Section 6, this mapping explicitly represents the computation steps of the transformation, and can therefore support verification that these maintain *Inv* and decrease any variant, Q. For a separate-models transformation $\tau : S \to T$ the machine invariant expresses $\Gamma_S \cup Asm0 \cup Inv$. For an update-in-place transformation $\tau : S \to S$ the invariant expresses $\Gamma_S \cup Asm@pre \cup Inv$. The machine represents the transformation *at any state during its computation.*

This mapping is suitable to support the proof of syntactic correctness, semantic preservation and semantic correctness by using internal consistency proof in B, a more complex mapping is used for the proof of confluence and termination, using refinement proof [36].

The general form of a B machine M_τ representing a separate-models transformation τ with source language S and target language T is:

```
MACHINE Mt SEES SystemTypes
VARIABLES
  /* variables for each entity type and feature of S */
  /* variables for each entity type and feature of T */
INVARIANT
  /* typing definitions for each entity type and feature of S and T */
  GammaS &
  Asm0 & Inv
INITIALISATION
  /* var := {} for each variable */
OPERATIONS
  /* creation operations for entity types of S, restricted by Asm */
  /* update operations for features of S, restricted by Asm */
  /* operations representing transformation steps */
END
```

SystemTypes defines the type *Object_OBJ* and any other type definitions required, eg., of enumerated types.

The operations to create and update S elements are used to set up the source model data of the transformation. Subsequently, the operations representing transformation steps are performed.

If $Asm0$ consists of universally quantified formulae $\forall s : S_i \cdot \psi$, then the instantiated formulae $\psi[sx/s]$ are used as restrictions on operations creating $sx : S_i$ (or subclasses of S_i). Likewise, operation $setf(sx, fx)$ modifying feature f of S_i has a precondition $\psi[sx/s, fx/s.f]$. All these operations will include the preconditions $Asm1$ from Asm which concern only the target model.

As an example, the transformation of Figure 1 can be defined by the following partial machine:

```
MACHINE Mt SEES SystemTypes
VARIABLES objects, as, x, br, bs, y, ar
INVARIANT
  objects <: Object_OBJ &
  as <: objects & bs <: objects &
  x : as --> INT & br : as --> FIN(bs) &
  y : bs --> INT & ar : bs --> FIN(as) &
  !a.(a : as => x(a) > 0) &
  !a.(a : as => !b.(b : bs => (b : br(a) <=> a : ar(b)))) &
  !b.(b : bs => #a.(a : as & y(b) = x(a)*x(a) & b : br(a)))
INITIALISATION
  objects, as, x, br, bs, y, ar := {}, {}, {}, {}, {}, {}, {}
```

The invariant expresses Γ_S and the *Inv* properties of the transformation. $\#b.P$ is B syntax for $\exists b \cdot P$, $!a.P$ is B syntax for $\forall a \cdot P$. & denotes conjunction, $<:$ denotes $\subseteq$ and $-- >$ is $\to$ (the total function type constructor). A universal set *objects* of existing objects is maintained, this is a subset of the static type *Object_OBJ* declared in *SystemTypes*.

Occurrences $E@pre$ or $f@pre$ in *Inv* or *Post* are interpreted by the 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 the operations for transformation computation steps.

The operations representing computation steps are derived from the rule implementations δ_i of the constraints Cn. This modelling approach facilitates verification using weakest precondition calculation, compared to more abstract encodings. If Cn has the form

$S_i \rightarrow forAll(s \mid SCond\ implies\ Succ)$

then the operation representing a computation step δ_i of Cn is:

```
delta_i(s) =
  PRE s : sis & SCond & not(Succ) &
    C1 & ... & Cn-1 & def(Succ)
  THEN
    stat'(Succ)
  END
```

where $stat'(P)$ encodes the procedural interpretation $stat(P)$ of P in B program-like statements, called *generalised substitutions*. These have a similar syntax to the programming language described in Section 2, and use the same weakest-precondition semantics. B has an additional statement form $v := e1 \| w := e2$ of *parallel assignment*: the assignments are performed order-independently, with the values of $e1$, $e2$ being simultaneously assigned to v, w. The ANY WHERE THEN statement of B corresponds to our creation statement.

If the implementation of τ defines a non-standard rule implementation of Cn, this implementation could be encoded in B in place of the above definition of *delta_i*.

If τ's implementation requires that all constraints $C1$, ..., $Cn-1$ are established before Cn, this ordering can be encoded by including $C1$, ..., $Cn-1$ in the preconditions of *delta_i*, as above (cf., the *ECond* conditions of Section 4). $not(Succ)$ can be omitted if negative application conditions are not checked by the implementation of Cn. For the mapping to B, $def(Succ)$ includes checks that numeric expressions in *Succ* are within the size bounds of the finite numeric types *NAT* and *INT* of B, and that $objects \neq Object\_OBJ$ prior to any creation of a new object.

The computational model of a transformation τ expressed in M_τ therefore coincides with the definition of transformation computation described in Section 2: a computation of τ is a sequence of transformation steps executed in an indeterminate order, constrained only by the need to maintain *Inv*, and, if a specific implementation I is defined, to satisfy the ordering restrictions of I's behaviour.

For the example of Figure 1, using a fixpoint implementation, the resulting completed B machine has:

```
OPERATIONS
  create_A(xx) =
    PRE xx : INT & xx > 0 & objects /= Object_OBJ & bs = {}
    THEN
        ANY ax WHERE ax : Object_OBJ - objects
        THEN
            as := as \/ { ax } || objects := objects \/ { ax } ||
            x(ax) := xx ||
            br(ax) := {}
        END
    END;

  setx(ax,xx) =
    PRE ax : as & xx : INT & xx > 0 & bs = {}
    THEN
      x(ax) := xx
    END;

  r1(ax) =
    PRE ax : as & not( #b.(b : bs & y(b) = x(ax)*x(ax) & b : br(a)) ) &
      objects /= Object_OBJ & x(ax)*x(ax) : INT
    THEN
```

```
        ANY b WHERE b : Object_OBJ - objects
        THEN
           bs := bs \/ { b } || objects := objects \/ { b } ||
           y(b) := x(ax)*x(ax) ||
           br(ax) := br(ax) \/ { b } || ar(b) := { ax }
         END
     END
END
```

$r1$ defines the transformation step of the postcondition constraint. The machine is generated automatically by the UML-RSDS tools from the UML specification of the transformation[7]. UML-RSDS encodes the semantics of all cases of updates to associations, including situations with mutually inverse association ends, as in this example (the last two assignments of $r1$).

Using these machines we can verify syntactic correctness and semantic preservation properties of a model transformation, by means of *internal consistency* proof of the 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: $\exists\, v.I$ where v is the tuple of variables of the machine, and I its invariant.
- That the initialisation establishes the invariant: $[\mathit{Init}]I$
- That each operation maintains the invariant:

  $$\mathit{Pre} \wedge I \;\; \Rightarrow \;\; [\mathit{Code}]I$$

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

B machines implicitly satisfy the *frame axiom* for state changes: variables v which are not explicitly updated by an operation are assumed not to be modified by the operation. This corresponds to the assumption made in our framework that v is unmodifed by activity act if $v \notin \mathit{wr}(act)$.

We can follow the scheme for proof indicated in Section 4 using B, as follows (for separate models transformations):

1. Internal consistency proof of $M_{\mathcal{T}}$ establishes that Inv is an invariant of the transformation.
2. By adding the axioms of Γ_T to the INVARIANT clause, the validity of these during the transformation and in the final state of the transformation can be proved by internal consistency proof, establishing syntactic correctness.
3. By adding φ and $\chi(\varphi)$ to the INVARIANT of $M_{\mathcal{T}}$, for $\varphi \in \mathit{Pres}$, semantic preservation of φ can be proved by internal consistency proof. Creation and update operations to set up the source model must be suitably restricted by φ.
4. At termination of the transformation, all the application conditions of the transformation rules are false. The inference

   $$(\bigwedge \neg\, (ACond)) \;\; \Rightarrow \;\; \mathit{Post}$$

   can be encoded in the ASSERTIONS clause of $M_{\mathcal{T}}$ and proved using the invariants $\Gamma_S \cup \mathit{Inv} \cup Asm0$.

For update-in-place transformations, termination, confluence and semantic correctness proof needs to use suitable Q variants for each constraint, considered below.

Using Atelier B version 4.0, 24 proof obligations for internal consistency of the above machine Mt are generated, of which 18 are automatically proved, and the remainder can be interactively proved using the provided proof assistant tool.

In order to prove that a postulated Q measure is actually a variant function for a constraint, refinement proof in B can be carried out, with an abstraction of the transformation model machine $M_{\mathcal{T}}$ defined as $M0_{\mathcal{T}}$:

[7] In practice, single-letter feature, variable and entity type names should be avoided, since these have a special meaning in B AMN.

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

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

delta represents a transformation step of the constraint for which q is the postulated variant. The operator $q :: s$ assigns an unspecified element of s to q. Each constraint C_i may have a corresponding variant q_i, the operation *delta_i* for abstracted transformation steps of C_i then has the form:

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

where the constraint implementations of C_1, ..., C_{i-1} are designed to terminate prior to any execution of *delta_i* and therefore their variants are assumed to be 0 in the precondition of *delta_i*.

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 *qi* variants. Refinement proof then attempts to verify that the explicit definition of each *qi* obeys the abstract specification, ie, that it is strictly decreased by every execution of *delta_i*.

Refinement obligations in B are [31]:

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

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

- Each refined operation *PRE* Pre_R *THEN* $Code_R$ *END* satisfies the pre-post relation of its abstract version:

$$Inv_A \wedge Inv_R \wedge Pre_A \;\Rightarrow\; Pre_R \wedge [Code_R]\neg\ [Code_A]\neg\ (Inv_A \wedge Inv_R)$$

Refinement proof is usually manually intensive, with the majority of proof obligations requiring interactive proof.

To verify confluence of a transformation, it is sufficient to show that there is a unique state (up to structure isomorphism) where all the variants *qi* have $qi = 0$. This can be verified in the refinement by adding an *ASSERTIONS* clause of the schematic form:

```
ASSERTIONS
  q1 = 0 & ... & qn = 0  =>  tstate = f(sstate)
```

This clause expresses that the target model state *tstate* has specific values in terms of the source model state *sstate*, when all the *qi* are zero.

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 implementation: that the desired semantics of target model elements relative to source model elements also holds at termination. An example of this form of confluence proof is given in [39]: in this case one direction (that the computed transitive closure of a relation r is always a subset of the actual transitive closure) of the equality is part of the transformation invariant, and the other direction follows from $Q = 0$.

The above procedures can be used as a general process for proving *Inv* and *Pres* properties, and for proving termination, semantic correctness and confluence, by means of suitable postulated Q variant functions. 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 computation 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 the metamodels of Section 2. For TGG it may be preferable to combine the operations which update the input model and those which express transformation steps, since a transformation step in TGG involves the simultaneous update of source, target and correspondence models.

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 DO-178C [19]. As with Z3, it can be argued that our translation of OCL to B accurately represents the semantics of (our classical logic version of) OCL, so that proof in B is sound with respect to logical deduction over OCL sentences in first-order logic, however the numeric types *NAT* and *INT* in B are bounded, and correspond to 32-bit unsigned and signed integers, respectively, so that soundness only applies if the same meaning is given to OCL *Integer*. In addition, the type *Real* must be excluded from constraints and transformations in order to carry out valid proof analysis in B. *Object_OBJ* is finite in B, so that an a-priori upper bound should be set on the maximum number of objects.

The size of the B formal model M_τ is linear in terms of the size of the model transformation τ, however the complexity of the transformation rules has a considerable effect on the proof effort required. Postcondition constraints using *forAll* quantifications nested within *exists* in their succedents cannot be effectively verified, and the use of conjunctive-implicative form is necessary instead. Recursive update operations cannot be represented. B is not suitable for establishing satisfiability properties asserting the existence of models of certain kinds, and tools such as Z3, UMLtoCSP [15], Alloy [2] or USE [29] are more appropriate for these.

7.1 Transformation verification tools

Table 9 lists some existing formalisms/technologies which can be used for transformation verification based on proof, and identifies their appropriateness or limitations for different verification tasks.

The most comprehensive technology appears to be B, which is also the only one to directly support inductive proof over computation steps. However, B requires substantial expertise in logic and set theory to use successfully, as most verification tasks involve interactive proof: automated proof may only resolve a small percentage of refinement obligations in particular. B also does not provide any counter-example finding capabilities. A disadvantage of many formalisms is that they are based on a two-valued first-order or relational logic, in contrast to the 3-valued logic of OCL used in UML. However, the complexity of the full OCL value system, involving both *invalid* and *null* values, and its lack of clear semantics, means that tools which do attempt to handle full OCL necessarily make specific assumptions about the semantics, which may not match the specifier's intentions [11].

Z3 is well-suited to proof of properties within a single state, but transformations may only have decidable theories if they satisfy the *stratification* property (ie., that only navigations in a consistent direction up or down the composition heirarchies of the metamodels are used), and Z3 is also limited by its restricted representation capabilities for OCL, ie., it has no direct representations of operators such as *select*, *collect*, etc, in contrast to B. This limitation also applies to Alloy. The USE tool works directly on UML and OCL, so reducing the semantic gap between the analysis formalism and the transformation being analysed, however USE has no proof capabilities. B lacks support for real numbers, and uses bounded integers, whilst Z3 lacks support for strings and subtyping. HOL-OCL has a comprehensive representation of OCL, but proof in HOL-OCL is primarily interactive.

Formalism	*Capabilities*	*Limitations*
Alloy [2]	Constraint analysis; counterexample construction	Bounded search space: incomplete counter-example detection. Uses relational logic.
B [36]	Proof (interactive and automated)	Refinement proof can be highly time-intensive. Two-valued logic used.
UML-RSDS [49]	Syntactic analysis, maps to B, Z3, USE	Requires transformation to be written in UML-RSDS. Uses 2-valued logic.
USE [29]	Counterexample construction	Bounded search space.
Z3 [51]	Satisfaction checking, proof by failure of counterexample search	Incomplete counterexample detection, proof. Limited expressiveness for OCL. Uses 2-valued logic.
HOL-OCL [10]	Interactive and automated proof	Tool-specific semantics for OCL.

Table 9. Verification technologies for model transformations

Therefore a heterogeneous approach to transformation verification tool support is necessary in general, with technologies being selected on the basis of their appropriateness for particular tasks.

An important area which has not been developed previously is the definition of proof techniques for compositions of transformations. We identify some rules in [39]. In particular, for sequential compositions such as Figure 6, the syntactic correctness of the composed transformation $\tau_1;\ \tau_2$ follows from that of τ_1 and τ_2 separately, provided that the assumptions Asm_2 of τ_2 can be ensured at termination of τ_1, either because these and Asm_1 are implied by the overall assumptions Asm of the composition, and Asm_2 are not invalidated by τ_1, or because τ_1 establishes Asm_2. Likewise, semantic preservation and termination properties compose over sequential composition. However, for confluence of $\tau_1;\ \tau_2$, confluence of τ_1 is required, and additionally that τ_2 is *isomorphism-preserving*: it maps isomorphic source models to isomorphic target models.

8 Correctness by construction techniques

If transformations are specified in a platform-independent declarative manner, using postconditions, preconditions and invariants expressed in the transformation specification metamodel (Figure 5), then platform-independent implementations (expressed in the metamodel of Figure 7) can be derived from them, and given the existence of mappings to particular transformation languages, such as ETL, ATL, etc, platform-specific implementations can then be generated from these implementation models. These platform-specific implementations should then be correct-by-construction with respect to the specifications: they will satisfy semantic correctness without the need for further proof. In UML-RSDS we generate executable Java implementations directly from the platform-independent implementations [38].

The transformation specification in the metamodel of Figure 5 plays the role of a *Computation-independent model* (CIM) in model-driven development terminology, ie., a model without explicit algorithmic details, whilst the platform-independent implementation (Figure 7) plays the role of a *Platform-independent model* (PIM).

The basis for the synthesis of a correct-by-construction implementation of mapping rules, is the definition of a procedural interpretation, $stat(P)$, for certain OCL predicates P. $stat$ maps from expressions over a language (or a union of languages) L to behaviours over L:

$$stat : Exp(L) \nrightarrow Statement(L)$$

The intent behind this mapping is that $\mathit{stat}(P)$ should establish P, assuming the definedness of expressions in P:

$$\mathit{def}(P) \;\Rightarrow\; [\mathit{stat}(P)]P$$

for $P \in \mathrm{dom}(\mathit{stat})$.

The design-level activity $\mathit{stat}(P)$ associated with a transformation specification postcondition predicate P is defined systematically based on the structure of P. $\mathit{stat}(P)$ can be read as 'Make P true'. Table 10 shows some of the main cases of this definition.

P	$\mathit{stat}(P)$	*Condition*
$x = e$	$x := e$	x is assignable, $x \notin \mathit{rd}(e)$
$e : x$ $x \rightarrow \mathit{includes}(e)$	$x := x \rightarrow \mathit{including}(e)$	x is assignable, set-valued, $x \notin \mathit{rd}(e)$
$e\ /: x$ $x \rightarrow \mathit{excludes}(e)$	$x := x \rightarrow \mathit{excluding}(e)$	x is assignable, collection-valued, $x \notin \mathit{rd}(e)$
$e <: x$ $x \rightarrow \mathit{includesAll}(e)$	$x := x \rightarrow \mathit{union}(e)$	x is assignable, set-valued, $x \notin \mathit{rd}(e)$
$e\ /<: x$ $x \rightarrow \mathit{excludesAll}(e)$	$x := x - e$	x is assignable, collection-valued, $x \notin \mathit{rd}(e)$
$x \rightarrow \mathit{isDeleted}()$ (single object x)	$E := E \rightarrow \mathit{excluding}(x)$	Each entity type E containing x
$\mathit{obj}.\mathit{op}(e)$ $\mathit{objs}.\mathit{op}(e)$	$\mathit{obj}.\mathit{op}(e)$ for $x : \mathit{objs}$ do $x.\mathit{op}(e)$	Single object obj Collection objs
$P1\ \mathit{and}\ P2$	$\mathit{stat}(P1);\ \mathit{stat}(P2)$	$\mathit{wr}(P2) \cap \mathit{wr}(P1) = \{\}$ $\mathit{wr}(P2) \cap \mathit{rd}(P1) = \{\}$
$E \rightarrow \mathit{exists}(x \mid x.\mathit{id} = v$ $\mathit{and}\ P1)$ $E \rightarrow \mathit{exists}(x \mid P1)$	if $E.\mathit{id} \rightarrow \mathit{includes}(v)$ then $x := E[v];\ \mathit{stat}(P1)$ else $(x : E;$ $\mathit{stat}(x.\mathit{id} = v\ \mathit{and}\ P1))$ $(x : E;\ \mathit{stat}(P1))$	E is a concrete entity type with $E \rightarrow \mathit{isUnique}(\mathit{id})$ E is a concrete entity type, $P1$ not of form $x.\mathit{id} = v\ \mathit{and}\ P2$ for unique id attribute of E

$e \rightarrow \mathit{exists}(x \mid x.\mathit{id} = v$
$\mathit{and}\ P1)$
 	
$e \rightarrow \mathit{exists}(x \mid P1)$
	if $e \rightarrow \mathit{includes}(E[v])$ then
	$(x := E[v];\ \mathit{stat}(P1))$
	else skip
	if $e \rightarrow \mathit{notEmpty}()$ then
	$(x := e \rightarrow \mathit{any}();$
	$\mathit{stat}(P1))$ else skip
		Non-writable expression
		e with element type E,
		$E \rightarrow \mathit{isUnique}(\mathit{id})$
		Non-writable expression
		e, $P1$ not of
		above form

$E \rightarrow \mathit{exists1}(x \mid P1)$	if $E \rightarrow \mathit{exists}(x \mid P1)$ then skip
	else $\mathit{stat}(E \rightarrow \mathit{exists}(x \mid P1))$
		E is a concrete entity
		type or non-writable
		expression e with
		element type E

$E \rightarrow \mathit{forAll}(x \mid P1)$	for $x : E$ do $\mathit{stat}(P1)$	P type 1, localised
$P1\ \mathit{implies}\ P2$	if $P1$ then $\mathit{stat}(P2)$ else skip	

Table 10. Definition of $\mathit{stat}(P)$

Updates to association ends may require additional further updates to inverses of the association ends, updates to entity type extents or to features may require further updates to derived and other data-dependent features, and so forth. These updates are all included in the *stat* activity. In particular, for $x \rightarrow \mathit{isDeleted}()$, x is removed from every association end in which it resides, and further cascaded deletions may occur if these ends are mandatory/composition ends.

The clauses for $X \rightarrow \mathit{exists}(x \mid x.\mathit{id} = v\ \mathit{and}\ P1)$ test for existence of an x with $x.\mathit{id} = v$ before creating such an object: this has implications for efficiency but is necessary for correctness: two distinct X elements with the same primary key value should not exist. This design strategy is a case of the well-known principle of 'check before enforce' used in QVT, ETL, ATL and other transformation languages.

$stat(E \rightarrow forAll(x \mid P1))$ has special definitions for type 2 and 3 quantified formulae, and for type 1 non-localised formulae, based on fixpoint iteration ([38]).

The write frame of $stat(P)$ is equal to $wr(P)$, the read frame includes $rd(P)$.

As an example of these definitions, $stat(R)$ for the postcondition of the transformation of Figure 1 is:

```
for a : A
do
   (b : B; b.y := a.x*a.x;
    a.br := a.br \/ { b };
    b.ar := b.ar \/ { a })
```

The selection of a suitable PIM for a transformation CIM is based upon the data-dependency relations of the postconditions of the CIM.

Given the set $Post$ of postconditions of a transformation τ, we wish to find an ordering of the postcondition constraints such that the sequential composition of the 'natural' implementations $stat(Cn)$ of the constraints $Cn \in Post$ achieves the conjunction $\bigwedge Post$ of the postconditions.

To find such an ordering we consider the following properties of constraints:

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

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

"Cm depends on Cn".

A transformation implementation with rule implementations r_i of postconditions C_i (ie., with each $r_i.applies.relation \; = \; C_i$) ordered as $r_1, \ldots, r_n$ should satisfy the *syntactic non-interference* conditions:

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

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

A transformation implementation with the ordering $r_1, \ldots, r_n$ of rule implementations 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.

If the ordering $r_1, ..., r_n$ satisfies semantic non-interference, then by induction it can be proved that the corresponding sequential composition of rule implementations establishes the conjunction of the C_i [38]. Thus any semantically non-interfering ordering of the r_i is equivalent in this sense.

The resulting PIM implementation therefore has activity r_1; ...; r_n where $r_i \in RuleImplementation$ implements C_i: $r_i.applies.relation = C_i$. In turn, each r_i has $r_i.behaviour = stat(C_i)$.

As described in Section 5, depending upon the internal data-dependencies of C_i, its implementation $stat(C_i)$ can be defined as a bounded or fixpoint iteration of individual transformation steps δ_i. For a type 1 localised constraint $S_i \rightarrow forAll(s \mid SCond\ implies\ Succ)$, this leads to an implementation of the form

$$for\ s : S_i\ do\ \delta_i(s)$$

where δ_i has activity $stat(SCond\ implies\ Succ)$.

If an ordering of the $Post$ constraints of τ can be found that satisfies semantic non-interference, then semantic correctness therefore holds for an implementation constructed using this ordering. Termination and confluence properties can be established by syntactic analysis of the individual constraints, for type 1 constraints, following the process of Section 5. For other forms of constraint, these properties may require verification of suitable variants (Section 7). Syntactic correctness and semantic preservation properties may need proof, eg., using Z3 or B.

8.1 Reverse-engineering of model transformation implementations

The above process may be applied in reverse in order to abstract existing MT code to declarative specifications. A statement $\mathit{Code} \in \mathit{Statement}$ can be abstracted to a postcondition predicate P if $\mathit{stat}(P) = \mathit{Code}$.

In particular, an assignment $x := e$ abstracts to $x = e[x@\mathit{pre}/x]$ and $\mathit{Code}1;\ \mathit{Code}2$ abstracts to $P1\ \mathit{and}\ P2$ if $\mathit{stat}(P1) = \mathit{Code}1$, $\mathit{stat}(P2) = \mathit{Code}2$ and $\mathit{wr}(P1) \cap \mathit{wr}(P2) = \{\}$ and $\mathit{wr}(P2) \cap \mathit{rd}(P1) = \{\}$. Likewise for bounded loops, conditionals, etc. In cases where the data dependency conditions do not hold in the code, the construct $\neg\ [\mathit{Code}]\neg\ (v_1' = v_1\ \mathit{and}\ \ldots\ \mathit{and}\ v_k' = v_k)$ can be used to extract a pre-post specification from Code, where $\mathit{wr}(\mathit{Code}) = \{v_1, ..., v_k\}$. Eg., $\neg\ [x := x * 5;\ x := x + 1]\neg\ (x' = x)$ is $x' = (x * 5) + 1$, ie., $x = (x@\mathit{pre} * 5) + 1$.

If P is of the form $S_i \rightarrow \mathit{forAll}(s \mid F(s))$ and $\mathit{rd}(P) \cap \mathit{wr}(P) \neq \{\}$, then $\mathit{stat}(P)$ is a general fixpoint iteration of the schematic form

```
while not(P)
do
  (select s : Si with not(F(s));
   stat(F(s)))
```

Thus any fixed-point iteration of this form, but with a more determinate means of iterating through the $s : S_i$, will abstract to P, provided that the iteration is guaranteed to reach all elements of S_i. Section 10 gives an example.

These techniques provide a basis for mapping hybrid or imperative MT code into MT specifications in the metamodels of Section 2.

9 Case study 1: UML to Java code synthesis

We consider a small fragment of this refinement transformation τ, to illustrate how the above verification techniques can be applied to such transformations. Figure 9 shows the parts of the source and target language metamodels which we consider here.

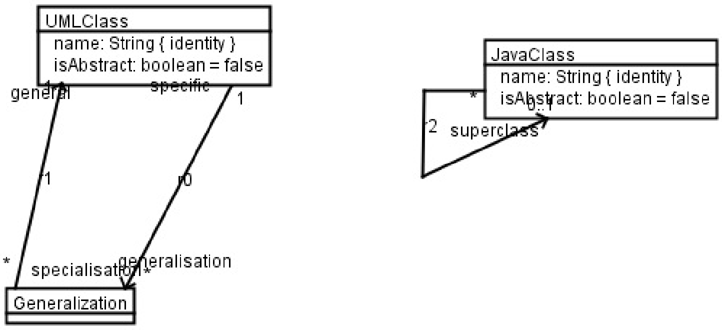
UMLClass
name: String { identity }
isAbstract: boolean = false
general
specific
1
1
1
0
*
specialisation
generalisation
*
Generalization
JavaClass
name: String { identity }
isAbstract: boolean = false
*
0..1
2
superclass

Fig. 9. UML and Java metamodels

The theory Γ_S of the source UML language includes the uniqueness property of class names. The assumptions $\mathit{Asm}0$ on the source model of the transformation include that there are no cases of multiple inheritance or cycles of inheritance in the source model. Asm is $\mathit{Asm}0$ together with the assumption that the target model is empty: $\mathit{JavaClass} = \mathit{Set}\{\}$, etc.

The required Java language properties Γ_T are that Java class names are unique and that there is no multiple inheritance or cycles of inheritance.

The transformation mappings (the specification postconditions Post of τ) express properties such as:

$$\begin{aligned}(R1) :\ & \mathit{UMLClass} \rightarrow \mathit{forAll}(c \mid \mathit{JavaClass} \rightarrow \mathit{exists}(cj \mid \\ & \quad cj.\mathit{name} = c.\mathit{name}\ \mathit{and}\ cj.\mathit{isAbstract} = c.\mathit{isAbstract}))\end{aligned}$$

and

$$\begin{array}{l}(R2) : \mathit{UMLClass}{\rightarrow}\mathit{forAll}(c, d \mid\ \ d : c.\mathit{generalisation}.\mathit{general}\ \mathit{implies} \\ \qquad\qquad\qquad\qquad\qquad \mathit{JavaClass}[d.\mathit{name}] : \mathit{JavaClass}[c.\mathit{name}].\mathit{superclass})\end{array}$$

defining how the poststate UML and Java models should correspond. $\mathit{JavaClass}[s]$ denotes the Java class with name s. This specification is written using the 'Map objects before links' pattern [38].

In $R1$ the mapping endpoints are the non-modifiable $c \in \mathit{UMLClass}$ from the UML input model of the transformation, and $cj \in \mathit{JavaClass}$ (modifiable) from the Java output model. Similarly c, d are input endpoints of $R2$.

Inv expresses conservativeness properties such as $(\mathit{Inv}1)$:

$$\begin{array}{l}\mathit{JavaClass}{\rightarrow}\mathit{forAll}(cj \mid \mathit{UMLClass}{\rightarrow}\mathit{exists}(c \mid \\ \qquad\qquad\qquad c.\mathit{name} = cj.\mathit{name}\ \mathit{and}\ c.\mathit{isAbstract} = cj.\mathit{isAbstract}))\end{array}$$

and $(\mathit{Inv}2)$:

$$\begin{array}{l}\mathit{JavaClass}{\rightarrow}\mathit{forAll}(cj, dj \mid\ \ dj : cj.\mathit{superclass}\ \mathit{implies} \\ \qquad\qquad\qquad \mathit{UMLClass}[dj.\mathit{name}] : \mathit{UMLClass}[cj.\mathit{name}].\mathit{generalisation}.\mathit{general})\end{array}$$

which are inverses of the corresponding postcondition Post properties.

For this transformation, the language interpretation mapping χ can be defined as:

$$\begin{array}{l}\mathit{UMLClass} \ \longmapsto\ \mathit{JavaClass} \\ \mathit{UMLClass} :: \mathit{name} \ \longmapsto\ \mathit{JavaClass} :: \mathit{name}\end{array}$$

This is a partial language morphism, as features such as *general* and *generalisation* have no interpretation in the target language.

An example of a semantic preservation proof is the preservation of the property φ that no class name contains the space character:

$$\mathit{UMLClass}{\rightarrow}\mathit{forAll}(c \mid \mathit{not}(\text{“ ”} : c.\mathit{name}{\rightarrow}\mathit{characters}()))$$

This translates to $\chi(\varphi)$:

$$\mathit{JavaClass}{\rightarrow}\mathit{forAll}(c \mid \mathit{not}(\text{“ ”} : c.\mathit{name}{\rightarrow}\mathit{characters}()))$$

By using $\mathit{Inv}1$ we can deduce this from the original property φ of UML classes. Notice that this proof is independent of the particular implementation chosen: any implementation that maintains Inv will also ensure semantic preservation of φ.

A specific implementation I of τ with behaviour $\mathit{stat}(R1);\ \mathit{stat}(R2)$ is chosen, which performs all computation steps $r1(c)$ of $R1$ before any computation step $r2(c, d)$ of $R2$. This implementation satisfies the semantic non-interference property of Section 8 because

$$\begin{array}{rcl}\mathit{rd}(R1) & = & \{\mathit{UMLClass}, \mathit{UMLClass} :: \mathit{name}, \mathit{UMLClass} :: \mathit{isAbstract}\} \\ \mathit{wr}(R1) & = & \{\mathit{JavaClass}, \mathit{JavaClass} :: \mathit{name}, \mathit{JavaClass} :: \mathit{isAbstract}\} \\ \mathit{rd}(R2) & = & \{\mathit{UMLClass}, \mathit{UMLClass} :: \mathit{name}, \mathit{UMLClass} :: \mathit{generalisation}, \\ & & \quad \mathit{JavaClass}, \mathit{Generalization} :: \mathit{general}\} \\ \mathit{wr}(R2) & = & \{\mathit{JavaClass} :: \mathit{superclass}\}\end{array}$$

so that $R1 < R2$ and $\mathit{stat}(R2)$ is syntactically non-interfering wrt $R1$.

The transformation step for $R1$ is:

$$\begin{array}{l}r1(c : \mathit{UMLClass}) \\ (cj : \mathit{JavaClass};\ cj.\mathit{name} := c.\mathit{name};\ cj.\mathit{isAbstract} := c.\mathit{isAbstract})\end{array}$$

and similarly for $r2$.

We can deduce that Inv is preserved by applications $r1$ of $R1$, from the form of $R1$: the newly created $cj : \mathit{JavaClass}$ produced by an application $r1(c)$ clearly satisfies $\mathit{Inv}1$, and it is not connected to any other Java class by *superclass* links, so *Inv*2 is preserved. Also, applications $r2$ of $R2$ preserve *Inv* since they do not create new Java classes, but only link existing instances in such a way that *Inv*2 is satisfied.

If no application of $R1$ or $R2$ is enabled, this means that the conclusions of these mappings are true for all source model elements that satisfy their assumptions, ie., that *Post* is true. Therefore, semantic correctness of the implementation I holds (also since this is the correctness-by-construction implementation).

Syntactic correctness of the transformation (using this implementation) means that it is guaranteed to produce syntactically valid Java programs, satisfying Γ_T, from valid UML models that satisfy *Asm*. Syntactic correctness follows from Γ_S, *Asm*, *Post* and *Inv*. For example, if there are two distinct Java classes $cj1$, $cj2$ with the same name in the target model, these must have been derived from the same UML class c (by *Inv*1 and the uniqueness of *name* for UML classes). But $R1$ implies that c is mapped to a unique Java class with name $c.name$. Likewise, if there is a situation of multiple or cyclic inheritance in the Java target model, by *Inv*2 there must be a corresponding situation of multiple inheritance or cyclic inheritance in the UML model which it was derived from, contradicting *Asm*. These proofs could be formalised using internal consistency proof in B (the stratification property does not hold for this specification, so a formalisation in Z3 may not be effective for proof).

Termination follows from the fact that the computations of both $R1$ and $R2$ are bounded iterations, over *UMLClass* and over *UMLClass* × *UMLClass*, respectively.

Clearly applications of $r1$ are order-independent, since different applications update entirely disjoint data items. Applications of $r2$ could interfere with each other, if there was multiple inheritance in the UML model, eg., distinct classes $d1$ and $d2$ in some $c.generalisation.general$: only one superclass $d1$ or $d2$ of c could be represented in the Java model. But *Asm* ensures that this cannot occur. Therefore confluence holds.

10 Case study 2: Re-expression of trees as graphs

This example is defined in ETL in [28] and illustrates how existing transformations can be verified by our techniques, by reverse-engineering the ETL implementation to a representation in the transformation metamodels of Section 2. ETL contains language mechanisms typical of hybrid transformation languages, such as ATL or GrGen, including essential use of transformation traces.

Figure 10 shows the source S (on the lhs) and target T (on the rhs) metamodels of this transformation.

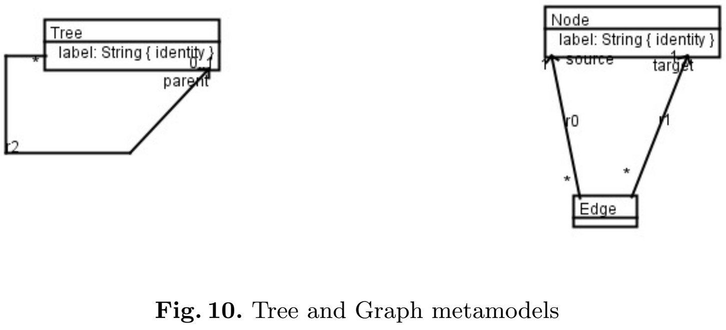

Fig. 10. Tree and Graph metamodels

The aim of the transformation is to re-express tree structures as graphs, with explicit graph edges in place of links from a tree instance to its parent.

An assumption *Asm*0 is that there are no cycles in the *parent* relation:

$$Tree{\rightarrow}forAll(t \mid t.parent{\rightarrow}closure(){\rightarrow}excludes(t))$$

It is also assumed that the target model is empty: $Edge = Set\{\}$ and $Node = Set\{\}$.

A required property of Γ_T is that there are no duplicate edges:

$$\begin{aligned}&\mathit{Edge}\rightarrow \mathit{forAll}(e1 \mid \mathit{Edge}\rightarrow \mathit{forAll}(e2 \mid \\ &\qquad e1.\mathit{source} = e2.\mathit{source}\ \mathit{and}\ e1.\mathit{target} = e2.\mathit{target}\ \mathit{implies}\ e1 = e2))\end{aligned}$$

An invariant *Inv* can be formulated, which expresses that the only nodes and edges in the target model are those (uniquely) derived from some source model elements:

$$\mathit{Node}\rightarrow \mathit{forAll}(n \mid \mathit{Tree}\rightarrow \mathit{exists}1(t \mid t.\mathit{label} = n.\mathit{label}))$$

$$\begin{aligned}&\mathit{Edge}\rightarrow \mathit{forAll}(e \mid \mathit{Tree}\rightarrow \mathit{exists}1(t \mid t.\mathit{parent}.\mathit{size} > 0\ \mathit{and} \\ &\qquad e.\mathit{source}.\mathit{label} = t.\mathit{label}\ \mathit{and}\ e.\mathit{target}.\mathit{label} = t.\mathit{parent}.\mathit{label}\rightarrow \mathit{any}()))\end{aligned}$$

The transformation is implemented in [28] using trace lookup and implicit rule invocation:

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node
  { n.label := t.label;
    if (t.parent.isDefined())
    { var edge := new Graph!Edge;
      edge.source := n;
      edge.target := t.parent.equivalent();
    }
  }
```

The $obj.\mathit{equivalent}()$ expression looks up in the transformation trace *Trace* to check if *obj* has already been mapped to a target element *tobj*, if so, it returns such an element, otherwise it invokes any applicable rules (in this case, *Tree*2*Node* itself) to map *obj* to a target element, which is then returned.

The semantics of the above rule can therefore be expressed in the activity language of our transformation implementation metamodel as a behaviour:

```
Tree2Node(t : Tree, nout : Node)
(if (Node.label->contains(t.label))
 then nout := Node[t.label]
 else
  (n : Node;
   n.label := t.label;
   if (t.parent.size > 0)
   (edge : Edge;
    edge.source := n;
    if Trace->exists( tr | tr.source = t.parent->any() )
    then
      edge.target := Trace->select( tr |
         tr.source = t.parent->any() )->collect( target )->any()
    else
      Tree2Node(t.parent->any(),edge.target)
   );
   nout := n
 )
)
```

This makes explicit the implicit recursive call in the ETL code. We assume that check-before-enforce semantics is used for the above rule, otherwise duplicate nodes and edges could be created by distinct calls of *Tree*2*Node* operating on the same tree branch.

The operation code constitutes computation steps for a specification postcondition *Post*0:

$$\begin{aligned}&\mathit{Tree}\rightarrow \mathit{forAll}(t \mid \mathit{Node}.\mathit{label}\rightarrow \mathit{excludes}(t.\mathit{label})\ \mathit{implies} \\ &\quad (t.\mathit{parent}.\mathit{size} = 0\ \mathit{implies}\ \mathit{Node}\rightarrow \mathit{exists}(n \mid n.\mathit{label} = t.\mathit{label}))\ \mathit{and} \\ &\quad (t.\mathit{parent}.\mathit{size} > 0\ \mathit{implies}\ \mathit{Node}\rightarrow \mathit{exists}(n \mid n.\mathit{label} = t.\mathit{label}\ \mathit{and} \\ &\qquad \mathit{Edge}\rightarrow \mathit{exists}(\mathit{edge} \mid \mathit{edge}.\mathit{source} = n\ \mathit{and} \\ &\qquad\qquad \mathit{edge}.\mathit{target} = \mathit{Node}[t.\mathit{parent}.\mathit{label}]\rightarrow \mathit{any}()))))\end{aligned}$$

This is a type 3 constraint, with an implementation as a fixpoint iteration $stat(Post0)$ which iterates the quantified body of $Post0$ until no tree remains without a matching node. This can be compared to the ETL $Tree2Node(t, n)$ which performs corresponding actions for t and all ancestors of t. Unlike $stat(Post0)$ it uses a fixed order of iteration, from descendants to parents. $stat(Post0)$ should compute the same target model as $Tree2Node$ applied to each leaf node of the tree(s) in the source model.

To verify properties of such recursive computations, induction on call depth can be used. That is, assuming that the call $Tree2Node(t.parent{\rightarrow}any(), edge.target)$ creates a correct target model graph for the tree structure at and above $t.parent$, we can argue that $Tree2Node(t, nout)$ does so for the tree structure at and above t.

Termination can be shown by arguing that every call of $Tree2Node$ terminates: any chain of *parent* links must be finite since loops are forbidden by Asm and because models are finite.

However the implementation appears excessively complex, and in order to produce an improved and more easily verified version of this transformation, we can use the correctness-by-construction technique of Section 8, starting from a specification with the following two postcondition constraints $Post1$:

$$Tree{\rightarrow}forAll(t \mid Node{\rightarrow}exists(n \mid n.label = t.label))$$

to relate trees to nodes, and $Post2$:

$$\begin{array}{l} Tree{\rightarrow}forAll(t \mid t.parent.size > 0 \ implies \\ \qquad Edge{\rightarrow}exists1(e \mid e.source = Node[t.label] \ and \\ \qquad\qquad e.target = Node[t.parent.label]{\rightarrow}any())) \end{array}$$

to relate *parent* links to edges. These are both localised type 1 constraints.

This new specification is an example of the 'Map objects before links' specification pattern [38], which should be used in such cases of recursive structures in the source metamodel.

The constraints satisfy the syntactic non-interference property if ordered as above, and are both of type 1, satisfying internal syntactic non-interference, so a semantically correct, terminating and confluent implementation can be automatically synthesised as $stat(Post1)$; $stat(Post2)$:

```
for t : Tree do delta1(t);
for t : Tree do delta2(t)
```

where $delta1$ is:

```
delta1(t : Tree)
(n : Node;
 n.label := t.label)
```

and $delta2$ is:

```
delta2(t : Tree)
(if t.parent.size > 0
 then
   if Edge->exists( e | e.source = Node[t.label] and
                        e.target = Node[t.parent.label]->any() )
   then skip
   else
     (e : Edge;
      e.source := Node[t.label];
      e.target := Node[t.parent.label]->any()
     )
)
```

In this case we can prove that the invariant properties hold: after any sequence of transformation steps, each existing target model node must be derived from exactly one source model tree, and similarly for edges. Duplicate edges cannot occur because $Post$ and Inv together imply that two edges $e1$ and $e2$ with common source nodes and common target nodes, must be derived from a single tree t with a unique parent $t1$ and (from the second postcondition), that therefore $e1 = e2$. Therefore syntactic correctness holds.

11 Case study 3: Refactoring class diagrams

This update-in-place transformation rationalises a class diagram by removing duplicate copies of attributes from sibling classes [27]. Figure 11 shows the metamodel of the single language of this transformation.

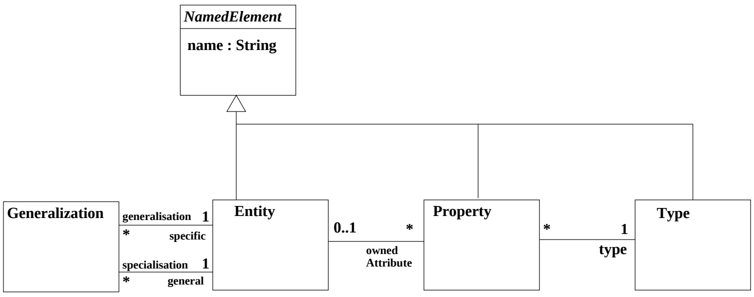

Fig. 11. Basic class diagram metamodel

There are three mapping rules, of which the simplest is rule 1: **Pull up common attributes of all direct subclasses:** If the set $g = c.specialisation.specific$ of all direct subclasses of a class $c : Entity$ has two or more elements, and all classes in g have an owned attribute with the same name n and type t, add an attribute of this name and type to c, and remove the copies from each element of g (Figure 12).

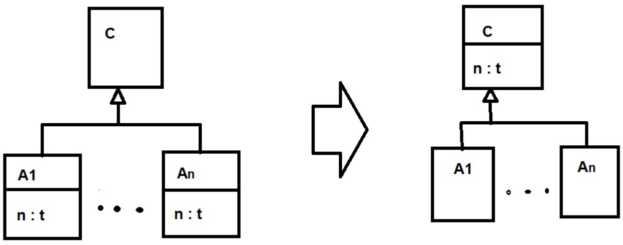

Fig. 12. Rule 1 specification

To prove syntactic correctness, the following properties need to be established for the termination state of the transformation:

1. Single inheritance: $generalisation.size \leq 1$ for all classes.
2. No duplicated attribute names in classes: $allAttributes \rightarrow isUnique(name)$ where $allAttributes$ is defined recursively as

$$allAttributes = ownedAttribute \cup generalisation.general.allAttributes$$

These constraints can also be assumed as the preconditions Asm of the transformation.

Additionally, there is a postcondition $Post$ that there should be no cases of attributes satisfying the application condition of Rule 1 (Figure 12) in the termination state.

A model-level semantic preservation property is that the semantics of the transformed model must be equivalent to the semantics of the source model: where the semantics is the set of possible collections of objects which could exist for the leaf classes $e : Entity$ in the model, ie., if:

$$sem(m) = \{possible\ object\ configurations\ for\ leaf\ classes\ of\ m\}$$

then we need to show $sem(m) = sem(n)$. This can be internalised as the invariant that $c.allAttributes$ for each leaf $c \in Entity$ is not essentially changed by the transformation steps (the names and types are preserved, although not the exact *Property* objects), nor is the set of leaf classes; $Equiv1$:

$$\begin{aligned} &Entity{\rightarrow}select(e \mid e.specialisation.size = 0) \ = \\ &\qquad\qquad Entity@pre{\rightarrow}select(e \mid e.specialisation@pre.size = 0) \end{aligned}$$

and $Equiv2$:

$$\begin{aligned} &Entity{\rightarrow}forAll(e \ \mid \ e.specialisation.size = 0 \ implies \\ &\qquad\qquad\qquad\qquad e.allAttributes \approx e.allAttributes@pre) \end{aligned}$$

where $atts1 \approx atts2$ is $atts1{\rightarrow}forAll(p1 \mid atts2{\rightarrow}exists(p2 \mid p2.name = p1.name \ and \ p2.type = p1.type)) \ and \ atts2{\rightarrow}forAll(p2 \mid atts1{\rightarrow}exists(p1 \mid p1.name = p2.name \ and \ p1.type = p2.type))$.

The GrGen.NET implementation of rule 1, from [27], is as follows:

```
rule rule1 {
  c:Class;
  :SuperOf(c,g1); :SuperOf(c,g2);
  g1:Class -:ownedAttribute-> a1:Property -:type-> t:Type;
  g2:Class -:ownedAttribute-> a2:Property;
  :SameAttribute(a1,a2);
  negative {
    g3:Class;
    :SuperOf(c,g3);
    g1;
    negative {
      g3 -:ownedAttribute-> a3:Property;
      :SameAttribute(a1,a3);
    }
  }
  modify
    c -:ownedAttribute-> a4:Property -:type-> t;
    eval {
      a4._name = a1._name;
    }
    exec(RemoveAttributeFromSubclasses(c, a4) ;> [createInverseEdges]);
  }
}
```

Class is used instead of *Entity* here, for consistency with the UML metamodel.

In the *exec* clause, applications of *RemoveAttributeFromSubclasses* and *createInverseEdges* are explicitly chained after the main rule by means of invocation.

This rule implementation can be abstracted to the metamodel of Figure 7 in a similar way to the ETL example of Section 10:

```
rule1(c : Class)
(if there exist g1, g2, a1, a2, t satisfying
    the guard conditions
 then
    select such g1, g2, a1, a2, t;
    (a4: Property;
     a4.type := t;
     c.ownedAttribute := c.ownedAttribute->including(a4);
     a4._name := a1._name;
     RemoveAttributeFromSubclasses(c,a4))
)
```

In this case, *createInverseEdges* has no actions to perform, so is omitted.

To establish the properties *Asm* in the post-state, one technique, as described in Section 4, is to show that these are invariants of the transformation, ie., they are preserved by each transformation computation step. We can either prove the preservation at the specification level using the specifications (eg., using a mapping relation based on Figure 12) of the steps, and then show semantic correctness of the step implementations wrt the step specifications, or prove directly the preservation for each step implementation.

For the GrGen implementation we take the second option, using the representation $rule1$ of the implementation in the metamodel of Figure 7. Since the $rule1$ code does not modify *generalisation*, the first Asm property is trivially preserved. For the second, the new attribute $a4$ with name $a1.\_name$ is introduced into c, in the *modify* clause. However, in the invoked $RemoveAttributeFromSubclasses(c, a4)$ operation, any attribute with the same name as $a4$ is removed from each class in $c.specialisation.specific$. The property 2 is therefore maintained, since *allAttributes* is not otherwise modified for any class by the rule implementation.

A formal proof of this argument could be constructed by translating the GrGen rule implementations to operations of a B machine, and expressing the properties as invariants of this machine. Internal consistency proof of the machine will include the invariance of the properties over transformation steps.

Termination follows since the number of *Property* instances in the model is strictly decreased by each rule application, ie, $Property.allInstances()\rightarrow size()$ gives an upper bound for a variant of the transformation.

At termination of the implementation, $rule1$ cannot be applied, ie., there are no cases of classes c satisfying the application conditions of $rule1$. But this implies that there are no cases of same-named and -typed attributes in all (at least 2) direct subclasses of a class, as required.

Confluence does not hold, since alternative orders of applications of some rules (rules 2 and 3 of [27]) may produce non-isomorphic terminal models.

For model-level semantic correctness, we can prove that the above predicates $Equiv1$ and $Equiv2$ are invariant. They are clearly true initially, and rule 1 preserves the set of leaf classes and the total set of $(name, type)$ pairs of their *allAttributes* properties. Thus the semantics of the model is preserved.

An example of a heterogeneous system of transformations could be the transformation of case study 3 followed by that of case study 1: overall semantic correctness of this sequential composition would follow from the correctness of the individual transformations, and by the establishment of the preconditions of the UML-to-Java transformation by the refactoring transformation. Likewise, overall termination holds.

12 Evaluation

Our approach has been implemented using the UML-RSDS language and toolset [38, 49]. Transformation specifications are defined by UML use cases, with preconditions, invariants and postconditions, whilst designs are defined using activities in the statement language presented here. Syntactic analysis is performed on the specifications, and these can be automatically translated to Z3 and B AMN for semantic analysis. Designs are generated from specifications using the approach of Section 8, executable code in Java is automatically synthesised from the designs. Thus all steps of the general process of Figure 8 have been implemented except for the reverse-engineering of existing transformations into language-independent representations. Currently we use a manual process to perform this step, but automation (for ETL and GrGen) is being investigated. The techniques defined here have been applied to many cases of transformation verification. Examples include the class diagram refactoring case study of [27], the computation of the transitive closure of a relation [39], a large-scale migration transformation [33], and the slicing of state machines [34]. The example of [27] is a semantically complex update-in-place transformation (of which the simplest rule is discussed in Section 11). We were able to prove termination, syntactic correctness and semantic correctness for the UML-RSDS implementation using manual proof with less than 1 person day effort. The case study of [39] was formally proved using proof in B to establish termination and confluence. This required approximately 5 person days of interactive proof, primarily concerning the refinement obligations to establish the necessary variant properties. The example of [33] has 66 entity types and features, and represents a realistic migration problem. We were able to identify failures of semantic preservation and syntactic correctness in the proposed migration mapping, and to establish termination, confluence and semantic correctness by syntactic analysis. The effort required was approximately 3 person days. The state machine slicing algorithms of [34] form part of a large and complex software engineering tool for model slicing. The model-level semantic preservation of the slicing algorithms was shown by induction over the individual transformation steps which rewrite state machines into simpler forms. This involved approximately 5 person days of manual proof.

We have found that the organisation of proof steps described in Section 4 are generally very effective in carrying out manual or tool-supported proof: the verification effort is broken down into separate verification of the transformation invariants and variants, relative to a given implementation, and then verification of syntactic and semantic correctness and semantic preservation using these invariants and variants. Proof of syntactic correctness and semantic preservation can in some cases be carried out independently of particular implementations by relying instead upon the invariant properties. This permits reuse of proof effort, if the implementation is changed, provided the modified implementation also maintains the invariants.

Table 11 shows examples of the extent of automation of internal consistency and refinement proof using B for some transformations. Even for internal consistency proof, the proof effort is higher for the transformations (the computation of transitive closure, and the balancing of binary trees) which use type 2 or 3 constraints, compared to those using only type 1 constraints.

Case study	*Total proof obligations*	*Auto-proved*	*Interactively proved*	*Auto-proof percentage*
Transitive closure				
(internal consistency)	17	11	6	65%
(refinement)	48	26	22	54%
Balancing binary trees				
(internal consistency)	29	19	10	66%
(refinement)	47	27	20	57%
UML to RDB				
(internal consistency)	31	26	5	84%
A to B				
(internal consistency)	24	18	6	75%

Table 11. B proof automation on case studies

In order to minimise the proof effort required for verification, we recommend the combination of syntactic analysis, satisfaction checking and correctness-by-construction synthesis to develop new transformations. This requires restrictions on the form of the transformation specification, ie., that these should satisfy syntactic or semantic non-interference, but most practical cases of refinements, migrations, re-expressions and other separate-models transformations can be defined to satisfy these restrictions. The Conjunctive-implicative pattern, and related patterns, such as the Map objects before links pattern, described in [38] are recommended for the structuring of transformation specifications and implementations, in order to reduce proof effort. In contrast, recursive descent structuring, or the use of implicit rule invocation via a mechanism such as ETL *equivalent*/*equivalents* greatly complicate verification.

Even in cases where semantic non-interference does not hold, as in the case study of Section 11 (where applications of rule 3 can introduce new cases of classes to which rule 1 can be applied, and vice-versa), correctness-by-construction synthesis can still be used to generate a semantically correct implementation from the transformation specification [38]. Proof of the variant property for designated variant functions will however be necessary to ensure termination, and likewise proof will be required to establish confluence, if this holds.

Limitations of our approach are the reliance upon the serialisation of implementations, in order to carry out weakest precondition reasoning, and the problems of scale encountered with large low-level transformation implementations, such as the Kermeta implementation in [27], which we were unable to verify.

13 Related work

In [38] we introduced the correctness-by-construction MT approach using UML-RSDS, and we described design patterns which can be used to define modular, efficient and verifiable transformations. In [36] an overview of verification techniques for UML-RSDS is given. In the present paper, we provide detailed semantic foundations for MT verification, not specific to UML-RSDS, and define in detail MT language-independent verification techniques using syntactic analysis, satisfaction checking and theorem proving.

In [25], metamodels for the requirements, specification and design of model transformations are introduced, together with a process for the development of transformations using these languages. We follow the approach of [25], and extend this to deal with the generation of verification conditions from transformation specifications, and the mapping of transformation specifications to verification formalisms. In contrast to the *transML* approach, we take advantage of the many similarities between MT languages to define a language-independent transformation implementation representation, to avoid a multiplicity of metamodels for individual languages. Here we have focussed upon the high-level design and low-level design stages of [25]. In future work we intend to integrate the techniques of this paper with the other elements of the *transML* approach of [25].

In this paper we have primarily used the idea of a verification or transformation model in order to perform semantic analysis of a model transformation. The paper [32] introduces one of the first attempts to use the verification model approach: relational transformation specifications are formalised in the B AMN specification language, which is then used to prove syntactic correctness of the transformations. The concept of verification model is described (as 'transformation models') in [8], and much subsequent work on transformation verification has used this approach. Eg., [14] shows how QVT-R and TGG specifications can be mapped to a verification model consisting of OCL formulae, which captures the semantics of the specifications. This verification model can then be analysed using any OCL tool. This work has subsequently been extended to consider a declarative subset of ATL [12, 13]. In contrast to our approach, the verification model of [13] expresses only the intended post-state of the transformation, and does not represent the transformation steps or dynamic behaviour of the transformation. It is therefore less appropriate for proof of invariance, termination or confluence properties. Invariant-based properties such as model-level semantic preservation/equivalence seem to require modelling of transformation steps, and not simply the termination states of a transformation. The approach of [13] appears restricted to type 1 constraints in our terms.

Other related work is [9], which uses a verification model based on rewriting logic to analyse QVT-like transformations, and [16], which maps ATL into a verification model based on the Coq theorem prover. Alloy has been used to analyse UML and OCL specifications and model transformations in QVT [3, 2]. Alloy provides bounded satisfaction-checking capabilities, but in a restricted relational language, which limits the forms of transformation specification which can be analysed. Eg., nested collections cannot be represented. Translations from model transformation languages to different formalisms have also been used to perform termination analysis [50, 46], proof of syntactic correctness [23], counterexample generation [13] and proof of semantic preservation [40]. Table 12 summarises such approaches. It can be seen that these deal primarily with declarative transformation languages, and are often limited to restricted subsets of these and to specific verification properties.

In this paper we extend the verification model concept to cover all elements of a practical transformation language (UML-RSDS), and to include hybrid and imperative model transformation languages. We represent transformations from multiple transformation languages in a common representation and use mappings from these metamodels of transformation specifications and implementations to verification models in different formalisms (such as B, Z3, etc) to support the verification of a range of verification properties. This enables analysis of transformations expressed in a wide range of transformation languages, and of transformation systems involving multiple transformation languages. In contrast to [13], our representation of transformations includes behavioural details (Figure 7), enabling us to represent hybrid and imperative transformations, and to reason about transformation executions using induction over computation steps.

We have also given detailed language-independent verification techniques, and we have defined criteria that semantic mappings to verification formalisms should ideally satisfy, ie., they should be institution comorphisms. Institutions and institution comorphisms are used as a basis to support the use of multiple logical frameworks in the HETS environment [41]. It is possible to construct such morphisms based upon our mappings to Z3 and B AMN, these morphisms satisfy the soundness direction of the co-morphism property (that validity of a translated property in the verification

Approach	*Transformation languages*	*Formalisms*	*Analyses*	*Mappings*	*Results*
Asztalos et. al. [4]	VMTS	MCDL, MCIL	semantic correctness	Automated	Automated proof that implementation achieves required postconditions
Becker et. al. [6, 7]	Graph transformation	symbolic verifier	avoidance of invalid states	Automated	invariance proof/ counterexamples
Boronat et al [9]	QVT-R subset	Maude	invariance proof, counterexamples	Manual	Partial invariance proof, counterexample generation
Buttner et. al. [13]	Declarative ATL	OCL, Alloy	syntactic correctness (counterexamples)	Automated	Partial counterexample generation for syn. corr. of declarative ATL
Calegari et. al. [16]	ATL subset	CIC, Coq	syntactic correctness	Manual	Interactive proof of syn. corr.
Giese et. al. [20]	TGG	Isabelle/ HOL	model-level semantic equivalence	Manual	Manual proof of semantic equivalence
Inaba et. al. [23]	UnCAL subset + annotations	Monadic 2nd order logic	syntactic corr., counter-example generation	Automated	Automated decision procedure for syn. corr. of restricted UnCAL transformations
Massoni et. al. [40]	none	Alloy, PVS	example generation, proof	Manual, restricted	Proves simple UML refactorings semantically preserving
Rensink et. al. [46] checkVML	GXL	Promela, SPIN	invariance, termination	Automated	bounded model-checking
GROOVE	GXL	state-transition system	invariance, termination	Automated	bounded model-checking
Stenzel et. al. [48]	QVT-O	KIV	syn. corr., model-level sem. pres.	Manual	Interactive proof of syn. corr., model-level sem. pres.
Varro et. al. [50]	graph transform- ations	Petri Nets	termination	Manual Not exact semantically	Partial termination analysis of graph transformations with NACs

Table 12. Verification model approaches

model implies validity of the property in the original OCL/FOL representation), but not the converse direction, due to incompleteness of the Z3 and B formalisms.

Model-level semantic correctness (also known as model-level semantic preservation [35]), that is, the preservation of the internal semantics of models by a transformation, is specifically considered by several works. The paper [48] considers this verification property for QVT-O transformations, and uses a dynamic-logic based formalism to verify that certain semantic properties of source models are preserved in the target models. Our approach potentially facilitates such verification, because the *Post* and *Inv* predicates of a transformation $\tau : S \rightarrow T$ precisely characterise how a target model n relates to its source m. A language-level interpretation χ also characterises how the language elements of S are expressible in terms of T. The derived mapping $Mod(\chi)$ of models supports analysis of the semantic preservation of static model-level properties relative to χ, and formal proof of such preservation can be carried out by inductive proof using the B representation of transformations, provided that the semantics is representable in B [39]. The concept of a transformation invariant is applicable to proof of model-level semantic preservation, ie., an invariant can express that semantic preservation holds for all target elements created up to an arbitrary intermediate stage of the transformation, or for all restructurings of the model so far carried out – in each case these are inductions over transformation steps. An attempt at using such invariant-based reasoning is made in [48], however this is complicated by the recursive descent style of implementation used for the transformation, resulting in a complex invariant and proof task. The approach of [48] relies upon a strict composition hierarchy in models, which we do not require for invariant-based reasoning. The paper [1] also considers model-level semantics preservation, for a code-generation transformation. Proof is not used, instead model-checking is used to confirm that required properties are preserved for specific models. Likewise, in [42, 43] preservation of semantic properties on a model-by-model basis is verified by using a structural correspondence (a bisimulation) between the source and target models. An invariant-based approach is used to verify model-level semantic equivalence in [20], however they use manual proof in Isabelle/HOL to show invariance of semantic equivalence under transformation steps, whilst a formalisation using B can potentially automate more of the proof effort.

There has been considerable work on the formalisation of OCL and UML for semantic analysis [2, 10, 11, 47]. The complexity of modelling OCL *null* and *invalid*, together with inconsistencies in the OCL semantic definitions of these values, means that systems such as [11] must make additional assumptions to provide a reasoning framework for full OCL. Instead, we choose to rule out such values completely and to work with classical logic, which has the advantage of unambiguity and the existence of many powerful proof and analysis tools. We agree with [11] that OCL *invalid* should not be used for modelling, but we would go further and forbid the use of *null* also. By considering that 0..1 multiplicity association ends are collection-valued (of size 0 or 1), a simple and uniform treatment of many data structure properties can be given, without the need to test for *null* values.

OCL was originally intended as a language for logical expression evaluation, to express declarative constraints of UML models. However, transformation languages such as QVT and Kermeta now use OCL to define executable effects, to update models. The imperative use of OCL as a programming language introduces semantic problems, for example, how a *forAll* or other iteration should behave if (logically) its evaluation can be terminated before all elements are processed, whilst (imperatively) some intended side-effect behaviour could still arise if additional elements are considered. We rule out such cases by clearly distinguishing the logical and imperative use of OCL. Table 10 defines precisely when an expression E is interpreted imperatively, ie., as $stat(E)$. We exclude side-effects when an expression is used logically. Thus the logical evaluation of iterators can be made more efficient. In an imperative interpretation $stat(E)$ of expression E, however, the computation only terminates when E is established (which may require fix-point iteration, etc.).

For example, logical evaluation of $s \rightarrow forAll(x \mid x.att = v)$ can return *false* as soon as some $x \in s$ with $x.att \neq v$ is found, and no updates are performed. But $stat(s \rightarrow forAll(x \mid x.att = v))$ is $for\ x : s\ do\ x.att\ := \ v$ and always iterates completely through all elements of s, assigning v to $x.att$ for each $x \in s$. The logical and imperative interpretations are related naturally by the property

$$def(E) \ \Rightarrow \ [stat(E)]E$$

This relationship also forms the basis for reverse-engineering existing transformation code in hybrid or imperative languages to specifications.

Conclusions

We have shown how a systematic language-independent framework and techniques for model transformation verification can be given, and we have illustrated the use of these techniques on representative case studies of different kinds of transformation. A significant benefit of formally modelling transformation specifications, implementations and verification properties, is that the mapping of these into verification formalisms such as B or Z3 can be automated. Such mappings have been incorporated into the UML-RSDS tools [49].

We have shown the problems which arise with regard to verification if transformation implementations use techniques such as recursive descent or implicit rule invocation. We therefore recommend that transformations are structured in accordance with patterns such as Conjunctive-implicative form and Map objects before links in order to make verification feasible.

Acknowledgements

Richard Paige of York University and Steffen Zschaler of King's College London have contributed to the ideas presented here. The ETL specification presented in Section 10 is due to Kolovos et al., and appeared in [28]. The GrGen specification presented in Section 11 is due to Pieter Van Gorp and appeared in [27].

References

1. L. Ab Rahim, J. Whittle, *Verifying semantic conformance of state machine-to-Java code generators*, MODELS 2010, LNCS, 2010.
2. K. Anastasakis, B. Bordbar, J. Kuster, *Analysis of Model Transformations via Alloy*, Modevva 2007.
3. K. Anastasakis, B. Bordbar, G. Georg, I. Ray, *On challenges of model transformation from UML to Alloy*, Software Systems Modelling, vol. 9, no. 1, 2010.
4. M. Asztalos, P. Ekler, L. Lengyel, T. Levendovszky, G. Mezei, T. Meszaros, *Automated verification by declarative description of graph rewriting-based model transformations*, MPM 2010.
5. T. Baar, *The definition of transitive closure in OCL: limitations and applications*, EPFL, Switzerland.
6. B. Becker, D. Beyer, H. Giese, F. Klein, D. Schilling, *Symbolic invariant verification for systems with dynamic structural adaption*, ICSE 2006, ACM Press.
7. B. Becker, L. Lambers, J. Dyck, S. Birth, H. Giese, *Iterative development of consistency-preserving rule-based refactorings*, ICMT 2011, LNCS vol. 6707, 2011.
8. J. Bezivin, F. Buttner, M. Gogolla, F. Jouault, I. Kurtev, A. Lindow, *Model Transformations? Transformation Models!*, ATLAS group, University of Nantes, 2006.
9. A. Boronat, R. Heckel, J. Meseguer, *Rewriting logic semantics and verification of model transformations*, FASE 2009, pp. 18–33, 2009.
10. A. Brucker, B. Wolff, *The HOL-OCL book*, Technical report 525, ETH Zurich, 2006.
11. A. Brucker, M. Krieger, B. Wolff, *Extending OCL with null-references*, MODELS 2009 Workshops, LNCS 6002, pp. 261–275, 2010.
12. F. Buttner, J. Cabot, M. Gogolla, *On validation of ATL transformation rules by transformation models*, MoDeVVa 2011.
13. F. Buttner, M. Egea, J. Cabot, M. Gogolla, *Verification of ATL transformations using transformation models and model finders*, ICFEM 2012.
14. 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.
15. 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.
16. D. Calegari, C. Luna, N. Szasz, L. Tasistro, *A type-theoretic framework for certified model transformations*, in FM 2011, LNCS vol. 6527, pp. 112–127, 2011.
17. Drey, Z., Faucher, C., Fleurey, F., Mahe, V., Vojtisek, D., *Kermeta Language Reference Manual*, https://www.kermeta.org/docs/KerMeta-Manual.pdf, April, 2009.

18. H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer, *Information preserving bidirectional model transformations*, FASE 2007, pp. 72–86, 2007.
19. FAA, DO-178C, Software considerations in airborne systems and equipment certification, January 2012.
20. H. Giese, S. Glesner, J. Leitner, W. Shafer, R. Wagner, *Towards verified model transformations*, proceedings of 3rd international workshop on model-driven engineering, verification and validation (MODEVVA), 2006.
21. J. Goguen, R. Burstall, *Institutions: abstract model theory for specification and programming*, Journal of the ACM, 39: 95–146, 1992.
22. 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.
23. K. Inaba, S. Hidaka, Z. Hu, H. Kato, K. Nakano, *Graph-transformation verification using monadic second-order logic*, PDPP '11, 2011.
24. F. Jouault, I. Kurtev, *Transforming Models with ATL*, in MoDELS 2005, LNCS Vol. 3844, pp. 128–138, Springer-Verlag, 2006.
25. E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, O. Marchi dos Santos, *trans*ML: A Family of Languages to Model Model Transformations, MODELS 2010, pages 106–120, Springer-Verlag, LNCS volume 6394, 2010.
26. E. Jakumeit, S. Buchwald, M. Kroll, *GrGen.NET: the expressive, convenient and fast graph rewrite system*, International Journal on Software Tools for Technology Transfer (STTT), 12: 263–271, 2010.
27. 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.
28. D. S. Kolovos and R. F. Paige and F. Polack, *The Epsilon Transformation Language*, ICMT, 2008, pp. 46–60.
29. M. Kuhlmann, M. Gogolla, *From UML and OCL to relational logic and back*, MODELS 2012, Springer LNCS, vol. 7590, 2012.
30. J. Kuster, *Definition and validation of model transformations*, SoSyM vol. 5, no. 3, pp. 233–259, 2006.
31. K. Lano, *The B Language and Method*, Springer-Verlag, 1996.
32. K. Lano, *Using B to Verify UML Transformations*, MODEVA 06, 2006.
33. K. Lano, S. Kolahdouz-Rahimi, *Migration case study using UML-RSDS*, TTC 2010, Malaga, Spain, July 2010.
34. K. Lano, S. Kolahdouz-Rahimi, *Slicing techniques for UML models*, Journal of Object Technology, vol. 10, 2011.
35. K. Lano, S. Kolahdouz-Rahimi, I. Poernomo, *Comparative evaluation of model transformation specification approaches*, International Journal of Software Informatics, Vol. 6, Issue 2, 2012.
36. K. Lano, S. Kolahdouz-Rahimi, T. Clark, *Comparison of model transformation verification approaches*, Modevva workshop, MODELS 2012.
37. K. Lano, S. Kolahdouz-Rahimi, *Model-driven development of model transformations*, ICMT 2011, 2011.
38. K. Lano, S. Kolahdouz-Rahimi, *Constraint-based specification of model transformations*, Journal of Systems and Software, to appear, 2012.
39. K. Lano, S. Kolahdouz-Rahimi, T. Clark, *Verification of model transformations*, Dept. of Informatics, King's College London, 2012.
40. T. Massoni, R. Gheyi, P. Borba, *Formal refactoring for UML class diagrams*, 19th Brazilian symposium on Software Engineering, 2005.
41. T. Mossakowski, C. Maeder, K. Luttich, *The Heterogeneous Tool Set*, University of Bremen, Germany, 2012.
42. A. Narayanan, G. Karsai, *Towards verifying model transformations*, GT-VMT 2006, ENTCS, 2006.
43. A. Narayanan, G. Karsai, *Verifying model transformations by structural correspondence*, GT-VMT 2008.
44. OMG, *Object Constraint Language v2.3.1 Specification*, formal/2012-01-02, 2012.
45. I. Poernomo, J. Terrell, *Correct-by-construction Model Transformations from Spanning tree specifications in Coq*, ICFEM 2010.
46. A. Rensink, A. Schmidt, D. Varro, *Model checking graph transformations: A comparison of two approaches*, ICGT 2004, LNCS vol. 3256, 2004.
47. M. Soeken, R. Wille, R. Dreschsler, *Encoding OCL data types for SAT-based verification of UML/OCL models*, University of Bremen, 2012.
48. K. Stenzel, N. Moebius, W. Reif, *Formal verification of QVT transformations for code generation*, MODELS 2011, Springer LNCS vol. 6981, 2011.
49. UML-RSDS toolset and manual, `http://www.dcs.kcl.ac.uk/staff/kcl/uml2web/`, 2013.
50. D. Varro, S. Varro-Gyapay, H. Ehrig, U. Prange, G. Taentzer, *Termination analysis of model transformations by Petri Nets*, ICGT 2006, LNCS vol. 4178, 2006.
51. Z3 Theorem Prover, http://research.microsoft.com/en-us/um/redmond/projects/z3/, 2012.

A Structure isomorphism

As described in Section 2, a structure m for a language L can be considered to be a tuple $((E_1{}^m, \ldots, E_k{}^m), (f_1^m, ..., f_l^m))$ of the sets $E_i{}^m$ of instances of each entity type E_i of L, and of the maps $f_j^m : E_i{}^m \rightarrow \mathit{Typ}$ representing the values of the data features of these entity types.

Structures are considered isomorphic if they cannot be distinguished on the basis of feature values. Let $h_i : E_i{}^m \rightarrow E_i{}^{m'}$ be a family of bijections between the entity type instance sets for two structures m and m' of the same language L. Then m and m' are isomorphic $m \simeq m'$ if:

1. $h_i(x) = x' \;\Rightarrow\; f_j^m(x) = f_j^{m'}(x')$ for each attribute feature f_j of E_i, $x \in E_i{}^m$
2. $h_i(x) = x' \;\Rightarrow\; h_k(f_j(x)) = f_j^{m'}(x')$ for each single-valued role feature $f_j : E_i \rightarrow E_k$ of E_i, $x \in E_i{}^m$
3. $h_i(x) = x' \;\Rightarrow\; h_k(\!|\, f_j(x) \,|\!) = f_j^{m'}(x')$ for each set-valued role feature $f_j : E_i \rightarrow \mathit{Set}(E_k)$ of E_i, $x \in E_i{}^m$.
4. $h_i(x) = x' \;\Rightarrow\; f_j(x);\ h_k = f_j^{m'}(x')$ for each sequence-valued role feature $f_j : E_i \rightarrow \mathit{Sequence}(E_k)$ of E_i, $x \in E_i{}^m$.

This has the consequence that m and m' satisfy the same sentences of $\mathit{Sen}(L)$.

B Notations and logical foundations

In this paper we use a number of different formal notations: internally within transformations and their input and output languages L we use an OCL-like notation, eg.:

$A{\rightarrow}\mathit{forAll}(a \mid a.x > 0)$

However by the semantic mapping of a language L to a first-order set theory (FOL) language $\mathcal{L}_L$, such constraints can be considered as being simply alternative syntax for conventional logical formulae, eg.:

$\forall\, a : A \cdot x(a) > 0$

These alternative notations have no semantic distinction. Likewise we treat the usual model transformation concept of 'model', ie., an instance of a modelling language defined by a metamodel, as equivalent to the FOL concept of a mathematical model and interpretation for a FOL theory. Unlike [5], we consider that models are implicitly infinite. Models have finite interpretations for all entity types, but depend upon finite set theory and also upon infinite sets such as interpretations of the set of integers. Thus even the $\varnothing$ structure for a language is infinite. The concept of a structure, as an interpretation for the symbols of a language which does not necessarily satisfy all the language constraints, is also considered equivalent for OCL and FOL.

We also use FOL notation for metalogical reasoning and definitions.

The Godel Completeness Theorem holds for first order set theory: every consistent set of sentences of $\mathcal{L}_L$ has a model (which will be infinite, since all models contain an interpretation of $\mathbb{N}$). Thus if a property φ is valid in all models of a consistent theory Γ_L, it must be provable from Γ_L.

Z3 and B AMN have distinct notations for logic and set theory. Only a subset of the OCL/FOL notations can be expressed in a semantically equivalent manner in Z3 and B AMN, and we have explicitly identified in Sections 6 and 7 where these issues arise. For Z3, numeric types and operators can be considered equivalent to the OCL/FOL forms, eg., Z3 Int expresses OCL Integer and FOL $\mathbb{Z}$. The Z3 syntax

```
(forall ((a A)) (> (x a) 0))
```

equivalently expresses the example constraint. However, OCL sets are modelled by Z3 lists, so that operators such as $\rightarrow$*including*() on sets need to be encoded by conditional expressions, to avoid multiple copies of elements in lists. Strings are encoded as atomic elements or as lists of integers. Subtyping cannot be directly represented. The relation of provability $\vdash_{Z3}$ in Z3 is more restricted than $\vdash$ for FOL.

In B, the key semantic distinction from OCL/FOL is the finiteness of INT and Object_OBJ, and the lack of representation for real numbers. Thus numeric values outside the range of INT have no denotation in B AMN and must be avoided in the transformation being analysed.

The constraint in B notation is:

```
!a.(a : A  => x(a) > 0)
```

In order to obtain semantic convergence between OCL/FOL and B AMN, a finitary set theory could be used for the former. In this version of set theory (ZF set theory with the axiom of infinity replaced by its negation), all sets are finite. The standard models of such a set theory are those based on the set V_ω of hereditarily finite sets: thus the models themselves remain infinite. Completeness is true for finitary set theory.

In UML-RSDS we use `int` and `double` in the specification language to indicate the use of the bounded numeric types. For attributes and values of these types, additional definedness clauses are needed, eg.: $def(e)$ includes the condition

$$-2^{31} + 2 \leq e \;\; and \;\; e \leq 2^{31} - 2$$

for expressions e whose values should be in `int`. Likewise, prior to creation of an object, some test that sufficient resources exist to complete the creation should be performed.