Verification Techniques for Model Transformations

K. Lano
Dept. of Informatics, King’s College London

This presentation will:

• Introduce concept of model transformation;

• Identify important verification properties for transformations:

• Define verification techniques using syntactic analysis and semantic analysis techniques.
Introduction

Model transformations (MT) are a key element of model-driven development (MDD) approaches for software construction.

Transformations are used for:

- Refinement: eg., code generation of Java from UML
- Migration: eg., mapping EHR data from one format to another, or one Business Process Model notation to another
- Analysis: eg., identifying hazardous states in drug dispenser device behaviour
- Quality improvement: eg., refactoring class diagrams.

For critical applications, eg., avionics or medical device software, patient records, need assurance of transformation correctness.
Transformation example: language translation

Rule-based language translation can be expressed as transformation of analysed sentences.

Eg.: “from the house” has structure Prep + Det + Noun.
Translation to Russian omits Det (if it is “the”, “a” or “an”), translates Prep and Noun, to RPrep, RNoun, then modifies RNoun to case required by RPrep, eg., genitive case for “from”.
Result is RPrep + genitive(RNoun).

Rule:- Prep + Det + Noun $\rightarrow$ RPrep + genitive(RNoun)
Translation of sentence structure

NP

from NP

the house

RNP

ot doma
Obstacles to transformation development and verification include:

- Emphasis has been on transformation coding, not specification
- Transformations defined at rule level – individual rewrite or mapping rules, no link to global pre/post specification
- Many different MT languages: Kermeta, ATL, QVT-R, ETL, UML-RSDS, etc (each research group has its own, we have 2!). Usually lack formal semantics
- Many different forms of transformation (update-in-place, mapping, merging, etc). Can uniform verification technique be used for these?
Solutions

- Formal foundations for MT languages (in logic/set theory)
- Language-independent MT specifications (in UML + OCL)
- Use transformation *invariants* (predicates preserved by rule applications) and *variants* (values reduced by rule applications) to relate rules to global specification.
Transformation Metamodels

Representation

Synthesis

Semantic mapping

ETL, QVT, ATL, GrGen, Kermeta

B, Z3, USE, Alloy, etc

Verification process
Example: Basic graph metamodel
Example: what does this transformation do?

Precondition/language constraint: no self-loops:

\[ \forall n : Node \cdot n \not\in n.\text{neighbours} \]

A single transformation rule:

\[ \text{neighbours.size} \leq 1 \implies \text{self} \rightarrow \text{isDeleted()} \]

operating on Node.

Rule is applied repeatedly to the graph, removing nodes with 0 or 1 neighbour, until no such nodes remain. Incident edges of such nodes are also deleted.

But what is the global pre/post behaviour?
Verification properties

Transformations from a language $S$ to a language $T$ should:

- produce target models valid according to $T$ (*syntactic correctness*);
- transformations should preserve key properties of source model (*semantic preservation*);
- implementations should be terminating and confluent, and correct with respect to transformation specification (*semantic correctness*).
Properties for language translation example

Syntactic correctness: valid English sentences/phases are mapped to valid Russian ones.

Semantic preservation: if sentence $s$ is mapped to sentence $s_1$, then these have same meaning.

Termination: The rules always produce a result.

Confluence: If different rules could apply to same text, their results are the same. Eg.: “Alex is a French restaurant manager”.

Semantic correctness: no global specification to compare to.
Properties for graph example

Syntactic correctness: rule applications preserve validity of model, so true.

Semantic preservation: if nodes $n$ and $n1$ connected in original graph and are both in final graph, then are they still connected?

Termination: Node.size is a variant.

Confluence: ?

Semantic correctness: no global specification to compare to.
Transformation verification techniques

Three general approaches have been used for MT verification:

1. syntactic analysis of transformation text;
2. mapping of the transformation to a formalism supporting semantic analysis;
3. specifying transformations in a proof formalism and then synthesising a correct-by-construction implementation.

We use (1) and (2). Approach (2) could be termed the verification model approach: semantics of transformation is expressed as model in a logical formalism.
Language-independent specification of model transformations

Many MT languages exist, with individual syntaxes, but usually share common concepts.

We use UML use cases to specify transformations $\tau$, with assumptions $Asm$ (use case preconditions), effects $Post$ (use case operational postconditions), invariants $Inv$.

Operate on source and target metamodels specified as class diagrams, $S$ and $T$ (may be same).

Additional predicates:

1. $Ens$ – declarative postconditions, including $\Gamma_T$

2. $Pres$ – properties which $\tau$ should preserve, under interpretation $\chi$ from $S$ to $T$. 
Correctness properties (mapping transformations)

(I) syntactic correctness:

\[ Asm0, Post, Inv, \Gamma_S \vdash_{\mathcal{L}_{S \cup T}} Ens \]

\(\Gamma_S\) is formal theory of \(S\).

(II) semantic preservation:

\[ Asm0, Post, Pres, Inv, \Gamma_S \vdash_{\mathcal{L}_{S \cup T}} \chi(Pres) \]

(III) semantic correctness:

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

where \(\text{activity}\) is algorithm of transformation.

Invariance of Inv (IV):

\[ Asm, \Gamma_S \vdash_{\mathcal{L}_{S \cup T}} Inv \]
and for transformation steps $\delta$:

$$
\Gamma_S \vdash_{\mathcal{L}_{S\cup T}} Inv \Rightarrow [\delta]Inv
$$

Implementations should satisfy (V) *termination* and (VI) *confluence*.

*Post* constraints $Cn$ should have implementing activities $\text{stat}(Cn)$.

Key proof technique is establishing invariance of $Inv$, then using it to establish $Ens$ and $\chi(Pres)$.

For update-in-place transformations, $Inv$, $Ens$ and $Post$ may relate current/end state to initial state ($v@pre$).

Variant $Q : \mathbb{N}$ reduced by every transformation step.
Correctness properties (update-in-place transformations)

(I) syntactic correctness:

\[ \text{Asm}@\text{pre}, \text{Post}, \text{Inv}, Q = 0, \Gamma_S \vdash_{\mathcal{L}_S} \text{Ens} \]

(II) semantic preservation:

\[ \text{Asm}@\text{pre}, \text{Post}, \text{Pres}@\text{pre}, \text{Inv}, Q = 0, \Gamma_S \vdash_{\mathcal{L}_S} \chi(Pres) \]

(III) semantic correctness:

\[ \Gamma_S \vdash_{\mathcal{L}_S} \text{Asm} \Rightarrow ([\text{activity}]\text{Post})[v/v@\text{pre}] \]

Invariance of Inv (IV):

\[ \text{Asm}, \Gamma_S \vdash_{\mathcal{L}_S} \text{Inv}[v/v@\text{pre}] \]

and for transformation steps \( \delta \):

\[ \Gamma_S \vdash_{\mathcal{L}_S} \text{Inv} \Rightarrow [\delta]\text{Inv} \]
Conceptual execution model

Executions of mapping $\tau$ are sequence of rule applications $\delta$ from initial state $(m, \emptyset)$ satisfying $Asm$ to terminal $(m, n)$ satisfying $Post$.

For update-in-place transformations can represent as starting state $(m, m)$ and terminal $(m, n)$.

$Inv$ is maintained through computation. To show termination + confluence, need variant $Q : \mathbb{N}$ strictly decreased by each step.
Verification example

Key to verifying graph transformation is invariant $Inv$, based on analysis of which nodes $n$ are deleted:

$$n : \text{Node}@pre \rightarrow \text{Node} \text{ implies } \exists_1 p \cdot p \text{ path from } n \text{ to } n1 : \text{Node}@pre \text{ with } n1.\text{neighbours}@pre.\text{size} \leq 1$$

In other words, if $n$ is deleted, this is because there is unique loop-free sequence of edges (in original graph) from $n$, to an element with 0 or 1 neighbour in original graph.

Call consequent property $P(n)$.

Proof of invariance of $Inv$ is direct. Holds initially, as $\text{Node} = \text{Node}@pre$. $\text{Asm}$ is also invariant.

A suitable $Q$ measure is:

$$\text{card}(\{n : \text{Node} \mid P(n)\})$$
Strictly decreased by each rule application. 0 in end state, so confluence holds.

Declarative global postcondition can then be expressed as:

\[(Ens) : \quad \text{Node} = \{ n : \text{Node}@pre \mid \neg P(n) \}\]

Inference (I) holds: Inv implies that only \(P(n)\) nodes are deleted, \(Q = 0\) that all \(P(n)\) nodes are deleted.
Syntactic analysis

For each predicate $P$ define write frame $wr(P)$ of $P$, and read frame $rd(P)$ – sets of entities and features which it may update or access.

Dependency ordering $Cn < Cm$, “$Cm$ depends upon $Cn$”, is defined between $Post$ constraints by

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

Use case with $Post$ postconditions $C_1, \ldots, C_n$ should satisfy syntactic non-interference:

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

Implies that activities $stat(C_j)$ of $C_j$ cannot invalidate earlier constraints $C_i$, for $i < j$. 

21
**Syntactic analysis**

Syntactic checks for determinacy + definedness of constraints can be applied.

*Type 1 constraints* $C_i$ satisfy condition

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

For these, confluence, semantic correctness and termination can be established by syntactic checks to ensure that distinct applications of constraint cannot semantically interfere (*internal syntactic non-interference*).

$\text{stat}(C_i)$ for type 1 constraints is fixed for-loop iteration over their source domains.

$\text{Inv}$ can be calculated as inverse of $\text{Post}$ in such cases.
Analysis of recurrent constraints

Constraint $Cn$ of form

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

is termed recurrent if

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

Generally require implementation by fixpoint computation.

Our example has write and read frame $\{Node, neighbours\}$.

A measure $Q : S \times T \rightarrow \mathbb{N}$ on source and target model data is used to establish termination, confluence and correctness of recurrent constraints.

$Q$ should be a variant function for applications of constraint:

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

\[
\Gamma_S, Inv \vdash Q(smodel, tmodel) = 0 \equiv \\
\{ s \mid s \in S_i \land SCond \land \neg (Succ) \} = \{ \}
\]

*Succ* abbreviates the constraint rhs \( \exists t : T_j \cdot TCond \text{ and } Post \).

Confluence requires that \( Q = 0 \) state is unique (VI), ie., for each starting state of source and target models there is unique (up to isomorphism) possible terminal state of the models (produced by applying the constraint until it cannot be applied further) in which \( Q = 0 \).
Verification using B AMN

B specification consists of a linked collection of machines. Each machine encapsulates data and operations. Metamodels are represented by sets and maps for entity types + features.

Verification model $M_\tau$ representing semantics of transformation $\tau$ has form:

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

where \( Pres' \) is \( \chi(Pres) \), and \( Ens \) and \( Pres' \) are only included if they are invariant.
Verification model in B for example has:

MACHINE Mt SEES SystemTypes
VARIABLES nodes, neighbours
INVARIANT nodes <: Node_OBJ &
   neighbours : nodes --> FIN(nodes) &
   /* Asm: */ !nodex.(nodex : nodes => nodex /: neighbours(nodex))
INITIALISATION
   nodes, neighbours := {}, {} 
OPERATIONS
   create_Node() =
      PRE nodes /= Node_OBJ
      THEN
         ANY nodex WHERE nodex : Node_OBJ - nodes
         THEN
            nodes := nodes \ { nodex } ||
            neighbours(nodex) := {} 
         END
addneighbours(nodex, nodexx) =
    PRE nodex : nodes & nodexx : nodes & nodexx /= nodex
    THEN
    neighbours(nodex) := neighbours(nodex) \ {nodexx}
    END;

phase1(nodex) =
    PRE nodex : nodes & card(neighbours(nodex)) <= 1
    THEN
    nodes := nodes - { nodex } ||
    neighbours := {nodex} <<-1 neighbours
    END
END
Properties (I), (II) and (IV) of $\tau$ can be verified by internal consistency proof of $M_{\tau}$: that invariant is established by initialisation, and is preserved by each operation.

In general, about 80% of B internal consistency proof obligations are automatically provable.

In order to prove that some $Q$ is a variant for a constraint, refinement proof in B is used.
Machine $M0_\tau$ defines variant property of $q$:

MACHINE M0 \ SEES SystemTypes
VARIABLES /* variables for source model data */ , q
INVARIANT
    /* typing of source data & */ q : NAT
INITIALISATION
    /* source data := {} || */ q := 0
OPERATIONS
    /* creation and update operations for source
     model: these may set q arbitrarily in NAT */

    phase1() =
        PRE q > 0
        THEN q :: 0..q-1
        END
END
phase1 represents a transformation step of constraint for which \( q \) is postulated variant.

Each constraint \( C_i \) may have a corresponding variant \( q_i \).

Original \( M_\tau \) machine used to define refinement of \( M0_\tau \), with refinement relation giving explicit definition of \( q \).

To verify confluence of a constraint, prove there is essentially a unique state where \( q = 0 \):

**ASSERTIONS**

\[
q = 0 \implies T_{\text{data}} = f(S_{\text{data}})
\]
**Analysis using Z3**

Z3 is a satisfaction-based automated proof tool, able to decide formulae in several decidable subsets of first order logic. For properties (I) and (II) to be decidable, both \( S \) and \( T \) must be *stratified* languages (associations point in consistent direction).

Entity types encoded as uninterpreted sorts, and features as uninterpreted functions.

Eg., \( \Gamma_S; \text{ Asm} \) of case study encoded as:

```
(declare-sort Node)
(declare-fun neighbours (Node) List(Node))
(assert (forall ((nodex Node))
           (not (member (nodex (neighbours nodex)))))
```

To prove \( \varphi \) from this theory, assertion of \( \text{not}(\varphi) \) is added, and result that combined theory is unsatisfiable is obtained.
## Comparison

<table>
<thead>
<tr>
<th>Capability</th>
<th>Syntactic analysis</th>
<th>$B$</th>
<th>$Z3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automated</td>
<td>Data-dependency, determinacy, completeness for all constraints.</td>
<td>80% for (I), (II), (IV).</td>
<td>100% for decidable FOL</td>
</tr>
<tr>
<td>verification</td>
<td>(V), (VI), (III)</td>
<td>Only 20% for (V), (VI), (III).</td>
<td>subsets.</td>
</tr>
<tr>
<td></td>
<td>for some type 1 constraints.</td>
<td></td>
<td>Proves (I), (II) properties.</td>
</tr>
<tr>
<td>Encoding</td>
<td>Works directly on specification of $\tau$.</td>
<td>Supports all of OCL, except reals and rationals.</td>
<td>Indirect OCL encoding.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No wpc calculation.</td>
</tr>
</tbody>
</table>
Comparison

Syntactic analysis effective to identify errors in data dependencies, definedness and determinacy. Establishes properties (III), (V) and (VI) for internally syntactically non-interfering type 1 constraints.

Proof in B provides means of proving properties (I), (II), (IV) by internal consistency proof, and properties (V), (VI), (III) by refinement proof (but requires high expertise for interactive proof). Supports direct encoding of UML and OCL.

Z3 provides automated proof, for restricted forms of transformation. Z3 lacks high-level data structures – so OCL and UML encoding indirect. Can be used to prove properties (I) and (II) directly.
Conclusions

• Syntactic checking essential, also sufficient in many cases

• Care is needed that verification model is sound representation (e.g., B integers are bounded, unlike OCL or Z3)

• Attempting proof in B or Z3 can help identify errors

• Systematic verification by hand also useful, using \( Inv, Q \) and rules

• \( Inv \) derived as inverse of transformation: expresses the origin of changes to model.

Approach and techniques implemented for UML-RSDS tools (http://www.dcs.kcl.ac.uk/staff/kcl/uml2web)