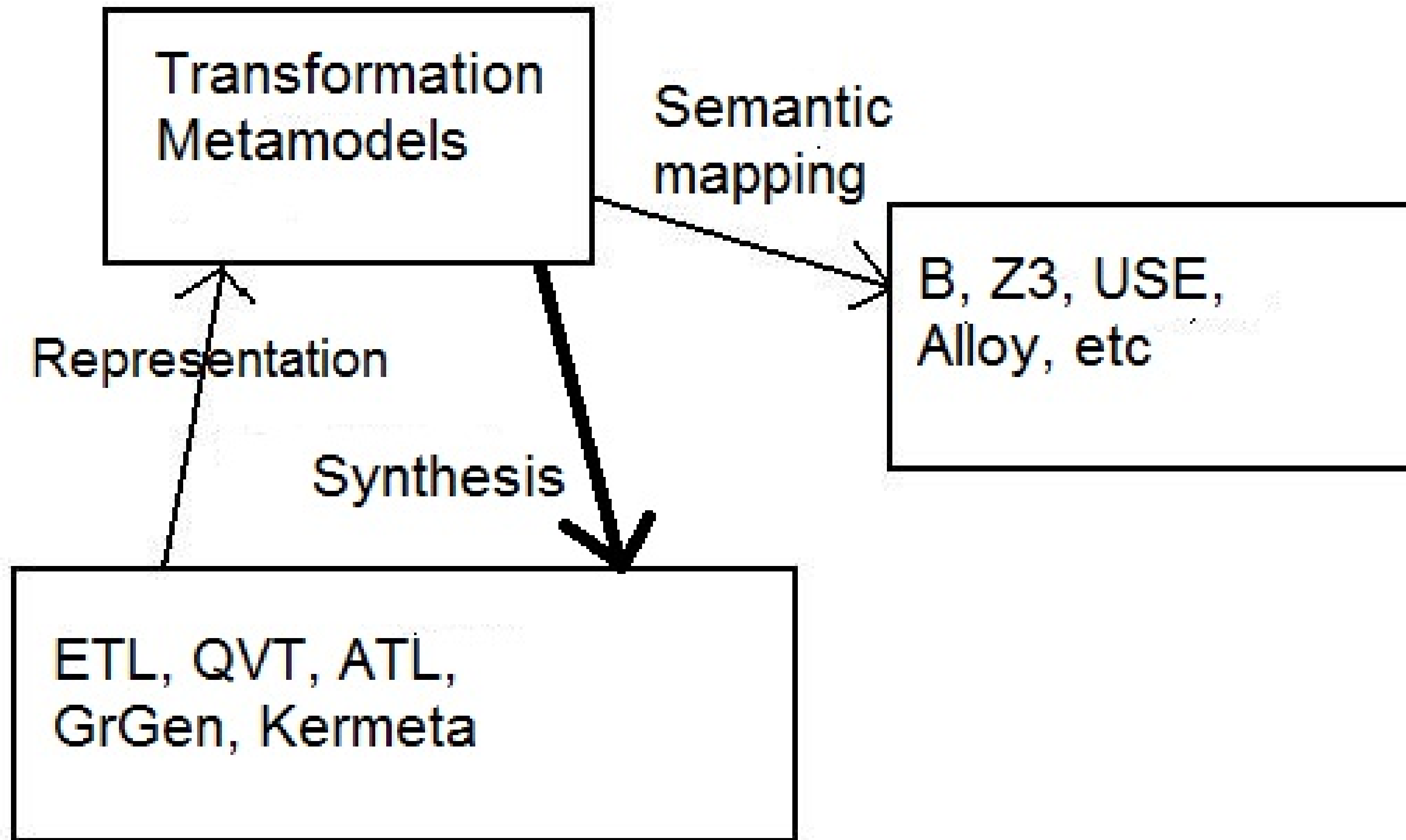
Language-Independent Model Transformation Verification

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

King's College London; University of Middlesex

Language-independent MT verification

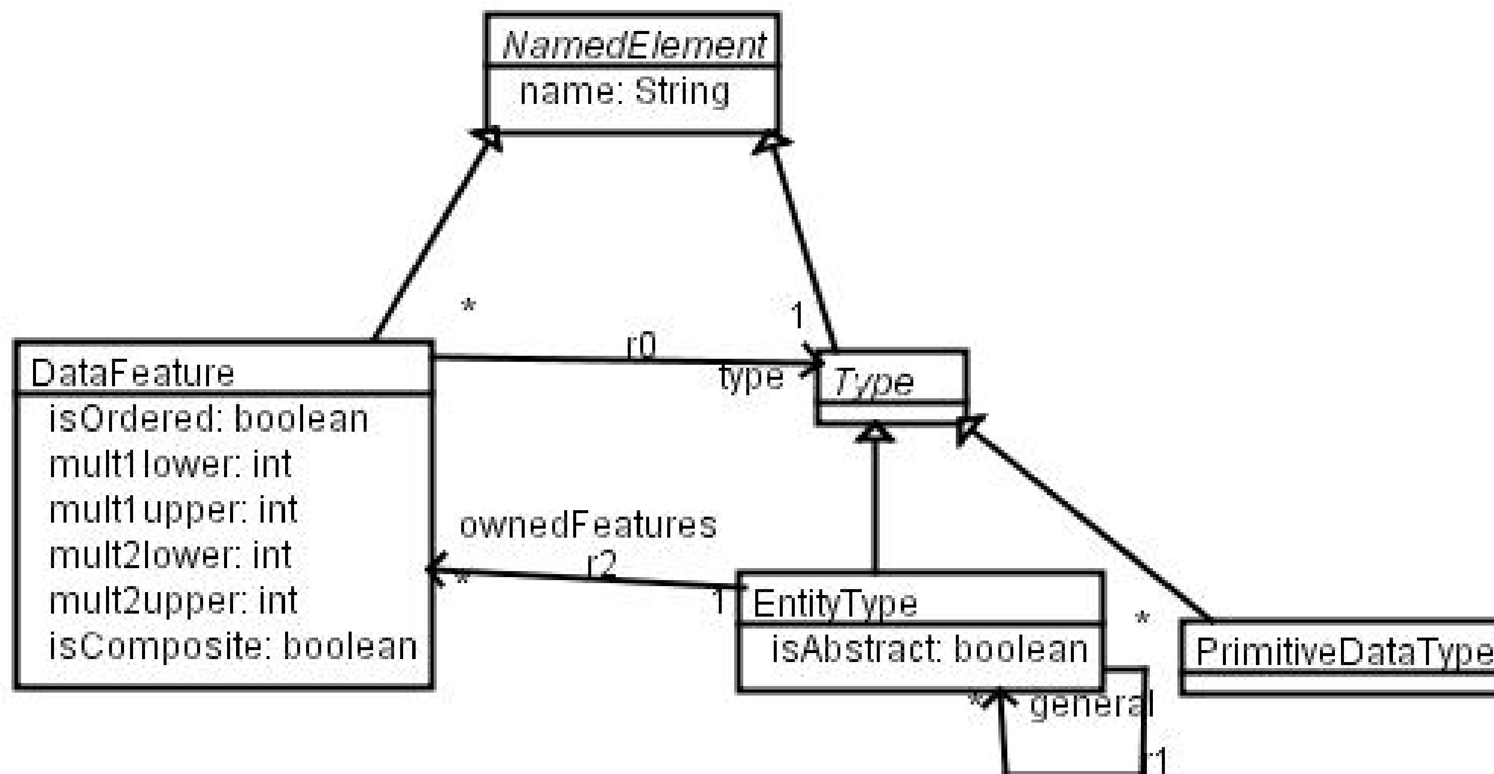
- One hinderance to MT verification is large number of different MT languages which exist.
- Instead of language-specific analysis, we define analysis process applicable to multiple MTLs, via common intermediate representation.
- Illustrate approach by applying it to ATL.



Verification process

Metamodels for MTLs

- Based on *transML* specification, implementation metamodels
- Provide semantic representations of transformations
- MT representations in metamodels can be mapped to verification formalisms (such as theorem provers or constraint satisfaction checkers).



Metamodel ($\mathcal{LMM}$) for modelling languages

Formal semantics for languages

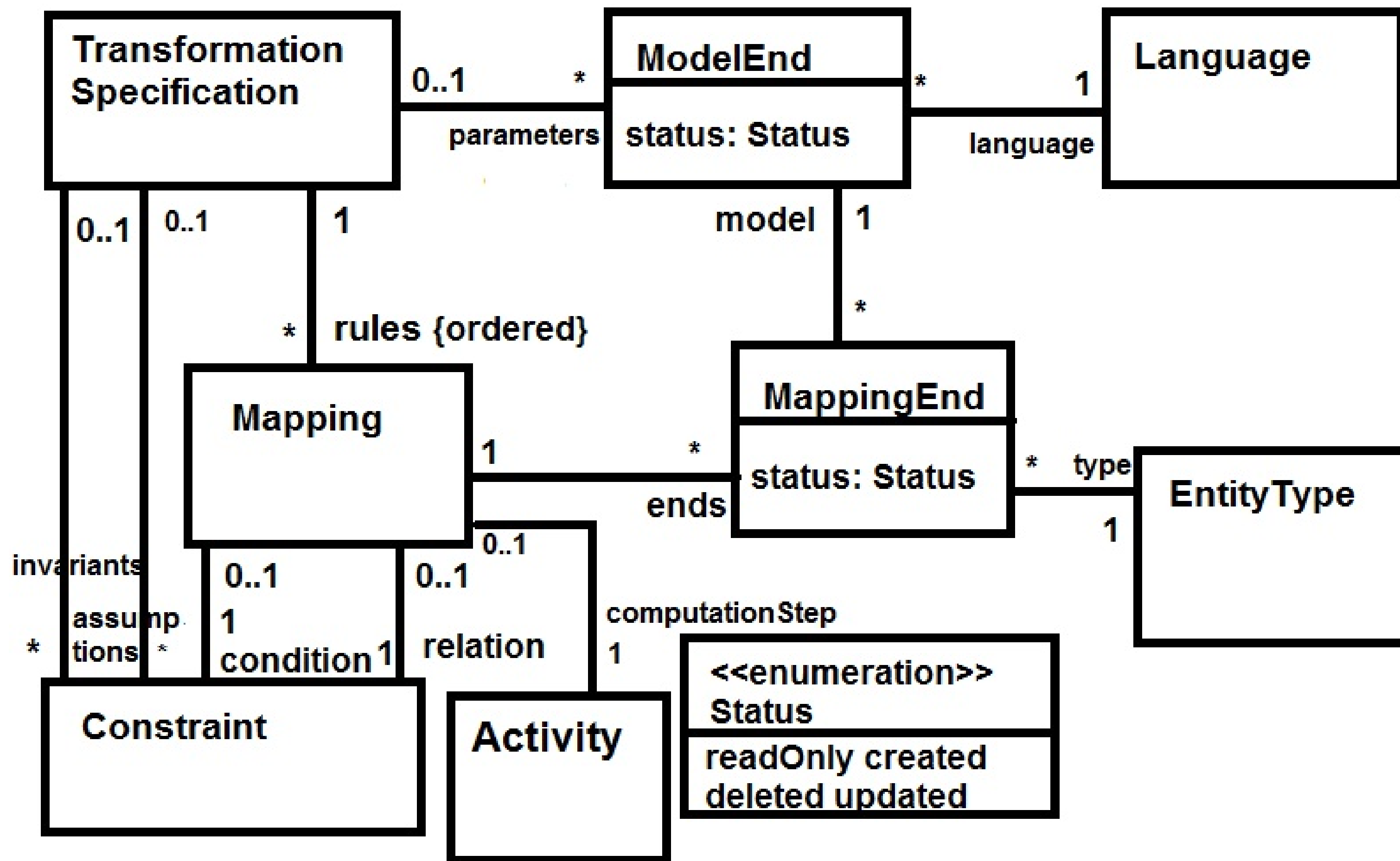
- Metaclass *Language* representing languages has set *entityTypes* of *EntityType*, set *features* of *DataFeature*, and set *constraints* of *Constraint*
- Set of expressions over base language L (a class diagram) is $Exp(L)$
- To support verification, define proof theory and (logical) model theory wrt language L, by associating formal 1st-order language/logic $\mathcal{L}_L$ to L (of $\mathcal{LMM}$).

Modelling concept	$\mathcal{LMM}$ *rep.*	*Formal semantics*
Entity type E	*EntityType* element	Type symbol E', extent of E
Single-valued attribute $att : Typ$ of E	*DataFeature* element $mult2upper = 1$, $mult2lower = 1$, $type \in PrimitiveDataType$	Function symbol $att' : E' \to Typ'$
Single-valued role r of E with target type $E1$	*DataFeature* element $mult2upper = 1$, $mult2lower = 1$, $type \in EntityType$	Function symbol $r' : E' \to E1'$

Unordered many-valued role r of E with target entity type $E1$	*DataFeature* element $mult2upper \neq 1$ or $mult2lower \neq 1$, $isOrdered = false$, $type \in EntityType$	Function symbol $r' : E' \to \mathbb{F}(E1')$
Supertype $E1$ of E	$E1 \in E.general$	Type $E1'$, $E' \subseteq E1'$

Transformation representation

- A transformation τ characterised by collection of mapping specifications, define intended relationships τ should establish between input (source) and output (target) models, at termination. Postconditions *Post* of τ.
- Each mapping has corresponding *computation step*, defines application of mapping to specific source elements that satisfy its condition.



Transformation specification metamodel $\mathcal{TMM}$

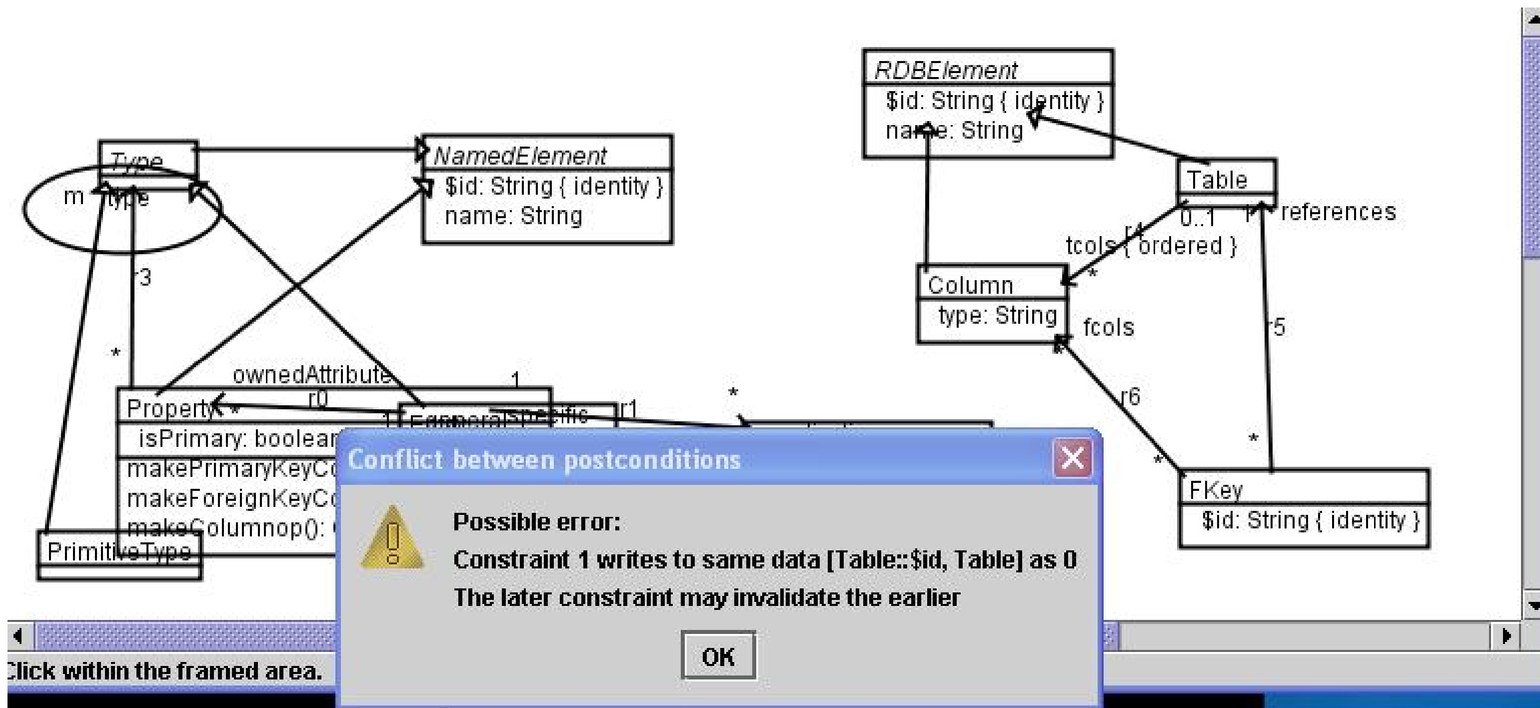
Transformation metamodel

- *rules.relation* constraints express postconditions *Post* of transformation.
- The *assumptions* express preconditions *Asm* of transformation.
- The *invariants* define properties *Inv* true initially, and preserved by each computation step.

Verification techniques

Verification techniques can be applied to $\mathcal{TMM}$ representation:

- Syntactic analysis to identify definedness conditions $def(Cn)$, determinacy conditions $det(Cn)$ of each mapping Cn.
- Data dependency analysis, to compute read-frame $rd(Cn)$ and write-frame $wr(Cn)$ for each Cn, to identify cases of invalid data-dependency and data-use.
- Syntactic analysis to establish confluence and termination in case of transformations not involving fixed-point iteration.
- Translation to B AMN, to verify transformation invariants, syntactic correctness (conformance of target models to target language) and model-level semantic preservation (source and target models have equivalent semantics).
- Translation to Z3, to identify counter-examples to syntactic correctness.



Data dependency analysis example

Mapping from ATL to transformation metamodel

Assume that rule inheritance has been statically expanded out so only leaf rules exist.

Denote $\mathcal{TMM}$ expression equivalent of an ATL expression e by e'.

ATL element	*Transformation metamodel representation*
Module	TransformationSpecification
Matched rule	Mapping(s)
Lazy rule	Operation invoked from mapping constraint
Unique lazy rule	Operation using caching to avoid re-application
Implicit rule invocation	Object lookup using object indexing or traces
Using variable	Variable defined in mapping condition
Functional helper	Query operation
Attribute helper	Operation defining navigation expression
Action blocks	Operation with activity, invoked from mapping constraint

Mapping from ATL to transformation metamodel

Given an ATL module

```
module M;
create OUT : T from IN : S;
rule r1
{ from s1 : S1, ..., sm : Sm (SCond)
  using { v1 : Typ1 = val1; ... }
  to t1 : T1 (TCond1), ..., tk : Tk (TCondk)
  do (Stat)
}
...
rule rn { ... }
```

$\mathcal{TMM}$ representation of M is a transformation specification M', each matched rule r_i is represented by mappings $r_i Matching$ and $r_i Initialisation$ of M'.

Input conditions $SCond$ translated to condition conjunct $SCond'$ of $r_i Matching$ and $r_i Initialisation$. `using` variables are expressed by conjunct $vt = valt'$ of condition.

If an action block $Stat$, new update operation $opri(s2 : S2, ..., sm : Sm, t1 : T1, ..., tk : Tk)$ is introduced, with activity given by $Stat'$ ($Stat$ as a UML activity).

Relation of $r_i Matching$ has context $S1$ and predicate $T1 \rightarrow exists(t1 \mid t1.\$id = \$id)$ *and* ... *and* $Tk \rightarrow exists(tk \mid tk.\$id = \$id)$

Creates target model elements corresponding to source model elements that satisfy $SCond'$, establishes tracing relations from $s1$ to $t1$, ..., tk.

Relation of $r_i Initialisation$ has context $S1$ and predicate

$$
\begin{array}{l}
T1 \rightarrow exists(t1 \mid t1.\$id = \$id \ and \ ... \ and \\
\quad Tk \rightarrow exists(tk \mid tk.\$id = \$id \ and \\
\qquad TCond1' \ and \ ... \ and \ TCondk' \ and \\
\qquad s1.opri(s2, ..., tk))...)
\end{array}
$$

The tj corresponding to $s1$ are looked-up and their features initialised.

Lazy rules r with first input entity type $S1$ translated to operations rop of $S1$. Calls $thisModule.r(v1, ..., vm)$ to rule interpreted as calls $v1.rop(v2, ..., vm)$.

rop stereotyped as $\ll cached \gg$ for unique lazy rules.

All $r_i Matching$ rules are listed before all $r_i Initialisation$ rules in $M'.rules$.

Translation from ATL to $\mathcal{TMM}$ can be extended to update-in-place transformations, i.e., to *refining* mode of ATL.

Translated transformation represents usual execution semantics of ATL: target objects are created in first phase, followed by links between objects.

Computation steps (rule applications) of translation correspond to original specification steps; an invariant of translation is also invariant of original.

Syntactic correctness proof using invariants of transformation also demonstrates syntactic correctness for original transformation.

Similarly, for model-level semantic preservation.

Counter-example ($sourcem, targetm$) from satisfaction-checking tool for translated transformation, also counter-example for original ATL transformation.

For refining mode transformations, termination and confluence of translated specification establishes these properties for original specification.

Evaluation

(1) UML to relational database ATL transformation, three lazy rules and one matched rule with action block:

```
rule makeTable {
from c : Entity ( c.generalisation->size() = 0 )
to t : Table ( rdbname <- c.name )
do {
  for ( att in c.ownedAttribute )
  { if ( att.isPrimary )
    { t.tcols->includes(thisModule.makePrimaryKeyColumn(att)); }
  }
  for ( att in c.ownedAttribute )
  { if ( att.type.oclIsKindOf(Entity) )
    { t.tcols->includes(thisModule.makeForeignKeyColumn(att)); }
  }
  for ( att in c.ownedAttribute )
  { if ( att.type.oclIsKindOf(PrimitiveType) or
```

```
        att.isPrimary = false )
     { t.tcols->includes(thisModule.makeColumn(att)); }
     }
  }
} }

lazy rule makePrimaryKeyColumn {
  from p : Property ( p.isPrimary )
  to c : Column ( rdbname <- p.name,
                  type <- p.type.name )
}

lazy rule makeForeignKeyColumn {
  from p : Property ( p.type.oclIsKindOf(Entity) )
  to c : Column
     ( rdbname <- p.name,
       type <- "String" ),
     f : FKey
```

```
    ( references <- p.type,
      fcols <- Set{ c } )
}

lazy rule makeColumn {
  from p : Property ( true )
  to c : Column
    ( rdbname <- p.name,
      type <- p.type.name )
}
```

Matched rule translated to constraints $C0$ and $C1$ with context *Entity*:

```
generalisation.size = 0   implies
    Table->exists( t | t.$id = $id  and  t.rdbname = name )

generalisation.size = 0   implies
    Table->exists( t | t.$id = $id  and  self.makeTable1op(t) )
```

$C1$ has condition and relation of *makeTableMatching*, creates table objects. $C2$ has condition and relation of *makeTableInitialisation*, creates and links columns to tables.

makeTable1op(t) represents *makeTable*'s action block, invoking operations *makePrimaryKeyColumnop*, *makeForeignKeyColumnop* and *makeColumnop* of *Property* corresponding to lazy rules:

```
makeColumnop(): Column
post:
  Column->exists( c | c.$id = $id  and  c.rdbname = name  and
                      c.type = type.name  and result = c )
```

Evaluation

Verification properties for case study include: (i) termination; (ii) confluence; (iii) columns of foreign keys are always contained in set of columns of tables: $fcols \subseteq Table.tcols$ on $FKey$.

(i) can be shown by data-dependency analysis – that bounded iteration implementation is sufficient for $C0$ and $C1$. (ii) fails because $Table :: tcols$ is ordered. Proof of (iii) by establishing it as a transformation invariant.

Property	*ATL via $\mathcal{TMM}$*
Termination	By data-dependency analysis
Foreign key columns contained in table columns	20 proof obligations, 15 automatically proved

Evaluation

(2) Refactoring transformation operating on class diagrams. Applies ‘pull up feature’ refactorings.

ATL refining mode solution with a single rule:

```
rule r1 {
  from c : Entity, p : Property
    (c.specialisation->size() > 1 and
     c.specialisation.specific.ownedAttribute->includes(p) and
     c.ownedAttribute.name->excludes(p.name) and
     c.specialisation.specific->forAll( g |
          g.ownedAttribute->exists( q |
               q.name = p.name and q.type = p.type ) ) )
  to c : Entity
    ( ownedAttribute <- c.ownedAttribute->including(p) )
  do
  { for ( p1 in c.specialisation.specific.ownedAttribute )
    { if (p1.name = p.name)
```

```
      { p1->isDeleted(); }
    }
  }
}
```

No *Matching* phase required in semantic representation. Mapping ends (here $c : Entity$) in both `from` and `to` clauses are updated by rule.

Required correctness properties: (i) termination; (ii) preservation of single inheritance: invariant $generalisation \rightarrow size() \leq 1$ on *Entity*; (iii) preservation of no attribute name conflicts within classes: invariant $ownedAttribute \rightarrow isUnique(name)$ of *Entity*.

Termination shown by proving that $Property.allInstances() \rightarrow size()$ is a variant, i.e., number of *Property* instances in model strictly decreased by applying the rule.

Property	*ATL via* $\mathcal{TMM}$
Termination	6 proof obligations, 5 automatically proved
Single inheritance	13 proof obligations, all automatically proved
No name conflicts	15 proof obligations, 13 automatically proved

Conclusions

Previous work on MT verification has been language-specific.

We provide general process for MT verification by expressing MTL semantics in $\mathcal{TMM}$, reverse-engineering transformations to this semantic representation, and analysis techniques on $\mathcal{TMM}$.

Currently working on ETL and Flock.