Abstract
In this paper we apply the UML-RSDS notation and tools to the “Hello World” case studies and explain the underlying development process for this model transformation approach.

1 Specification of model transformations
In UML-RSDS a transformation specification is written in first-order logic and OCL, and consists of one or more predicates:

1. A global specification, Cons, of a model transformation, expresses in a platform-independent manner the overall effect of the transformation, as a relation between the source and target models. It is expressed in the (disjoint) union of the languages of these models. It is intended to hold true at termination of the transformation.

2. A predicate Asm expresses the assumptions made about the source and target models at the start of the transformation, for example, that the target model is empty and that the source model is syntactically correct wrt the source language. It also is a predicate in the union language.

3. Ens is a predicate in the target language (possibly also in the source language if the transformation modifies both source and target models), expressing additional properties of the transformation result, which should be consequences of Cons.

4. Pres is a predicate in the source language, expressing properties of the source model which should be true (possibly via some syntactic interpretation) in the target model.

*Research supported by the HoRTMoDA EPSRC project
The specification is therefore independent of any specific model transformation implementation language, and can be used as the basis for development in many such languages. By making explicit the semantic assumptions on source and target models, the specification assists in the verification (formal or informal) of model transformations. The set-theoretic semantics of Chapter 6 of [4] is used for OCL.

Various common patterns for the specification predicates have been identified, for example Cons can often be written in conjunctive-implicative form [6], as a conjunction of constraints of the form

\[ \forall s : S \cdot SCond \implies \exists t : T \cdot Post \]

where \( S \) is a source language entity and \( T \) is a target language entity. This pattern is applicable to re-expression transformations such as model migrations, and to abstraction and refinement transformations.

The patterns assist in the derivation of explicit PSM designs from the specification, consisting of a sequence of phases, which apply specific rules or operations to achieve the specification constraints. Provided that the updates defined in Post do not affect the data read in SCond or Post, and that the extent of \( S \) is fixed throughout the transformation (\( ^\ast \)), then a constraint of the above form can be implemented by an iteration

\[ \text{for } s : S \text{ do } s.op() \]

where \( op \) implements the constraint for a particular \( S \) object.

This iteration constitutes a single phase in the design. The possible orderings of phases are determined by defining a partial order over the target language entities: \( T_1 < T_2 \) if \( T_1 \) is used in Cons to define a feature of \( T_2 \) (or a feature of a subclass of \( T_2 \)). Any phase that creates \( T_2 \) instances must therefore be preceded by all phases that create \( T_1 \) instances.

The restriction (\( ^\ast \)) is termed the non-interference condition.

The iterative phase activities derived from the constraints are also terminating and they establish the truth of their corresponding constraint, by construction. The PSM design can then be used to synthesise code in Java, using the UML-RSDS toolset [5]. The resulting executable is a stand-alone implementation of the transformation, operating upon simple text format files defining input and output models.

## 2 Simple transformation tasks

Here we give the specifications and implementations of the simple transformation tasks in [1]. All of these tasks satisfy the restrictions described above, so they can be specified and designed directly in UML-RSDS.
2.1 Hello world transformation

This has the global specification (Cons predicate):

\[ \exists g : Greeting \cdot g.text = "Hello" \text{ and } \exists p : Person \cdot g.whom = p \text{ and } p.name = "World" \]

This specification is implemented by a static operation \( op \) of the \( Greeting \) class with the above predicate as its postcondition:

\[
\text{op}() \quad \text{post:} \quad \begin{align*}
Greeting \rightarrow \exists (g \mid g.text = "Hello" \ & \ \text{Person} \rightarrow \exists (p \mid g.whom = p \ & \ p.name = "World")
\end{align*}
\]

2.2 Graph properties

Figure 1 shows the basic graph metamodel in the UML-RSDS tools, and an operation specification and its generated Java code. We assume that the following constraint \( Asm0 \) of the source model holds:

\[
\forall g : \text{Graph} \cdot g.edges.src \subseteq g.nodes \text{ and } g.edges.trg \subseteq g.nodes
\]

The queries are simple examples of abstraction transformations, and can be specified as follows:

\[
\forall g : \text{Graph} \cdot \exists r : \text{IntResult} \cdot r.num = g.nodes \rightarrow \text{size}()
\]

Figure 1: Graph metamodel and queries in UML-RSDS
creates for each graph a result object recording the number of nodes in the graph.

This constraint is written using the syntax

\[ \text{IntResult} \rightarrow \exists r \mid r.\text{num} = \text{nodes} \rightarrow \text{size}() \]

in UML-RSDS, as a postcondition of the transformation use case, with context \text{Graph}, and an operation with this postcondition can be automatically generated to implement the specification, together with a phase definition to execute it for all elements of \text{Graph}.

Likewise for the other queries:

\[ \forall g : \text{Graph}. \exists r : \text{IntResult} \cdot r.\text{num} = g.\text{edges} \rightarrow \text{select}(\text{src} = \text{trg} \text{ and } \text{trg} \neq \{\}) \rightarrow \text{size}() \]

returns the number of looping edges, and is implemented by an operation \text{op2}().

\[ \forall g : \text{Graph}. \exists r : \text{IntResult} \cdot r.\text{num} = g.\text{edges} \rightarrow \text{select}(\text{src} = \{\} \text{ or } \text{trg} = \{\}) \rightarrow \text{size}() \]

returns the number of dangling edges and can be implemented by an iteration of an operation \text{op3}() on graphs.

\[ \forall g : \text{Graph}. \exists r : \text{IntResult} \cdot r.\text{num} = (g.\text{nodes} - (g.\text{edges} \cup g.\text{edges}.\text{src} \cup g.\text{edges}.\text{trg})) \rightarrow \text{size}() \]

returns the number of nodes that are not the source or target of any edge. \(-\) denotes set subtraction and \(\cup\) set union. This is implemented by \text{op4}().

We extend the final query problem by defining an auxiliary entity which records the 3-cycles in the graph (Figure 2).

![Figure 2: Extended graph metamodel](image)
The specification Cons of this transformation then defines how unique elements of ThreeCycle are derived from the graph, and returns the cardinality of this type in the end state of the transformation:

\( (C1) : \forall g \cdot \forall e_1 : g.\text{edges}; e_2 : g.\text{edges}; e_3 : g.\text{edges} \cdot e_1.\text{trg} = e_2.\text{src} \text{ and } e_2.\text{trg} = e_3.\text{src} \text{ and } e_3.\text{trg} = e_1.\text{src} \text{ and } (e_1.\text{src} \cup e_2.\text{src} \cup e_3.\text{src}) \rightarrow \text{size}() = 3 \text{ implies } \exists tc : \text{ThreeCycle} \cdot tc.\text{elements} = (e_1.\text{src} \cup e_2.\text{src} \cup e_3.\text{src}) \text{ and } tc : g.\text{cycles} \)

\( (C2) : \forall g : \exists r : \text{IntResult} \cdot r.\text{num} = g.\text{cycles} \rightarrow \text{size}() \)

The order of nodes in a cycle is not distinguished by C1, if this was required then elements should be ordered (a sequence). Because of Asm0, each three-cycle will consist of nodes in a single graph. The unique existential quantifier \( \exists_1 \) specifies that there must exist exactly one object satisfying the quantified properties, ie, duplicated cycles are not included in cycles.

Each constraint is refined by a specific phase in the design. The exists1 quantifier is implemented by checking that there is no existing ThreeCycle with the required property, before creating such an element.

Constraint C2 is simply implemented by a phase, countCycles, consisting of an operation of Graph with the constraint as its postcondition. The overall transformation is implemented as the global activity Graph.createCycles(); Graph.countCycles(), since IntResult depends on ThreeCycle. A further phase could be introduced to remove all ThreeCycle instances from the model, once the counts have been recorded:

ThreeCycle \rightarrow \text{isDeleted}()

An alternative approach would be to evaluate the set of three cycles in a single expression:

\[ g.\text{edges} \rightarrow \text{collect}(e_1, e_2, e_3 \mid \{e_1, e_2, e_3\}) \rightarrow \text{asSet()} \rightarrow \text{select}(s \mid s \rightarrow \text{size}() = 3 \& s.\text{src} = s.\text{trg}) \rightarrow \text{size}() \]

but we consider that the approach using ThreeCycle is more clear.

### 2.3 Reverse edges

The global specification Cons for this transformation is:

\[ \forall e : \text{Edge} \cdot e.\text{src} = e.\text{try}@\text{pre} \text{ and } e.\text{try} = e.\text{src}@\text{pre} \]

The suffix @\text{pre} denotes the value of the expression at the start of the transformation. This is the usual style of specification for update-in-place transformations.

This specification can be implemented by an iteration of an operation of Edge, of the form:
reverse()
post:
    \( \text{src} = \text{try} \odot \text{pre} \) & \( \text{try} = \text{src} \odot \text{pre} \)

Here in contrast \( \text{pre} \) denotes the value of the expression at the start of the operation. This will also actually be the value at the start of the transformation since updates to one edge do not affect any other edge.

The iteration is expressed as a bounded loop:

\[
\text{for } e : \text{Edge} \text{ do } e.\text{reverse()}
\]

or more concisely as \( \text{Edge}.\text{reverse()} \).

By standard rules of inference we can deduce that this iteration is confluent, and that it achieves the specification Cons \([5]\).

### 2.4 Simple migration

The metamodels for this re-expression transformation are shown in Figure 3, together with extracts from example input and output models (on the left and right hand sides, respectively).

![Figure 3: Graph migration metamodels](image)

Figure 3: Graph migration metamodels
We make the additional assumption $Asm_1$ that the target model is empty at the start of the transformation:

$$ModelElement_2 = \{\}$$

We can specify this transformation by three constraints, defined as the post-conditions of a single use case of the system:

\[(C1) : \forall n_1 : Node_1. \exists n_2 : Node_2. n_2.id_2 = n_1.id_1 \text{ and } n_2.text = n_1.name\]

\[(C2) : \forall e_1 : Edge_1. \exists e_2 : Edge_2. e_2.id_2 = e_1.id_1 \text{ and } e_2.text = \text{""} \text{ and } e_2.src_2 = Node_2[e_1.src_1.id_1] \text{ and } e_2.trg_2 = Node_2[e_1.trg_1.id_1]\]

\[Node_2[e_1.src_1.id_1] \text{ denotes the set of } Node_2 \text{ objects with primary key } id_2 \text{ value in the set } e_1.src_1.id_1.\]

\[(C3) : \forall g_1 : Graph_1. \exists g_2 : Graph_2. g_2.id_2 = g_1.id_1 \text{ and } g_2.gcs = Node_2[g_1.nodes.id_1] \cup Edge_2[g_1.edges.id_1]\]

A design can be automatically generated from these constraints, and implements each constraint by a separate phase in a three-phase algorithm:

1. phase1: Map all $Node_1$ elements to $Node_2$ elements.
2. phase2: Map all $Edge_1$ elements to $Edge_2$ elements.
3. phase3: Map all $Graph_1$ elements to $Graph_2$ elements.

The ordering of the phases follows from the ordering of the entities $Node_2 < Edge_2 < Graph_2$ in the target language, based upon the dependencies between these entities in the specification constraints ($Edge_2$ instances depend upon $Node_2$ instances, etc).

The definition of the individual phases also follows from the form of the constraints. The following three operations, of $Node_1$, $Edge_1$ and $Graph_1$, respectively, are derived from the constraints:

\[\text{toNode2()}\]
\[\text{post: } Node_2 -> exists( n_2 \mid n_2.id_2 = id_1 \& n_2.text = name )\]

\[\text{toEdge2()}\]
\[\text{post: } Edge_2 -> exists( e_2 \mid e_2.id_2 = id_1 \& e_2.text = "" \& e_2.src_2 = Node_2[src_1.id_1] \& e_2.trg_2 = Node_2[trg_1.id_1] )\]
toGraph2()

post:

\[ Graph2 \rightarrow \text{exists}( g2 \mid g2.\text{id} = \text{id1} \&
\]
\[ g2.\text{gcs} = \text{Node2}[\text{nodes.\text{id1}}] \cup \text{Edge2[\text{edges.\text{id1}}}] \)

The phases are then defined as bounded iterations of these operations:

1. phase1: Node1.toNode2()
2. phase2: Edge1.toEdge2()
3. phase3: Graph1.toGraph2()

Each iteration is confluent and terminating by construction. It can be directly shown that the sequential composition of these phases achieves the specification.

Model migration transformations typically have an associated language interpretation \( \chi \), which defines a syntactic mapping of the source language to the target language, identifying how the concepts of the source language are expressible in the target language. In this case we can define \( \chi \) as follows:

- \( \text{ModelElement1} \mapsto \text{ModelElement2} \)
- \( \text{ModelElement1 :: id1} \mapsto \text{ModelElement2 :: id2} \)
- \( \text{Node1} \mapsto \text{Node2} \)
- \( \text{Node1 :: name} \mapsto \text{Node2 :: text} \)
- \( \text{Edge1} \mapsto \text{Edge2} \)
- \( \text{Edge1 :: src1} \mapsto \text{Edge2 :: src2} \)
- \( \text{Edge1 :: trg1} \mapsto \text{Edge2 :: trg2} \)
- \( \text{Graph1} \mapsto \text{Graph2} \)
- \( \text{Graph1 :: nodes} \mapsto \text{Graph2 :: gcs \cap Node2} \)
- \( \text{Graph1 :: edges} \mapsto \text{Graph2 :: gcs \cap Edge2} \)

No information about the source model is lost by the migration, since the transformation is injective: distinct source models will be mapped to distinct target models. The identity attributes provide a trace facility: a source model element with id1 value \( v \) corresponds to the target model elements with id2 equal to \( v \).

The preservation property \( \text{Pres} \) as \( \text{Asm0} \) is preserved by the transformation, ie, elements of a graph \( g : \text{Graph1} \) are mapped to elements of the transformed form \( g' : \text{Graph2} \) of \( g \). The interpretation \( \chi(\text{Asm0}) \) of \( \text{Asm0} \) is:

\[ \forall g : \text{Graph2} :: (g.\text{gcs} \cap \text{Edge2}).\text{src2} \subseteq g.\text{gcs} \cap \text{Node2} \text{ and (g.\text{gcs} \cap \text{Edge2}).trg2} \subseteq g.\text{gcs} \cap \text{Node2} \]

which follows from the corresponding closure properties of \( \text{src2} \) and \( \text{trg2} \).

The existence of an interpretation implies [6] that there is a reverse transformation, given by:

\[ (R1) : \forall n2 : \text{Node2} ::
\]
\[ \exists n1 : \text{Node1} :: n1.\text{id1} = n2.\text{id2} \text{ and } n1.\text{name} = n2.\text{text} \]
\[(R2) : \forall e2 : Edge2 :\]
\[\exists e1 : Edge1 : e1.id1 = e2.id2 \text{ and} \]
\[e1.src1 = Node1[e2.src2.id2] \text{ and } e1.trg1 = Node1[e2.trg2.id2]\]

\[(R3) : \forall g2 : Graph2 :\]
\[\exists g1 : Graph1 : g1.id1 = g2.id2 \text{ and} \]
\[g1.nodes = Node1[g2.gcs.id2] \text{ and} \]
\[g1.edges = Edge1[g2.gcs.id2]\]

These reverse constraints are also implementable by corresponding operations and a phased algorithm. The reverse mapping however is not injective, and loses information (the text of edges), so is an abstraction.

2.5 Delete nodes

The global specification of this update-in-place transformation can be written as:

\[\forall g : Graph :\]
\[g\text{-edges} \rightarrow \text{select(src.name }= n1 \text{ or trg.name }= n1) \rightarrow \text{isDeleted()} \text{ and} \]
\[g\text{-nodes} \rightarrow \text{select(name }= n1) \rightarrow \text{isDeleted()}\]

The predicate on \(g\) also serves as the definition of an operation \text{remove}(s : String) of Graph that implements the transformation. Since edges depend on nodes, edges are deleted before nodes (the reverse to the ordering used in construction of a model).

2.6 Insert transitive edges

This can be considered as a simple example of a quality-improvement model transformation. Such transformations are typically update-in-place transformations, and have an associated quality measure \(Q : \mathbb{N}\) on the models, used to show termination of the transformation. The transformation aims to reduce \(Q\) to 0 in the target model. In this case \(Q\) is the number of pairs of distinct non-dangling edges \(e1, e2\) of the source model with \(e1.trg = e2.src\) and with no existing edge from \(e1.src\) to \(e2.trg\).

Under the assumption Asm2 that there are not already any duplicate edges in the graph:

\[\forall e1, e2 : Edge : e1.src = e2.src \text{ implies } e1.trg \neq e2.trg\]

the specification of this transformation can be written as:

\[(Cons) :\]
\[\forall g : Graph : \forall e1 : g\text{-edges} @ \text{pre} ; e2 : g\text{-edges} @ \text{pre} :\]
\[e1.trg = e2.src \text{ and } e1.src \neq \{\} \text{ and} \]
\[e1.trg \neq \{\} \text{ and } e2.trg \neq \{\} \text{ implies} \]
\[\exists e3 : Edge : e3.src = e1.src \text{ and } e3.trg = e2.trg \text{ and } e3 : g\text{-edges}\]
This satisfies the non-interference condition (since the created \(e^3\) edges are distinct and are not included in the sets of edges being iterated over), so permitting an implementation using fixed iterations. If instead the transitive closure \(R^+\) of \(R\) was required, Cons would use \(edges\) instead of \(edges@pre\), and a more complex implementation strategy would be required, using repeated iteration until a fixed point is reached [6].

The implementation is generated as a double iteration over edges.

3 Transforming specific models

Source and target metamodels are defined using the visual class diagram editor of UML-RSDS. Metamodels cannot contain multiple inheritance, and all non-leaf classes must be abstract. Metamodels can be saved to a file by the Save data command.

Source models can be defined in text files, which are then read by the executable implementation of the transformation metaclass, in a textual form. A test model of the simple graph metamodel can be defined as follows:

\[
\begin{align*}
g &: \text{Graph} \\
n1 &: \text{Node} \\
n1.name &= "n1" \\
n1 &: g.nodes \\
n2 &: \text{Node} \\
n2.name &= "n2" \\
n2 &: g.nodes \\
e &: \text{Edge} \\
e.src &= n1 \\
e.trg &= n2 \\
e &: g.edges
\end{align*}
\]

This defines a single edge from the first to the second node. Alternative models can be defined in a similar way.

The UML-RSDS toolset, the full code of the transformations, and the generated Java executables, can be found at http://www.dcs.kcl.ac.uk/staff/kcl/ttc11.zip. UML-RSDS can be executed by the command `java UmlTool`. The directory `output` is used to store metamodels, input and output models, and the generated Java code. The command `Load data` loads a metamodel from a file (eg, `mig2.txt` for the migration metamodel). The command `Synthesis Java` generates the Java executable of a transformation, this generated executable is the `Controller.java` file in the `output` directory. This can be compiled and used independently of the toolset.

4 Conclusion

We have shown that UML-RSDS can specify the case study transformations in a direct manner, both as high-level specifications and as explicit designs. UML-RSDS has the advantage of using standard UML and OCL notations to specify...
transformations, reducing the cost of learning a special-purpose transformation language. Our method has the advantage of making explicit all assumptions on models (eg, Asm0 above) and providing global specifications (Cons, Asm, Ens) of transformations, independent of specific rules.

The transformation designs are closely related to high-level logical specifications (PIM specifications) and in many cases can be automatically derived from the specifications, as illustrated in this paper.

References