GMF Model Migration Case Study
Transformation Specification

K. Lano, S. Kolahdouz-Rahimi
Dept. of Informatics, King’s College London, Strand, London, UK*

July 7, 2011

Abstract
In this paper we apply the UML-RSDS notation and tools to the GMF model migration case study and explain how to use the UML-RSDS tools.

1 Model transformation specification in UML-RSDS

UML-RSDS is a model-driven development method with an associated toolset. It was originally designed as a general-purpose method for synthesising verified executable systems from high-level specifications [3], and has been adapted for the synthesis of transformation implementations from specifications [8]. Modelling is carried out using UML 2: class diagram models, use cases, state machines, activities, object models and interactions.

In UML-RSDS the initial specification of a transformation 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 union of the languages of these models. It is intended to hold true at termination of the transformation. It is the postcondition of the use case 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. It is the precondition of the use case of the transformation.

*Research supported by the HoRTMoDA EPSRC project
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.

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.

Various common patterns for the specification predicates have been identified, for example $Cons$ can often be written in conjunctive-implicative form \[7\], 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 designs from the specification, consisting of a sequence of phases, which apply specific rules or operations to achieve the specification constraints.

Provided that the set $wr(Cn)$ of features and entities written by the constraint $Cn$ is disjoint from the set $rd(Cn)$ of features and entities that it reads (the non-interference condition), then a constraint of the above form can be implemented by an iteration

```plaintext
for s : S do s.op()
```

where $op$ is defined as:

```plaintext
op()
post: SCond[self/s] ⇒ T→exists( t | Post[self/s] )
```

This iteration constitutes a single phase in the design. The possible orderings of phases are determined by analysing the data dependencies between the constraints: if $wr(C_1) \cap rd(C_2) \neq \{\}$ then the phase for $C_1$ must precede (or be combined with) that for $C_2$.

The iterative phase activities are also terminating (since only bounded iteration is used), 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 \[6\]. The resulting executable is a stand-alone implementation of the transformation, operating upon simple text format files defining input and output models.
2 GMF model migration

This case study [1] is a re-expression transformation which involves a complex restructuring of the data of a model: actual figures are replaced by references to figures, and references from a figure to subfigures are recorded by explicit objects.

Figure 1 shows the unified metamodels of the source (GMF version 1.0) and target (GMF version 2.1) languages. Since most of the data of a model may remain unchanged by the transformation, we specify the transformation as an update-in-place mapping. Figure 1 is the target metamodel version of the Figure class, figures 1 is the target version of the gallery figure list association end.

Class diagrams can be created using the visual class diagram editor of the UML-RSDS tool (executed by invoking java UmlTool).

We assume in Asm that the input model is a syntactically correct version 1.0 model and that the new entities have no instances:

\[
\text{Figure} 1 = \{\}
\]
\[
\text{FigureDescriptor} = \{\}
\]
\[
\text{ChildAccess} = \{\}
\]

For simplicity of specification, we decompose the transformation into a first transformation which creates the new data from the old, without deleting any
data, and a second transformation which removes the version 1.0 data which is not in version 2.1.

For clarity, we use conventional mathematical notation here, the specification must however be written in the ASCII syntax for OCL when entered into the toolset (Appendix A).

The first transformation is specified by the following Cons constraints:

\[(C1) : \forall f : \text{Figure} \cdot \exists rf : \text{RealFigure} \cdot rf.name = f.name \text{ and} \exists fd : \text{FigureDescriptor} \cdot fd.actualFigure = rf\]

For each source model figure, there is a unique target model real figure, with a figure descriptor.

\[(C2) : \forall f : \text{Figure} \cdot \text{RealFigure}[f.name].children = \text{RealFigure}[f.children.name]\]

For each source model figure, the target model real figure has children the corresponding children (real figures). The notation \(E[x]\) is the instance of \(E\) identified by the key value \(x\), if this is a single value, or the set of \(E\) instances identified by \(x\), if \(x\) is a set.

\[(C3) : \forall fg : \text{FigureGallery} \cdot\]

\(fg.figures1 = \text{RealFigure}[fg.figures.name] \text{ and } fg.descriptors = \text{FigureDescriptor} -\text{select}(\text{actualFigure} : fg.figures1)\)

For each figure gallery, its figures (\(\text{figures1}\)) in the target model are the real figures corresponding to the source model figures of the gallery, its descriptors are the descriptors of these figures. Although in this constraint \(\text{figures1}\) is both written and read, the update only affects the local data of one \(\text{FigureGallery}\) object \(fg\), and no other object is modified, so no other application of the rule is affected.

\[(C4) : \forall f : \text{Figure}; fd : \text{FigureDescriptor}; d : f.referencingElements .\]

\(fd.actualFigure = \text{RealFigure}[f.name] \text{ implies } d.figure = fd \text{ and } (d : \text{DiagramLabel implies} \exists ca : \text{ChildAccess} \cdot d.accessor = ca \text{ and } ca : fd.accessors)\)

The figure descriptor of a diagram element in the target model is that corresponding to the figure which contained the element in the source model. If the diagram element is a label of a nested figure, then an explicit child access object is defined to record the access ([1], page 3).
The following Ens properties can be shown from the constraints:

\[
\begin{align*}
Canvas &= Canvas@pre \\
FigureGallery &= FigureGallery@pre \\
Node &= Node@pre \\
Compartment &= Compartment@pre \\
Connection &= Connection@pre \\
DiagramLabel &= DiagramLabel@pre \\
DiagramElement &= DiagramElement@pre 
\end{align*}
\]

and likewise for the associations which are not modified by the transformation.

Each of the Cons constraints satisfy the non-interference condition, so can be implemented by simple iterations. The phase for \( C_1 \) must precede the phases for the other three constraints, but they can be executed in any order, so the transformation can be decomposed into several separate processes if required. Only \( C_4 \) uses the DiagramElement class and its subclasses, so an input model could be divided into two parts, with the instances of classes Figure, FigureGallery required for \( C_1 \) to \( C_3 \), and instances of the other classes required for \( C_4 \).

\( C_1 \) and \( C_2 \) are implemented by iterations over Figure of operations copyFigure and copyChildren, respectively. \( C_3 \) is implemented by an iteration of an operation copyFigures over FigureGallery. \( C_4 \) is implemented by an iteration of an operation createReferences on Figure.

The BNF syntax of the OCL subset used in UML-RSDS is defined in Appendix A. Metamodels are stored in text files in the output subdirectory, but should not be edited directly, only via the graphical editor of UML-RSDS.

- **copyFigure** is:

  \[
  \text{copyFigure}() \\
  \text{post:} \\
  \quad \text{RealFigure} \rightarrow \exists ( rf \mid rf.\text{name} = \text{name} \& \\
  \quad \quad \text{FigureDescriptor} \rightarrow \exists ( fd \mid fd.\text{actualFigure} = rf ) )
  \]

- **copyChildren** is:

  \[
  \text{copyChildren}() \\
  \text{post:} \\
  \quad \text{RealFigure}[\text{name}].\text{children} = \text{RealFigure}[\text{children.name}]
  \]

- **copyFigures** is:

  \[
  \text{copyFigures}() \\
  \text{post:} \\
  \quad \text{figures1} = \text{RealFigure}[\text{figures.name}] \& \\
  \quad \text{descriptors} = \text{FigureDescriptor} \rightarrow \exists ( \text{actualFigure} : \text{figures1} )
  \]

- **createReferences** is:

  \[
  \text{createReferences}(fd : \text{FigureDescriptor}, d : \text{DiagramElement}) \\
  \text{post:}
  \]

5
\[fd.actualFigure = RealFigure[\text{name}] \Rightarrow d.figure = fd &\]
\[
(d : DiagramLabel \Rightarrow \text{ChildAccess} \text{→exists}(ca | d.accessor = ca & ca : fd.accessors))\]

These are derived directly from the constraints.

`createReferences` is invoked by an activity

\[
\text{for \, fd : FigureGallery do for \, d : referencingElements do createReferences(fd,d) of an operation \text{copyReferences} \, \text{of} \, Figure.}\]

Finally an activity

\[
\text{loadModel("gmf1.txt") ; Figure.copyFigure() ; FigureGallery.copyFigures(); Figure.copyReferences(); Figure.copyChildren(); saveModel("out.txt")}\]

is generated for the system, defining the overall algorithm. Appendix B gives the BNF syntax of activities.

The second transformation removes all instances of `Figure` and all elements and links specific to the source metamodel. It is an update-in-place transformation, with `Cons` specification

\[
\begin{align*}
\text{Figure}@\text{pre}.\text{referencingElements} &= \{\} \\
\text{FigureGallery.figures} &= \{\} \\
\text{Figure} \rightarrow \text{isDeleted}() \\
\end{align*}
\]

This can be coded as the postcondition of an operation `cleanModel` of `Canvas`.

The two transformations are composed by executing one after the other, using an intermediate file to hold the target model of the first transformation, which serves as the source model of the second. For convenience we have combined the transformations into a single executable, `Controller.class` in the `gmf/output` directory.

An example source model (`gmf1.txt`) is as follows:

\[
\begin{align*}
c & : \text{Canvas} \\
c1 & : \text{Compartment} \\
c2 & : \text{Compartment} \\
c1 & : c.\text{ compartments} \\
c2 & : c.\text{ compartments} \\
n1 & : \text{Node} \\
n2 & : \text{Node} \\
n1 & : c.\text{ nodes} \\
n2 & : c.\text{ nodes} \\
l & : \text{DiagramLabel} \\
l & : c.\text{ labels} \\
\end{align*}
\]
The new model generated from this is:

c : Canvas
c1 : Compartment
c2 : Compartment
c : c.compartments
c1 : c.compartments
c2 : c.compartments
n1 : Node
n2 : Node
n1 : c.nodes
n2 : c.nodes
l : DiagramLabel
l : c.labels
fg : FigureGallery
fg : c.figures
rf1 : RealFigure
rf1.name = "f1"
rf2 : RealFigure
rf2.name = "f2"
fd1 : FigureDescriptor
fd1.actualFigure = rf1
fd2 : FigureDescriptor
fd2.actualFigure = rf2
fd1 : fg.figures1
fd1 : fd1.accessors
l.figure = fd1
n1.figure = fd1
c1.figure = fd1
n2.figure = fd2
c2.figure = fd2
c : ChildAccess
c : c.accessors
l.accessor = ca
ca : ca.accessors

The metamodel, Java executable, and test models are in the gmfdirectory in gmf.zip.
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, and loaded by `Load data`.

Source models can be defined in text files, which are then read by the executable implementation `Controller.class` of the transformation, in a textual form. An example is shown above for GMF.

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, `gmfmm3.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. It is compatible with Java SDK version 1.4.1 and later versions, the only specialised Java package used is Java reflection, to load models.

4 Conclusion

We have shown that UML-RSDS can specify the GMF case study transformation in a direct and concise 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 and providing global specifications (Cons, Asm, Ens) of transformations, independent of specific rules.

One deficiency is a lack of graphical specification for transformation rules, ie, by diagrams at the abstract or concrete syntax level. We intend to support such specification as a supplement to the formal specifications of rules.

The transformation designs are closely related to the high-level logical specifications (PIM specifications).

References

A Expression syntax of UML-RSDS

UML-RSDS uses both classical set theory expressions and OCL. It only uses sets and sequences, and not bags or ordered sets, unlike OCL. Symmetric binary operators such as ∪ and ∩ are written in the classical style, rather than as operators on collections. Likewise for the binary logical operators.

A logical op is one of =>, & or. An equality op is one of =, /=, >, <, <=, >=, / (not-in). A factor op is one of +, /, *, -, \ (union), ^ (concatenation of sequences), / \ (intersection). An fe_sequence is a comma-separated sequence of factor expressions. Identifiers can contain ".".

B Activity syntax of UML-RSDS

The following concrete syntax is used for a subset of UML structured activities:
\[< \text{statement} > ::= < \text{loop\_statement} > | < \text{creation\_statement} > | < \text{conditional\_statement} > | < \text{sequence\_statement} > | < \text{basic\_statement} > \]
\[< \text{loop\_statement} > ::= \text{“while”} < \text{expression} > \text{“do”} < \text{statement} > | \text{“for”} < \text{expression} > \text{“do”} < \text{statement} > \]
\[< \text{conditional\_statement} > ::= \text{“if”} < \text{expression} > \text{“then”} < \text{statement} > \text{“else”} < \text{basic\_statement} > \]
\[< \text{sequence\_statement} > ::= < \text{statement} > \text{“;”} < \text{statement} > \]
\[< \text{creation\_statement} > ::= < \text{identifier} > \text{“=”} < \text{identifier} > \]
\[< \text{basic\_statement} > ::= < \text{basic\_expression} > \text{“;=”} < \text{expression} > | \text{“skip”} | \text{“return”} < \text{expression} > | \text{“(”} < \text{statement} > \text{“)”} | < \text{call\_expression} > \]