

# Specification and Verification of Model Transformations using UML-RSDS

K. Lano, S. Kolahdouz-Rahimi  
Dept. of Computer Science, King's College London  
Work carried out within the HoRTMoDA EPSRC project

June 3, 2010

## Abstract

In this paper we describe techniques for the specification and verification of model transformations using a combination of UML and formal methods. The use of UML 2 notations to specify model transformations facilitates the integration of model transformations with other software development processes, and the reflexive application of model transformations. Extracts from three large case studies of the specification of model transformations are given, to demonstrate the practical application of the approach.

## 1 Introduction

Model transformations are mappings of one or more software engineering models (*source* models) into one or more *target* models. The models considered may be graphically constructed using languages such as the Unified Modelling Language (UML) [19], or can be textual notations such as programming languages or formal specification languages.

The concepts of Model-driven Architecture (MDA) [16] and Model-driven Development (MDD) use model transformations as a central element, principally to transform high-level models (such as Platform-Independent Models, PIMs) towards more implementation-oriented models (Platform-Specific Models, PSMs), but also to improve the quality of models at a particular level of abstraction.

The following properties are important for model transformations:

**Syntactic correctness** Can it be shown that a transformation maps correct models of the source language into correct models of the target language?

**Definedness** Is the transformation applicable to every model of the source language?

**Determinacy/Uniqueness** Does the specification define a unique target model from a given source model?

**Rule completeness** The intended effect of a transformation rule can be unambiguously deduced from its explicit specification.

**Language-level semantic correctness** Is there an interpretation  $\chi$  from the source language  $L1$  to the target language  $L2$  such that, for any  $M1$  and  $M2$  which are models of  $L1$  and  $L2$  and are related by the transformation:

$$M1 \models \varphi \Rightarrow M2 \models \chi(\varphi)$$

for each sentence  $\varphi$  of  $L1$ ?

**Confluence** Is the effect of the transformation independent of any alternative orders of application of transformation rules which are allowed by the specification?

In Section 2 we introduce the UML-RSDS specification approach for model transformations, and compare it with other approaches. Section ?? describes particular verification techniques for such specifications. In Section 4 we illustrate the use of UML-RSDS on the UML to relational database schema transformation, in Section 5 we give extracts from the specification of a transformation-based slicing tool. Section 6 describes a transformation from the UML 1.4 activity diagram notation to that of UML 2.2.

## 2 Specification Techniques for Model Transformations

A large number of formalisms have been proposed for the definition of model transformations: *declarative* approaches such as graph grammars [3] or the Relations notation of QVT [21], *hybrid* approaches such as ATL [5] and Epsilon [9], and *imperative* approaches such as Kermeta [7].

Ideally, any specification language for model transformations should support validation, modularity, verification, and the implementation of transformations. Modularity is the key property which supports the other three properties. Transformations described as single large monolithic relations cannot be easily understood, analysed or implemented. Instead, if transformations can be decomposed into appropriate smaller units, these parts can be (in principle) more easily analysed and implemented, and the analysis and implementation of the complete transformation can be composed from those of its parts.

UML-RSDS (UML Reactive System Development Support) is a subset of UML with a precise axiomatic semantics [14], [15] (Chapter 6). A UML-RSDS toolset supports the specification and analysis of systems in this subset, and the generation of executable code from specifications. UML-RSDS can be used to define model transformations in a declarative fashion by *constraints*, which express implicitly how two (or more) models are related, and what changes to one model need to be made (to preserve the truth of the constraints) when a change in another model takes place.

For example, the well-known tree to graph transformation (Figure 1) can be specified by two global constraints<sup>1</sup>:

$$\begin{aligned}
 t : \text{Tree} \text{ implies } & \exists n : \text{Node} \cdot n.name = t.name \\
 t : \text{Tree} \text{ and } t.parent \neq t \text{ and} \\
 n : \text{Node} \text{ and } n.name = t.name \text{ and} \\
 n1 : \text{Node} \text{ and } n1.name = t.parent.name \text{ implies} \\
 & \exists e : \text{Edge} \cdot e.source = n \text{ and } e.target = n1
 \end{aligned}$$


Figure 1: Tree to graph transformation metamodels

This model expresses that there is a mapping between the tree objects in the source model and the node objects in the target model, and that there is an edge object in the target model for each non-trivial relationship from a tree node to its parent. The *identity* constraint means that tree nodes must have unique names, and likewise for graph nodes.

<sup>1</sup> $\exists x : C \cdot P$  abbreviates the OCL formula  $C.allInstances() \rightarrow exists(x | P)$

The constraints correspond directly to the informal requirements for the transformation. In the UML-RSDS approach, such constraints are implemented by identifying for each operation *op* of the system if *op* can potentially invalidate a constraint, either by making the antecedent true or the succedent false [13]. For operations that can affect the constraint the tool generates additional code for *op* which ensures the constraint is preserved.

In this case, a creation operation *createTree* will have additional code generated which creates a *Node* element for the new *Tree* object, and the operation *setparent* will have additional code to create a new *Edge* element if the parent of a tree node is set to be different to itself (and if matching graph nodes for the parent and tree node already exist).

Code generation directly from such high-level constraints may produce highly inefficient code, however. Instead, the constraints can be refined by defining explicit transformation rules as operations, specified by precondition and postcondition predicates, defining how particular elements of the source model are mapped to the target model. These operations are usually placed in new classes, external to the source or target metamodels, but referring to these metamodels.

In this example, the mapping from trees to graph nodes could be expressed by two explicit operations, using the OCL notation of UML:

```
mapTreeToNode(t : Tree) : Node
post:
  Node.allInstances()→includes(result) and
  result.name = t.name
```

```
mapTreeToEdge(t : Tree) : Edge
pre: t ≠ t.parent and
  Node.name→includes(t.name) and
  Node.name→includes(t.parent.name)
post:
  Edge.allInstances()→includes(result) and
  result.source = Node[t.name] and
  result.target = Node[t.parent.name]
```

The notation *Node[x]* refers to the node object with primary key (in this case name) value equal to *x*, it is implemented in the UML-RSDS tools by maintaining a map from the key values to nodes. In OCL it would be expressed as

```
Node.allInstances()→select(name = x)→any()
```

Likewise, *Node.name* abbreviates

```
Node.allInstances()→collect(name)
```

The explicit transformation rules generally permit more efficient implementation than the purely constraint-based specifications. They can be related to the global declarative form by showing, using reasoning in the axiomatic semantics (Chapter 6 of [15]) of UML, that they do establish the constraints:

$$t : Tree \Rightarrow [mapTreeToNode(t)] (\exists n : Node \cdot n.name = t.name)$$

and hence

$$[for t : Tree do mapTreeToNode(t)] (\forall t : Tree \cdot \exists n : Node \cdot n.name = t.name)$$

because the individual applications of *mapTreeToNode(t)* are independent and non-interfering.  $[stat]P$  is the weakest precondition of predicate *P* with respect to statement *stat* (Chapter 6 of [15]).

The UML-RSDS approach to model transformations is a hybrid approach, using standard UML and OCL notations for specification, and the UML-RSDS MDA tool [13] for UML-RSDS notation to generate executable versions of a transformation from its specification. The correctness of individual transformation steps can be verified by a translation to the B formalism, and the correctness of compositions of steps can be verified by using inference rules.

In this approach individual transformation rules are specified as operations using pre/post pairs in the OCL notation. The precondition of the rule identifies when it is applicable, and to which elements of the source model. The postcondition identifies what changes to elements and connections should be made in the target model.

Rules are grouped into *rulesets*, which are UML classes. The attributes of a ruleset are common data used by its contained rules (operations). All rules in a ruleset can be applied in the same phase of a transformation. A ruleset has an algorithm or *application policy* which controls the order and conditions of application of its rules: this can be specified as a UML activity, state machine or other UML behaviour formalism, as the *classifierBehavior* of the ruleset class. In this paper we will use an abstract program notation which is a specific concrete syntax for a subset of UML structured activities. Since the ruleset algorithm is a UML *Behavior*, it can itself be specified by a *BehavioralFeature*, and by pre and post condition constraints. Ruleset verification checks that the composition of rules defined by the algorithm satisfies these constraints, using inferences based on the structure of the algorithm. For example, if  $P \Rightarrow [r1]Q$  and  $Q \Rightarrow [r2]R$  for rules  $r1$  and  $r2$ , then  $P \Rightarrow [r1; r2]R$  for the sequential combination of  $r1$  and  $r2$ .

For the tree to graph transformation, the ruleset activity is defined by:

```
for t : Tree do mapTreeToNode(t) ;
for t : Tree do mapTreeToEdge(t)
```

This also defines the rule application order for the complete transformation.

The overall control of phase ordering and application within a model transformation can also be defined by means of an activity or a state machine. At this level also the task of verification can be decomposed into steps based on the structure of the state machine or activity, using the ruleset specifications.

This three-level structure provides an appropriate concept of modularity, which supports verification, validation and modification of a transformation: individual rules and rulesets can be reused for different transformations, and rulesets can be validated and verified independently of other parts of the transformation. At the same time, only standard UML and OCL notations are used to define transformations, improving reuse of transformations and the integration of transformations with other UML tools.

In terms of the properties discussed in the introduction, UML-RSDS supports the proof of syntactic correctness, semantic correctness, definedness, determinacy and confluence. Completeness checks upon individual rules are also supported. Syntactic correctness can be checked by translating rulesets into B machines and rules into operations of these machines. The proof of internal correctness of the machine will demonstrate that applications of the rules establish the properties of the target language, which are expressed as invariants of the machine. Semantic correctness can be shown by the use of inference rules for [ ]. Definedness is shown by checking that rules are only invoked at points in a transformation where their precondition is true, and that expressions are always defined at the point where they are evaluated. For unbounded loops and recursions, proof of termination using variants is necessary. Determinacy is shown by analysing operation postconditions to check that they are unambiguous, and by establishing confluence for unordered loops.

UML-RSDS has the following advantages compared to other model transformation approaches:

- Standard UML and OCL is used, so that developers do not need to learn a new notation to specify model transformations.
- Transformations can be analysed by other UML tools, such as OCL checkers [24].

- The abstract specification level, using constraints, is inherently bidirectional [28], supporting transformation in both directions between two metamodels.
- Since model transformation specifications are themselves UML models, transformations can be applied to themselves, supporting reflection, a capability absent from most model transformation approaches [6].
- The notation has a formal semantics which supports verification.

UML-RSDS is not restricted to single-target, single-source transformations; mappings involving any number of metamodels can be defined.

### 3 Verification Techniques for Model Transformations

For UML-RSDS specifications of model transformations, syntactic correctness and language-level semantic correctness of individual rules can be verified by translating the specifications to the B formalism, following the set-theoretic definition of UML semantics given in [15].

For the tree to graph example, the transformation metamodels and operation *mapTreeToNode* can be formalised in B as:

```

MACHINE Tree2Graph
SEES String_TYPE
SETS Tree_OBJ; Node_OBJ
VARIABLES trees, nodes, name, nname
INVARIANT
    trees <: Tree_OBJ & nodes <: Node_OBJ &
    name : trees >-> String & nname : nodes >-> String
INITIALISATION
    trees, nodes, name, nname := {}, {}, {}, {}
OPERATIONS
mapTreeToNode(t) =
    PRE t : trees
    THEN
        IF #n.(n : nodes & nname(n) = name(t))
        THEN skip
        ELSE
            ANY nx WHERE nx : Node_OBJ - nodes
            THEN
                nodes := nodes \ / { nx } ||
                nname(nx) := name(t)
            END
        END
    END
END

```

This translation is performed automatically by the UML-RSDS tool. Internal consistency proof of the machine in B then demonstrates syntactic correctness of the *mapTreeToNode* rule: that it maintains the identity constraint of names in the target model. More precisely, it shows that if the source model satisfies the source language constraints, and the existing target model satisfies the target language constraints, then applying the *mapTreeToNode* operation with a true precondition results in a target model which also satisfies the target language constraints.

The same procedure can be used to verify language-level semantic correctness: that source-language properties remain true, in interpreted form, in the transformed model. An example would be the property that a tree has no cycles under the *parent* relationship: this translates into an acyclicity property of the resulting graph.

Checks on definedness, uniqueness and rule completeness can be carried out by syntactic analysis of the rulesets and individual transformation rules. Definedness of a transformation

$\tau$  is ensured if the termination condition  $trm(S_\tau)$  of the B statement  $S_\tau$  corresponding to  $\tau$  is equivalent to *true* [12]. That is,  $[S_\tau]true$  is *true*. In turn, this is ensured if the precondition of each operation is guaranteed to be true each time it is invoked within  $\tau$ , if unbounded loops and recursions can be proved to terminate, and expression evaluations are well-defined. Uniqueness is ensured if all operations have deterministic postconditions, and if all unordered iterations are confluent. In the case of rule completeness, the checks include that for every object  $x$  created or modified in the target model by the rule, all data features of  $x$  are explicitly set by the rule, unless their values can be deduced from default initialisations of the target language, or from derivation constraints from explicitly set features.

Confluence of a specification is the most complex property to establish. We can however define rules which show that bounded unordered loops are confluent in specific cases. A *for*  $x : s$  *do* *acts* loop is a form of loop activity in UML structured activities. In the general case an execution of the loop consists of a collection of executions of  $acts[v/x]$ , one for each element  $v$  of  $s$  at the start of the loop. These executions may occur in any order and may be concurrent.

We assume that the loop body is a single operation call with its parameter ranging over  $s$ .

The inference rule: from

$$v : s \Rightarrow [acts(v)]P(v)$$

derive

$$[for\ v : s\ do\ acts(v)](\forall v : s@pre \cdot P(v))$$

is valid for such loops, provided that one execution of *acts* does not affect another: the precondition of each  $acts(v)$  has the same value at the start of  $acts(v)$  as at the start of the loop, and if  $acts(v)$  establishes  $P(v)$  at its termination,  $P(v)$  remains true at the end of the loop.

Technically, we can say that the  $acts(v)$  are *strongly confluent* with respect to  $P$ , permitting concurrent execution within the iteration statement *stat*: *for*  $v : s$  *do*  $acts(v)$ , if, for  $i \in \mathbb{N}_1$ :

1.  $P(v) \circ \downarrow(acts(v), j) \equiv P(v) \circ \downarrow(stat, i)$  for each  $v \in s \circ \uparrow(stat, i)$  and  $j \in Occ_{stat, i}(acts(v))$ , the occurrences of  $acts(v)$  within the occurrence  $i$  of *stat*.
2.  $Pre_{acts(v)} \circ \uparrow(acts(v), j) \equiv Pre_{acts(v)} \circ \uparrow(stat, i)$  for each  $v \in s \circ \uparrow(stat, i)$  and  $j \in Occ_{stat, i}(acts(v))$ .

$P \circ t$  means  $P$  holds at time  $t$ ,  $e \circ t$  is the value of  $e$  at time  $t$ , and  $\uparrow(S, i)$  and  $\downarrow(S, i)$  are the start and end times of the  $i$ -th occurrence of  $S$  [14].

Together these properties show complete non-interference between the separate invocation instances within the *for* loop, with regard to the truth of  $P$  on elements of  $s$ .

In the common case where the source and target models are separate, and the postcondition of the iterated operation only updates the target model by predicates of the form  $\exists e : E \cdot e.pk_1 = pk$  and  $e.att_1 = v_1$  and ... and  $e.att_n = v_n$  where  $E$  is an entity of the target metamodel, and  $pk_1$  a primary key for  $E$ , and  $pk$  a primary key of the source model class, then the first condition holds: the separate iterations affect distinct variables  $e.att_i$  with the created objects  $e$  being distinct in different iterations, and the effects of the iterations on the set  $E.allInstances()$  are cumulative and non-interfering.

An alternative form, *weak confluence*, instead requires only that the separate invocations do not interfere provided they are executed sequentially:

1.  $P(w) \Rightarrow [acts(v)]P(w)$  for each  $v, w \in s \circ \uparrow(stat, i)$ , with  $v \neq w$ .
2.  $Pre_{acts(w)} \Rightarrow [acts(v)]Pre_{acts(w)}$  for each  $v, w \in s \circ \uparrow(stat, i)$ , with  $v \neq w$ .
3.  $\#active(acts(v)) \leq 1$  for each  $v \in s \circ \uparrow(stat, i)$ .
4.  $\#active(acts(v)) + \#active(acts(w)) \leq 1$  for each  $v, w \in s \circ \uparrow(stat, i)$ ,  $v \neq w$ .

$\#active(m)$  is the number of active executions of action  $m$  at a time point.

In either version of confluence, the inference rule of the for loop can be used to deduce that  $Pre \Rightarrow [stat](\forall v : s@pre.P(v))$  if  $Pre \Rightarrow Pre_{acts(v)}$  for each  $v$ , and if  $Pre_{acts(v)} \Rightarrow [acts(v)]P(v)$  for each  $v$ .

For the tree to graph transformation, both iterations over *Tree* satisfy strong confluence with regard to the respective postconditions of their iterated operations, which allows us to deduce that the constraints of this transformation are achieved by the *Tree2Graph* activity.

Data-dependency analysis or formal proof can be used to establish sufficient conditions for confluence. If we consider only cases where the source and target models are separate, then the writable data items (variables) that can occur in transformation operation postconditions are assumed to be: attributes of the transformation class (ruleset), or features  $obj.f$  of some object  $obj$  of the target metamodel created or modified by this operation. Transformation operations are assumed to not modify any data of the source model.

General syntactic conditions necessary for strong confluence of an unordered loop for  $x : C$  do *stat* are that:

1. The variables written in *stat* are disjoint from those read in *stat*.
2. Written variables  $ref.f$  of *stat* have distinct object references  $ref$  in separate iterations, so that no two iterations modify the same data:

$$o1 \neq o2 \Rightarrow o1.ref \neq o2.ref$$

for  $o1, o2 : C$ .

3. An exception to this is that common written variables are permitted, provided that the updates to these in different iterations are order-independent in their effects.

The following operation is an example where the first condition fails: the variable  $s$  is both written and read in the postcondition. The second condition holds for the features of  $x$  if  $name$  and  $name1$  are primary keys.

*op1()*  
 post:  $\exists x : D \cdot x.name1 = name$  and  $x : s$  and  $x.count = s.size$

The following operation is an example where the third condition fails: the writes to variable  $s$  are not order-independent ( $att1$  and  $att2$  are integer attributes,  $s$  is a set of integers).

*op2()*  
 post:  $\exists x : D \cdot x.name1 = name$  and  
 ( $att1 > 0$  implies  $att2 : s$ ) and ( $att1 \leq 0$  implies  $att2 /: s$ )

Generally, cumulative changes to such shared data, using a single occurrence of an associative update such as  $x : s$  or  $x /: s$  are necessary to ensure confluence.

For weak confluence, we can relax the condition that written data is not read: written variables  $s$  can occur in read form as  $s@pre$ . The following is an example of a weakly confluent operation that is not strongly confluent (assuming that  $name$  and  $name1$  are primary keys):

*op3()*  
 post:  $\exists x : D \cdot x.name1 = name$  and  $x : s$  and  $t = t@pre + 1$

where  $t$  is an integer variable.

## 4 Case Study: UML to Relational Database Schemas

We have used UML-RSDS to specify the well-known transformation from UML class diagrams to relational database schemas. Our specification differs from existing transformations for this mapping, because we avoid the use of recursion between rules. Instead a sequential ordering of separate transformation steps is used.

A simple transformation rule in this transformation is the introduction of a primary key, to a class which does not have one, using the metamodel of Figure 2:

```

introducePrimaryKey(c : UMLClass)
pre:
  c.stereotypes.name → includes("persistent")
  c.ownedAttribute.stereotypes.name → excludes("identity")
  c.feature.name → excludes(c.name + "Id")
post:
  Property.allInstances() → exists(a |
    a.oclIsNew() and
    a.name = c.name + "Id" and
    c.ownedAttribute =
      (c.ownedAttribute)@pre → including(a) and
    a.type = IntegerType and
    Stereotype.allInstances() → exists(s |
      s.oclIsNew() and
      s.name = "identity" and
      a.stereotypes = Set{ s } )

```

$a.oclIsNew()$  specifies that a new object  $a$  is created.  $e@pre$  refers to the value of  $e$  at the start of the operation.

The UML-RSDS toolset checks the completeness of postconditions: for each object which is newly created in an operation, the operation should set all the data features of the object whose value cannot be inferred from default values or derivation relationships. It also permits the implicit specification of changes to inverse association ends (ie., the predicate  $a.classifier = c$  in the above case), to superset association ends (*feature*) and deletion propagation of parts of a composite aggregation. Code for these implicit changes is included in the code generated from the postcondition.

This transformation rule is defined in a metaclass, *TransformationRules*, in the metamodel. This metaclass represents the ruleset to which *introducePrimaryKey* ( $c : UMLClass$ ) belongs. A behaviour can be attached to this class, to identify how the rule should be applied, and, in the case of several rules, in which order they should be applied.

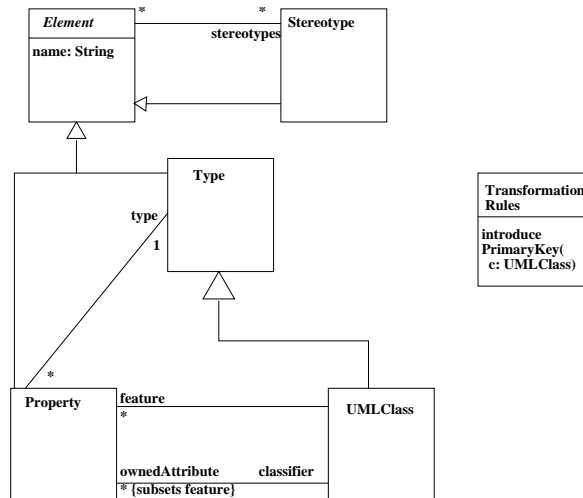


Figure 2: Metamodel for primary key transformation

In this example, the rule should be applied by iterating it over all classes in the source model, in an arbitrary order:

```

introducePrimaryKeys()
for c : UMLClass

```

```

do
  if c.stereotypes.name → includes("persistent") and
     c.ownedAttribute.stereotypes.name →
       excludes("identity") and
     c.feature.name → excludes(c.name + "Id")
  then
    introducePrimaryKey(c)

```

The inference rule: from

$$v : s \Rightarrow [acts(v)]P(v)$$

derive

$$[for v : s do acts(v)](\forall v : s@pre \cdot P(v))$$

is valid for bounded loops, provided that one execution of *acts* does not affect another: the precondition of each *acts*(*v*) has the same value at the start of *acts*(*v*) as at the start of the loop, and if *acts*(*v*) establishes *P*(*v*) at its termination, *P*(*v*) remains true at the end of the loop.

If inheritance has been removed from the model, then the separate iterations of *introducePrimaryKey* are independent and non-interfering, provided they do not overlap in their executions, so it can be deduced that the ruleset satisfies an overall pre-post specification defining the introduction of primary keys to each persistent class.

This ruleset is one step within the model transformation which maps a UML class diagram to a relational database schema (Figure 3).

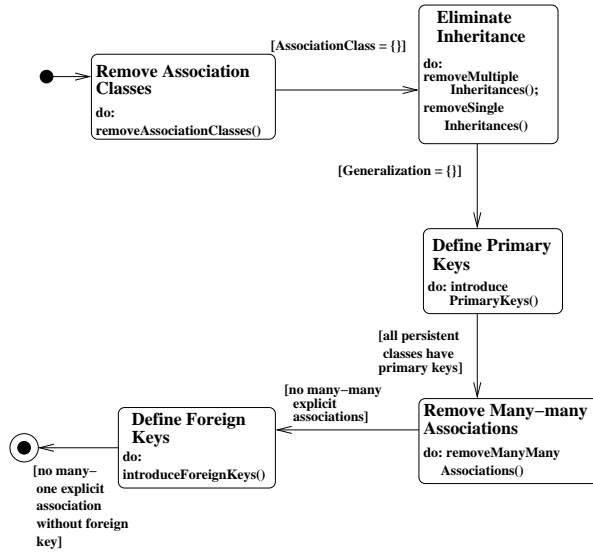


Figure 3: Transformation algorithm

The correctness of a transformation can be demonstrated from that of its individual phases and rulesets. Syntactic correctness follows if there is a series of intermediate languages  $\mathcal{L}_{I_1}, \dots, \mathcal{L}_{I_n}$  such that each ruleset  $\tau_i$  transforms from  $\mathcal{L}_{I_i}$  to  $\mathcal{L}_{I_{i+1}}$ , and so that the starting language for the algorithm combining these rulesets is  $\mathcal{L}_1$ , and the final language is  $\mathcal{L}_2$ .

In the example of Figure 3, the intermediate languages are subsets of the UML class diagram metamodel, which are successively reduced to smaller languages (without the metaclasses *AssociationClass*,

*Generalization*, and with restricted forms of association, etc) until a metamodel isomorphic to that of the relational data model can be used in the final step.

The semantic correctness of this algorithm can be shown by defining suitable state invariants. For example, the entire transformation preserves the property that no class has two different features with the same name, because each individual ruleset preserves this property. Only ruleset specifications are needed for this high-level verification, not the detailed specifications of individual rules.

The transformation specifications can be used directly as input to the code generation tool of the UML-RSDS toolset, which automatically synthesises Java code representing the metamodel (in a similar manner to Kermeta code) and code implementing the transformation rules and any algorithm for their combination.

For the introduce primary key rule, this code is, in part:

```
public void introducePrimaryKey(UMLClass c)
{ if (!(Element.getAllname(
    c.getstereotypes().contains("persistent") &&
    !(Element.getAllname(
        Element.getAllstereotypes(
            c.getownedAttribute()).
            contains("identity"))))
    { return; }
  Property a = new Property();
  Controller.inst().addProperty(a);
  Controller.inst().addownedAttribute(c,a);
  Stereotype s = new Stereotype();
  Controller.inst().addStereotype(s);
  Controller.inst().setname(s,"identity");
  Controller.inst().setstereotypes(a,
    (new SystemTypes.Set()).add(s).getElements());
}
```

The implicit effects of the changes to *ownedAttribute* are explicitly coded in the definition of *addownedAttribute* in the generated code.

In contrast to other specifications of the UML to relational database transformation, our approach uses sequential composition of transformation steps, instead of recursive invocation of rules. This approach has the advantage of simplifying verification, and reducing dependencies between rules. The individual transformation steps can be reused in different contexts, independently of this transformation. New steps can also be added (for example, to remove qualified associations) without the need to modify the details of existing steps. This set of transformations has been incorporated into the UML-RSDS toolset.

## 5 Case Study: State Machine Slicing

Model transformations can be used to carry out the slicing of UML state machines. We have specified a toolset for state machine slicing, using model transformations defined in UML-RSDS. In this section we will give extracts from the specification to illustrate the use of UML-RSDS for a substantial application of model transformations. The state machine metamodel of Figure 4 will be used. The dashed-line associations are used to compute slices and are additional to the UML 2 metamodel for state machines.

Slicing of programs has been a widely-used analysis technique for many years [29] and has also been used for reverse-engineering and re-factoring of code. In general this technique considers a specific point within the program code, such as the end point of the program, and a set of variables of interest at this point, and traces back through the program, discarding any statements which do not contribute to the values of the variables of interest at the selected point.

In the formulation of Harman and Danicic [4] a slice is considered as a transformed version  $S$  of an artifact  $C$  which has a lower value of some complexity measure, but an equivalent

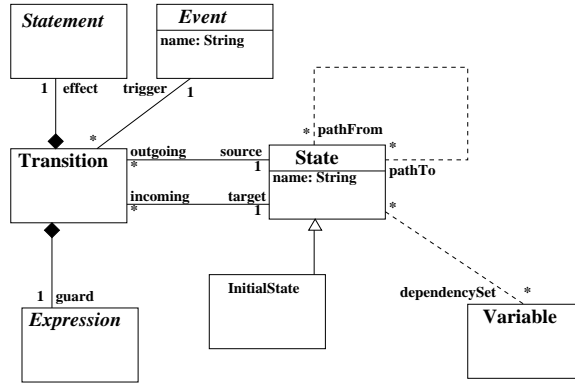


Figure 4: State machine metamodel

semantics with respect to the sliced data:

$$S <_{syn} C \wedge S =_{sem} C$$

The slicing may be structure-preserving, so that  $S$  has the same structure as  $C$  although containing only a subset of its elements, or may be *amorphous*, with a possibly completely different structure.

We use the following criteria for slicing a state machine  $M$ :  $S <_{syn} M$  if  $S$  is syntactically smaller than  $M$ , with no more elements in any category (states, transitions, transition actions, etc) than  $M$ , and fewer elements in at least one category.  $S =_{sem} M$  if for all input sequences  $e$  of events, starting from  $S$  and  $M$  in their initial states, and with the same initial values for their common variables, the state  $s$  of interest is reached by  $S$  as a result of the input sequence whenever it is reached by  $M$  as a result of the same sequence, and then the value of the variables  $V$  of interest in the state  $s$  of interest are the same in the two models.

This definition means that an analyser can deduce properties about  $M$  from properties of  $S$ , for properties which concern the values of  $V$  in  $s$  over all paths to  $s$ . In particular if predicate  $P$  is a state invariant of  $s$  in  $S$ , then it is also a state invariant of  $s$  in  $M$ .

In order to achieve independence of semantic variations of UML state machines, we can define *strict semantic equality*  $S =_{sem}^{str} M$  which only requires the behaviour of  $S$  and  $M$  to be the same for input sequences  $e$  which always trigger explicit transitions in  $M$  at every step.

We will consider transformations which are valid under one or both of these semantics (if a transformation preserves  $S =_{sem} M$  then also it preserves  $S =_{sem}^{str} M$ ).

The following transformations are used to slice state machines:

- *removeStates*: Remove states (and their incoming and outgoing transitions) which cannot be reached from the initial state, or which have no outgoing path to the selected state of the slice.
- *sliceTransitions*: Slice transition actions to remove assignments which cannot affect the value of the variables of interest in the selected state.
- *deleteTransitions*: Delete transitions with a *false* guard.
- *mergeTransitions*: Merge two transitions which have the same sources, targets and actions. The guard of the resulting transition is the disjunction of the original guards.
- *replaceVariablesByConstants*: Replace a feature  $v$  by a constant value  $e$  throughout a state machine, if  $v$  is initialised to  $e$  on the initial transition of the state machine, and is never subsequently modified.

- *mergeStates*: Merge a group  $K$  of states into a single state  $k$  if the states are connected only by actionless transitions and all transitions which exit  $K$  are triggered by events distinct from any of the events that trigger internal transitions of  $K$ .

For example, transitions can be merged if their sources, targets and actions are the same:  $tr1 : s1 \rightarrow_{op(x)[G1]/acts} s2$  and  $tr2 : s1 \rightarrow_{op(x)[G2]/acts} s2$  can be replaced by:

$$tr : s1 \rightarrow_{op(x)[G1 \text{ or } G2]/acts} s2$$

Likewise, state merging can be used to reduce a set of states to a single state: A group  $K$  of states can be merged into a single state  $k$  if:

1. All transitions between the states of  $K$  have no actions.
2. All transitions which exit the group  $K$  are triggered by events distinct from any of the events that trigger internal transitions of  $K$ . If two transitions that exit  $K$  have the same trigger and different targets, they must have disjoint guard conditions.
3. Each event  $\alpha$  causing exit from  $K$  cannot occur on states within  $K$  which are not the explicit source of a transition triggered by  $\alpha$ .

This transformation rule is only valid for the  $S =_{sem}^{str} M$  semantics, an alternative state merging transformation is valid for the  $S =_{sem} M$  semantics.

If the initial state is in a group  $g$  which is merged to a single state  $p$ , then  $p$  is initial in the new state machine.

The overall transformation is defined by the activity:

```
removeStates() ;
deleteTransitions() ;
calculateDependencies() ;
sliceTransitions() ;
replaceVariablesByConstants() ;
mergeTransitions() ;
mergeStates()
```

Semantic correctness of the slicing transformation can be proved by analysis of the individual steps. For example, for the property

$$InitialState.allInstances() \rightarrow size() = 1$$

that there is a unique initial state, only the rulesets *removeStates* and *mergeStates* can fail to preserve this property, and so it is sufficient to establish their correctness with respect to the property, in order to verify the correctness of the entire algorithm.

## 6 Case Study: Model Migration of Activity Diagrams from UML 1.4 to UML 2.2

This transformation was one of the case studies for the 2010 transformation tool competition [25]. It involves the transformation of models of the UML 1.4 activity diagram language [17] into models of the UML 2.2 activity diagram language [20].

In UML 1.4 the language of activity diagrams was a variant of the state machine language. However in UML 2.2, a separate language is defined. The structure of these two languages are quite similar, so the transformation can be specified in a direct manner based on the structure of the source language.

Figure 5 shows a screen shot of the UML-RSDS system with the transformation source metamodel on the left, and the target metamodel on the right.

We assume that there is only one composite state in the source model, the *top* state of the model, an OR-composite state (Page 2-158 of the UML 1.4 superstructure specification

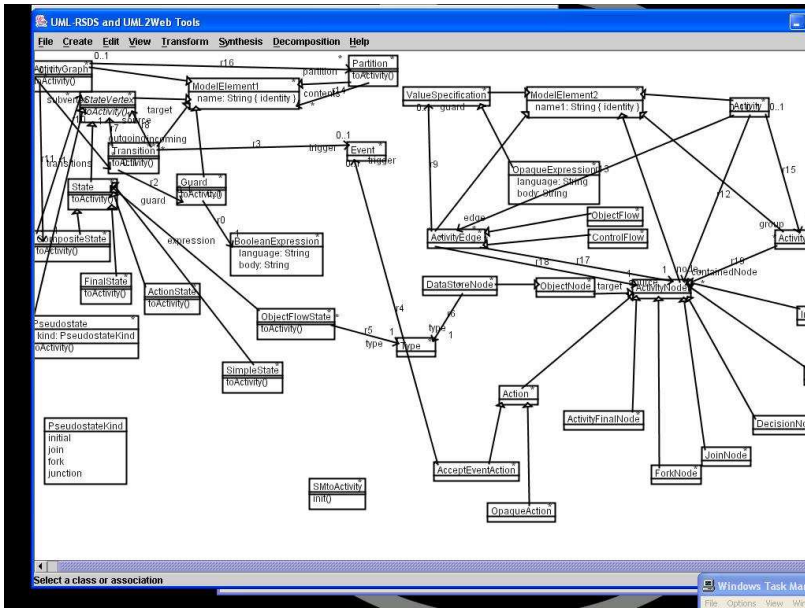


Figure 5: Model Migration case study metamodels

[18]). Then *top.subvertex* in the source model is interpreted by *node* in the target model. An alternative mapping, instead of from *ActionState* to *OpaqueAction*, would be from *ActionState* to *Action*, by mapping the entry action of the UML 1.4 state to UML 2.2.

The transformation is mainly a direct mapping from source language metaclasses and features to corresponding target language metaclasses and features. The only parts requiring significant logic in the transformation are (i) the mapping of different kinds of pseudostate to different kinds of activity nodes, and (ii) the mapping of transitions to control or object flows, depending on what state vertices they connect, and of transitions with triggers to control nodes which receive the trigger event or signal. Figure 6 shows the mapping of a signal-triggered transition from a UML 1.4 *SimpleState* to a semantically equivalent UML 2.2 diagram with an *AcceptEventAction* to consume the triggering event.

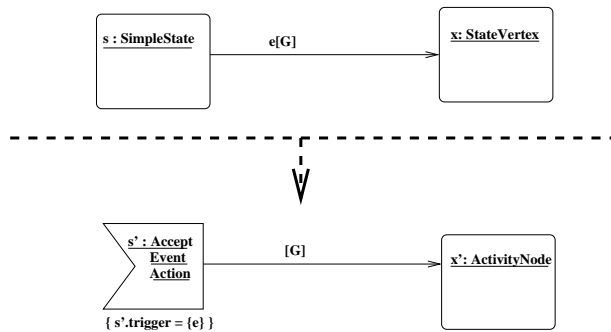


Figure 6: Mapping of signal-triggered transitions

The transformation can be formalised by a set of constraints, which define how the source and target models are related. For example, the correspondence of final states and activity final nodes could be defined by a constraint  $C1$ :

$$f : FinalState \text{ implies } \exists n : ActivityFinalNode \cdot n.name = f.name$$

Likewise for the other source language metaclasses and features. We assume that *name* is

a primary key (identity) attribute for the source model elements. An artificial element id attribute could be introduced instead if *name* is not an identity attribute.

Because of the similarity in structure between the source and target metamodels, the transformation can be specified by rules placed in the source language metaclasses, one per metaclass, to define how that entity should be mapped into the target model. This makes the transformation easy to understand and modify, and also facilitates the use of inheritance.

For states, we have six operations, one for each concrete subclass of *StateVertex*. For example, in *ObjectFlowState*:

```
toActivity()
post:
  ∃ n : DataStoreNode · n.name = name and n.type = type
```

It is assumed that UML 1.4 types can be mapped without change into UML 2.2 types. Similar operations are defined in *FinalState* and *ActionState*.

In *Pseudostate* we define:

```
toActivity()
post:
  (kind = initial implies ∃ n : InitialNode · n.name = name) and
  (kind = join implies ∃ n : JoinNode · n.name = name) and
  (kind = fork implies ∃ n : ForkNode · n.name = name) and
  (kind = junction and incoming.size = 1 implies ∃ n : DecisionNode · n.name = name) and
  (kind = junction and incoming.size > 1 implies ∃ n : MergeNode · n.name = name)
```

A junction state is mapped to a decision node if it has one incoming transition, otherwise to a merge node.

A simple state is assumed to be used in an activity diagram in order to wait for an event to occur (since action states cannot have triggers on their outgoing transitions, page 3-159 of [18]). These states are therefore mapped to *AcceptEventAction* instances:

```
toActivity()
post:
  outgoing.size = 1 and outgoing.trigger.size = 1 implies
  ∃ n : AcceptEventAction · n.name = name and
  n.trigger = outgoing.trigger
```

in *SimpleState*.

In *CompositeState*:

```
toActivity()
  subvertex.toActivity()
```

In this case the composite state is mapped by mapping each of its contained states. We could also specify the creation of a *StructuredActivityNode* with the same name as the composite state, and with *node* set equal to the activity nodes mapped from *subvertex*.

Guards are mapped to opaque expressions:

```
toActivity()
post:
  ∃ oe : OpaqueExpression ·
  oe.name = name and
  oe.language = expression.language and
  oe.body = expression.body
```

in *Guard*.

Transitions are mapped to particular activity edges:

```
toActivity()
pre:
  ActivityNode.name → includes(source.name) and
```

*ActivityNode.name* → *includes(target.name)*

**post:**

(*source* : *ObjectFlowState* or *target* : *ObjectFlowState* implies  
 $\exists f : \text{ObjectFlow} \cdot f.name = name$  and  
 $f.source = \text{ActivityNode}[source.name]$  and  
 $f.target = \text{ActivityNode}[target.name]$  and  
 $f.guard = \text{OpaqueExpression}[guard.name]$  ) and  
(*source* /: *ObjectFlowState* and *target* /: *ObjectFlowState* implies  
 $\exists f : \text{ControlFlow} \cdot f.name = name$  and  
 $f.source = \text{ActivityNode}[source.name]$  and  
 $f.target = \text{ActivityNode}[target.name]$  and  
 $f.guard = \text{OpaqueExpression}[guard.name]$  )

A transition is mapped to an object flow if it has source or target in *ObjectFlowState*, otherwise to a control flow.

The notation *Classname*[*pkset*] can be used to obtain a set of objects of *Classname*, from a set *pkset* of primary key values. For example *OpaqueExpression*[*guard.name*] abbreviates

*OpaqueExpression.allInstances()* → *select( oe | oe.name : guard.name )*

The partitions of the source model are mapped to activity partitions of the target model, with corresponding contents:

*toActivity()*

**post:**

$\exists ap : \text{ActivityPartition} \cdot ap.name = name$  and  
 $ap.containedNode = \text{ActivityNode}[contents.name]$  and  
 $ap.containedEdge = \text{ActivityEdge}[contents.name]$

in *Partition*.

Finally, activity graphs are mapped to activities:

*toActivity()*

**post:**

$\exists a : \text{Activity} \cdot a.name = name$  and  
 $a.node = \text{ActivityNode}[top.subvertex.name]$  and  
 $a.group = \text{ActivityPartition}[partition.name]$  and  
 $a.edge = \text{ActivityEdge}[transitions.name]$

The overall algorithm is specified as the activity of the transformation metaclass *SMtoActivity*:

*init()* ;

*CompositeState.toActivity()* ;

*Guard.toActivity()* ;

*Transition.toActivity()* ;

*Partition.toActivity()* ;

*ActivityGraph.toActivity()*

In general, the mapping of subordinate parts of an element must be performed before the mapping of the element itself.

## 7 Conclusions and Further Work

A similar concept of rule sets with *control expressions* was considered in [11]. Our approach extends this idea, by providing both declarative and imperative specifications for rulesets, and with a concept of refinement between these alternatives. Our approach permits the separate verification of individual rules, of ruleset algorithms against ruleset pre-post specifications, and of complete transformations specified as compositions of rulesets.

Further work includes automating verification steps and the decomposition of verification, using the specification structure, and automated checks on completeness, consistency and confluence of transformations.

## References

- [1] 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.
- [2] J. Cuadrado, J. Molina, *Modularisation of model transformations through a phasing mechanism*, Software Systems Modelling, Vol. 8, No. 3, pp. 325–345, 2009.
- [3] H. Ehrig, G. Engels, H-J. Rozenberg (eds), *Handbook of Graph Grammars and Computing by Graph Transformation*, Volume 2, World Scientific Press, 1999.
- [4] M. Harman, D. Binkley, S. Danicic, *Amorphous Program Slicing*, Journal of Systems and Software, 68 (1): 45 – 69, October 2003.
- [5] F. Jouault, I. Kurtev, *Transforming Models with ATL*, in MoDELS 2005, LNCS Vol. 3844, pp. 128–138, Springer-Verlag, 2006.
- [6] F. Jouault, I. Kurtev, *On the interoperability of model-to-model transformation languages*, Science of Computer Programming, 68, pp. 114–137, 2007.
- [7] Kermet, <http://www.kermet.org>, 2010.
- [8] S. Kolahdouz-Rahimi, *Model Transformation Specification in UML-RSDS*, ICST PhD Symposium, 2010.
- [9] D. Kolovos, R. Paige, F. Polack, *The Epsilon Transformation Language*, in ICMT 2008, LNCS Vol. 5063, pp. 46–60, Springer-Verlag, 2008.
- [10] I. Kurtev, K. Van den Berg, F. Joualt, *Rule-based modularisation in model transformation languages illustrated with ATL*, Proceedings 2006 ACM Symposium on Applied Computing (SAC 06), ACM Press, pp. 1202–1209, 2006.
- [11] Kuske, S., *Transformation Units – A structuring principle for graph transformation systems*, dissertation, Universitat Bremen, 2000.
- [12] K. Lano, *The B Language and Method*, Springer-Verlag, 1996.
- [13] K. Lano, *Constraint-Driven Development*, Information and Software Technology, 50, 2008, pp. 406–423.
- [14] K. Lano, *A Compositional Semantics of UML-RSDS*, SoSyM, vol. 8, no. 1, February 2009, pp. 85–116.
- [15] K. Lano (ed.), *UML 2 Semantics and Applications*, Wiley, 2009.
- [16] OMG, *Model-Driven Architecture*, <http://www.omg.org/mda/>, 2004.
- [17] OMG, *UML Specification, version 1.4*, <http://www.omg.org/spec/UML/1.4/>, 2001.
- [18] OMG, *UML Superstructure Specification, version 1.4*, <http://www.omg.org/spec/UML/1.4/>, OMG document 01-09-67, 2001.
- [19] OMG, *UML superstructure, version 2.1.1. OMG document formal/2007-02-03*, 2007.
- [20] OMG, *UML Specification, version 2.2*, <http://www.omg.org/spec/UML/2.2/>, 2007.
- [21] OMG, *Query/View/Transformation Specification*, ptc/05-11-01, 2005.
- [22] OMG, *Meta Object Facility (MOF) Core Specification*, OMG document formal/06-01-01, 2006.
- [23] OptXware, *The Viatra-I Model Transformation Framework Users Guide*, 2010.
- [24] Richters, M. *A UML-based Specification Environment*, <http://www.db.informatik.uni-bremen.de/projects/USE>, 2001.
- [25] L. Rose, D. Kolovos, R. Paige, F. Polack, *Model Migration Case for TTC 2010*, Dept. of Computer Science, University of York, 2010.
- [26] A. Schurr, *Specification of graph translators with triple graph grammars*, WG '94, LNCS vol. 903, Springer, 1994, pp. 151–163.
- [27] C. Snook, P. Wheeler, M. Butler, *Preliminary Tool Extensions for Integration of UML and B*, IST-2000-30103 deliverable D4.1.2, 2003.
- [28] P. Stevens, *Bidirectional model transformations in QVT*, SoSyM vol. 9, no. 1, 2010.
- [29] M. Weiser, *Program slicing*, IEEE Transactions on Soft. Eng., 10, July 1984, pp. 352–357.