Abstract

The purpose of our research is to evaluate and compare different approaches for the specification, verification and implementation of model transformations, and to make recommendations for a transformation specification language which is modular, verifiable, and supports reuse and implementation.

In this paper we survey existing approaches to model transformations and propose a new specification and implementation approach for transformations. We describe case studies, of state machine slicing, and re-architecting systems for achieving quality of service in service-oriented architectures, which are used to evaluate model transformation specification approaches and languages.

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 graphical languages such as the Unified Modelling Language (UML) [12], or can be textual notations such as programming languages or formal specification languages.

The concepts of Model-driven Architecture (MDA) [11] 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 Platform-Specific Models (PSMs), but also to improve the quality of models at a particular level of abstraction.

As part of the MDA, a standard for Queries, Views and Transformations (QVT) was developed [14], and defined different notations for specifying and implementing model transformations.

The research area of model transformations remains very active, and certain fundamental issues have as yet only partially been solved:

Specification issues Model transformations are relations between languages, i.e., in the binary case, they identify for a pair \((M_1, M_2)\) of models of the respective languages, if \(M_1\) (the source model) is transformed by the transformation to \(M_2\) (the target model).

But for convenience and comprehensibility of specification, transformations are usually defined by transformation rules which relate specific elements of \(M_1\) to specific elements of \(M_2\). This introduces problems of dependency and consistency between rules: for instance, one rule may read data created by another rule, so must be applied after it. Different orders of application of rules to a source model may result in different target models, so that a model transformation is not uniquely specified by a set of element-to-element rules.

Verification issues Transformation rules do not form a good unit of modularity for composition [8] or verification, and higher-level units of modularity such as phases [4] have been proposed instead.

2 Case Studies

Two case studies will be used to investigate the requirements of practical applications of transformations:

1. Slicing of state machines [10]: transformations can be used to reduce the size and complexity of a state machine when only certain aspects of its behaviour are considered. This is a generalisation of program slicing, and can be useful in improving the structure and comprehensibility of the state machine, and in testing. A set of transformation rules have been defined for this process, and these are combined using a complex algorithm in order to carry out slicing.

2. Quality of Service Requirements: Much research has been dedicated to developing good formal methods for checking that overall system Quality of Service (QoS) requirements hold for queued
3 Specification Techniques for Model Transformations

A large number of formalisms have been proposed for the definition of model transformations: the pure relational approach of [1, 2], graphical description languages such as graph grammars [6] or the visual notation of QVT [14], hybrid approaches such as Epsilon [7] and implementation-oriented languages such as Kermeta [5].

As discussed in the introduction, a key problem with the specification of model transformations is that, semantically, model transformations are relations from one (or more) entire model to one or more entire models, but that such a global description is impractical for non-trivial languages and transformations. Instead, model transformations are specified in terms of relations between individual elements in the source model(s) and individual elements in the target model(s). The model-to-model relation is then derived from some composition of these individual relations.

A related problem is that languages such as UML are defined by highly complex metamodels, in which meta-entities depend on each other. A transformation which operates on some \( c : \text{Class} \) in the source model may need to also transform the \( \text{Property} \) elements which are owned attributes of \( c \), the \( \text{Operation} \)s which are owned operations of \( c \), and so forth. In terms of transformation rules, a rule for \( \text{Class} \) will typically depend on rules for \( \text{Property} \) and \( \text{Operation} \), which in turn depend on rules for other metaclasses, including superclasses of \( \text{Class} \), such as \( \text{Type} \).

Rules may be phased, with one set of rules which create objects in the target model being executed before another set of rules which link these objects together.

Ideally, any specification language for model transformations should support validation, modularity, verification, and the implementation of transformations:

**Modularity** It is possible to compose two or more model transformations to form a new model transformation. Compositions could be conditional, iterative, sequencing, or logical combinations such as conjunction.

**Validation** It is possible to analyse the specification to ensure it represents the correct intended transformation. This might be done by inspection, animation, testing, etc.

**Verification** It should be possible to prove that a transformation is semantically correct, ie., that all constraints of the source model remain true in the target model (possibly under some interpretation).

**Implementation** The specification can be used to automatically generate an efficiently executable implementation of the transformation, which is correct with respect to the specification.

Modularity is the key property which supports the other three properties. Transformations described as single large monolithic relations or graph mappings 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.

Three general styles of specification have been used for defining model transformations:

**Declarative** Transformations are described abstractly, eg, as mathematical relations between source and target models [1, 2].

**Imperative** Transformations are defined as programs which explicitly define the details of how a source model is transformed into a target model [5].

**Hybrid** A combination of declarative and imperative, eg, a wide-spectrum specification language in which a declarative description can be refined within the same notation into a program-like description [7].

The declarative style has the advantage that (in principle), transformations can be described more clearly and concisely, omitting the details of strategies for selecting and modifying model elements, and avoiding issues of ordering of application of rules on specific elements. The imperative style on the other hand makes it easier and more direct to implement the transformation in an executable form. The hybrid style attempts
to combine the other two styles in order to obtain the advantages of both.

4 The UML-RSDS approach

Our proposed approach to model transformations is a hybrid approach, using standard UML and OCL notations for specification, and using the UML2Web MDA tool [9] to generate executable versions of a transformation from its specification. A hybrid approach is used to enable all stages of development of transformations to be carried out in a single language, so enhancing reuse and verification.

In this approach declarative transformation rules are specified as pre/post pairs in the OCL notation of UML [13]. The precondition of the rule identifies when it is applicable, and to which elements of the source model. The postcondition identifies what elements and connections should exist in the target model.

A simple transformation rule in this notation is the introduction of a primary key, to a class which does not have one, using the metamodel of Figure 1:

\[
\text{introducePrimaryKey}(c : \text{UMLClass})
\]

**pre:**
\[
c.\text{stereotypes}.\text{name} \rightarrow \text{includes}(\text{"persistent"})
\]
\[
c.\text{ownedAttribute}.\text{stereotypes}.\text{name} \rightarrow \text{excludes}(\text{"identity"})
\]
\[
c.\text{feature}.\text{name} \rightarrow \text{excludes}(c.\text{name} + \text{"Id"})
\]

**post:**
\[
a : \text{Property}
\]
\[
a.\text{oclIsNew}()
\]
\[
s : \text{Stereotype}
\]
\[
s.\text{oclIsNew}()
\]
\[
a.\text{name} = c.\text{name} + \text{"Id"}
\]
\[
s.\text{name} = \text{"identity"}
\]
\[
a.\text{stereotypes} = \text{Set}\{s\}
\]
\[
c.\text{ownedAttribute} =
\]
\[
(c.\text{ownedAttribute})\#\text{pre} \rightarrow \text{including}(a)
\]
\[
a.\text{type} = \text{IntegerType}
\]
\[
a : \text{C} \text{abbreviates C.allInstances()} \rightarrow \text{includes}(a).
\]
\[
a.\text{oclIsNew}() \text{specifies that a new object } a \text{ is created.}
\]
\[
e@\text{pre} \text{refers to the value of } e \text{ at the start of the operation.}
\]

The precondition will be true for any model element \( c \) which is a persistent class without an identity attribute. For such elements the transformation defined by the postcondition will be applied to create a new model from the old, in which a new identity attribute is added to the selected class.

This transformation rule is defined in a metaclass, \( \text{TransformationRules} \), in the metamodel. This metaclass represents the ruleset to which \( \text{introducePrimaryKey} \) belongs. A ruleset may have 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 control-flow formalism. 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.

A simple transformation rule in this notation is the introduction of a primary key, to a class which does not have one, using the metamodel of Figure 1:

\[
\text{introducePrimaryKey}(c : \text{UMLClass})
\]

**pre:**
\[
c.\text{stereotypes}.\text{name} \rightarrow \text{includes}(\text{"persistent"})
\]
\[
c.\text{ownedAttribute}.\text{stereotypes}.\text{name} \rightarrow \text{excludes}(\text{"identity"})
\]
\[
c.\text{feature}.\text{name} \rightarrow \text{excludes}(c.\text{name} + \text{"Id"})
\]

**post:**
\[
a : \text{Property}
\]
\[
a.\text{oclIsNew}()
\]
\[
s : \text{Stereotype}
\]
\[
s.\text{oclIsNew}()
\]
\[
a.\text{name} = c.\text{name} + \text{"Id"}
\]
\[
s.\text{name} = \text{"identity"}
\]
\[
a.\text{stereotypes} = \text{Set}\{s\}
\]
\[
c.\text{ownedAttribute} =
\]
\[
(c.\text{ownedAttribute})\#\text{pre} \rightarrow \text{including}(a)
\]
\[
a.\text{type} = \text{IntegerType}
\]
\[
a : \text{C} \text{abbreviates C.allInstances()} \rightarrow \text{includes}(a).
\]
\[
a.\text{oclIsNew}() \text{specifies that a new object } a \text{ is created.}
\]
\[
e@\text{pre} \text{refers to the value of } e \text{ at the start of the operation.}
\]

The precondition will be true for any model element \( c \) which is a persistent class without an identity attribute. For such elements the transformation defined by the postcondition will be applied to create a new model from the old, in which a new identity attribute is added to the selected class.

This transformation rule is defined in a metaclass, \( \text{TransformationRules} \), in the metamodel. This metaclass represents the ruleset to which \( \text{introducePrimaryKey} \) belongs. A ruleset may have 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 control-flow formalism. 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.

A simple transformation rule in this notation is the introduction of a primary key, to a class which does not have one, using the metamodel of Figure 1:

\[
\text{introducePrimaryKey}(c : \text{UMLClass})
\]

**pre:**
\[
c.\text{stereotypes}.\text{name} \rightarrow \text{includes}(\text{"persistent"})
\]
\[
c.\text{ownedAttribute}.\text{stereotypes}.\text{name} \rightarrow \text{excludes}(\text{"identity"})
\]
\[
c.\text{feature}.\text{name} \rightarrow \text{excludes}(c.\text{name} + \text{"Id"})
\]

**post:**
\[
a : \text{Property}
\]
\[
a.\text{oclIsNew}()
\]
\[
s : \text{Stereotype}
\]
\[
s.\text{oclIsNew}()
\]
\[
a.\text{name} = c.\text{name} + \text{"Id"}
\]
\[
s.\text{name} = \text{"identity"}
\]
\[
a.\text{stereotypes} = \text{Set}\{s\}
\]
\[
c.\text{ownedAttribute} =
\]
\[
(c.\text{ownedAttribute})\#\text{pre} \rightarrow \text{including}(a)
\]
\[
a.\text{type} = \text{IntegerType}
\]
\[
a : \text{C} \text{abbreviates C.allInstances()} \rightarrow \text{includes}(a).
\]
\[
a.\text{oclIsNew}() \text{specifies that a new object } a \text{ is created.}
\]
\[
e@\text{pre} \text{refers to the value of } e \text{ at the start of the operation.}
\]

The precondition will be true for any model element \( c \) which is a persistent class without an identity attribute. For such elements the transformation defined by the postcondition will be applied to create a new model from the old, in which a new identity attribute is added to the selected class.

This transformation rule is defined in a metaclass, \( \text{TransformationRules} \), in the metamodel. This metaclass represents the ruleset to which \( \text{introducePrimaryKey} \) belongs. A ruleset may have an algorithm or application policy which controls the order and conditions of

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

\[
\text{introducePrimaryKeys()}
\]
\[
\text{for } c : \text{UMLClass}
\]
\[
\text{do}
\]
\[
\text{if } c.\text{stereotypes}.\text{name} \rightarrow \text{includes}(\text{"persistent"}) \text{ and }
\]
\[
c.\text{ownedAttribute}.\text{stereotypes}.\text{name} \rightarrow \text{excludes}(\text{"identity"}) \text{ and }
\]
\[
c.\text{feature}.\text{name} \rightarrow \text{excludes}(c.\text{name} + \text{"Id"})
\]
\[
\text{then}
\]
\[
\text{introducePrimaryKey}(c)
\]

A for \( x : s \text{ do acts} \) loop is a form of loop activity in UML structured activities. It can also be represented as a WHILE loop in B, and a for-each loop in Java. 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 be concurrent.

The inference rule: from

\[
v : s \Rightarrow [\text{acts}(v)]P(v)
\]
derive

\[
\forall v : s \models \text{acts}(v) \rightarrow (v : s \models \pre P(v))
\]

is valid for such loops, provided that one execution of \text{acts} does not affect another: the precondition of each \text{acts}(v) has the same value at the start of \text{acts}(v) as at the start of the loop, and if \text{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 \text{introducePrimaryKey} are independent and non-interfering, provided they do not overlap in their executions, so it can be deduced that the ruleset satisfies the pre-post specification:

\begin{verbatim}
introducePrimaryKeys()
post:
UMLClass.allInstances() \rightarrow\forall c \in
   (c.ownedAttribute.stereotype.name)@pre \rightarrow includes(“persistent”) and
   (c.ownedAttribute.stereotype.name)@pre \rightarrow excludes(“identity”) and
   (c.feature.name)@pre \rightarrow excludes(c.name + “Id”) implies
   Property.allInstances() \rightarrow\exists a \in
      a.ocIsNew() and
      a.name = c.name + “Id” and
      c.ownedAttribute =
         (c.ownedAttribute)@pre \rightarrow including(a) and
      a.type = IntegerType and
      Stereotype.allInstances() \rightarrow\exists s \in
         s.ocIsNew() and
         s.name = “identity” and
      a.stereotype = Set{ s })
\end{verbatim}

The ruleset can be used as part of a model transformation which maps a UML class diagram to a relational database schema.

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

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.

5 Conclusions and Further Work

In the remainder of our research we will test out different model transformation approaches on the case studies, and consider what extensions are necessary to the UML-RSDS approach to make it practical for large-scale application.

References