ReMoDeL Compiled

The Cross-Compilation of ReMoDeL to Java by Example

Technical Report

Revision: 2.2

Date: 25 January 2023

Anthony J H Simons
Department of Computer Science
University of Sheffield

Contents

1. Introduction

This document provides a technical overview of the compilation strategy for models, metamodels and model transformations expressed in **ReMoDeL v3**, a high-level syntax for defining models and model transformations. It assumes little prior knowledge of parsing, compilation or code generation. It should be read in conjunction with the companion document *ReMoDeL Explained: An Introduction to ReMoDeL by Example* [1], which explains the notions of models, metamodels and model transformations.

The surface syntax of *ReMoDeL* supports designers in creating abstract designs that closely model the concepts and relations in any given domain. *ReMoDeL* may be considered a high-level language for creating *domain-specific models*, which need tools to interpret, transform or compile and execute them.

1.1 Compilation Strategies

High-level languages can be *interpreted*, or *compiled*.

- An *interpreter* is a program that reads the surface syntax of the language, builds models of the language's expressions, and then simulates their execution.
- A *compiler* is a program that reads the surface syntax of the language, and then generates machine-level instructions that are directly executable on hardware.
- A *cross-compiler* is an intermediate strategy, that converts the surface syntax of the language into another intermediate language, for which a compiler already exists.

All these strategies start with a *parser*, a program for reading the surface syntax and building an internal model of the language's expressions, known as a *parse tree*. The parse tree is checked for correct syntax, as it is constructed. Afterwards, it may be checked to ensure that all concepts are defined before use, or all values have the types expected by the operations. The parse tree may then be used in different ways:

- An *interpreter* will submit the parse tree to an *evaluator*, which simulates the execution of the expressions in the parse tree directly.
- A *compiler* will submit the parse tree to a *code generator*, which will output low-level statements in a directly executable machine language.
- A *cross-compiler* will submit the parse tree to a *code translator*, which will output mid-level statements in another programming language.

1.2 ReMoDeL Compilation Strategy

ReMoDeL v3 uses a *cross-compilation* approach, in which expressions in the surface syntax are converted into packages, classes and methods in the Java programming language. The standard Java compiler is then used to convert these to bytecode, which is interpreted on the standard Java Virtual Machine (JVM).

The syntax of *ReMoDeL v3* is succinct and declarative. The translations into Java may generate up to *three times* the number of lines of code in the older *ReMoDeL v2* format, which encoded models directly in Java. The current approach brings the benefits of model translation to *ReMoDeL* itself.

2. Compiling a Metamodel

A *ReMoDeL* metamodel is cross-compiled into a Java package, containing a set of Java classes, whose fields and methods are derived from the concepts, attributes, relationships and operations in the metamodel. The following correspondences exist:

- *metamodel* – the name of a metamodel is used to generate the name of a Java package. The package name is "*meta.*" plus the lowercase name of the metamodel.
- *concept* – each defined concept corresponds to a *public* Java class having the same name, which belongs to the above package. The class has a single *public* default constructor of the same name.
- *attribute* – each named attribute corresponds to a *protected* Java field having the same name, with default initialisation. The field has *public* getter and setter methods.
- *reference* – each named reference corresponds to a *protected* Java field having the same name, initialised to *null*. The field has *public* getter and setter methods.
- *component* – each named component corresponds to a *protected* Java field having the same name. A single component is initialised to *null*; and collections are initialised to an empty collection of the right type. The field has *public* getter and setter methods.
- *operation* – each named operation corresponds to a declared *public* method. The operation body is translated using code templates for each kind of expression.

We illustrate these correspondences with examples of metamodels taken from the companion document *ReMoDeL Explained* [1].

2.1 The InTree Metamodel

Figure 1 shows the *InTree* metamodel in the *ReMoDeL* visual notation, describing the shape of in-trees in which nodes refer to their parent node [1]. The *Tree* concept is simply a container for a list of *Nodes*. *Tree* also provides an operation to detect the *root Node*. Every *Node* except the *root* has a parent *Node*.

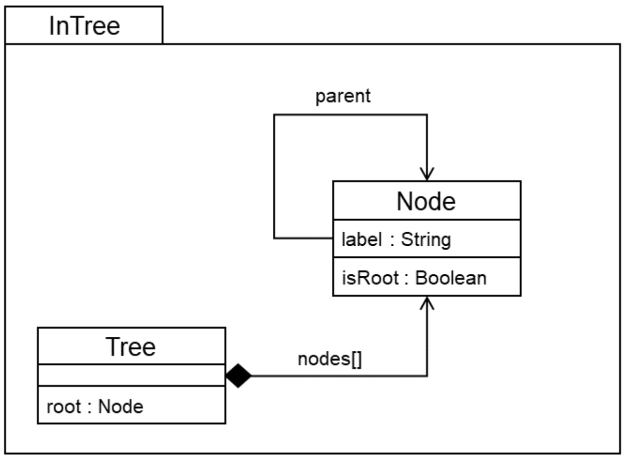

Figure 1: The InTree visual metamodel, with two concepts

The same metamodel is shown in the *ReMoDeL* metamodel syntax in figure 2. This metamodel has the name *InTree*, and contains two concepts called *Node* and *Tree*, where a *Tree* contains a list of *Nodes*, and each labelled *Node* may refer to a parent *Node*.

```
metamodel InTree {
   concept Node {
      attribute label : String
      reference parent : Node
      operation isRoot : Boolean {
         parent = null
      }
   }
   concept Tree {
      component nodes : Node[]
      operation root : Node {
         nodes.detect(node | node.isRoot)
      }
   }
}
```

Figure 2: The InTree textmetamodel, with two concepts

2.2 The InTree Java Translation

The *InTree* metamodel is translated into a Java package called *meta.intree*. The two concepts in the metamodel are translated into two Java classes. Figure3 illustrates the translation of the *Node* concept into a Java class, whose full name is *meta.intree.Node.*

```
package meta.intree;
import remodel.util.*;

public class Node extends Top {
   protected String label = "";
   public String getLabel() {
      return label;
   }
   public Node setLabel(String label) {
     this.label = label;
     return this;
   }

   protected Node parent;
   public Node getParent() {
      return parent;
   }
   public Node setParent(Node parent) {
      this.parent = parent;
      return this;
   }

   public Boolean isRoot() {
      return parent == null;
   }
   public Node() {
   }
}
```

Figure 3: The translated meta.intree.Node class in Java

The class belongs to the package *meta.intree;* and it imports classes from *remodel.util*, a utility library. One of these is the class called *Top*, which is the root class for translated *ReMoDeL* concepts. The fields and methods of the *Node* class are generated according to the following regular translation conventions:

- *properties* – every attribute, reference or component property called *X* of the type *T* is converted into a *protected* Java field of the same name.
- *properties* – as above, are also provided with *public* methods *T getX()* and *N setX(T)*, where *N* is the self-type of the owning class.
- *attributes* – are initialised to default values (here, the empty String "").
- *references* – are initialised to *null* (implicitly, by default Java rules).
- *a default constructor* is provided to create an instance with no initial field values.

These conventions are like those used by *Java Beans* [2], classes designed in a particular way to support runtime discovery of how to set object properties. The standard method-naming conventions allow a compilers to infer which method to use when setting a property with a given name. Setter-methods follow the *Builder Pattern* [3], always returning *this*, the current object. This allows setter-methods to be invoked sequentially, in a nested fashion.

Figure 4 illustrates the translation of the *Tree* concept into a Java class, whose full name is *meta.intree.Tree*. This class defines a list-valued field, *nodes*:

```
package meta.intree;
import remodel.util.*;

public class Tree extends Top {
   protected PureList<Node> nodes = new PureList<Node>(Node.class);
   public PureList<Node> getNodes() {
      return nodes;
   }
   public Tree setNodes(PureList<Node> nodes) {
      this.nodes = nodes;
      return this;
   }

   public Node root() {
      return nodes.detect(new Predicate<Node>(){
        public boolean apply(Node node) {
           node.isRoot();
        });
   }
   public Tree() {
   }
}
```

Figure 4: The translated meta.intree.Tree class in Java

In addition to the conventions described above, the following translation conventions also apply, where collection types are present:

- *collections* – the list-type *T[]* is translated into the *remodel.util* class *PureList<T>*. The set-type *T{}* is translated into the *remodel.util* class *PureSet<T>*.
- *components* – having a list-type *T[]* or set-type *T{}* are also initialised respectively to the empty *PureList<T>,* or empty *PureSet<T>*.
- *collection methods* – apart from basic *add* and *remove* methods (used internally), the collection classes *PureList<T>* and *PureSet<T>* have constructive insertion, removal, composition and decomposition methods that always return new collections.
- *higher order methods* – higher-order collection methods, such as: *detect, select, reject, collect*, accept further *Predicate* or *Function* lambda-expression arguments.

- *lambda expressions* – a lambda expression passed as a predicate is translated by an anonymous *Predicate* subclass, with an *apply()* method returning *true* or *false*.

Apart from the explicit initialisation of *String, PureListT>* and *PureSet<T>* types, the compiler relies on Java's own default initialisation rules for *int, double, char, boolean* and reference types, which all receive the empty value, *zero, false or null*.

2.3 The OutTree Metamodel

Figures 5 and 6 show the *OutTree* metamodel, describing the shape of out-trees in which nodes contain their own children [1]. The *Tree* is simply a container for the root *Node*.

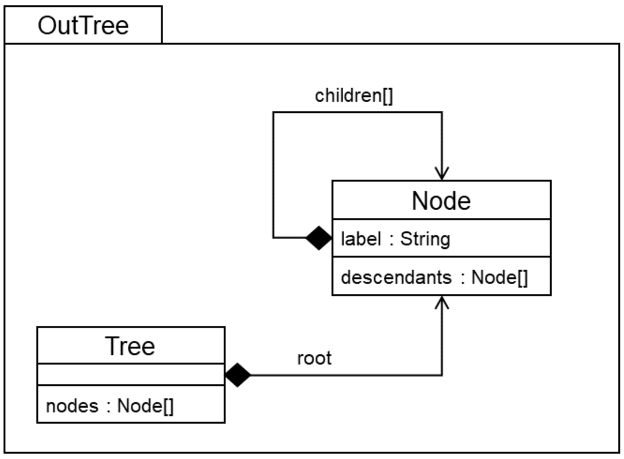

Figure 5: The visual OutTree metamodel, with two concepts

```
metamodel OutTree {
   concept Node {
      attribute label : String
      component children : Node[]
      operation descendants : Node[] {
         children.append(
            children.collate(child | child.descendants))
      }
   }
   concept Tree {
      component root : Node
      operation nodes : Node[] {
         root.asList.append(root.descendants)
      }
   }
}
```

Figure 6: The text OutTree metamodel, with two concepts

This metamodel has the name *OutTree*, and contains two concepts called *Node* and *Tree*. *Tree* contains the *root* Node; and each labelled *Node* may contain further *Node* children. What is new is that the complete list of *nodes* must now be computed by an operation of *Tree* (c.f. *InTree* in figure 2, in which the list was stored). This allows examination of a more complex method translation for the operation.

2.4 The OutTree Java Translation

The *OutTree* metamodel is translated into a Java package called *meta.outtree*, containing two classes. Figure 7 shows the translation of the *Node* concept into a Java class, whose full name is *meta.outtree.Node.* The class belongs to the package *meta.outtree;* and it imports classes from the *remodel.util* utility library. The *Node* class inherits from the library class *Top*, and uses the library class *PureList*.

```
package meta.outtree;
import remodel.util.*;

public class Node extends Top {
   protected String label = "";
   public String getLabel() {
      return label;
   }
   public Node setLabel(String label) {
     this.label = label;
     return this;
   }

   protected PureList<Node> children =
      new PureList<Node>(Node.class);
   public PureList<Node> getChildren() {
      return children;
   }
   public Node setChildren(PureList<Node> children) {
      this.children = children;
      return this;
   }

   public PureList<Node> descendants() {
      return children.append(children.collate(
         new Function<Node, PureList<Node>>() {
            public PureList<Node> apply(Node child) {
               return child.descendants();
            }
      }));
   }
   public Node() {
   }
}
```

Figure 7: The translated meta.outtree.Node class in Java

This generated Java class contains fields and methods according to the translation scheme described previously. We highlight the following new aspects:

- *mapping operations* – collections offer two mapping operations, *collect()* and *collate().* The former accepts a lambda-expression returning a single element; the latter, seen here, accepts a lambda-expression returning a collection.
- *lambda expressions* – a lambda expression passed as a function is translated by an anonymous *Function* subclass, with an *apply()* method returning the result type. Here, the function accepts a *Node* and returns a *PureList<Node>*.
- The *collate()* operation, when applied to a *PureList<Node>,* merges the results returned by the lambda-expression as a single *PureList<Node>*. When applied to a

PureSet<T>, it returns a single merged set. The result type of the lambda-expression must follow the type of the original collection.

The ordering of the *descendants()* method performs a breadth-first traversal of the *Node's* children, in which the immediate *children* of the *Node* appear in the result, before their recursive *descendants()*. This is achieved by the order of arguments to *append()*. The only *Node* not included in *descendants()* is the top node on which this is invoked.

Figure 8 illustrates the translation of the *Tree* concept into a Java class, whose fully qualified name is *meta.outtree.Tree*. This class defines a single root field, with an operation for calculating the complete list of nodes:

```
package meta.outtree;
import remodel.util.*;

public class Tree extends Top {
   protected Node root;
   public Node getRoot() {
      return root;
   }
   public Tree setRoot(Root root) {
      this.root = root;
      return this;
   }

   public PureList<Node> nodes() {
      return asList(root).append(root.descendants());
   }
   public Tree() {
   }
}
```

Figure 8: The translated meta.outtree.Tree class in Java

The complete breadth-first traversal of the tree by *nodes()* is now apparent. The *root* node is converted into a singleton list, and onto this is appended the root's descendants, as computed above. We highlight the following:

- *asList* and *asSet* – are available to every *ReMoDeL* concept, as if they were invoked operations, defined for the *Top* concept and inherited by every other concept.
- They are translated into *static methods* of the library class *remodel.util.Top*, by transposing the object receiver into an argument. This allows Java generic type inference to work correctly on the argument.

The translation of *asList* and *asSet* is the same, whether the receiver is a single concept or a collection. The library class *Top* provides multiple overloaded versions of the static methods, to convert the Java object correctly.

3. Compiling Higher-Order Operations

Figure 7 above gave an example of a higher-order operation *collate()* and how this is translated into Java. Here, we discuss the general approach to compiling higher-order operations. *ReMoDeL* collections support four kinds of higher-order operation:

- **predicates**:
  ```
  forall(Predicate): Boolean,
  exists(Predicate): Boolean
  ```
- **filtering**:
  ```
  select(Predicate): Collection,
  reject(Predicate): Collection,
  detect(Predicate): Element
  ```
- **mapping**:
  ```
  collect(Function): Collection,
  collate(Function): Collection
  ```
- **reducing**:
  ```
  reduce(Reduction): Element
  ```

The receiver of these invocations is a *list* or *set*, with the *ReMoDeL* list type *T[]* or set type *T{}* for some element type *T*. Each of the higher-order operations accepts a lambda-expression (a *Predicate, Function* or *Reduction*) and returns either a *Boolean* value, a *Collection* (of the same type as the receiver), or an *Element* (of the receiver).

3.1 Translation of Lambda-Expressions

A simple lambda-expression is any expression of the following general form, in which the lambda-variable *element* may optionally be given an explicit type:

```
(element | expression-referring-to-element)
(element : T | expression-referring-to-element)
```

The *expression-referring-to-element* may be any kind of *ReMoDeL* expression involving the *element*. The lambda-variable *element* has the type *T*. This is either given explicitly, or inferred from the collection-type of the receiver. The result-type of the lambda-expression depends on the expression's result, but is always *Boolean* for predicates.

Another kind of lambda-expression has two lambda-variables *elem1, elem2* which may optionally be given an explicit type:

```
(first, second | expression-combining-first-and-second)
(first, second : T | expression-combining-first-and-second)
```

The *expression-combining-first-and-second* may be any kind of *ReMoDeL* expression that performs some combination of the elements. The two lambda-variables *first* and *second* have the same type *T*. This is either given explicitly, or inferred from the collection-type of the receiver. The result-type of this kind of lambda-expression is also *T*.

Lambda-expressions are translated either as a *Predicate*, or a *Function*, or a *Reduction*. This choice is determined by what the higher-order method expects as its argument (see above). The types *Predicate*, *Function* and *Reduction* are provided as Java interfaces in the *ReMoDeL* standard library *remodel.util*. These have the form as shown in figure 9, in which the argument and result types are expressed as generic parameters.

```
public interface Predicate<T> {
   public abstract boolean apply(T element);
}

public interface Function<T, S> {
   public abstract S apply(T element);
}

public interface Reduction<T> {
   public abstract T apply(T first, T second);
}
```

Figure 9: The interfaces Predicate, Function and Reduction

Any given lambda-expression is translated into an instance of an *anonymous subclass* of one of these interfaces, whose *apply()* method translates the body of the lambda-expression. This captures the notion of a closure, that is, a dynamically created body of code, which exists within a surrounding static scope and may refer freely to variables in this scope.

Three examples of generated Java expressions that construct an instance of an anonymous subclass are shown in figure 10. This uses the Java syntax for anonymous classes, where the anonymous concrete type to be instantiated is given by a construction call to the abstract interface, followed by an overriding class-body, in which a concrete implementation of the interface's abstract method is provided:

```
new Predicate<Node>() {
   public Boolean apply(Node node) {
      return node.isRoot();
   }
};

new Function<Node, PureList<Node>>() {
   public PureList<Node> apply(Node child) {
      return child.descendants();
   }
};

new Reduction<Integer>() {
   public Integer apply(Integer first, Integer second) {
      return first + second;
   }
}
```

Figure 10: Anonymous Predicate, Function and Reduction instances

The *ReMoDeL* compiler is able to replace the generic type parameters by the actual types found in context. The first two examples are taken from the generated expression code found in the translations of figures 4 and 7. The *Predicate* instance accepts a *Node* and returns a *Boolean* result. The *Function* instance accepts a *Node* and returns a *List<Node>* (in this example). The *Reduction* instance accepts two *Integers* and returns the *Integer* sum.

3.2 Translation of Collection Operations

The anonymous *Predicate*, *Function* or *Reduction* instances are passed as arguments to the translation of the higher-order *Collection* operations. These are predefined in the *ReMoDeL* standard library *remodel.util*, as the methods of the *PureSet<T>* and *PureList<T>* classes, which respectively implement the *ReMoDeL* types *T{}* and *T[]*.

```
public PureList<T> reject(Predicate<T> predicate) {
   PureList<T> result = new PureList<T>(getElemType());
   for (T element : elements) {
      if (! predicate.apply(element))
         result.elements.add(element);
   }
   return result;
}

public <S> PureList<S extends T> select(Predicate<T> predicate) {
   PureList<S> result = new PureList<S>();
   for (T element : elements) {
      if (predicate.apply(element))
         result.elements.add((S) element);
   }
   return result;
}

public <S> PureSet<S> collect(Function<T, S> function) {
   PureSet<S> result = new PureSet<S>()
   for (T element : elements) {
      result.elements.add(function.apply(element);
   }
   return result;
}

public T detect(Predicate<T> predicate) {
   for (T element : getElements()) {
      if (predicate.apply(element))
         return element;
   }
   return null;
}

public T reduce(Reduction<T> reduction) {
   T result = null;
   Iterator<T> = getElements().iterator();
   if (iter.hasNext())
      result = iter.next();
   while (iter.hasNext())
      result = reduction.apply(result, iter.next());
   return result;
}
```

Figure 11: Some examples of translated higher-order operations.

Figure 11 shows examples of the translation of higher-order operations, including some defined in *PureCollection, PureList* and *PureSet*. All of the operations *select(), reject(), collect()* and *collate()* create and return collections of the same kind (set or list), whose result-type is inferred from the context. The implementation of *select()* optionally allows the result type to be more specific than the receiver type. This supports filtering heterogeneous collections to extract specifically-typed subsets.

The implementations wrap an underlying Java collection *elements*, which is traversed but not modified. This is either accessed directly, or dynamically through the method *getElements()*. Methods always build up a new collection as the result. When translating the *ReMoDeL* creation expression for an empty collection, a Java *Class*-object is stored, denoting the element type, and this is used to model the type of added elements.

We could have used the Java 1.8 streaming API to implement higher-order operations, but chose not to, because some of the generated code would have been more complicated. Our Java translation follows the *ReMoDeL* expressions closely.

3.3 The Graph Metamodel

Figures 12 and 13 show the metamodel for a *Graph*, a single-rooted, directed graph in which the connections between each *Vertex* pair is modelled explicitly by an *Edge*. The *Graph* is simply the collections of its *vertices* and *edges*. This presents a more interesting use of higher-order operations to find the root of the *Graph*.

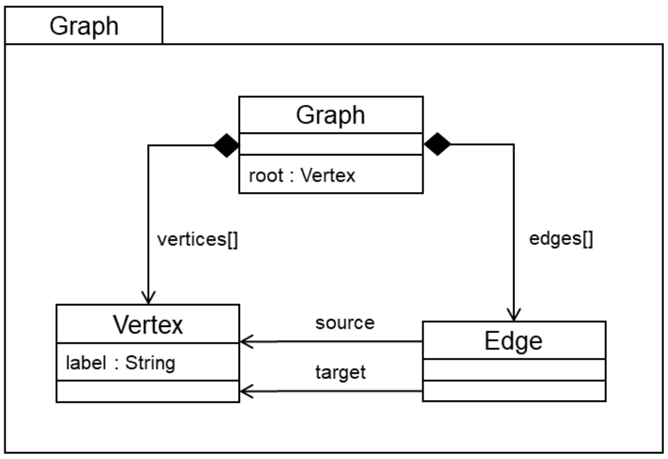

Figure 12: A visual metamodel for Graph, with three concepts

```
metamodel Graph {
   concept Graph {
      component vertices : Vertex[]
      component edges : Edge[]
      operation root : Vertex {
         vertices.detect(vertex |
            not edges.exists(edge | edge.source = vertex))
      }
   }
   concept Vertex {
      attribute label : String
   }
   concept Edge {
      reference source : Vertex
      reference target : Vertex
   }
}
```

Figure 13: A text metamodel for Graph, with three concepts

3.4 The Graph Java Translation

Figure 14 shows just a part of the translation of this metamodel, focusing on the translation of the *Graph* concept, generating the Java class *meta.graph.Graph*. The other two classes are generated following a similar pattern, described in section 2.

```
package meta.graph;
import remodel.util.*;

public class Graph extends Top {
   protected PureList<Vertex> vertices =
         new PureList<Vertex>(Vertex.class);
   public PureList<Vertex> getVertices() {
      return vertices;
   }
   public Graph setVertices(PureList<Vertex> vertices) {
      this.vertices = vertices;
      return this;
   }
   protected PureList<Edge> edges =
         new PureList<Edge>(Edge.class);
   public PureList<Edge> getEdges() {
      return edges;
   }
   public Graph setEdges(PureList<Edge> edges) {
      this.edges = edges;
      return this;
   }
   public Vertex root() {
      return vertices.detect(new Predicate<Vertex>() {
         public boolean apply(Vertex vertex) {
            return !edges.exists(new Predicate<Edge>() {
               public boolean apply(Edge edge) {
                  return edge.getSource() == vertex;
               }
            });
         }
      });
   }
   public Graph() {
   }
}
```

Figure 14: The translated meta.graph.Graph class in Java

We highlight the translation of the *root()* method, which returns the single root *Vertex* of the *Graph*. This invokes two higher-order methods, *detect()* and *exists()*. The outer invocation has a predicate which searches the *vertices* for the single *vertex* satisfying a nested condition, expressed by the inner invocation, which searches the *edges* until it cannot find any *edge* which has the *vertex* as its *source*.

The anonymous subclasses of *Predicate* contain implementations of the *apply()* method that may capture the values of variables that occur within the enclosing scope. So, for example, the inner invocation (filtering *edges*) is able to refer to the variable *vertex*, defined in the scope of the outer invocation. This variable occurrence is *effectively final* in Java, meaning that it cannot be modified. Java's rules for compiling anonymous classes ensure that captured variables are copied into the inner context.

4. Compiling Concept Inheritance

ReMoDeL concepts enter into classification relationships. A concept may stand alone, or may be defined as a subtype of some other concept. A subtype concept is defined using the *inherit* keyword, indicating that the subtype inherits all of the properties of the supertype. The translation follows slightly different conventions, to ensure that access is preserved to predefined methods declared in the root concept *Top*.

- *standalone concept* – the translation must always generate a Java class that extends the *remodel.util.Top* class. This must be added to the import list.
- *extended concept* – the translation must always generate a Java class that extends the class corresponding to the translation of the concept from which it inherits. In this case, *remodel.util.Top* should not be added to the import list.
- *setter-methods* – the translation of an extended concept must redefine all setter methods to return a more specifically typed *this* reference. The generated code merely has to type-downcast the result of the inherited setter-method.

4.1 The ERM Metamodel

Figure 15 illustrates a visual metamodel describing the concepts present in an *Entity Relationship Model* (ERM). The metamodel is possibly incomplete, but shows the overall structure. The metamodel specifies that a *Diagram* consists of a set of *Entities* and a set of *Relationships*, where each *Entity* consists of a set of *Attributes*, and each *Relationship* consists of two *EndRoles* called the *source* and *target*, each referring to an *Entity*.

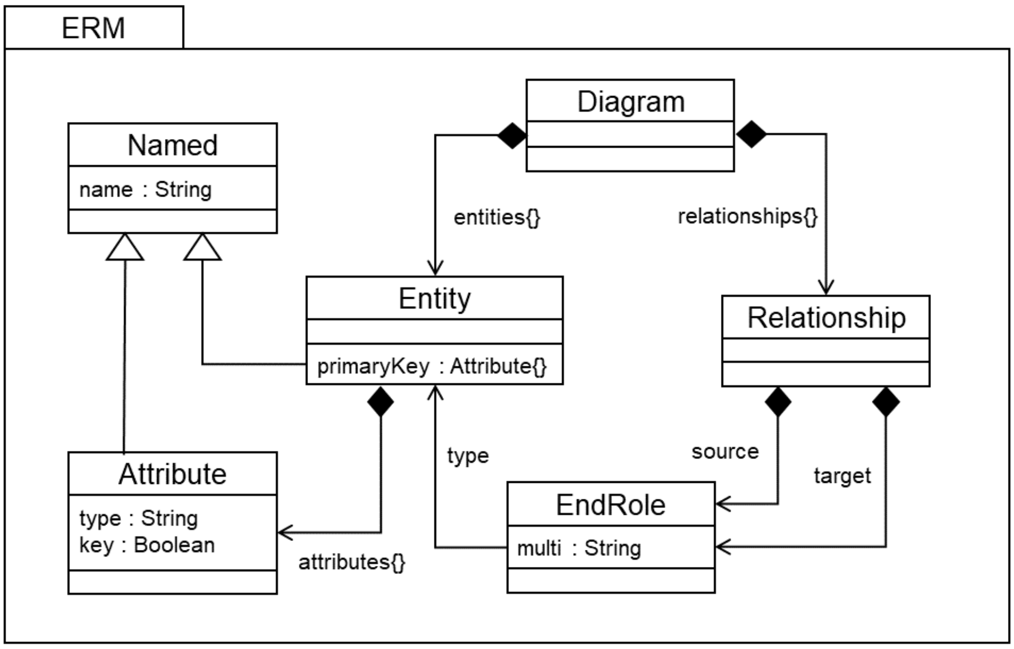

Figure 15: The visual ERM Metamodel

The *Entity* and *Attribute* concepts inherit from the *Named* concept. This is shown using the generalisation arrow linking the subtypes to the supertype. By virtue of this relationship, both *Entity* and *Attribute* inherit the *name* attribute that is defined in *Named*. We will explore the consequences of this in the translation to Java, in the following section.

```
metamodel Erm {
   concept Named {
      attribute name : String
   }
   concept Entity inherit Named {
      component attributes : Attribute{}
      operation primaryKey : Attribute{} {
         attributes.select(attribute | attribute.key)
      }
   }
   concept Attribute inherit Named {
      attribute type : String
      attribute key : Boolean
   }
   concept Relationship {
      component source : EndRole
      component target : EndRole
   }
   concept EndRole {
      reference type : Entity
      attribute multi : String
   }
   concept Diagram {
      component entities : Entity{}
      component relationships : Relationship{}
   }
}
```

Figure 16: The text ERM metamodel

Figure 16 shows the same metamodel in the textual format. This shows the detail of the *primaryKey* operation, which filters a subset of an *Entity's Attributes*, returning only those *Attributes* in which *key* is *true*.

4.2 The ERM Java Translation

Figure 17 shows the Java translation of the standalone concept *Named*. This follows the same translation scheme as described previously. The *ReMoDeL* attribute *name* is translated by the compiler into a Java field, also called *name*, along with its getter- and setter-methods.

```
package meta.erm;
import remodel.util.*;

public class Named extends Top {
   protected String name = "";
   public String getName() {
      return name;
   }
   public Named setName(String name) {
      this.name = name;
      return this;
   }
   public Named() {
   }
}
```

Figure 17: The translated meta.erm.Named class in Java

Next, we focus on how the *ReMoDeL* compiler determines whether a translation requires any special Java import statements. If the generated Java class relies on any class provided in the *ReMoDeL* standard library, then the compiler must generate an import statement:

```
import remodel.util.*;
```

This makes any library class available in the new context. However, if no library class is required, then it is an error to include such an import statement (it would raise a warning in most Java IDEs). In this case, the import statement should be suppressed. In figure 17, it is clear that *Named* refers to the *remodel.util.Top* class, so this library is imported.

Figure 18 shows the Java translation of the extended concept *Attribute*, which inherits from *Named*. In the Java translation, the generated *Attribute* class *extends* the *Named* class. By virtue of inheriting from *Named*, this class transitively inherits from *Top*, so does not refer to *Top* directly. Since it makes no reference to other library classes, the library import statement is suppressed, to avoid generating Java code that issues *unused import* warnings.

```
package meta.erm;

public class Attribute extends Named {
   protected String type = "";
   public String getType() {
      return type;
   }
   public Named setType(String type) {
      this.type = type;
      return this;
   }
   protected boolean key;
   public boolean getKey() {
      return key;
   }
   public Attribute setKey(boolean key) {
      this.key = key;
      return this;
   }
   public Attribute setName(String name) {
      return (Attribute) super.setName(name);
   }
   public Attribute() {
   }
}
```

Figure 18: The translated meta.erm.Attribute class in Java

Next, we focus on how the *ReMoDeL* compiler handles the types returned by setter-methods. Recall that all generated setter-methods must return *this*, the current object, so that the setter-methods may be sequentially nested, following the Builder Pattern [3]. This pattern is widely used in transformation rules (see section 5).

In figure 17, the class *Named* provided a method *setName(String)* which returned *this*, of the type *Named*. If an instance of *Attribute* were to use this inherited method to set its own *name*, the result would be returned as a *Named* instance, not an *Attribute* instance. For this reason, in figure 18, the method *setName(String)* is redefined to override the returned type of *this*, to ensure the type is *Attribute*.

If this were not done, then the following sequence of Java invocations would fail:

```
new Attribute().setName("length").setType("Integer");
```

because the result of *setName()* is a *Named* object, for which the method *setType()* is not defined. By providing the overriding method, which casts down the result type from *Named* to *Attribute*, this problem is avoided. The redefined method is generated following a template, which uses the original method and casts down the result type.

```
package meta.erm;
import remodel.util.*;

public class Entity extends Named {
   protected PureSet<Attribute> attributes =
           new PureSet<Attribute>(Attribute.class);
   public PureSet<Attribute> getAttributes() {
      return attributes;
   }
   public Entity setAttributes(PureSet<Attributes> attributes) {
      this.attributes = attributes;
      return this;
   }
   public PureSet<Attribute> primaryKey() {
      return attributes.select(new Predicate<Attribute>() {
         public boolean apply(Attribute attribute) {
            return attribute.getKey();
         }
      });
   }
   public Entity setName(String name) {
      return (Entity) super.setName(name);
   }
   public Entity() {
   }
}
```

Figure 19: The translated meta.erm.Entity class in Java

Figure 19 shows the Java translation of the extended concept *Entity*, which also inherits from *Named*. Similar to figure 18, the *Top* class is not referred to directly (but is inherited transitively through *Named*). However, in this case, we must still supply an import statement, because the *Entity* class uses the *PureSet<T>* library class.

Figure 19 also redefines the *setName(String)* method to override the type of the result, so that *this* has the more specific type *Entity* (rather than *Named*). In this way, any generated Java class which extends another Java class must override any inherited setter-method, in order to return the more specific type of *this*.

5. Compiling a Model Transformation

A *ReMoDeL* transformation is cross-compiled into a Java class, in which the individual mapping or merging rules are translated as the methods of this class. A transformation belongs to a group, which is translated into a Java package owning the generated class.

The following correspondences exist:

- *transformation name* – the name of the transformation becomes the name of a Java class that is the translation of the transformation.
- *transformation group* – the declared group becomes (part of) the name of a Java package containing the class denoting the transformation.
- *source metamodel* – each declared source metamodel is mapped to a Java package corresponding to the translation of this metamodel.
- *target metamodel* – the declared target metamodel is mapped to the Java package corresponding to the translation of this metamodel.
- *mapping rule* – each mapping rule corresponds to a public method of the generated class accepting an argument from the source metamodel. The rule caches its result from the target metamodel, based on the source key.
- *merging rule* – each merging rule corresponds to a public method of the generated class accepting an argument from each source metamodel. The rule caches its result from the target metamodel, based on a key computed from all sources.
- *function rule* – each auxiliary function corresponds to a public method of the generated class. Its result is not cached.

5.1 Model Transformation

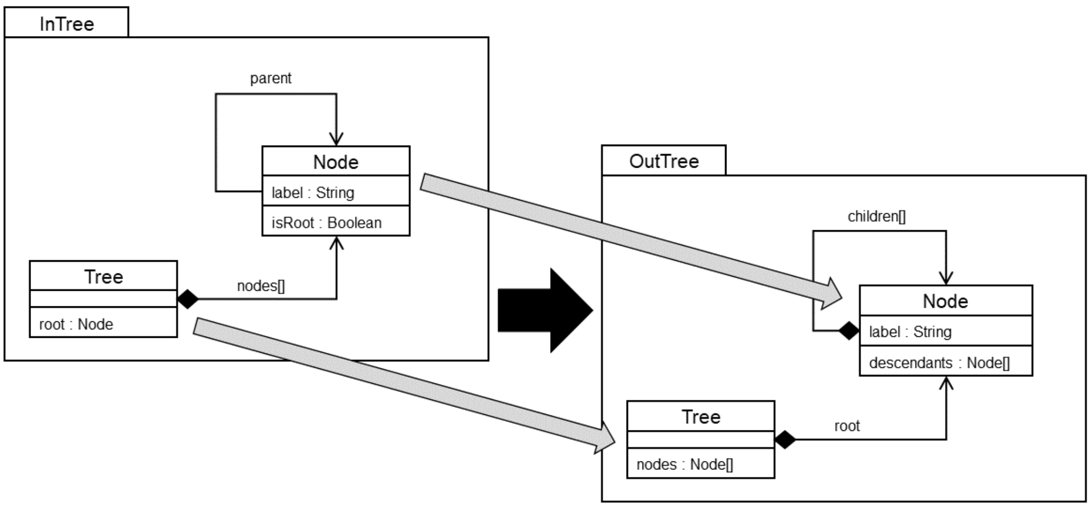

Figure 20: Visualising the InTree to OutTree transformation

Figure 20 illustrates a model transformation called *InTreeToOutTree* in the *ReMoDeL* visual notation (taken from [1]). This transformation converts an instance of the source *InTree* metamodel into an instance of the target *OutTree* metamodel. The same metamodel is described using the *ReMoDeL* textual notation in figure 21, in which the transformation group *Trees* is declared.

```
transform InTreeToOutTree : Trees {

   metamodel source : InTree
   metamodel target : OutTree

   mapping inTreeToOutTree (inTree : InTree_Tree) : OutTree_Tree {
      create OutTree_Tree(
         root := inNodeToOutNode(inTree.root, inTree))
   }

   mapping inNodeToOutNode (inNode : InTree_Node,
                            inTree : InTree_Tree) : OutTree_Node {
      create OutTree_Node(label := inNode.label,
         children := inTree.nodes.select(child | child.parent = inNode)
            .collect(node | inNodeToOutNode(node, inTree)))
   }
}
```

Figure 21: The textual InTree to OutTree transformation

The transformation group *Trees* will be used to construct a Java package having the name: *rule.trees*. By default, all transformations belong to a Java package whose prefix starts with *rule*, and whose suffix is the name of the transformation group converted to lowercase.

The transformation name *InTreeToOutTree* will be used as the name of the generated Java class. This class will belong to the Java package *rule.trees*. Other transformations belonging to the same group, such as *InTreeToGraph* (taken from [1]) will be added to this package.

5.2 Translation of a Model Transformation

Figure 22 shows an outline Java translation (eliding the method bodies) of the transformation in figure 21, which generates the Java class *rule.trees.InTreeToOutTree*. The approach is similar to the translation of a *ReMoDeL* concept, creating package, import and inheritance declarations. This is deliberate, following a standardised approach to cross-compilation. Aspects of the Java translation are highlighted below:

- *owning package* – the package owning the Java class is named *rule.trees*, after the *Trees* transformation group declaration in figure 21.
- *class name* – the name of the Java class corresponds exactly to the name of the transformation: *InTreeToOutTree* in figure 21.
- *inheritance* – the generated class *InTreeToOutTree* extends the library class *TopRule*. This is the common ancestor of all generated transformation classes, which supplies a shared context in which to cache rule results. *TopRule* extends *Top*.
- *main method* – a public static *main()* method is generated, which allows the generated class to be executed on command-line arguments, representing the path name(s) to the source model(s) to be transformed.
- *apply method* – a public *apply* method is generated, which accepts a source *Model<S>* as input, and returns a target *Model<T>* as output, where *S* and *T* are the types of the root elements, respectively of the source and target models.
- *rule methods* – every mapping rule is translated into a public method from *S* to *T*, where *S* and *T* are respectively the types of a source and target element.
- *constructor* – a default constructor is provided, named after the transformation.

- *import statements* – Java import statements ensure that the generated class has access to *Model, TopRule* and Java IO classes *File* and *IOException*.

```
package rule.trees;

import remodel.util.*;
import remodel.meta.Model;
import java.io.File;
import java.io.IOException;

public class InTreeToOutTree extends TopRule {
   public static void main(String[] args) throws IOException {
      ...
   }
   public Model<meta.outtree.Tree> apply(Model<meta.intree.Tree> tree) {
      ...
   }
   public meta.outtree.Tree inTreeToOutTree(meta.intree.Tree inTree) {
      ...
   }
   public meta.outtree.Node inNodeToOutNode(meta.intree.Node inNode,
         meta.intree.Tree inTree) {
      ...
   }
   public InTreeToOutTree() {
   }
}
```

Figure 22: Outline of generated rule.trees.InTreeToOutTree class.

The default constructor has an empty body. Details of the elided bodies of the other methods are given in the following sections.

5.3 Generated main() Method

Figure 23 shows the structure of the generated *main()* method. This checks the number of command line arguments (we expect only one), creates a source *Model*, reads this from a file created from the pathname *args[0]*, creates an instance of the transformation class *InTreeToOutTree*, invokes its *apply()* method on the source *Model* to yield a target *Model* and finally writes this to an output file sharing the prefix directory of the input file.

```
public static void main(String[] args) throws IOException {
   if (args.length < 1)
      throw new IOException("Missing args[] path to input model.");
   Model<meta.intree.Tree> source0 = new Model<>("inTree", "InTree");
   source0.read(new File(args[0]));
   System.out.println("Successfully read: " + args[0]);
   InTreeToOutTree transform = new InTreeToOutTree();
   Model<meta.outtree.Tree> target = transform.apply(source0);
   target.write(new File(toOutPath(args[0])));
   System.out.println("Successfully wrote: " + toOutPath(args[0]));
}
```

Figure 23: Generated main() method for rule.trees.InTreeToOutTree

The generated *main()* method allows the transformation class to be directly executed on command-line argument(s). Here, only one is expected (because we have one source metamodel). The generated *main()* method is really provided as a convenience, for the sake of testing the generated transformation class. See section 5.4 for how to perform the transformation programmatically within Java.

The pathname *args[0]* must include a directory prefix, which is expected. Typically, this is some subdirectory of the top-level Eclipse project directory. In this case, a suitable value for *args[0]* might be: "*meta/intree1.mod*". The reason for this is that the output file will be created in the related standard location: "*meta/out.mod*", where *out.mod* is the standard name for output model files. This file may be renamed, if it desired to keep it. The *main()* method also outputs its progress to standard output.

5.4 Generated apply() Method

The normal way to use a transformation is to create an instance of the transformation class, here *rule.trees.InTreeToOutTree*, and invoke its *apply()* method on a source *Model*, to yield a target *Model*. This is usually coded directly in hand-written Java code. Figure 24 shows the generated *apply()* method for *InTreeToOutTree*; and also the style in which it would be used in a bespoke Java program:

```
public Model<meta.outtree.Tree>
            apply(Model<meta.intree.Tree> inTree) {
   return new Model<>("outTree", "OutTree",
      inTreeToOutTree(inTree.getRoot()));
}

// Program fragment using the apply() method

Model<meta.intree.Tree> source = ... // from somewhere
Model<meta.outtree.Tree> target = null;
InTreeToOutTree transform = new InTreeToOutTree();
target = transform.apply(source);
```

Figure 24: Generated apply() method and its usage

The *apply()* method always accepts a source *Model* and returns a target *Model*. These are concrete instantiations of the generic class *Model<T>*, which is part of the standard package *remodel.meta* containing *ReMoDeL's* own meta-metamodel types. *Model<T>* encapsulates a model, including the model's *name*, its corresponding metamodel *type*, and the *root* element of the model. The root element could be of any type, generated from some *ReMoDeL* concept definition. The principal methods of *Model<T>* are:

- *String getName()* – returns the name of the model.
- *String getType()* – returns the metamodel type name.
- *T getRoot()* – returns the root element of the model.

Within any given model, one element is the root of this model. That is, every other element in the model is a component (or transitively is a component) of the root element. The type of this root element is used to instantiate *Model<T>*. Therefore:

- The source model is of the type: *Model<meta.intree.Tree>*
- The target model is of the type: *Model<meta.outtree.Tree>*

The type names are always expressed using their Java package-qualified forms, to distinguish the different source and target types, which otherwise have the same names.

The method *getRoot()* is used in the generated *apply()* method in figure 24, to access the *root* of the source metamodel. This *root* is passed as an argument to the *top rule*, the first rule in the transformation, which by convention must always act on the *root* element.

The class *Model<T>* also provides methods to read and write models in the *ReMoDeL* serial text format for models (see [1]). These methods use *ModelReader* and *ModelWriter* from the standard package *remodel.io*, which contains all readers, writers and compilers for the *ReMoDeL* toolset.

5.5 Translation of Mapping Rules

The mapping rules from the transformation in figure 21 are translated into the *rule methods* of the class *rule.trees.InTreeToOutTree*, as shown in figure 25. The first rule in this listing is known as the *top rule*. The top rule is used to inform the compiler about the required types of the *apply()* method (see section 5.4), so by convention it must be listed first.

```
public meta.outtree.Tree inTreeToOutTree(meta.intree.Tree inTree) {
   final String ruleName = "inTreeToOutTree";
   Object ruleKey = getKey(inTree);
   if (hasEntry(ruleName, ruleKey))
      return getEntry(ruleName, ruleKey);
   meta.outtree.Tree result = new meta.outtree.Tree()
      .setRoot(inNodeToOutNode(inTree.root(), inTree));
   return putEntry(ruleName, ruleKey, result);
}

public meta.outtree.Node inNodeToOutNode(meta.intree.Node inNode,
      meta.intree.Tree inTree) {
   final String ruleName = "inNodeToOutNode";
   Object ruleKey = getKey(inNode, inTree);
   if (hasEntry(ruleName, ruleKey))
      return getEntry(ruleName, ruleKey);
   meta.outtree.Node result = new meta.outtree.Node()
      .setLabel(inNode.getLabel())
      .setChildren(inTree.getNodes().select(
         new Predicate<meta.intree.Node>() {
            public Boolean apply(meta.intree.Node child) {
               return child.getParent() == inNode;
            }
         }).collect(
         new Function<meta.intree.Node, meta.outtree.Node() {
            public meta.outtree.Node apply(meta.intree.Node node) {
               return inNodeToOutNode(node, inTree);
            }
         }));
   return putEntry(ruleName, ruleKey, result);
}
```

Figure 25: Generated rule methods for rule.trees.InTreeToOutTree

All rule methods are generated according to a template pattern, which ensures that the rule is *idempotent*, that is, if the method is invoked multiple times on the same source input(s), it always returns the same target output. The pattern is as follows:

- *ruleName* – is used as an indexing feature for the rule method.
- *ruleKey* – is a key computed from the source inputs to the rule method.

- *hasEntry(ruleName, ruleKey)* – returns true if a result value has previously been cached for this rule under the given key.
- *getEntry(ruleName, ruleKey)* – fetches a cached result value for the given rule and key, which must exist if *hasEntry()* returns true.
- *putEntry(ruleName, ruleKey, result)* – caches a result value for the given rule and key; and also returns the result.

The three methods *hasEntry(), getEntry()* and *putEntry()* are defined in the library class *TopRule*, from which every transformation class inherits. These access a shared context, used by all the rules within a given transformation class. The context indexes each rule *result* firstly by the *ruleName*, and secondly by the *ruleKey*.

One important aspect to emphasise is that all Java class types generated from the source and target metamodels must be referenced using their full Java package-qualified names. This is because it is expected that different metamodels will inevitably overlap in the names used for their concepts; and the translated Java classes will also have overlapping names.

Here, both metamodels happen to use the identical concept names: *Tree* and *Node*. These are translated into distinct classes in the source and target packages, having the same simple class names. So that these are not confused, they are always referenced by their distinct Java package-qualified names: *meta.intree.Tree* and *meta.outtree.Tree*.

5.6 Focus on the inTreeToOutTree() Method

Figure 26 focuses on the first of the two mapping rules that were shown in figure 25. The method has a standard signature. It is named after the mapping rule *inTreeToOutTree*. It accepts an instance of the source type *meta.intree.Tree*, and returns an instance of the target type *meta.outtree.Tree*.

```
public meta.outtree.Tree inTreeToOutTree(meta.intree.Tree inTree) {
    final String ruleName = "inTreeToOutTree";
    Object ruleKey = getKey(inTree);
    if (hasEntry(ruleName, ruleKey))
       return getEntry(ruleName, ruleKey);
    meta.outtree.Tree result = new meta.outtree.Tree()
       .setRoot(inNodeToOutNode(inTree.root(), inTree));
    return putEntry(ruleName, ruleKey, result);
}
```

Figure 26: Focus on the mapping rule inTreeToOutTree.

The structure of the method is as follows. The code that is shaded in grey is the template code, designed to index the result of the rule. If the *context* (defined in *TopRule*) contains an entry for the *ruleName* "*inTreeToOutTree*" and the *ruleKey inTree*, then the cached result is returned immediately.

Otherwise, the transformation is computed, and the result is cached in the *context*, before it is returned. The transformation creates an instance of *meta.outtree.Tree* and sets its *root* to the result of transforming the *inTree's root: meta.intree.Node*, into a *meta.outtree.Node*. This invokes the second mapping rule, *inNodeToOutNode()*, to translate the node. We highlight the following aspects of the Java translation:

- The names of the method arguments correspond to the names of the mapping rule arguments and the types correspond to package-qualified source types.
- the name of the method's return value is conventionally *result*, which is declared as a local variable in the body of the method, to support the caching behaviour. Its type is the package-qualified target type.
- The body of the method constructs an instance of the target type, using the *Builder Pattern* [3], in which the instance is first constructed using a default constructor, and those fields to be initialised are set using a sequenced nesting of setter-methods. This allows more, or fewer fields to be set.

5.7 Focus on the inNodeToOutNode() Method

Figure 27 focuses on the second of the two mapping rules that were shown in figure 25. The method has a standard signature and is named after the mapping rule *inNodeToOutNode*. It accepts two source arguments, *inNode: meta.intree.Node*, and *inTree: meta.intree.Tree*, and returns an instance of the target type *meta.outtree.Node*.

```
public meta.outtree.Node inNodeToOutNode(meta.intree.Node inNode,
      meta.intree.Tree inTree) {
   final String ruleName = "inNodeToOutNode";
   Object ruleKey = getKey(inNode, inTree);
   if (hasEntry(ruleName, ruleKey))
      return getEntry(ruleName, ruleKey);
   meta.outtree.Node result = new meta.outtree.Node()
      .setLabel(inNode.getLabel())
      .setChildren(inTree.getNodes().select(
         new Predicate<meta.intree.Node>() {
            public Boolean apply(meta.intree.Node child) {
               return child.getParent() == inNode;
            }
         }).collect(
         new Function<meta.intree.Node, meta.outtree.Node() {
            public meta.outtree.Node apply(meta.intree.Node node) {
               return inNodeToOutNode(node, inTree);
            }
         }));
   return putEntry(ruleName, ruleKey, result);
}
```

Figure 27: Focus on the mapping rule inNodeToOutNode.

The structure of the method is as follows. The code that is shaded in grey is the template code, designed to index the result of the rule. If the *context* (defined in *TopRule*) contains an entry for the *ruleName* "*inNodeToOutNode*" and the composite *ruleKey (inNode, inTree)*, then the cached result is returned immediately. Note that the key is based on both source arguments (although the second argument is always the same *inTree*).

Otherwise, the transformation is computed, and the result is cached in the context, before it is returned. The transformation creates an instance of *meta.outtree.Node* and sets its *label* to the same as the *inNode's label*, and sets its *children* to the result of searching all the *inTree's* nodes to find those *child* nodes that refer to *inNode* as a *parent*, and transforming each of these to a node of the type *meta.outtree.Node*.

The body of the method makes use of the Builder Pattern [3], in which the *result* is created using a default constructor, followed by the sequentially nested invocation of setter-methods *setLabel()* and *setChildren()*. These can only work as intended if they return *this*, the instance of *meta.outtree.Node* on which they were invoked (see section 4.2).

The way in which the *children* are calculated makes use of higher-order collection operations, namely *select()* and *collect()*. These are composed sequentially, with the list resulting from *select()* being the receiver for *collect()*.

- The *select()* method receives a *Predicate* argument to filter the source nodes, returning true if a node satisfies the predicate. This follows the pattern for implementing predicate arguments as anonymous subclasses of *Predicate*, whose method body performs a *boolean* test (see section 3.1)
- The *collect()* method receives a *Function* argument to transform each of the source nodes into a target node. This follows the pattern for implementing function arguments as anonymous subclasses of *Function,* whose method body transforms the argument, yielding the result (see section 3.1).

In fact, the body is a recursive call to the *inNodeToOutNode* method. The recursion halts, when the filtered list of source children is empty, so there is nothing to transform.

6. Further Aspects of Compilation

Here, we round up some remaining aspects of the *ReMoDeL* cross-compilation strategy that were not covered in earlier sections. These include:

- translation of identifiers
- translation of merging rules
- translation of function rules

6.1 Translation of Identifiers

The *ReMoDeL* expression language uses identifiers to name different kinds of property: attributes, references, components and operations. These occur in different contexts, such as when being declared, or when being accessed (read), or when being assigned (written), or when being invoked.

All *ReMoDeL* type identifiers are expected to follow "*CapitalCase*" conventions (initially capitalised, with internal capitalisation at word boundaries in multi-word identifiers). The following translation schemes are applied:

- Basic type identifiers occurring in non-parametric contexts are mapped to the equivalent Java basic type identifier (in lowercase), except for the *String* type, which is not a basic type in Java. The type *Decimal* is mapped to *double*.
- Basic type identifiers occurring in parametric contexts (as the element type of a *PureList<T>* or *PureSet<T>*) are mapped to the equivalent Java class-wrapper for the basic type (in "*CapitalCase*"). The type *Decimal* is mapped to *Double*.
- Other *ReMoDeL* concept type identifiers are mapped to a Java class having the same name (in "*CapitalCase*").
- In the translation of a metamodel, all *ReMoDeL* types that are not predefined (basic or collection types) come from the same metamodel and are declared to belong to the

corresponding Java package; and are mapped to Java class identifiers using the unqualified short name of the class.
- In the translation of a transformation, all *ReMoDeL* types that are not predefined (basic or collection types) are assumed to come from different metamodels, and are mapped to a Java class identifier using the full package-qualified name.

All *ReMoDeL* property and variable identifiers are expected to follow "*camelCase*" conventions (initially lowercase, with internal capitalisation at word boundaries in multi-word identifiers). The following translation schemes are applied:

- Variable and property identifiers occurring in a declaration context are mapped to a Java identifier having the same "*camelCase*" convention.
- Property identifiers that denote operations or rules are also checked to ensure that explicit parentheses are added (for arguments) even when empty.
- Variable identifiers occurring in a variable-access context are mapped to a Java identifier having the same name (and there is no variable-update context).
- Property identifiers (excluding executable properties) occurring in a property-access context are mapped to the access method *getX()* for a property named *X*.
- Property identifiers (excluding executable properties) occurring in a property-update context are mapped to the update method *setX()* for a property named *X*.
- Operation identifiers used in a property-access context are mapped to an invocation of the same-named method against the receiver (the property owner).
- Rule identifiers used in a function-call context are mapped to a call of the same-named rule method, with an implicit receiver (the transformation object).
- Special operation identifiers *{asSet, asList, asName, asType}* used in an invocation context are mapped to static method calls accepting the receiver as an argument.

The special identifier *self* is also used to refer to the current instance (the implicit receiver) when it occurs in the body of an operation. This has two translations:

- when *self* occurs outside the scope of a lambda-expression, it is mapped to the corresponding Java special identifier *this*.
- when *self* occurs inside the scope of a lambda-expression, it is mapped to *T.this*, where *T* is the name of the Java type enclosing the lambda-expression.

The special identifier *super* is translated unchanged, and is used in the same way as in Java for method-combination, invoking an inherited operation within its redefined version.

The special identifier *owner* may optionally be declared in a *component* property, having the type of its owning container. This is translated into a *transient* back-reference, which is not serialised in models (it is set automatically when adding a component to a container). The *owner* back-reference may be used in operations or transformation rules.

6.2 Translation of Merging Rules

Merging rules (distinguished by the keyword *merging*, rather than *mapping*) are expected to have two arguments taken from two different metamodels. The idea is that a merging rule constructs an instance of the target type by merging or folding together the information from two different source types. We may draw an analogy with aspect-weaving in Aspect-Oriented Programming (AOP) [4]. This kind of transformation may be useful to combine different views of a software system.

A partial sketch for a merging transformation is given in figure 28. This assumes two source metamodels. The first is a JSP (Jackson Structured Programming [5]) model, a tree of *Block* nodes capturing the sequence, selection and iteration structure. The second is a DFD (Dataflow Diagram [6]) model, a general graph consisting of *Process* nodes and *Dataflow* edges. The transformation seeks to construct a call-graph of *Procedure* nodes, following the tree-structure of the JSP diagram, in which *Procedure* nodes are also annotated with their arguments and result, represented by *Variable* nodes, which are translated from the *Dataflow* edges in the DFD diagram.

```
transform JspDfdToProc : Procs {

   metamodel source0 : JSP
   metamodel source1 : DFD
   metamodel target : Proc

   merging jspDfdToProcDiag (jsp : JSP_Diagram,
                     dfd : DFD_Diagram) : Proc_Diagram {
      create Proc_Diagram(
         root := blockProcToProc(jsp.root,
            dfd.processes.detect(proc : DFD_Process |
               proc.name = jsp.root.name), dfd)
      )
   }

   merging blockProcToProc(block : JSP_Block, proc : DFD_Process
                     dfd : DFD_Diagram) : Proc_Procedure {
      create Proc_Procedure(
         name := block.name,
         kind := block.kind,
         arguments := dfd.dataflows.select(flow : DFD_Dataflow |
               flow.target = proc)
            .collect(flow | flowToVariable(flow)),
         result := flowToVariable(
            dfd.dataflows.detect(flow : DFD_Dataflow |
               flow.source = proc)),
         children := block.children.collect(child : JSP_Block |
            blockProcToProc(child,
               dfd.processes.detect(proc2 : DFD_Process |
                  proc2.name = child.name),
               dfd)
         )
      )
   }

   mapping flowToVariable(flow : DFD_Dataflow) : Proc_Variable
     ...
}
```

Figure 28: Merging JSP and DFD views into a procedural model

The top rule takes a JSP *Diagram* and a DFD *Diagram* and produces a Proc *Diagram* (where *Proc* is the name of the procedural metamodel). To do this, it starts with the root *Block* of the JSP *Diagram* and seeks out the DFD *Process* having the same name as this *Block*, and then invokes the second rule with these arguments. The whole DFD *Diagram* is also supplied as an ancillary argument, to allow searching for further *Process* nodes.

The merging transformation in figure 28 is only trivially different from the mapping transformation in figure 21, in the following ways:

- it depends on two source metamodels, rather than just the one
- the top rule accepts an argument from each source metamodel
- otherwise, some rules are merging rules, others are mapping rules

Because of this, the translation bears a similarity with that of a mapping transformation; and is sketched in overview in figure 29.

```
package rule.procs;

import remodel.util.*;
import remodel.meta.Model;
import java.io.File;
import java.io.IOException;

public class JspDfdToProc extends TopRule {

   public static void main(String[] args) throws IOException {
      if (args.length < 2)
         throw new IOException("Missing args[] paths to input models.");
      Model<meta.jsp.Diagram> source0 = new Model<>("jsp", "JSP");
      source0.read(new File(args[0]));
      System.out.println("Successfully read: " + args[0]);
      Model<meta.dfd.Diagram> source1 = new Model<>("dfd", "DFD");
      source1.read(new File(args[1]));
      System.out.println("Successfully read: " + args[1]);
      JspDfdToProc transform = new JspDfdToProc();
      Model<meta.proc.Diagram> target = transform.apply(source0, source1);
      target.write(new File(toOutPath(args[0])));
      System.out.println("Successfully wrote: " + toOutPath(args[0]));
   }

   public Model<meta.proc.Diagram> apply(Model<meta.jsp.Diagram> jsp,
         Model<meta.dfd.Diagram> dfd) {
      return new Model<>("proc", "Proc",
         jspDfdToProcDiag(jsp.getRoot(), dfd.getRoot()));
   }

   public meta.proc.Diagram jspDfdToProcDiag(meta.jsp.Diagram jsp,
            meta.dfd.Diagram dfd) {
      final String ruleName = "jspDfdToProcDiag";
      Object ruleKey = getKey(jsp, dfd);
      if (hasEntry(ruleName, ruleKey))
         return getEntry(ruleName, ruleKey);
      meta.proc.Diagram result = new meta.proc.Diagram()
         .setRoot(blockProcToProc(jsp.getRoot(),
            dfd.getProcesses().detect(new Predicate<meta.dfd.Process>() {
               public boolean apply(meta.dfd.Process proc) {
                  return proc.getName().equals(jsp.getRoot().getName());
               }
            }),
            dfd));
      return putEntry(ruleName, ruleKey, result);
   }

   ...
}
```

Figure 29: Partial translation of the transformation in fig. 28

The main changes compared to the translation of a mapping rule transformation (see figures 22-24) are:

- The generated *main()* method seeks to find two input files on the *args[]* command line, one for each of the expected source models.
- The generated *apply()* method expects two source *Models*, one typed with the root JSP *Diagram* and the other typed with the root DFD *Diagram.*
- The top rule expects two source elements, a JSP *Diagram jsp* and a DFD *Diagram dfd*, and caches the result under a composite key computed using *getKey(jsp, dfd).*

Otherwise, the Java generation strategy is like what was described previously.

6.3 Translation of Function Rules

Function rules are ancillary functions, used as helpers in the transformation process. They are simple functions, possibly recursive, and they do not cache their results. The translation of such rules is therefore simpler than for other kinds of rule, which cache their results.

In the previous technical report *ReMoDeL Explained: an Introduction to ReMoDeL by Example* [1], we introduced an auxiliary function to test whether a general directed acyclic *Graph* was in fact a *tree*. That is, the role of the function was to act as a precondition, which could be invoked as part of a transformation from a *Graph* to a *Tree*.

```
# ReMoDeL coding of a helper function

function isTree(graph: Graph_Graph) : Boolean {
   graph.vertices.select(vertex: Graph_Vertex |
      not graph.edges.exists(edge: Graph_Edge |
         edge.source = vertex)).size = 1
}

// Java translation of a helper function

public boolean isTree(meta.graph.Graph graph) {
   return graph.getVertices().select(
      new Predicate<meta.graph.Vertex>() {
         public boolean apply(meta.graph.Vertex vertex) {
            return ! graph.getEdges().exists(
               new Predicate<meta.graph.Edge>() {
                  public boolean apply(meta.graph.Edge edge) {
                     return edge.getSource() == vertex;
                  }
               });
         }
      }).size() == 1;
}
```

Figure 30: Function rule and its simple translation to Java

Figure 30 shows both this function in the *ReMoDeL* expression language (upper) and its translation into Java (lower). Whereas the treatment of types, names and general program code is the same as presented earlier, the main difference is that no template code is generated for caching the result.

The result of the generated method may be returned directly. The Java code otherwise follows the functional style of the original *ReMoDeL* expression language. Like any other rule, this generated method is owned by a transformation class (the translation of the transformation which declared the function).

7. The ReMoDeL Distribution

The *ReMoDeL* toolset comes as a Java archive (jar-file). This is a compiled library, which you will import into your own *ReMoDeL* projects. Any given *ReMoDeL* project is created as a separate Java project. It includes the *ReMoDeL* toolset on its build-path as an external library. An individual *ReMoDeL* project may generate different Java packages (source and binary), depending on what metamodels and transformations you create in the *ReMoDeL* modelling languages.

7.1 The ReMoDeL 3.x Toolset

The *ReMoDeL* toolset comes as a binary Java archive file, containing the following packages:

- *remodel* – command-line tools for checking and compiling models
- *remodel.io* – reader, writer and compiler components
- *remodel.meta* – the model and metamodel components
- *remodel.expr* – the expression language components
- *remodel.util* – the standard runtime library

The command-line tools are intended to be used to check and compile your own *ReMoDeL* projects. These include two general-purpose main programs, which are sensitive to the filename extensions used to distinguish models, metamodels and transformations. Each program expects a single file pathname, relative to the IDE project directory in which they are executed.

- *Validate* – reads a model, metamodel or transformation file named on the input path: "*pathname/file.ext*"; and writes it out again as the file "*pathname/out.ext*" in the same directory, where *out* is a temporary output file for comparison with the input.
  - Metamodel (*.met*) and transformation (*.tra*) files are parsed as grammar trees; and, if syntactically correct, they are serialised back to text.
  - Model (*.mod*) files are parsed into instances of the compiled Java metamodel classes, and if syntactically correct, are serialised back to text.
- *Compile* - compiles a metamodel or transformation file named on the input path: "*pathname/file.ext*", creating a Java package consisting of one or more classes.
  - The metamodel file "*meta/Graph.met*" is compiled into the Java package "*meta.graph*" containing the Java classes for the metamodel concepts.
  - The transformation file "*rule/InTreeToGraph.tra*" is compiled into the Java package "*rule.trees*" containing the rule class *InTreeToGraph.java*.

If any syntax errors are found, these are reported as Java exceptions. A *SyntaxError* will report the expected syntax at a line number in the *ReMoDeL* source. A *SemanticError* will report other kinds of inconsistency, e.g. if a model references a non-existent Java class (because the metamodel was not compiled), or if a *ReMoDeL* property references a missing Java field (because a transformation rule is inconsistent with the metamodel).

Metamodel and transformation files may be validated at any time; model files may only be validated if the corresponding metamodel has previously been compiled. Metamodels may be compiled at any time; compiling a transformation will force prior re-compilation of any metamodel referenced by the transformation (for dependency's sake); and these are all assumed to be in the same directory.

7.2 An Example ReMoDeL Project

We assume the working examples of trees and graphs are to be provided as a separate *ReMoDeL* project. To create this, you first set up your Java project in your favourite Java IDE. Let us assume it is called *RM_Trees*. You then import the ReMoDeL 3.x toolset as an external library on your project's build path.

We expect the following directory structure under the *RM_Trees* project root directory:

- RM_Trees/src – the directory for Java source code, compiled by *ReMoDeL*
- RM_Trees/meta – the directory for *trees* metamodels (*.met* files)
- RM_Trees/rule – the directory for *trees* transformations (*.tra* files)
- RM_Trees/model – the directory for *trees* model instances (*.mod* files)
- RM_Trees/code – the directory for external code generation (not used here)

The names of the subdirectories: *meta, model, rule* are standard for all ReMoDeL projects. Under these directories will be placed the *ReMoDeL* language files relating to each group:

- metamodels – *InTree.met, OutTree.met, Graph.met*
- transformations – *InTreeToOutTree.tra, InTreeToGraph.tra, OutTreeToInTree.tra, ...*

Under the subdirectory called *models* will be placed the model instance files:

- models – *intree1.mod, outtree1.mod, graph1.mod, ...*

When compiling a metamodel file, or transformation file, this will generate new Java packages containing Java source classes under the *src* directory, for example:

- *meta.intree* – containing the Java classes compiled from *InTree.met*
- *meta.graph* – containing the Java classes compiled from *Graph.met*
- *rule.trees* – containing the Java classes compiled from transformations

The generated Java model transformations will have their own *main()* methods, so these may be executed directly. They expect on the command-line one (or more) paths to suitable model files that are the appropriate kind of input for the transformation. For example:

```
java InTreeToGraph model/intree1.mod
```

Mapping transformations will expect one input path, whereas merging transformations will expect two (or possibly more) input paths, according to the number of source metamodels required.

Whereas the above is styled as a command-line program executed in JDK, when running transformations in an IDE such as Eclipse, one would use the *Project* > *Run Configurations* option and set the command-line path argument under the *Arguments* tab.

8. References

[1] A J H Simons. ReMoDeL Explained (rev. 2.1): an introduction to ReMoDeL by example. Technical Report, 12 July, (University of Sheffield, 2022).

[2] Oracle. Trail: Java Beans™, The Java Tutorials (Oracle, 2020). *https://docs.oracle.com/javase/tutorial/javabeans/*

[3] J Bloch. *Effective Java, 2nd ed.* (Addison-Wesley, 2008).

[4] G Kiczales, J Lamping, A Mendhekar, C Maeda, C Lopes, J M Loingtier, J Irwin. "Aspect-oriented programming", Proc. 11th European Conference on Object-Oriented Programming (ECOOP '97), *Lecture Notes in Computer Science, 1241* (Springer, 1997), 220–242.

[5] M A Jackson. *Principles of Program Design*, (Academic Press, 1975).

[6] E Downs, P Claire and I Coe. *Structured Systems Analysis and Design Method: Application and Context, 2nd Ed.*, (Prentice Hall, 1991).