The University Of Sheffield.

ReMoDeL Explained

An Introduction to ReMoDeL 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 describes **ReMoDeL v3**, a high-level syntax for defining models and model transformations. It assumes no prior knowledge of Model-Driven Engineering and aims to introduce related concepts such as *models, metamodels* and *model transformations* from first principles, using simple examples. To give some initial context, the field of Model-Driven Engineering is introduced briefly below.

1.1 Model-Driven Engineering

Model-Driven Engineering (MDE) is a general strategy in software engineering that seeks to raise the level of abstraction at which we create and manage software systems. In the future, instead of focusing on detailed implementations in specific programming languages, we will create high-level designs using conceptual models that are much closer to the stakeholders' view of their business processes. These models will be checked, maintained and revised by automated tools, which will also generate the eventual software system.

MDE is concerned with any aspect of the design and implementation of software systems that can be represented and managed using models. Reverse-engineering a design from software is one activity. Checking a design for faults and repairing it is another activity. Improving the quality of a design by refactoring is another activity. Translating a design from one representation into another is a further activity. Generating an executable software system from a design is another activity. Each of these activities involves a process known as *Model Transformation* (MT), the conversion of models from one kind to another.

1.2 Model-Driven Development

Model-Driven Development (MDD) is the subfield of MDE which focuses specifically on generating executable software systems from high-level abstract designs. There are several perceived benefits. Firstly, the designs are independent of any particular programming language or execution platform, so if the latter change, the investment in the designs is not wasted or lost. Secondly, when the business needs change, it is far easier to update the design models than it is to intervene at the code level. Code is regenerated automatically from the managed designs. It is expected that this will allow Software Engineers to be more productive and more responsive to change requests.

MDD encompasses a range of approaches. Microsoft's *Software Factories* [1] supply a number of code templates and automatic code generators to build product-lines for similar kinds of software product. The Object Management Group (OMG), a consortium of technology providers and consumers, proposes a specific approach called *Model-Driven Architecture* (MDA) [2], based on a set of curated standards, which tool providers may choose to follow. MDA prescribes three levels of abstraction at which models of systems should be developed:

- Computation-Independent Model (CIM) – the business requirements model
- Platform-Independent Model (PIM) – the logical system design model
- Platform-Specific Model (PSM) – the template for implementation

The Eclipse Foundation provides a partial implementation of some of the OMG's standards in the *Eclipse Modelling Framework* (EMF) [3], which supports the creation of visual and textual editors for models. A number of earlier projects in MDD, such as ATL, Epsilon and

Agile UML are now embedded within EMF. Other independent approaches, such as ReMoDeL [4], are not so strongly tied to MDA.

1.3 ReMoDeL

ReMoDeL is an acronym for *Reusable Model Design Languages*, reflecting the original goal to specify an overlapping set of languages to model different aspects of software systems [4]. Some were closer to implementation (imperative, functional and object-oriented models) and others closer to design (entity-relationship, business process models).

- *ReMoDeL v1*: specified a set of metamodels, initially focusing on core programming and database concepts. These were published as XML specifications. Model transformations were imperative Java programs that manipulated the XML DOM-tree structures directly, which represented instances of these metamodels.
- *ReMoDeL v2:* specified a set of metamodels as Java packages, whose classes modelled the domain concepts directly, serialised as XML. Model transformations were hybrid declarative/imperative rules, using distinct Java patterns for mapping, merging, updating or splitting kinds of transformation.
- *ReMoDeL v3:* specifies metamodels and model transformations in a bespoke compact syntax, with compilers that convert these into executable Java libraries and programs. Concepts and transformations are expressed declaratively in a pure functional style. Mapping, merging and updating transformations are supported.

ReMoDeL v3 has its own compact syntax for expressing models, metamodels and model transformations. *ReMoDeL v3* syntax is about one third the size of the generated Java code, making it much easier to develop metamodels than in previous versions. This encourages rapid experimentation with different kinds of domain-specific language, since the designer is not burdened with code maintenance.

Designers are free to focus on the constraints captured in particular views of a software system. There may be many such views, expressed in different metamodels. Model-transformation then becomes a layered process of gradual refinement, which exploits the constraints found at different levels of abstraction. A software system should be the result of folding together models that capture the data, time and process views of the system.

2. Models and Metamodels

The notions of *models* and *metamodels* may be unfamiliar. A model is any constructed representation of something of interest. A model may capture certain details and ignore others. Informally, a model is a simplified version of the thing being modelled.

In Software Engineering, we are concerned almost exclusively with models of information. This can include models of data, models of processes and models of time. Models must be comprehensible to human Software Engineers, who are involved in the design process. In Model-Driven Engineering, models must also be machine-readable, since programs will transform models of one kind into models of another kind.

2.1 An InTree Model

The easiest way to explain these concepts is through examples. We will use the familiar idea of a tree, which is a well-known data structure in computer science. Figure 1 depicts a model of a tree, as a diagram expressing the structure of the tree model.

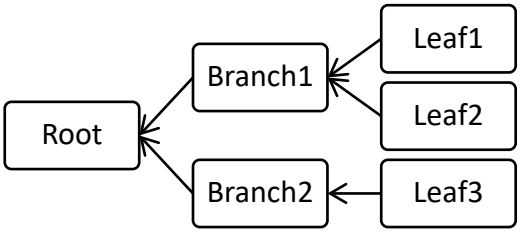

Figure 1: A visual model of an in-tree

The model consists of six labelled nodes, which are connected in a certain pattern. One node is labelled the *Root*, and five other dependent nodes are connected by an arrow to their parent node. Some nodes, labelled *Branch1, Branch2,* have further dependents. Other nodes, labelled *Leaf1, Leaf2, Leaf3,* have no dependents. This model is of a particular kind of tree, called an *in-tree*, in which the root node is reachable from every other node.

```
model intree1 : InTree {
   t1 : Tree(nodes = Node[
      n1 : Node(label = "Root"),
      n2 : Node(label = "Branch1", parent = n1),
      n3 : Node(label = "Branch2", parent = n1),
      n4 : Node(label = "Leaf1", parent = n2),
      n5 : Node(label = "Leaf2", parent = n2),
      n6 : Node(label = "Leaf3", parent = n3)
   ])
}
```

Figure 2: A textual model of an in-tree

The same logical model may be expressed either as a diagram, or as text. Figure 2 depicts the same tree as text, using the *ReMoDeL* model syntax. The keyword *model* introduces the model, which has the name *intree1* and the metamodel-type *InTree*. The model consists of seven typed nodes:

- *t1: Tree* – is a node representing the whole tree, consisting of six nodes
- *n1..n6: Node* – are the other nodes representing the parts of the tree

Each of the nodes has a unique identifier (*t1, n1..n6*). This allows nodes to refer to each other within the model. For example, the root node has the identifier *n1*, and the branch nodes *n2, n3* refer to this node *n1* as their parent. Similarly, the leaf nodes *n4, n5* refer to the branch node *n2* as their parent. The only node with no parent is the node *n1*. In this way, the textual notation in figure 2 captures exactly the structure of the tree in figure 1.

Each node has a number of internal properties. Properties can be simple attributes, or references to other nodes. For example, the node *n2* has a property called *label,* an attribute which is bound to the simple string value "Branch1". The same node *n2* has a property called *parent*, a reference which is bound to the node *n1*. The bound values of properties are indicated using the simple equals operator.

Some nodes are containers of other nodes. For example, the tree node *t1* has a property called *nodes*, which refers to a list of other nodes: *Node[n1 ... n6].* This makes *t1* the container of all the other nodes in the model, and we say that *n1..n6* are its components. Containers can be typed either as lists [...], or as sets {...}.

A model is written out as a collection of nodes, having a number of properties. The first time any given node is encountered, it is written out in full, with all of its properties. If the same node is encountered again, only its identifier is written.

2.2 The InTree Metamodel

The type of a model is a *metamodel* (a model of a model), which specifies the model's structure and behaviour. The tree of figure 1 is an *instance* of the *InTree* metamodel, depicted in figure 3 using the *ReMoDeL* graphical notation. This specifies what kinds of node are allowed in an *InTree*, and what properties they are expected to have.

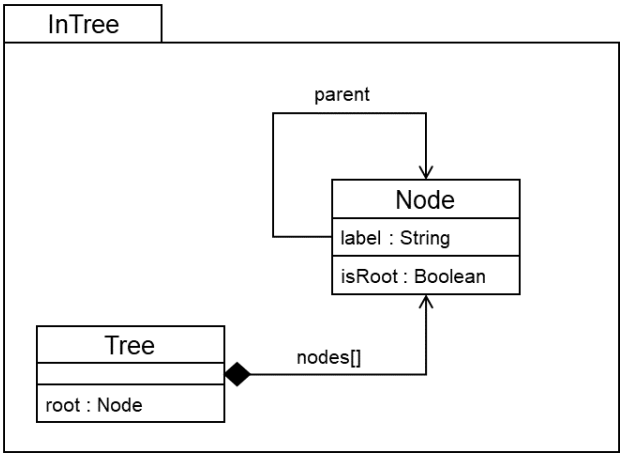

Figure 3: A visual metamodel for InTree

The metamodel is named *InTree*. The metamodel boundary encloses a couple of interlinked concepts, which specify the allowed types of node. These are drawn as rectangles, named *Tree* and *Node*, with two partitions listing their attributes and operations (see below). The references are indicated using labelled arrows drawn from the source to the target. A black diamond at the source-end indicates a stronger component reference.

Readers familiar with the UML notations [5] will notice that the visual syntax is similar to the UML class diagram. However, the ReMoDeL visual notation is strictly simpler, in that all relationships are directed and are labelled in one direction.

```
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 4: The textual metamodel for InTree

The same metamodel may also be expressed as text. Figure 4 depicts the same metamodel, using the *ReMoDeL* metamodel syntax. This captures all of the information in figure 3, and also specifies the operations. Below, we explain the meanings of certain keywords in the *ReMoDeL* metamodel syntax, drawing analogies with UML and Java:

- **metamodel** – defines a given metamodel (c.f. a UML or Java package). All concepts defined within the scope of the named metamodel belong to this namespace.
- **concept** – defines a concept within a metamodel (c.f. a UML or Java class). A concept is an entity that may have attributes, references, components and operations.
- **attribute** – defines a named property of a concept that stores a simply-typed value. Attributes store *String, Integer, Decimal, Boolean* and other simply-typed values.
- **reference** – defines a named property of a concept that refers to another concept elsewhere in the model (c.f. a weak reference, or a directed association in UML).
- **component** – defines a named property of a concept that is contained by, and managed locally by, the concept (c.f. an owned reference, or a composition in UML).
- **operation** – defines a named operation of a concept (c.f. a method in UML or Java). An operation always returns a result, and may optionally have parameters.

Every concept is named; and every property also declares a type after its name. Types are single, such as *Node*, or multiple, such as *Node[]*, the type of a *Node*-list. A set-type would be signalled as *Node{}*. Operations only require argument parentheses if they accept arguments. The syntax for the body of operations follows a mix of object-oriented and pure functional programming styles, using mapping and filtering:

- *Node.isRoot : Boolean* – returns true if the parent of the node is *null.*
- *Tree.root : Node* – filters the list of nodes for the only node that is a root.

The textual syntax is intended to be compiled to generate an executable metamodel, for example, by converting the metamodel into a Java package, and converting each concept into a corresponding Java class definition. The strong versus weak reference model also supports translation into C++, where memory management is explicit (a class must manage its own components, deleting these when it is deleted).

2.3 An OutTree Model

The *in-tree* is only one way in which we could choose to model a tree, in which every node refers to its parent. For example, we might prefer to model the tree as an *out-tree*, in which every node refers to its children. Figure 5 depicts an example of this kind of tree:

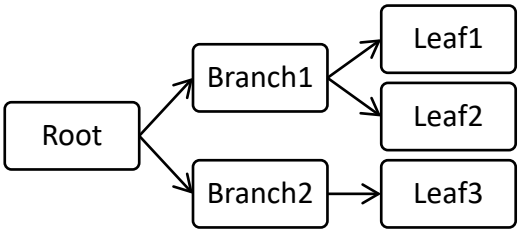

Figure 5: A visual model of an out-tree

The model consists of six labelled tree-nodes, which are connected in precisely the opposite direction to the tree in figure 1. One node labelled *Root* refers to nodes labelled *Branch1, Branch2*, where *Branch1* refers to *Leaf1, Leaf2* and *Branch2* refers to *Leaf3*. This tree is an *out-tree*, in which every node is reachable from the root.

```
model outtree1 : OutTree {
   t1 : Tree(root =
      n1 : Node(label = "Root", children = Node[
         n2 : Node(label = "Branch1", children = Node[
            n4 : Node(label = "Leaf1"),
            n5 : Node(label = "Leaf2")
         ]),
         n3 : Node(label = "Branch2", children = Node[
            n6 : Node(label = "Leaf3")
         ])
      ])
   )
}
```

Figure 6: A textual model for an out-tree

The same model may be represented either visually, or textually. Figure 6 depicts the same tree from figure 5 as text, using the *ReMoDeL* model syntax. The keyword *model* introduces the model, which has the name *outtree1* and the metamodel-type *OutTree*. The model consists of seven typed nodes:

- *t1: Tree* – encapsulates the whole tree, whose *root* is bound to the root *Node*.
- *n1..n6: Node* – represent other labelled nodes, which refer to their own children.

Contrasting figure 6 with figure 2, there is no single container for all the nodes; instead, *Tree* just refers to the root *Node n1*. Thereafter, each *Node* contains its own *children*. For example, the *Node n2* contains, as its *children*, the list *Node[n4, n5]*. This gives rise to the levels of indentation in figure 6, where nested nodes are introduced further to the right.

2.4 The OutTree Metamodel

The different structure of this model is defined by a different metamodel, as you would expect. Figure 7 depicts the *OutTree* metamodel in the ReMoDeL visual notation. This specifies the different allowed linkage between nodes. Overall, the declared *properties* of a

Tree in this *OutTree* metamodel are similar to the declared *properties* of a *Tree* in the *InTree* metamodel of figure 3: the difference is in which properties are provided by *components*, and which are provided by *operations*.

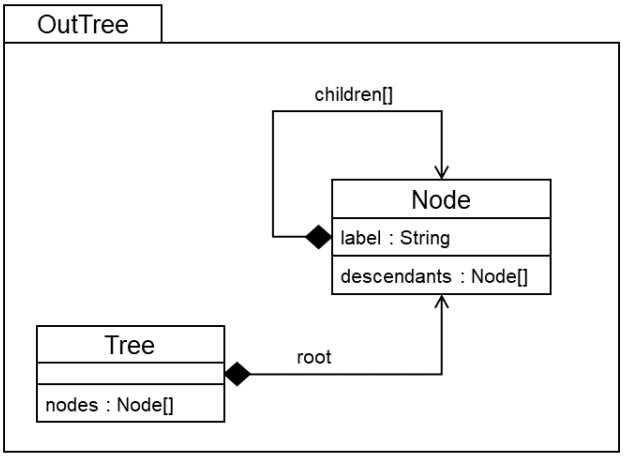

Figure 7: A visual metamodel for OutTree

The metamodel is named *OutTree*, and the boundary encloses two concepts named *Tree* and *Node*. A *Tree* has one *root* component referring to a *Node*. A *Node* has a *children* component referring to a list of *Nodes*. (In the visual notation, the list type indicator *[]* is added to the label on the arrow, when this refers to a list of the target node. Elsewhere, a list type is indicated by adding this indicator after the type-name: *Node[]*).

The same metamodel is depicted as text, using the ReMoDeL metamodel syntax, in figure 8. The *component* keyword indicates that the *Tree* concept is now a container just for the *root Node*; and that each *Node* is the container for its own *children Nodes*.

```
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 8: A textual metamodel for OutTree

In both metamodels (figures 4, 8), the *Tree* concept defines properties called *root* and *nodes*. Whereas in figure 4 the *nodes* are stored directly, in figure 8 the nodes are computed by an operation which walks through the tree. Likewise, whereas in figure 8 the root is stored directly, in figure 4 the root is computed by an operation which searches the nodes. This illustrates how different design decisions can be made in a metamodel. The featured operations in figure 8 work in the following way:

- *Tree.nodes : Node[]* – finds all nodes in the tree in top-down order, by treating the root as a list, and then appending all of the root's descendants to this list.
- *Node.descendants : Node[]* – finds the descendants of a node by collating the descendants of each child, found recursively, as a single list.

2.5 A Graph Model

Tree- and graph-like structures can either be modelled using nodes that refer directly to each other, or they can be modelled in a more explicit way, using distinct elements to represent the vertices (nodes) and edges (arcs) in the graph. An example of this is shown in figure 9, in which nodes *v1..v6* are *vertices*, and nodes *e1..e5* are *edges*.

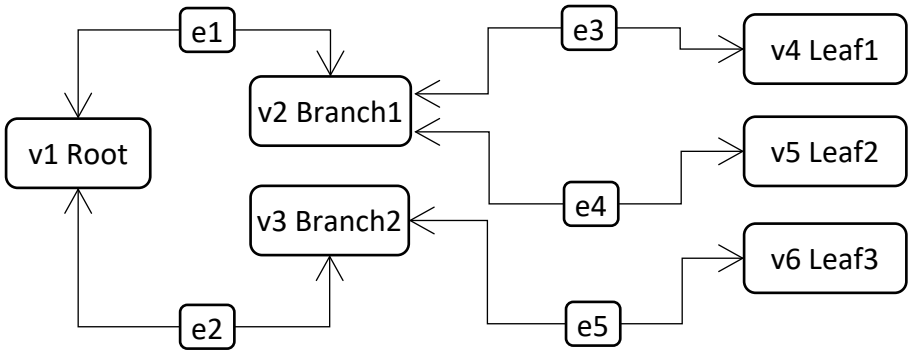

Figure 9: A visual model of a graph

The vertices do not refer directly to each other in this model. Instead, every edge connects a pair of vertices. So, the link from the vertex *v1* (the *Root*) to the vertex *v2* (*Branch1*) is represented by the edge *e1*, which refers to the pair of vertices. It is sometimes convenient to build models in this way, because the edges can be added after the vertices, independently.

```
model graph1 : Graph {
   g1 : Graph(vertices = Vertex[
      v1 : Vertex(label = "Root"),
      v2 : Vertex(label = "Branch1"),
      v3 : Vertex(label = "Branch2"),
      v4 : Vertex(label = "Leaf1"),
      v5 : Vertex(label = "Leaf2"),
      v6 : Vertex(label = "Leaf3")
   ], edges = Edge[
      e1 : Edge(source = v2, target = v1),
      e2 : Edge(source = v3, target = v1),
      e3 : Edge(source = v4, target = v2),
      e4 : Edge(source = v5, target = v2),
      e5 : Edge(source = v6, target = v3),
   ])
}
```

Figure 10: A textual model of a graph

The same model is depicted as text using the *ReMoDeL* model syntax in figure 10. From this it is clear that a *Graph* consists of a list of *Vertex[v1..v6]* and a list of *Edge[e1..e5]*.

- *g1: Graph* – is a container of lists of *vertices* and *edges*.
- *v1..v6: Vertex* – have a *label* attribute bound to a specified *String*.
- *e1..e5: Edge* – have a *source* and *target* reference bound to *Vertex* objects.

2.6 The Graph Metamodel

The more elaborate structure of a graph model is defined by the *Graph* metamodel, which specifies the relationships between the three types of concept: *Graph, Vertex* and *Edge*. This is depicted in the *ReMoDeL* visual notation in figure 11:

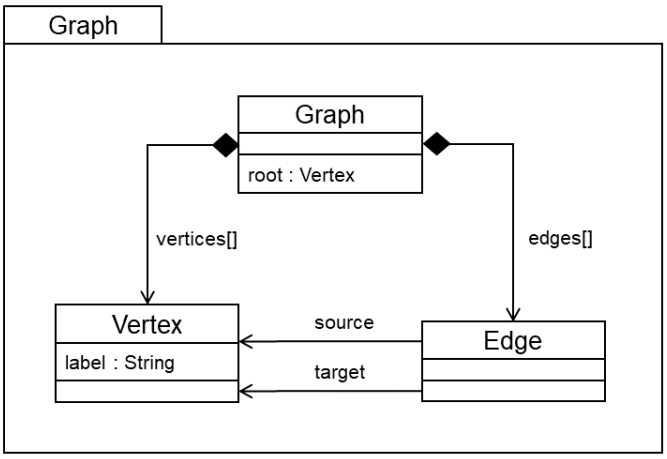

Figure 11: A visual metamodel for Graph

The figure names the metamodel *Graph*, and within the metamodel boundary shows the three concepts. *Graph* consists of a list of *Vertex[]* and a list of *Edge[],* which are its components. Each *Edge* has two references, one to its *source Vertex* and one to its *target Vertex*.

```
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 12: A textual metamodel for Graph

Figure 12 shows the same metamodel as text, using the ReMoDeL metamodel syntax. The *Graph* concept is now the container for every *Vertex* and every *Edge*. Each *Vertex* is merely labelled. Each *Edge* is a separate concept, denoting an arrow from a *source Vertex* to a *target Vertex*. It is possible to find the *root Vertex* by a searching operation:

- *Graph.root : Vertex* – searches for the only *Vertex*, for which no *Edge* exists, which has this *Vertex* as its *source* (this is the definition of a root!)

3. Model Transformation

The notion of *model transformation* may be unfamiliar. Model transformation (MT) is the process of converting models from one kind into another. In Software Engineering, we are concerned almost exclusively with transformations that affect models representing different views of a software system. Transformations may abstract a design, refine a design, translate a design, or fold together different views of the system.

Model transformations are classified along different, independent dimensions [5, 6]:

- **direction**: transformations are classified as *unidirectional* (they transform a source to a target model), or *bidirectional* if they are reversible (also transform a target to a source model);
- **domain**: transformations are classified as *endogenous* (*rephrasing* within the same language or metamodel) or *exogenous* (*translating* from one language or metamodel to another);
- **valency**: transformations are classified according to the number of source and target models as *mapping* (from one model to another), *merging* (of many models into one), *splitting* (of one model into many), or *updating* (modifying the same model in-place);
- **abstraction**: transformations are classified as *horizontal* (*migration*, preserving the abstraction-level), or *vertical* (*refinement*, becoming more concrete; or *abstraction*, becoming more abstract);
- **purpose**: transformations are classified as *translation* (of one model into another), *refactoring* (improving model quality), *enhancement* (inserting features into a model), *normalisation* (simplifying the model), *consistency maintenance* (of a model with respect to another).

Model transformations can be expressed as imperative programs that manipulate data structures in an opaque way, or as declarative rule-based languages that express transformations in a more transparent way. Consider the following:

- **Imperative Transformation**: programming language statements may create, modify or delete elements from the model in arbitrary ways. The transformation depends on the states of variables in the program and the order of statement execution, using assignment and iterative loops; and it is not easy to discern the logic being applied.
- **Declarative Transformation**: the transformation is expressed as a mapping from the input space to the output space. The transformation is more like a pure functional programming language (Haskell, ML, etc.) and does not depend on state variables. Repeated actions are accomplished using recursion, and variables are bound just once (not reassigned).
- **Pattern-Matching Transformation**: the transformation is expressed as a logical pattern, specifying a set of constraints between the input space and the output space. Rules are phrased using quantification: "*for all* X in the source, *there exists* Y in the target, such that certain properties hold". This is the most abstract way of specifying a transformation, and must be interpreted, or converted into an operational form before execution.

Clearly, there is a trade-off between the comprehensibility and executability of transformations in these different approaches. Other concerns include:

- *idempotence*, the notion that if the same transformation is invoked in multiple places on the same source entity, it should always yield the same target entity (not a duplicate copy);
- *independence*, the notion that different rules may specify different aspects of the same transformation independently, but these are folded together correctly in the target.

3.1 InTree to OutTree Transformation

The first example model transformation we will consider is a simple *translation*. The *InTree* and *OutTree* metamodels describe *Tree* and *Node* concepts, which support constructing tree models which are broadly similar, apart from the direction in which nodes are linked. It is therefore possible to provide a *mapping* transformation that performs a *horizontal migration* of a model from one metamodel to another (an *exogenous translation*, according to the taxonomy of model transformations).

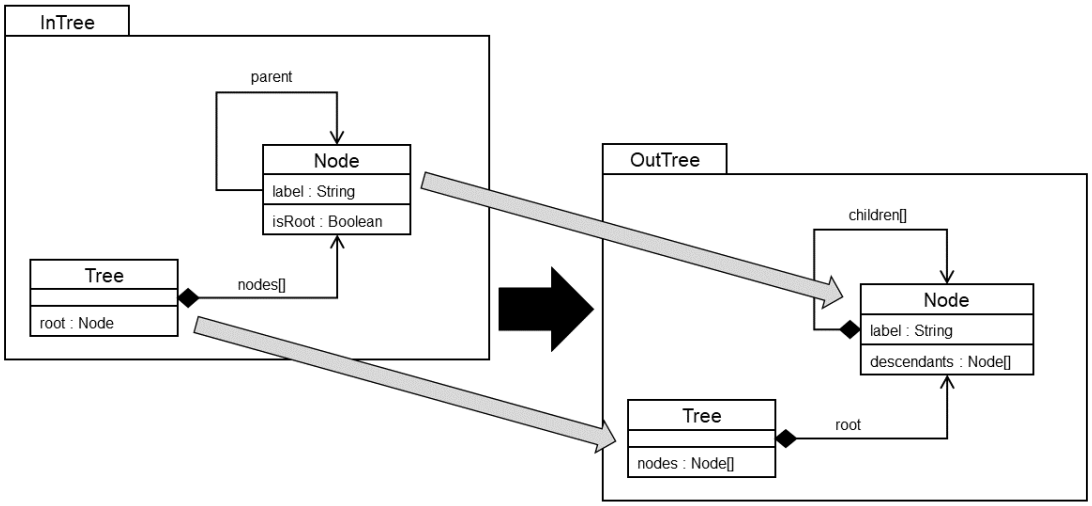

Figure 13: Visualising the InTree to OutTree transformation

Figure 13 illustrates the transformation as a *homomorphism* from one metamodel to another, which consists of a number of *morphisms* that map elements of one metamodel onto elements of the other. That is, the large black arrow denotes the transformation as a whole, which consists of several smaller grey arrows denoting individual rules that map elements from one metamodel onto elements of the other.

```
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 14: The InTree to OutTree transformation

Figure 14 shows the complete model transformation using the *ReMoDeL* textual syntax for transformations. The transformation is called *InTreeToOutTree*, and it belongs to a group of transformations known as *Trees*. It specifies the two metamodels being used (*InTree* for

source entities; and *OutTree* for target entities). It encapsulates a set of mapping rules, which perform the actual transformation. A few more keywords of the *ReMoDeL* syntax are:

- **transform** – the keyword introducing a model transformation;
- **metamodel** – the keyword introducing source, target metamodels;
- **mapping** – the keyword introducing a mapping rule.

All *source* metamodels are declared first, before the *target* metamodel, which always comes last. The mapping rules are declared after this, with the first rule (the *top rule*) being the rule that transforms the topmost element in the model, here *inTree : InTree_Tree*.

A mapping rule may have one or more *source* arguments, and yields one *target* result. The argument and result types are declared in their full metamodel-qualified form, because the type-names alone are not sufficient to distinguish which *Tree* or *Node* we mean (concepts with the same names exist in both metamodels).

Each rule is a function from inputs to output, whose body is a single nested expression in the *ReMoDeL* syntax. The rules have the following meaning:

- *inTreeToOutTree* – for each *inTree : InTree_Tree*, create a new *OutTree_Tree*, whose *root : OutTree_Node* node is the result of mapping the *root : InTree_Node*. This invokes the second rule:
- *inNodeToOutNode* – for each *inNode : InTree_Node*, create a new *OutTree_Node*, whose *label* is the same as that of the *inNode*, and whose *children* are found by filtering all the nodes of the *inTree* to select only those *child* nodes, whose *parent* refers to this *inNode*; and then recursively transform this list of *InTree_Node[]* nodes into corresponding *OutTree_Node[]* children.

To see how the whole model transformation works, imagine that we call the top-rule on the topmost element *t1: Tree* of the *in-tree* in figure 2. This gives the following recursive pattern of rule-invocation, shown in figure 15.

```
t1' := inTreeToOutTree(t1)
   n1' := inNodeToOutNode(n1, t1)
      n2' := inNodeToOutNode(n2, t1)
         n4' := inNodeToOutNode(n4, t1)
         n5' := inNodeToOutNode(n5, t1)
      n3' := inNodeToOutNode(n3, t1)
         n6' := inNodeToOutNode(n6, t1)
```

Figure 15: Recursive mapping rule invocation pattern

This shows how the elements *t1', n1'..n6'* of the target model are constructed in turn, by applying the rules to elements *t1, n1..n6* of the source model. The indentation in the figure indicates the nested calling of a rule by the next outer rule above it.

The recursion stops, when no further children are found for a given node. For example, the last application to *n6* and *t1* finds that none of the *t1.nodes* is a child of *n6*. Therefore, the *select* method returns an empty list, and the nested collect operation likewise returns an empty list, and the rule is not invoked recursively a further time.

In the body of these rules, we see some examples of the *ReMoDeL* syntax:

- *create* – is an operation that constructs a new instance of a concept type. The syntax is: *create Type(property1 := value1, ... propertyN := valueN)*, for as many properties of the type as are supplied (possibly an incomplete list).
- *select* – is a filtering operation applied to collections, which retains any object satisfying a test. The syntax is: *collection.select(element | condition-on-element)*.
- *collect* – is a transforming operation applied to collections, which maps each element to an element of another type, returning a collection of the same size. The syntax is: *collection.collect(element | transformation-of-element)*.

3.2 InTree to Graph Transformation

The second example is a slightly more elaborate translation. The *InTree* and *Graph* metamodels from figures 3 and 11 describe different sets of concepts: {*Tree, Node*} and {*Graph, Vertex, Edge*} respectively. Whereas in the *InTree* metamodel, the nodes refer directly to other nodes, in the *Graph* metamodel, the vertices are connected by explicit edges. Nevertheless, it should be possible provide a *mapping* transformation that performs a *horizontal migration* of a model from one metamodel to another (an *exogenous translation*, according to the taxonomy of model transformations).

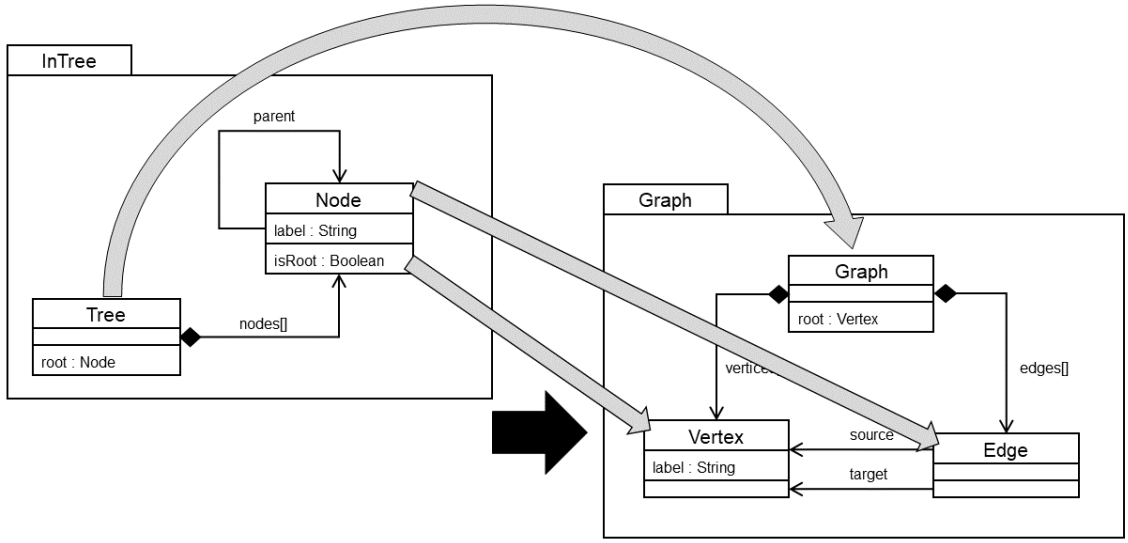

Figure 16: Visualising the InTree to Graph transformation

Figure 16 illustrates this transformation as a different *homomorphism* from one metamodel to another, which consists of a number of *morphisms* that map elements of one metamodel onto elements of the other. The large black arrow denotes the transformation as a whole, which consists of several smaller grey arrows denoting individual rules that map elements from one metamodel onto elements of the other.

The main difference between this and the earlier transformation in figure 13 is the additional rule required. The top rule maps a *Tree* to a *Graph*; another rule maps a *Node* to a *Vertex*; and a further rule maps a *Node* to an *Edge*. How this is done is described more completely in figure 17, which specifies the full details of each rule, using the *ReMoDeL* syntax.

```
transform InTreeToGraph : Trees {

   metamodel source : InTree
   metamodel target : Graph

   mapping inTreeToGraph (inTree : InTree_Tree) : Graph_Graph {
      create Graph_Graph(
         vertices := inTree.nodes.collect(node |
            inNodeToVertex(node)),
         edges := inTree.nodes.without(inTree.root)
                     .collect(node | inNodeToEdge(node))
   }

   mapping inNodeToVertex(inNode : InTree_Node) : Graph_Vertex {
      create Graph_Vertex(label := inNode.label)
   }

   mapping inNodeToEdge (inNode : InTree_Node) : Graph_Edge {
      create Graph_Edge(
         source := inNodeToVertex(inNode),
         target := inNodeToVertex(inNode.parent))
      )
   }
}
```

Figure 17: The InTree to Graph transformation

The transformation is called *InTreeToGraph*, and it belongs to the group of transformations known as *Trees*. It specifies the two metamodels being used (*InTree* for source entities; and *Graph* for target entities). The top rule is called *inTreeToGraph*.

As in the previous example, the mapping rules are functions from input to output, and have the following meaning:

- *inTreeToGraph* – for each *inTree : InTree_Tree*, create a new *Graph_Graph*, whose *vertices : Graph_Vertex[]* are the result of mapping the rule: *inNodeToVertex* over all the nodes of the *inTree*; and whose edges: *Graph_Edge[]* are the result of mapping the rule: *inNodeToEdge* over all but one of these nodes, that is, over all the nodes of the *inTree* except the *root* node.
- *inNodeToVertex* – for each *inNode : InTree_Node*, create a new *Graph_Vertex*, whose *label* is the same as that of the *inNode*.
- *inNodeToEdge* – for each *inNode : InTree_Node*, create a new *Graph_Edge*, whose *source* is the result of invoking *inNodeToVertex* on the *inNode*, and whose *target* is the result of invoking *inNodeToVertex* on the *inNode's parent* node.

One aspect of these rules is appealing. Each rule states declaratively how it maps its source to target elements. For example, the rule *inTreeToGraph* states that it uses *inNodeToVertex* to map nodes to vertices, and uses *inNodeToEdge* to map nodes to edges. Similarly, the rule *inNodeToEdge* uses *inNodeToVertex* again, to map the same nodes to source and target vertices. Does this cause problems, if the same nodes are mapped multiple times?

3.3 Idempotence of Rules

To see how applying rules multiple times to some source elements might cause problems, we can trace the pattern of rule-invocation. Imagine that we call the top rule on the topmost

element *t1: Tree* of the *in-tree* in figure 2. This gives the following recursive pattern of rule-invocation, shown in figure 18.

```
g1' := inTreeToGraph(t1)
   v1' := inNodeToVertex(n1)
   v2' := inNodeToVertex(n2)
   v3' := inNodeToVertex(n3)
   v4' := inNodeToVertex(n4)
   v5' := inNodeToVertex(n5)
   v6' := inNodeToVertex(n6)
   e1' := inNodeToEdge(n2)
      v2'' := inNodeToVertex(n2)
      v1'' := inNodeToVertex(n1)
   e2' := inNodeToEdge(n3)
      v3'' := inNodeToVertex(n3)
      v1''' := inNodeToVertex(n1)
   e3' := inNodeToEdge(n4)
      v4'' := inNodeToVertex(n4)
      v2''' := inNodeToVertex(n2)
   e4' := inNodeToEdge(n5)
      v5'' := inNodeToVertex(n5)
      v2'''' := inNodeToVertex(n2)
   e5' := inNodeToEdge(n6)
      v6'' := inNodeToVertex(n6)
      v3''' := inNodeToVertex(n3)
```

Figure 18: Recursive mapping rule invocation pattern

Initially, every rule creates a target element (indicated with a single prime *x′*) when invoked for the first time on a source element. However, the rule *inNodeToVertex* is eventually invoked many times on some source elements, mapping to the same target many times, indicated by adding further primes (viz. *x″* or *x‴*). So, for example, the node *n2* is mapped to *v2* a total of four times:

- firstly, when *n2* is mapped to *v2′* as one of the vertices of *Graph g1*
- secondly, when *n2* is mapped to *v2″* as the source of the *Edge e1*
- thirdly, when *n2* is mapped to *v2‴* as the target of the *Edge e2*
- fourthly, when *n2* is mapped to *v2⁗* as the target of the *Edge e4*.

From the point of view of writing a declarative specification, this is desirable. It says that: the *Edge* you create for a given *Node*, has a *source*, which is the result of transforming this *Node* to a *Vertex*, and a *target*, which is the result of transforming the *Node's parent* to a *Vertex*.

From a concrete programming point of view, this could lead to a disaster if, every time the rule *inNodeToVertex* were called, this created a new *Vertex*. The resulting model would then contain many duplicates of the same *Vertex*, which is not desirable (and is incorrect).

For this reason, every rule is designed to be *idempotent*, which means that if it is applied more than once to the same *source entity*, this always returns the *same target entity* (and not a new copy of it). The *ReMoDeL* compiler ensures, in the executable code generated for a transformation, that the results of each mapping rule are cached and retrieved, rather than recomputed, when that rule is applied to the same source argument(s) subsequently.

3.4 Inverse Transformations

In some approaches, model transformations are *bidirectional*. This is only possible if sufficient information is preserved in a transformation to support the inverse transformation (if information is lost, this is not possible). Some pattern-matching languages offer rule patterns which can be applied in either direction (forwards, or in reverse). In *ReMoDeL*, transformations are all *unidirectional*, but an *inverse transformation* may exist.

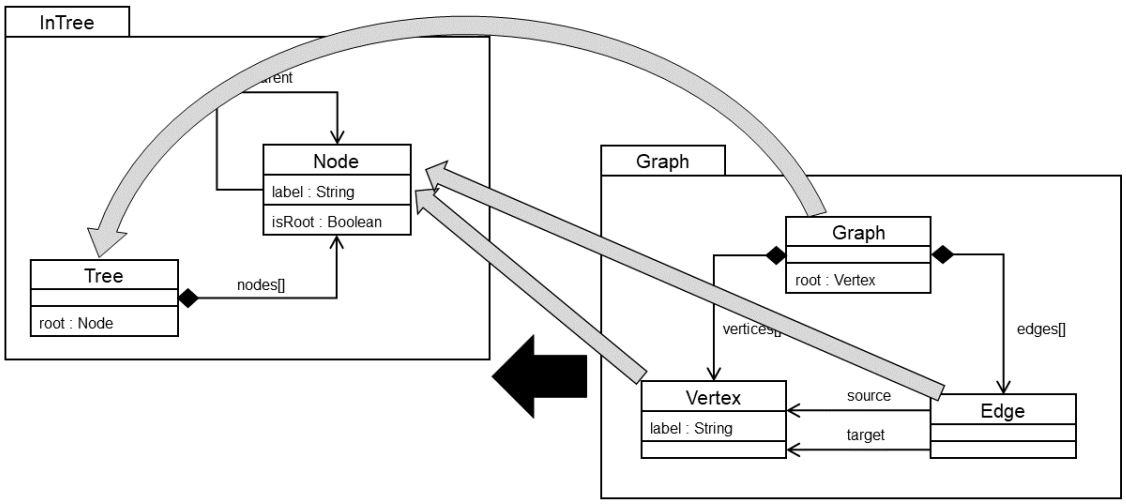

Figure 19: Visualising the GraphToInTree inverse transformation

Figure 19 shows the inverse transformation *GraphToInTree*, which transforms a *Graph* model back to an *InTree* model. This is possible because all of the different representations of trees preserve full information. Figure 20 shows the same transformation in the *ReMoDeL* textual notation:

```
transform GraphToInTree : Trees {

   metamodel source : Graph
   metamodel target : InTree

   mapping graphToInTree(graph : Graph_Graph) : InTree_Tree {
      create InTree_Tree(
         nodes := graph.vertices.collect(vertex |
            vertexToInNode(vertex, graph))
      )
   }

   mapping vertexToInNode(vertex : Graph_Vertex,
                          graph : Graph_Graph) : InTree_Node {
      if vertex = graph.root
         then create InTree_Node(label := vertex.label
         )
         else create InTree_Node(label := vertex.label,
            parent := vertexToInNode(graph.edges.detect(edge |
               edge.source = vertex).target, graph)
         )
   }
}
```

Figure 20: The inverse Graph to InTree transformation

The transformation rules have the following meanings:

- *graphToInTree* – for each *graph: Graph_Graph*, create a new *InTree_Tree* whose nodes are the result of mapping the *graph*'s *nodes* using the rule *vertexToInNode*.
- *vertexToInNode* – for each *vertex: Graph_Vertex*, create a new *InTree_Node*, whose *label* is the same as that of the *vertex*; and if the *vertex* is not the *root*, whose *parent* is the result of mapping the parent vertex, using *vertexToInNode* recursively. The parent vertex is the *target* of the *edge* referring to this *vertex* as its *source*. The *edge* is found by searching the *graph*'s *edges* for the only *edge* with this *vertex* as its *source*.

In the body of these rules, we see some more examples of the *ReMoDeL* syntax:

- *if...then...else* – is a conditional expression that tests a Boolean value and then returns the result of either the *then*-branch (in the *true* case), or the *else*-branch (in the *false* case). The two branches must return a value of the same type.
- *detect* – is a filtering operation applied to collections, which returns a single element passing a test. The syntax is: *collection.detect(element | condition-on-element)*.

In keeping with the pure functional flavour of *ReMoDeL*, the *if...then...else* conditional form is an expression, not a control statement; viz. it returns a value. The *detect()* filtering operation returns an element passing a test, or *null* if none exists. If many elements pass the test, then *detect()* returns the first such element of a *List*, or an arbitrary element of a *Set*.

The transformation *GraphToInTree* makes assumptions about the kind of *graph* to which it is applied. Firstly, it assumes a fully-connected directed acyclic *graph*, with only one *root* vertex. If this were not the case, then the rule *vertexToInNode* would be incorrectly specified. Secondly, it assumes that every *vertex* has at most one parent, such that the graph may be transformed exactly back into a tree.

```
model graph2 : Graph {
   g1 : Graph(vertices = Vertex[
      v1 : Vertex(label = "Root"),
      v2 : Vertex(label = "Branch1"),
      v3 : Vertex(label = "Branch2"),
      v4 : Vertex(label = "Leaf1"),
      v5 : Vertex(label = "Leaf2"),
      v6 : Vertex(label = "Leaf3")
   ], edges = Edge[
      e1 : Edge(source = v2, target = v1),
      e2 : Edge(source = v3, target = v1),
      e3 : Edge(source = v4, target = v2),
      e4 : Edge(source = v5, target = v2),
      e5 : Edge(source = v6, target = v3),
      e6 : Edge(source = v5, target = v3),
      e7 : Edge(source = v6, target = v1)
   ])
}
```

Figure 21: A textual model of a directed acyclic graph

What if the graph in question were a more general kind of directed acyclic graph, in which vertices could be linked to many parents and many children? This case is shown in figure 21, in which extra edges *e6, e7* have been added to the model from figure 10, to link source vertices *v5, v6* respectively to the extra targets *v3, v1*. This means that *v5, v6* have multiple parents.

If we apply the top rule *graphToInTree* to the *graph g1*, the transformation still works (after a fashion), and produces a tree model like that shown in figure 2. The reason that it produces a tree is because the *detect()* operation picks at most one edge to link any given vertex to a parent vertex. In fact, it picks the first such edges found in the model *graph2*, which means that the search never reaches *e6, e7*, which are ordered later in the list of *Edges*.

Some information about the graph is lost in this reverse transformation from *g1* to *t1'*. We can see this by applying the forwards transformation *inTreeToGraph* to *t1'*, which yields a graph *g1'*, identical to that shown in figure 10, and not the same as the more general graph in figure 21. That is, the edges *e6* and *e7* vanish.

From this, we may determine that:

- *InTreeToGraph* has an inverse transformation *GraphToInTree*; viz. it is always possible to convert an *InTree* into a *Graph*, which can be converted back into an identical *InTree*.
- *GraphToInTree* does not have a total inverse transformation; viz. while it is possible to convert a *Graph* to some kind of *InTree*, it is not always possible to convert this back into an identical *Graph*.

3.5 Partial Transformations

The rule *vertexToInNode* contains an expression, which depends critically on a graph being fully-connected, with every *vertex* apart from the *root* having a parent. The following is applied to every other vertex, after the *root* has been handled by a special case:

```
graph.edges.detect(edge | edge.source = vertex).target
```

That is, the expression assumes that there will always be a detected *edge*, from which it is possible to select the *target* vertex. If no *edge* were found for some *source* vertex, then this code would fail (*null* has no *target*). If graphs are allowed to have multiple roots (a *forest*, rather than a *tree*), this means that transformations to a *tree* will sometimes fail.

This raises the prospect of *partial transformations*, that is, transformations that are not applicable to every source model. The usual way to handle this is to specify a precondition, which accepts only those models to which the transformation may legally be applied.

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

mapping graphToInTree(graph : Graph_Graph) : InTree_Tree {
   if isTree(graph)
      then create InTree_Tree(
         nodes := graph.vertices.collect(vertex |
              vertexToInNode(vertex, graph))
      )
      else null
}
...
```

Figure 22: Defining a precondition (in a transformation)

Figure 22 shows how a precondition may be defined as a simple function, which is used in a transformation rule. Functions are distinguished from rules by the keyword:

- **function** – the keyword introducing an auxiliary function.

Functions are not idempotent, so compute their results every time they are called. Here, the function *isTree(Graph)* is a Boolean-valued predicate which can be invoked on a graph to see if it is in fact a tree (rather than a forest). The body of the function counts the number of roots in the *graph*, expecting this to be exactly one. The roots are found by selecting all vertices in the *graph* with no outgoing edges.

This precondition is tested inside the rule *graphToInTree*. If the precondition is *true*, the transformation may safely proceed; otherwise, it does not exist, and the result is therefore *null*. This is formally correct, if we wish to assert that there is no possible transformation from a graph to a tree, unless it is a single-rooted graph. Of course, it is dangerous to return *null* values, if other transformations expect non-null values.

In figure 22, we expressed the precondition as part of the transformation. It is also possible to specify constraints as part of the metamodel. Figure 23 shows the alternative style, in which *isTree: Boolean* is an operation of the *Graph* concept:

```
metamodel Graph {
   concept Graph {
      component vertices : Vertex[]
      component edges : Edge[]
      operation roots : Vertex[] {
         vertices.select(vertex |
            not edges.exists(edge | edge.source = vertex))
      }
      operation isTree : Boolean {
         roots.size = 1
      }
   }
   concept Vertex {
      attribute label : String
   }
   concept Edge {
      reference source : Vertex
      reference target : Vertex
   }
}
```

Figure 23: Defining a constraint (in a revised Graph metamodel)

Figure 23 is a revision of the *Graph* metamodel from figure 12. Here, we allow the possibility of multiple *roots* (a graph may now be a forest). However, the transformation back to an *InTree* is only possible if *isTree* holds.

Figure 24 shows the revised partial transformation from a *Graph* to an *InTree*. Here, we assume that the *Graph* metamodel is as specified in figure 23 (where graphs may be forests). The other changes in figure 24 to the original *GraphToInTree* transformation from figure 20 are that the precondition has been added in *graphToInTree*, and the way in which a *root* is detected is different in *vertexToInNode*, reflecting the change to the metamodel:

```
transform GraphToInTree : Trees {

   metamodel source : Graph
   metamodel target : InTree

   mapping graphToInTree(graph : Graph_Graph) : InTree_Tree {
      if graph.isTree
         then create InTree_Tree(
            nodes := graph.vertices.collect(vertex |
               vertexToInNode(vertex, graph))
         )
         else null
   }

   mapping vertexToInNode(vertex : Graph_Vertex,
                          graph : Graph_Graph) : InTree_Node {
      if graph.roots.has(vertex)
         then create InTree_Node(label := vertex.label
         )
         else create InTree_Node(label := vertex.label,
            parent := vertexToInNode(graph.edges.detect(edge |
               edge.source = vertex).target, graph)
         )
   }
}
```

Figure 24: The partial inverse Graph to InTree transformation

3.6 Other Kinds of Transformations

All the transformations considered so far are *translations* from one metamodel to another. Other kinds of transformation are possible. While we don't have room to give examples of these in this introduction, we can describe the flavour of several other kinds.

Enhancement – An example of this might be a transformation that ensures that every *Entity* in an *Entity Relationship Model* (ERM) is identifiable. The source and target metamodel is the same (ERM) and consists of a *Diagram* containing *Entities* and *Relationships*. The *Entities* contain *Attributes*, some of which may have an *id* property set to *true*, to indicate a candidate key.

The transformation is *endogenous*, consisting of a collection of *mapping rules*, with a rule to map each concept to the same concept (with changes included). The rule for entities will check whether the source *Entity* has at least one *Attribute* with *id = true*; and if not, the transformation will append to the target *Entity's* attributes a surrogate key *Attribute*, whose name is synthesised from the *Entity's* name: *entity.name.asName.concat("Id")*.

Merging or folding – An example of this might be a transformation that constructs a procedural call-graph (Proc) from two source models conforming to two metamodels, *Jackson Structured Programming* (JSP) and *Dataflow Diagram* (DFD). There are three metamodels altogether, the two sources and the target. The JSP model *Diagram* consists of named *Blocks,* each marked as one of *{sequence, selection, iteration}* and optionally having component *Block* children. The DFD model *Diagram* consists of named *Process* nodes with *Dataflow* edges indicating what *Datatype* flows from one *Process* to another.

The transformation is *exogenous*, consisting of a collection of *merging rules*, each of which accepts a pair of concepts from *source1* (JSP) and from *source2* (DFD). The rule for

Procedures will match up a *Block* and a *Process* having the same *name*, and will construct a *Procedure*, whose *Inputs* are obtained from the *Datatype*(s) on the *Dataflow* arrow(s) targeting the *Process*, and whose *Outputs* are likewise obtained from the *Dataflow* arrows exiting the same *Process*. The call-graph structure will be determined from the *Block* structure. The transformation may also generate named *Variables* within a *Procedure* to store the results of executing one sub-*Procedure*, so that these are available to the next sub-*Procedure*. The conditional expressions governing branching and looping will be taken from the *Blocks* marked as *{selection, iteration}*.

Normalisation – An example of this might be a transformation that constructs a normalised *Entity Relationship Model* (ERM) in third normal form, from a pre-normal ERM. The source and target metamodel is the same (ERM) and consists of a *Diagram* containing *Entities* and *Relationships*. Some *Entities* are related by many-to-many *Relationships*; and some are related by one-to-one *Relationships*. The rest are related by many-to-one *Relationships*.

The transformation is *endogenous*, consisting of a set of mapping rules that merge sets of *Entities* connected by one-to-one *Relationships*, and split many-to-many *Relationships* into a pair of many-to-one *Relationships* joined to an intermediate linker *Entity*. The merging rule uses an auxiliary function to compute transitive closures over one-to-one related *Entities*, such that any entity in this set is mapped to the same merged *Entity*. All *Relationships* joined to any one of the closure set are mapped to *Relationships* joined to the merged *Entity*. The name of the linker *Entity* is synthesized, and the composite key *Attributes* of the linker are created by renaming the key *Attributes* of the many-to-many linked *Entities*.

4 The Expression Language

So far we have seen only small examples of the *ReMoDeL* expression language. This is the syntax used to define the body of an *operation*, or the body of a *mapping rule.* The same executable syntax is used for both. The expression language is compact and has the flavour of both object-oriented and pure functional programming styles.

4.1 Basic Operations

ReMoDeL comes with a number of predefined types and operations. The predefined basic types are *Boolean, Integer, Decimal, Character* and *String*. These are all valid operands to comparison operators. The numerical types *Integer* and *Decimal* are valid operands to the arithmetic operators, which include modulo and power operators. The *Boolean* values are valid operands to the logical operators.

- **constants**: `null, false, true`
- **comparison**: `=, /=, <, >, <=, >=`
- **arithmetic**: `+, -, *, /, %, ^`
- **logical**: `and, or, not`
- **creation**: `create` *`Type`*`(`*`f1`*` := `*`e1`*`, ..., `*`fN`*` := `*`eN`*`)`
- **conditional**: `if` *`e1`* `then` *`e2`* `else` *`e3`*
- **compound**: *`e1`*`; ...;` *`eN`*

Note that the equality operator is a single equals-sign (not a double-equals sign) and the inequality operator is slash-equals (not exclamation-equals). The creation operator is used with an initialisation operator (colon-equals) to initialise the fields of created objects. Multiple initialisations are separated using a comma.

The conditional branching expression returns either the left or right branch value. These must be of the same type. No parentheses are needed to surround the test-expression (parentheses are used to alter the precedence of binary expressions). The compound expression returns the value of the final sub-expression, discarding the values of earlier sub-expressions. Multiple sub-expressions are separated using a semicolon. Note that all punctuation marks are separators, not terminators.

4.2 Concept Operations

Every type in *ReMoDeL* is a concept (including the basic types). Concepts exist in a classification hierarchy. A concept may declare that it inherits from a parent concept within the same metamodel, in which case it obtains the property declarations of the parent concept, before any properties it defines locally. If no explicit parent is declared, the concept inherits implicitly from *Top*, which is the root concept.

- **inheritance**: `concept` *`C1`* `inherit` *`C2`* `{ ... }`
- **conversion**: `asSet:` *`C`*`{}, asList:` *`C`*`[]`

Top provides two conversion operations that convert any concept into a list, or set. These are invoked like methods on a variable or expression denoting a concept. When invoked on a simple concept *C*, they return a singleton set *C{}* or singleton list *C[]* containing the concept.

When invoked on a collection (list or set), they return another collection of the requested type (conversions to the *same* type trivially return the same instance).

The basic concepts: *Boolean, Integer, Decimal, Character* and *String* are added to every metamodel as predefined concepts. Whenever any concept *C* is used in lists or sets, the appropriate set *C{}* or list *C[]* concept is added to the metamodel dynamically.

The basic concepts offer the binary operations described above. The *String* concept also provides further predefined operations, invoked like methods:

- **predicates**:
  ```
  isEmpty: Boolean
  startsWith(String): Boolean
  endsWith(String): Boolean
  contains(String): Boolean
  ```
  inspection:
  ```
  length: Boolean
  indexOf(Character): Integer
  charAt(Integer): Character
  ```
- **combination**:
  ```
  concat(String): String
  substring(Integer): String
  substring(Integer, Integer): String
  ```
- **case conversion**:
  ```
  asName: String
  asType: String
  ```

These support the manipulation of text. The *String*-appending operation is called *concat()* and the *String*-splitting operation is called *substring()*. These operations work like their equivalents in Java. The operations *startsWith(), endsWith()* and *contains()* test for the presence of substrings. The case conversion operations convert *String* values between "*nameCase*" (used for property names) and "*TypeCase*" (used for type names). All of these operations return new *String* values, and do not modify the original *String*.

4.3 Collection Operations

The predefined concepts *List* and *Set* inherit from a common *Collection* concept. These types are parametric in the type of their element. You will never see the type names *List*, *Set* directly in the expression language, but only specific type-instantiations, such as *Node[]* (c.f. a *List<Node>* in Java) or *Integer{}* (c.f. a *Set<Integer>* in Java).

Collection specifies a common set of operations, inherited by *Set* and *List*:

- **predicates**:
  ```
  isEmpty: Boolean
  has(Element) : Boolean
  count(Element): Integer
  ```
- **inspection**:
  ```
  size: Integer
  ```

- **combination**:
  ```
  with(Element): Collection
  without(Element): Collection
  ```

The operations *with()* and *without()* are the principal way of adding and removing elements singly in the pure functional expression language. All operations affecting collections are *constructive*, that is, they construct a new instance of the same kind of collection (they do not modify the collection). The operation *has()* tests whether an element is present in a collection; and *count()* counts the number of occurrences of an element.

The *Set* and *List* concepts provide distinct operations, in addition to those inherited from *Collection*. *Sets* contain unique elements (viz. no duplicates) and are ordered by element insertion. The *Set* operations that return *Set* results create a new *Set* value:

- **combination**:
  ```
  pick : Element
  union(Set): Set
  intersection(Set): Set
  difference(Set): Set
  ```

Lists may contain duplicate elements and are ordered by element insertion. The *List* operations that return *List* results create a new *List* value:

- **combination**:
  ```
  first: Element
  rest: List
  append(List): List
  ```

Lists may be used in recursive functions, using *lst.first* and *lst.rest* to select the head element and the tail of the list. *Sets* may also be processed recursively, using *set.pick* to select an element, and *set.without(set.pick)* to give the rest of the set. The *asList()* and *asSet()* operations convert from one to the other, where required, preserving order.

4.4 Higher-Order Operations

Collections play a major part in *ReMoDeL*, since rules frequently involve filtering collections to find a subset to which some other transformation is applied. Collections provide higher-order operations that accept a predicate or function argument (a *lambda expression*), applying this to every element of the collection. These operations fall into four groups:

- **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): Collection
  ```

A lambda expression is an anonymous function (or predicate) constructed dynamically. It has the syntactic form: (*variable* | *expression-containing-variable*). The variable denotes one element of the collection, and may be declared with a type, which must be the element-type of the collection (for convenience, the type may be omitted, in which case it is inferred from the type of the collection – see below for a discussion of type inference).

A lambda expression is a *predicate* if the body is a Boolean-valued expression. Otherwise, it is a *function*, returning some other typed value. When the lambda expression is passed as an argument to one of the higher-order collection operations, it is applied to every element of the collection. The meaning of these higher-order operations is as follows:

- *forall* – accepts a predicate and returns true if all elements of the collection pass this test. The syntax is: *collection.forall(element | condition-on-element)*.
- *exists* – accepts predicate and returns true if any element of the collection passes this test. The syntax is: *collection.exists(element | condition-on-element)*.
- *detect* – accepts a predicate and returns the first object passing this test, or *null* if none is found. The syntax is: *collection.detect(element | condition-on-element)*.
- *select* – accepts a predicate and returns a filtered collection, selecting objects that pass this test. The syntax is: *collection.select(element | condition-on-element)*.
- *reject* – accepts a predicate and returns a filtered collection, rejecting objects that pass this test. The syntax is: *collection.reject(element | condition-on-element)*.
- *collect* – accepts a function, which maps each element to an element of another type, and returns a collection of the same size, containing elements of the other type. The syntax is: *collection.collect(element | transformation-of-element)*.
- *collate* – accepts a function, which maps each element to a collection of another type but returns a single flat collection of the other element-type, formed by merging the results. The syntax is: *collection.collate(element | transformation-of-element)*.
- *reduce* – accepts a reduction (a reducing function of two arguments) which recursively combines the elements of the collection, returning a single element. The syntax is: *collection.reduce(elem1, elem2 | reduction-of-elem1-and-elem2)*

The filtering operations typically return a collection of the same kind as the collection on which they were invoked. Filtering a *List* returns a *List*, and filtering a *Set* returns a *Set*. The element-type of the result is typically the same as that of the collection. However, the *select* operation may return a collection of a more specific element type (this involves a runtime type cast). The *detect* operation returns one element of the element-type (or *null*).

The mapping operations return collections of the same kind, with a different element-type (according to the mapped lambda expression). Mapping over a *List* returns a *List*, and mapping over a *Set* returns a *Set* of the same size as the original collection. With the *collate* operation, the lambda expression is expected to map individual elements to *List* or *Set* results. The results are then merged either using *append* (for *Lists*) or *union* (for *Sets*).

The reducing operation *reduce* accepts a binary function that aggregates two elements into one element of the same type (e.g. the sum, or the maximum, of two numbers). If applied to an empty collection, the result is *null*. If applied to a singleton, the result is that element. Otherwise, the result is the accumulated aggregation of all the elements in the collection (mathematically, this is a limited kind of *left-fold* reduction).

The operations *select, reject, detect* and *collect* are used frequently. These are operations that filter, or transform *Sets* or *Lists* of items. Because filtering is common in transformations, we may see a *collect* (or *collate*) operation applied to the result of a previous *select* (or *reject*) operation. This gives rise to nested invocations, in which the collection resulting from one invocation may become the subject of the next invocation.

An example of nesting was shown in figure 14, in the rule *inNodeToOutNode*, where a *select* operation is used to filter the list *inTree.nodes*, and then a *collect* operation is applied to this result to convert every selected *InTree_Node* node to an *OutTree_Node*.

4.5 Variables

The expression language uses named variables to refer to different entities or values. The variables that are in scope at any time may include the following:

- The *attribute, reference,* or *component* properties of a *Concept* may be used within operations of the *Concept*, or any inheriting *Concept*.
- The formal arguments of any operation or transformation rule may be used within the scope of that operation or rule, which may also refer to properties in the outer scope.
- The lambda variables of any lambda expression may be used within the lambda body, which may also refer to variables in the outer scope.
- The following are special variables: *self* denotes the current concept instance; *super* denotes this in its supertype context; and *owner* denotes a component's (optional) back-reference to its owning container type. If present, *owner* is automatically set when the component is added to its container. The *owner* back-reference is transient and is never serialised in models.

The names of variables must be unique within their scope. An exception to this is that variables may also shadow the names of *ReMoDeL* keywords, such as *attribute* and *reference*, without ambiguity due to the context of usage.

4.6 Type Inference

The *ReMoDeL* toolset can perform limited type inference, to determine the types of certain expressions, where these are not given explicitly. For example, in figure 4, the *InTree* metamodel has an operation *root: Node* containing an expression:

```
nodes.detect(node | node.isRoot)
```

in which the type of the lambda variable *node* is not given. In this case, the *ReMoDeL* tools determine the type by inference. The heuristic used is that a lambda variable must always represent an element of a collection. The collection is denoted by the identifier *nodes*, which refers to a component declared in *Tree* with the list type: *nodes: Node[]*. The element type is therefore inferred to be *Node*. To avoid relying on type inference, it is possible to give explicit types to lambda variables, like this:

```
nodes.detect(node : Node | node.isRoot)
```

Type inference is more challenging in transformation rules, where there may be more than one type having the same name, in different metamodels. In figure 14, the *InTreeToOutTree* transformation has a mapping rule *inNodeToOutNode* containing an expression:

```
inTree.nodes.select(child | child.parent = inNode)
```

in which the type of the lambda variable *child* is not given. We need to know if this refers to an *InTree_Node* or an *OutTree_Node*. In this case, the heuristic performs qualified type inference, seeking the metamodel-qualified type of the collection. In this case, the expression *inTree.nodes* refers to a component declared in *InTree_Tree*, having the qualified list type *InTree_Node[]*, from which we infer that *child* has the element type *InTree_Node*.

This inference is only possible because the rule's argument (in figure 14) was declared with a metamodel-qualified type: *inTree: InTree_Tree* in the first place. If types are not given in their qualified forms, then another heuristic searches through the declared metamodels for a unique declaration of the type. Since this is not guaranteed, a semantic error may be raised if the type's owning metamodel cannot be resolved unambiguously.

To avoid relying on type inference, it is possible to give explicit metamodel-qualified types to lambda variables, like this:

```
inTree.nodes.select(child : InTree_Node | child.parent = inNode)
```

4.7 Type Conversion

ReMoDeL concepts exist in a classification hierarchy. Variables are polymorphic (typed with a general concept, but may receive objects of more specific types of concept). Sometimes, it may be desired to recover the specific type of an object stored in a general variable. This can be done simply by giving the access operation the more specific result type. This will create a type-cast at runtime.

References or components may be redeclared with more specific types, in concepts that inherit from the one in which they were first declared. Similarly, operations may be redeclared with more specific result types (argument types may not be changed).

Collections may in general store elements of heterogeneous types, so long as each element's type is a subtype of the collection's declared element type. This collection may be filtered by a *select()* operation that returns a collection of a more specific type than the declared type, so long as all the selected elements are of this type (which is checked at runtime). This is the only kind of type casting that may be performed on collections.

4.8 Default Initialisation

The declared properties of a concept are subject to default initialisation. This is to allow rules to choose how many properties they initialise explicitly. The basic types are always initialised to a default value; and the collection types are always initialised to an empty collection of the declared element-type. Only references to other concepts are initialised to *null*, the empty reference.

If not specified otherwise, *Boolean* attributes are initialised to *false*; *Integer* and *Decimal* attributes are initialised to zero; *String* attributes are initialised to the empty *String*; and *List* and *Set* components are initialised respectively to the empty list or empty set. This avoids many problems with *null* values.

4.8 Compilation to Java

ReMoDeL metamodels are compiled to a Java package, containing one class for each concept in the metamodel. ReMoDeL transformations are compiled to a Java package, containing a class representing the transformation, with an executable main() method, which reads the source model(s) and outputs the target model. This is described fully in the companion technical report, *ReMoDeL Compiled: The Cross-Compilation of ReMoDeL to Java by Example* [7].

5. References

[1] J Greenfield, K Short, S Cook, S Kent and J Crupi. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, ISBN 0-471-20284-3 (Microsoft, 2004).

[2] Object Management Group: MDA - The Architecture of Choice for a Changing World (OMG, 2014). https://www.omg.org/mda/

[3] Eclipse Foundation. The Eclipse Modelling Framework (EMF) (Eclipse, 2004-2022). https://www.eclipse.org/modeling/emf/

[4] AJH Simons. ReMoDeL: Reusable Model Design Languages (University of Sheffield, 2022). http://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/

[5] T Mens and P van Gorp. A Taxonomy of Model Transformation, Electronic Notes in Theoretical Computer Science 152 (2006) 125–142.

[6] M Biehl. Literature Study on Model Transformations, Technical Report, Embedded Control Systems, Royal Institute of Technology, Stockholm (Stockholm: KTH, 2010).

[7] AJH Simons. ReMoDeL Compiled (rev. 2.1): The Cross-Compilation of ReMoDeL to Java by Example, Technical Report, 12 July, (University of Sheffield, 2022).