# ReMoDeL: a pure functional object-oriented concept language for models, metamodels and model transformation.

Anthony J H Simons[1][a],
[1]*School of Computer Science, University of Sheffield, Regent Court, 211 Portobello, Sheffield S1 4DP, UK.*
*a.j.simons@sheffield.ac.uk*

Keywords: Model-Driven Engineering, Models, Metamodels, Model Transformation, Pure Functional, Object-Oriented, Concepts, Relationships, ReMoDeL.

Abstract: Model-Driven Engineering (MDE) is a broad discipline concerned with curating all aspects of system design using models. Model-Driven Architecture (MDA) is a highly publicised approach focusing on the generation of software systems from models. However, MDA consists of a large collection of complex, interlocking standards, which together are difficult to master and have only partial implementations. This motivated us to devise a much simpler language and toolset for MDE. The result is ReMoDeL (Reusable Model Design Language), a pure functional object-oriented language for describing concepts and relationships. ReMoDeL supports the creation of metamodels, models and model transformations. It leverages skills already known to programmers, such as inheritance and pure functional mapping. It integrates with any standard Java IDE and cross-compiles to Java, although ReMoDeL is more succinct (by 4x). ReMoDeL's pure functional transformations are in principle amenable to formal proof by induction. Practically, it offers a convenient and fast way to prototype different metamodels and transformations. We are using ReMoDeL to develop alternatives to UML and MDA (with different models and abstraction levels), with promising results.

## 1 INTRODUCTION

Model Driven Engineering (MDE) has been an active field in Software Engineering since the early 2000s (Whittle, et al., 2014). The goal is to enable the management of all aspects of systems design using models, abstractions of different views of the system, with the aim of increasing clarity and productivity. MDE encompasses many kinds of model translation, correction, refactoring and improvement, and reverse engineering of models from systems, whereas the subfield of Model-Driven Development (MDD) focuses more narrowly on the generation of systems from models (Mens & van Gorp, 2006, Biehl, 2010).

### 1.1 Languages for MDE

The best known proposal for MDD is the Object Management Group's Model-Driven Architecture (OMG, 2014b), built on a large collection of standards, including the Unified Modeling Language (OMG, 2017), the Meta Object Facility (OMG, 2016a), Query-View Transformation (OMG, 2016b), the Object Constraint Language (OMG, 2014a), XML Metadata Interchange (OMG, 2015) and the Common Warehouse Metamodel (OMG, 2003). Whereas companies report using some of the OMG's standards, such as UML and bespoke code translators to speed up development (Whittle, et al., 2017), there have been few complete implementations of the QVT model transformation standard. *SmartQVT* (Alizon, et al., 2008) was a Java implementation of the QVT-Operational language, not maintained since 2013. *ModelMorf* (TCS, 2007) implemented bidirectional transformations in the QVT-Relational language, with pattern matching, but has not survived. Lano attributes this lack of traction to the complexity of the whole MDA project:

"The problem with QVT-R is that large QVT-R specifications are difficult to write, understand or debug, and tend to have poor quality/high technical debt (including the *Rel2Core* transformation that appears in the standard itself)" (Lano, 2022).

Other influential model transformation languages originally developed outside the OMG remit include:

[a] https://orcid.org/0000-0002-5925-7148

the *ATLAS Transformation Language* (ATL) from INRIA, Nantes (Jouault, et al., 2008), a hybrid language with declarative and imperative aspects, *Kermeta* from IRISA, Rennes (Drey, et al., 2010), an object-oriented programming language with support for imperative model transformations, the *Epsilon Transformation Language (ETL)* from the University of York (Kolovos, et al., 2008), a hybrid model-to-model transformation language with lazy, guarded and greedy rule scheduling. Languages worthy of merit include *UML-RSDS (Reactive Systems Design Support)* from Kings College, London (Lano, 2016; 2018), and *Aocl* (Batory & Altoyan, 2020), which support a declarative style of specification in OCL. Whereas *Aocl* extends OCL with mapping functions, UML-RSDS uses pure OCL, cleverly converted by a compiler into executable mapping transformations, and can generate code in Java, C, C++ and C#.

Many of these languages have since migrated onto the Eclipse platform, to benefit from the graphical drawing tools and the Eclipse Modeling Framework (EMF) (Eclipse, 2008; 2021). *ATL, Kermeta, ETL* and *UML-RSDS* (as *AgileUML*) have followed this route to seek wider adoption. However, even one of the most complete Meta-CASE tools ever produced, *XMF-Mosaic* by the UK startup Xactium (Clark, et al., 2008) failed to capture the lasting interest of industry. In hindsight, this was seen as due to language and tool designers "developing elegant tools for researchers, not pragmatic tools for engineers" (Clark and Muller, 2012; Whittle, et al, 2017).

## 1.2 Simple Language for MDE

Our own development of ReMoDeL sought to avoid creating similar barriers to adoption. We have created a simple, declarative language in which designers can rapidly express sets of related concepts in a metamodel. Metamodels are cross-compiled to Java packages, containing one class per concept, automating all the usual Java bookkeeping. A model is an in-memory graph of instances of these classes, which can be read from, and serialised as text, in a format reminiscent of JSON (but not identical).

A model transformation is expressed as a set of pure functional mapping rules from source to target. Each rule is *idempotent* (when applied multiple times to the same source element, it always maps to the same target element). This promotes a desirable divide-and-conquer strategy of expressing rules in terms of simpler rules, without concerns for rule ordering or multiple firings. A model transformation compiles to a Java class, which may either run as a main program, or may be chained together in a series of model transformations.

A project in ReMoDeL consists of a new Java project in Eclipse (or other IDE), which has the ReMoDeL library on its build-path. The project only needs a *home* package with a shell program to run the compiler, and thereafter all ReMoDeL files are placed in subfolders {*meta, model, rule*} of the main project folder. This is all that is needed to get started.

## 2. REMODEL SYNTAX

The best way to present ReMoDeL is by example, and for this we use a popular "Hello World" introductory example found in the model transformation literature, the mapping between alternative tree- and graph-like representations of the same tree structure.

## 2.1 InTree Metamodel

Figure 1 illustrates an in-tree, a kind of tree in which every node refers to its parent node directly. Such a structure may be represented as the following serialised model text.

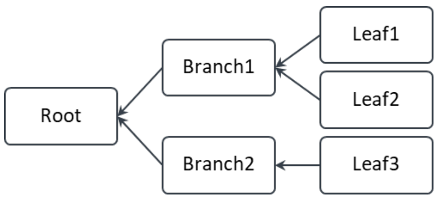

Figure 1: Model of an in-tree.

```
model tree1 : 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)
   ])
}
```

From this it should be clear that the *Tree t1* consists of a list of *Nodes n1-n6*, where most of these, apart from the root node, refer to their parent node. Every element of the model has a unique ID, such that they may refer to each other as desired. The type-declaration in the header *tree1:InTree*, states that the model *tree1* is an instance of the metamodel *InTree*.

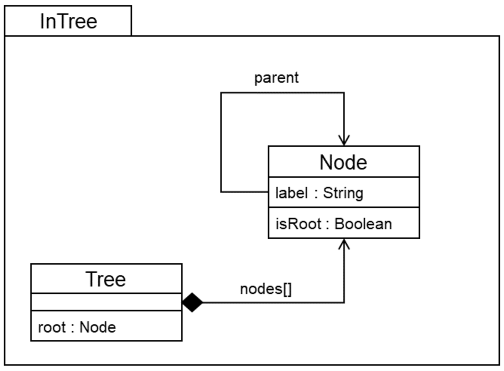

Figure 2: Metamodel for an InTree

Figure 2 illustrates the metamodel for an *InTree* using the graphical syntax adopted for ReMoDeL. This repurposes some UML notation (for the sake of familiarity), in which a metamodel is contained in a region reminiscent of a UML package, and concepts are drawn in a style reminiscent of a UML class. The same metamodel may also be expressed in the following textual format:

```
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)
      }
   }
}
```

From this it should be clear that the *InTree* metamodel consists of two concepts, *Node* and *Tree*. *Node* has an attribute *label* with a simple *String* type, and a weak reference to a *parent Node*. *Tree* owns a component list of *Nodes*; the list type is indicated by the square brackets following the *Node* type name.

Concepts may also have pure functional operations (parameters are optional), which return the value of their body expression. So, *Node* is able to determine if it is the root node, and *Tree* is able to filter its nodes to find the root node using *detect*, a higher-order filtering operation, whose lambda-expression tests each owned node in turn and returns the first found root node.

## 2.2 Graph Metamodel

Figure 3 illustrates an alternative graph representation for a tree-like structure, in which the vertices and edges are modelled explicitly as separate concepts. Such a structure may be represented in the following serialised model text.

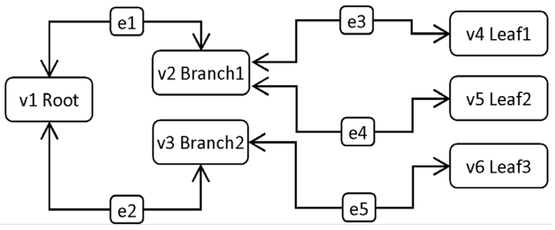

Figure 3: Model of a graph.

```
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),
 ])
}
```

From this, it should be clear that the *Graph g1* consists of a list of *Vertices v1-v6* and a list of *Edges e1-e5*. Each *Vertex* is labelled, and each *Edge* connects a given source and target *Vertex*.

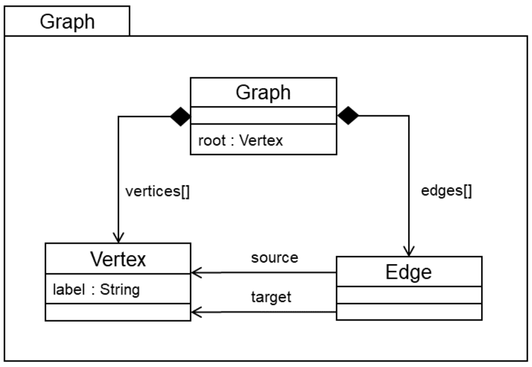

Figure 4: Metamodel for a Graph

Figure 4 illustrates the metamodel for a *Graph* using the ReMoDeL graphical notation. This defines the structure expressed above, which is obeyed by the model instance *graph1:Graph*. The same metamodel may also be expressed in textual format (*Graph* unambiguously names the metamodel and a concept):

```
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
   }
}
```

There are three concepts: *Graph*, which consists of lists of *Vertices* and *Edges*, *Vertex*, which has a simply-typed label attribute, and *Edge*, which has two references to its source and target *Vertex*.

The *Graph* operation to find the root vertex looks a little more involved than for the *InTree* case, but should be well understood by anyone familiar with pure functional programming: the *root* operation filters the vertices to detect a unique *Vertex* satisfying a given property, which is that there should not exist any *Edge*, whose source is that *Vertex*. Such a *Vertex* must be a root.

## 2.3 Model Transformation

Model transformations in ReMoDeL are provided as text files in the following format:

```
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)
   )
 }
}
```

The transformation is named *InTreeToGraph* and belongs to the transformation group *Trees*. The next lines introduce the source and target metamodels, saying that the transformation maps a *source:InTree* to a *target:Graph*. The transformation consists of three mapping rules: *inTreeToGraph* is the top-level rule (ordered first), which invokes subsidiary rules *inNodeToVertex* and *inNodeToEdge*.

The simplest rule *inNodeToVertex* takes an argument of the kind: *InTree_Node* and gives a result of the kind: *Graph_Vertex*. The concept-names are prefixed by their owning metamodel, since the source and target metamodels may sometimes contain concepts having the same name. The body of the rule creates a *Graph_Vertex*, whose label is initialised to the same value as that of the supplied *inNode*.

Similarly, the rule *inNodeToEdge* takes an argument of the kind *InTree_Node* and creates a result of the kind *Graph_Edge*. This rule invokes the previous rule to map the supplied *inNode* to its *source*, and the *parent* of this node to its *target*.

Finally, *inTreeToGraph* creates a *Graph*, whose *vertices* are obtained by mapping *inNodeToVertex* over the *inTree.nodes*, and whose *edges* are obtained by mapping *inNodeToEdge* over all but one of the *inTree.nodes*, the root node, which is excluded from the list by *without,* a pure functional list removal operation that returns a copy of the list without the specified element.

The higher-order mapping operation *collect* accepts a lambda-expression as its argument. This consists of a lambda-variable (here, *node*) separated by a vertical stroke from the lambda-body, which can be any expression including the lambda-variable. The effect of invoking *collect* on a list is to apply the lambda-expression to every *node* in the list, and collect a new list of the mapped results.

## 3. REMODEL SEMANTICS

ReMoDeL was designed to have a semantics based on pure functional programming semantics, for the sake of formal clarity. No operation, nor any rule, actually modifies any data structure destructively.

### 3.1 Properties of Transformations

A model transformation is a pure functional mapping from a source to a target model. The whole target is always created afresh by the transformation. Rules are completely declarative, stating exactly how to build the target in a compositional way using simpler rules. This brings certain desirable properties.

Firstly, no other part of the source or target model is modified by any mapping rule. This avoids hard problems found in languages with imperative updates to models, such as the hybrid languages ATL (Jouault, et al., 2008) and ETL (Kolovos, et al., 2008), in which it becomes important to control the order of rule-firing, so that target models are modified in some appropriate sequential order.

Secondly, rules are idempotent, mapping every element exactly once. Consider the number of times the rule *inNodeToVertex* is called on any given node. It is called once on every node by *inTreeToGraph* when creating the list of *Vertices*. It is called again by *inNodeToEdge* on all paired nodes connected by edges. During the whole model transformation, the *InNode n2* is given as an argument to *inNodeToVertex* on four separate occasions: once when being mapped to the *Vertex v2*, once when creating the *Edge e1*, and twice more when creating the *Edges e3* and *e4*. The idempotent property ensures that exactly one copy of the target *Vertex v2* is created, returning the same instance for all invocations.

Thirdly, models are constrained to be directed acyclic graphs (DAGs), to ensure that transformations always terminate. If a model were to contain cycles, then rules would call each other in infinite regress. But since it is sometimes profoundly useful to have back-references, especially to avoid passing the whole model as an extra argument to a rule, ReMoDeL allows any *component* to have a hidden back-reference called *owner*, which is set implicitly when the component is added to its master. Back-references are not mapped by transformations, and they are not serialised when a model is saved.

Fourthly, each transformation is unidirectional, describing a mapping in one direction only. This does not preclude defining an inverse transformation, to map the target back to the source. It is possible to write an inverse transformation *GraphToInTree*, which maps a graph-representation of a tree back to the original in-tree representation. However, the astute reader will have realised that a graph is a strictly more general kind of representation, since it may have more than one root node at the head of the DAG. Such a graph cannot be mapped (without loss) back to a tree. It is possible to define *partial* transformations in ReMoDeL.

While we have only shown an example of a *mapping* transformation, it is also possible to create *merging* transformations. These have more than one source metamodel and construct a target that blends information from both of the sources. An updating transformation is merely a *mapping* transformation that constructs a new instance of the target model. Both *mapping* and *merging* rules are idempotent, but for a *merging* rule, this is judged on the basis of a tuple of inputs from the source metamodels. Occasionally, it is desired to have a kind of rule that always constructs a new instance of the result. For this, ReMoDeL provides *function* rules, which map without idempotence.

Finally, for the more mathematically-inclined, ReMoDeL transformations have an interpretation within Category Theory as *homomorphisms* between general abstract datatype algebras. The individual mapping rules are *morphisms*, mappings between source and target elements of the algebras. This property is entailed by the fact that any kind of source or target metamodel, expressed in the concept-language, may ultimately be generalised as the "same kind" of abstract datatype algebra.

### 3.2 Properties of Metamodels

ReMoDeL was designed to be a conceptual modelling language, rather than a full programming language. A concept in ReMoDeL denotes any kind of modelled structural or behavioural entity that possesses a number of properties. The properties can only be drawn from the following kinds:

- an *attribute* – with a simply-typed value;
- a *reference* – referring to another concept (or list, or set) in the same metamodel;
- a *component* – a strong reference implying ownership of the related concept;
- an *operation* – a pure functional operation, to filter or access part of a concept.

Attributes have simple types taken from the set: {*Boolean, Character, Integer, Decimal, String*}. All references are directed, to preserve the DAG quality of models (see above). A *component* is a stronger kind of *reference*, meaning that the source concept owns the target concept. It also supports the implicit *owner* back-reference (see above). The intended difference in the semantics is that if an instance of a concept is deleted, it may simply forget its *references*, whereas its *components* should be deleted recursively in a cascading fashion. (This would support a future implementation of ReMoDeL in C++, which must manage the allocation of components explicitly).

ReMoDeL operations are without side-effects. They typically either query their owning concept to filter its stored properties for a subset of these, or they compute some kind of derived result. The modeller adds such query operations to a concept as needed, usually to make the process of writing a model transformation simpler.

ReMoDeL allows inheritance between concepts. A derived concept may *inherit* a base concept in its opening declaration (before the set of properties). The derived concept obtains all the properties of the base concept, but may override some by redefining them. A redefined reference or component may restrict the type of concept to which it refers. An operation may restrict its result type and may also replace its function body. This supports dynamic binding. ReMoDeL also allows inheritance between metamodels. A derived metamodel may *inherit* a base metamodel in its opening declaration (before the set of concepts). The semantics is similar to importing from a remote package, except that it is possible to override an inherited concept with a redefinition of that concept. Together, these features justify the acronym ReMoDeL (Reusable Model Design Language).

## 3.3 Properties of Expressions

ReMoDeL has an extensive expression language. It supports the *Boolean* constants: *false, true* and the operations *and, or* and *not*. It supports the empty reference *null*. It supports the six inequalities and a full range of arithmetic operators, including divide, modulo and exponent. It offers a ternary conditional *if – then – else* – which returns the value of one or other branch. It offers a range of standard *String* operations, which support searching, substring extraction, concatenation and case conversion, which is useful in rules that relate names and types of things.

As well as the basic types and concept types, ReMoDeL has pure functional list and set types. These are defined by appending [] or {} onto any type identifier, to denote respectively a list, or a set of that type. Any concept may use *asList* or *asSet* to convert it into a singleton of that type (when applied to collections, these may convert the collection-type). Lists and sets offer the common operations: *isEmpty, size, has, count, with and without*. Sets offer *union, intersection, difference* with the usual semantics and a *pick* operation for the axiom of choice, to select an arbitrary set element. Lists are ordered and offer *first, rest* and *append* with the usual semantics.

One of the most important and useful features of lists and sets is that they offer the following higher-order operations, which perform quantification, filtering, mapping and reduction on collections. Each of these has a lambda-expression as its argument:

- *exists(x | predicate(x))* – returns true if any element satisfies the predicate;
- *forall(x | predicate(x))* – returns true if all the elements satisfy the predicate;
- *detect(x | predicate(x))* – returns the first element satisfying the predicate;
- *select(x | predicate(x))* – returns those elements that satisfy the predicate
- *reject(x | predicate(x))* – returns those elements that fail to satisfy the predicate
- *collect(x | function(x))* – constructs a list (or set) of the result of applying the function to every element of the list (or set).
- *collate(x | function(x))* – appends a list (or unions a set) of the list- (or set-) results of applying the function to every element of the list (or set).
- *reduce(x, y | reduction(x, y))* – constructs a single value by combining all elements.

These operations are sensitive to the kind of collection on which they are invoked, and may return a collection of the same kind (whether a list, or a set). Filtering a heterogeneously typed collection may specify a more homogeneously typed result, which is useful, since the result's elements are automatically down-cast to the specific type, which may offer access to more specific operations.

The difference between the *collect* and *collate* operations is that *collect* expects the mapped function to return a single object, whereas *collate* expects the mapped function to return a list (or set), which must then be flattened. The *reduce* operation is a limited kind of left-fold reduction: if the collection is empty, it returns null; if the collection is a singleton, it returns that element; otherwise, it applies the binary reduction to combine multiple elements of the collection. This can be used to find the sum of a list, or the greatest element of a set, for example.

For further information about the expression language, the reader is invited to consult the report *ReMoDeL Explained* (Simons, 2023a). For further information about the cross-compilation model and how this preserves the semantics of the language, the reader is invited to consult the report *ReMoDeL Compiled* (Simons, 2023b).

## 4. REMODEL IN PRACTICE

ReMoDeL has been trialled in a number of student projects at the University of Sheffield. The learners are students on Computer Science undergraduate or postgraduate master's programmes. They undertake different kinds of model transformation problems in different conceptual domains.

### 4.1 Learning ReMoDeL

Undergraduates typically have a strong background in Java programming and one semester's exposure to functional programming in Haskell. Master's students may not have had functional programming experience. We find that the undergraduates adapt quickly to the pure functional style of transformation rules, although master's students also acquire this skill eventually. Once the idempotent property of mapping rules is understood, students soon adapt to the declarative divide-and-conquer style of mapping transformations.

A second acquired skill is in learning how to construct a suitable metamodel for the conceptual domain in question, especially how to factor out common features of different concepts in a concept hierarchy. This usually comes more easily to those with longer object-oriented programming experience, especially if they are familiar with refactoring classes in a class hierarchy.

Using ReMoDeL within a standard Java IDE (we have used Eclipse, IntelliJ IDEA and NetBeans) can present challenges for some who have never managed a Java project with an external library. Students learn about configuring the build-path of their project. Another skill is to keep the state of the IDE consistent. For example, Eclipse monitors the state of a Java project using metadata on its files and dependencies, which are updated only by the IDE's tools. Since ReMoDeL may independently create files and whole Java packages in a separate thread, it is possible for the metadata to become stale. In this case, the user must remember to refresh the project frequently.

### 4.2 Problem Domains

Another test of ReMoDeL is in seeing what kinds of problem domains it can be applied to with success. One of the first full-strength problems on which we tested its power to conceptualise domain models and capture complex chains of transformations was the UML to SQL transformation problem. This involved creating separate metamodels for the *UML Class Diagram*, the traditional *Entity-Relationship Model*, a normalised *Object Dependency Graph*, and an *SQL Database Model* capturing data definitions.

We developed a chain of transformations that could start either with UML, or a traditional ERM. *UMLtoERM* converted a class diagram into an ERM, in which UML's special semantic relationships generalisation, aggregation and composition, were mapped with associations to general relationships. *ERMtoNorm* normalised the ERM to 3NF, merging all entities connected one-to-one, splitting many-to-many relationships by introducing linker entities, and redirecting any affected relationships. *NormToEDG* converted all edges into directed, named references from the dependent to the master type. *EDGtoSQL* mapped all object types to tables, all attributes to columns, and all references to additional renamed columns, constructing primary and foreign keys that grouped these appropriately. For a complete account of this project, the reader is invited to refer to the technical reports (Simons, 2022a; 2022b).

### 4.3 Code Generation

To finish off the transformation chain, we wrote a simple code generator, to output SQL Data Definition Language from the final *SQL Database* model. This was achieved using a hand-written Java code generator class that followed the *Visitor Design Pattern* (Gamma, et al., 1995). An abstract *Visitor* was defined in the *SQL Database* metamodel, which compiled to a Java class. The hand-written code generator was a subclass of this *Visitor*, overriding the trivial method signatures. In this way, the metamodel could be recompiled without overwriting the hand-written generator methods.

## 5 CONCLUSIONS

We have succeeded in designing a fresh model transformation language, with a succinct syntax and clean semantics, and with a small runtime library that integrates easily with existing Java IDEs. Our initial experiments have shown that new users adapt rapidly to the language and compiler tools.

What is pleasing is that their focus of attention shifts to solving the real conceptual modelling and model transformation problems, rather than having to spend all their time understanding and fixing issues with the transformation architecture (Lano, 2022) or tools (Whittle, et al., 2017). A similar goal motivates the designers of DSL tools, such as MPS (JetBrains, 2024). But ReMoDeL is a general-purpose, rather than a domain-specific, modelling language.

Our future work arises out of different student projects, in which we have been able to explore new alternatives to UML, including a *Task Model* of clustered tasks and actors which are transformed into the *State Model* of the target system's GUI with authorisation (a transformation of system behaviour), or a simple *Impact Model* showing the CRUD effects of *Tasks* upon *Objects*, offering completeness checks on object lifecycles. These serve a good basis for cross-checking and refinement; we will extend and develop this family of models, which we tentatively name µML (the micro-Modelling Language).

## REFERENCES

Alizon, F., Belaunde, M., DuPré, G., Nicolas, B., Poivre, S. and Simonin, J. (2008). Les modèles dans l'action à France Télécom avec SmartQVT. *Ingénierie Dirigée par les Modèles, Congrès Journées Neptune* 5, Paris. *Génie Logiciel* 85, 35-42.

Batory, D. and Altoyan, N. (2020). Aocl : A pure-Java constraint and transformation language for MDE. *Proc. 8th Int. Conf. MODELSWARD*, 319-327.

Biehl, M. (2010). *Literature study on model transformations, Technical Report*, Embedded Control Systems, Royal Institute of Technology, Stockholm (Stockholm: KTH).

Clark, A., Sammut, P. and Willans, J. (2008). *Applied Metamodelling – a Foundation for Language-Driven Development*, 2nd ed., (Sheffield: Ceteva). Available: https://repository.mdx.ac.uk/item/82x67.

Clark, T., Muller, P.A. (2012). Exploiting model driven technology: a tale of two startups. *Software Systems Modeling* 11(4), 481–493.

Drey, Z., Faucher, C., Fleurey, F., Mahé V. and Vojtisek, D. (2010). *KerMeta Language Reference Manual*, November (Rennes: IRISA).

Eclipse Foundation (2008). *EMF: Eclipse Modelling Framework*, (Boston: Addison-Wesley). Available online: https://eclipse.dev/modeling/emf/

Eclipse Foundation (2021). *org.eclipse.emf.ecore Java package documentation*. https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software* (Boston: Addison-Wesley).

Jouault, F., Allilaire, F., Bézivin, J. and Kurtev, I. (2008). ATL: a model transformation tool, *Science of Computer Programming*, 72, 31–39.

JetBrains (2024). *Meta Programming System*. Available online: https://www.jetbrains.com/mps/

Kolovos, D., Paige, R. and Polack, F. (2008), The epsilon transformation language, *Theory and Practice of Model Transformations, Lecture Notes in Computer Science*, 5063, 46–60.

Lano, K. (2016), *Agile Model-Based Development using UML-RSDS* (Boca Raton: CRC Press).

Lano, K. (2018). *The UML-RSDS Manual, Technical Report*, Kings College, London. https://nms.kcl.ac.uk/kevin.lano/uml2web/umlrsds.pdf

Lano, K. (2022). Comment on Laurie Tratt's blog post: *UML: my part in its downfall*. Posted 10 March 2022. https://tratt.net/laurie/blog/2022/uml_my_part_in_its_downfall.html#comment-MSgn5NTaN8yM.

Mens, T. and van Gorp, P. (2006). A taxonomy of model transformations. *Electronic Notes in Theoretical Computer Science, 152* (Amsterdam: Elsevier), 125-142.

OMG (2003). *CWM – Common Warehouse Metamodel*, March. https://www.omg.org/spec/CWM/.

OMG (2014). *OCL – Object Constraint Language*, February. https://www.omg.org/spec/OCL/.

OMG (2014). *MDA – The Architecture of Choice for a Changing World*, June. https://www.omg.org/mda/.

OMG (2015). *XMI – XML Metadata Interchange*, June. https://www.omg.org/spec/XMI/.

OMG (2016). *MOF – Meta Object Facility*, October https://www.omg.org/spec/MOF/.

OMG (2016). *QVT – MOF Query/View/Transformation*, June. https://www.omg.org/spec/QVT/.

OMG (2017). *The Unified Modeling Language*, December. https://www.omg.org/spec/UML/.

Simons, A.J.H. (2022). ReMoDeL Data Refinement: data transformations in ReMoDeL, Part 1. *Technical Report*, 25 July, University of Sheffield. https://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/current/ReMoDeL_Data_Refinement_Part1.pdf

Simons, A.J.H. (2022). ReMoDeL Data Refinement: data transformations in ReMoDeL, Part 2. *Technical Report*, 25 July, University of Sheffield. https://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/current/ReMoDeL_Data_Refinement_Part2.pdf

Simons, A.J.H. (2023). ReMoDeL explained: an introduction to ReMoDeL by example. *Technical Report*, 25 January, University of Sheffield. https://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/current/ReMoDeL_Explained_25Jan23.pdf

Simons, A.J.H. (2023). ReMoDeL compiled: the cross-compilation of ReMoDeL to Java by example. *Technical Report*, 25 January, University of Sheffield. https://staffwww.dcs.shef.ac.uk/people/A.Simons/remodel/current/ReMoDeL_Compiled_25Jan23.pdf

Tata Consultancy Services (2007). *ModelMorf, a model transformer*. Now-defunct product website, no longer available.

Whittle, J., Hutchinson, J. and Rouncefield, M. (2014). The state of practice in Model-Driven Engineering. *IEEE Software, May-June* (Washington: IEEE Computer Society), 79-85.

Whittle, J., Hutchinson, J., Rouncefield, M., Burden, H. and Heldal, R. (2017). A taxonomy of tool-related issues affecting the adoption of model-driven engineering, *Software Systems Modeling*, 16 (2), 313-331.