Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Static Cyclic Scheduling for QVTr-like Declarative Transformations.

Edward D. Willink[a]

a. Willink Transformations Ltd., Reading, UK,

Abstract

THIS PAPER WAS WITHDRAWN BECAUSE SECTION 3.3.2 WAS WOOLLY AND ONCE FORMALIZED IT SEEMED DOUBTFUL THAT PARTITIONING WORKS AS CLAIMED. SURELY THE LOOP REMAINS? IT JUST SIMPLIFIES TO EASE SPECULATION / SPECIAL CYCLE TREATMENT.

ALSO REALLY NEED SOME RESULTS OF THE EXAMPLE.

A fully declarative transformation language such as QVT Relations provides a very powerful exposition of a model transformation. This avoids many hazards and inconveniences for the programmer, but provides significant challenges to the toolsmith. Minimally the declarative exposition must be supported and ideally its opportunities should be exploited. A naive execution may just search aimlessly for work that can progress and hope to eventually complete. A smarter tool may analyze the exposition carefully to sequence the necessary work effectively. Unfortunately this analysis is bedevilled by a variety of cyclic problems. We identify these problems and provide solutions so that an efficient almost completely static schedule can be used for execution.

Keywords QVTr, QVT Relations, Dependency Analysis, Declarative Model Transformation, Cyclic Scheduling

1 Introduction

The QVT specification [Obj08] introduced one imperative language, QVTo, and two declarative languages, QVTc and QVTr, as the solution to the early enthusiasm for model to model transformation. Only QVTo has a flourishing implementation. QVTc was never implemented despite the 'c' suggesting it is a core that 'o' and 'r' extend. The two QVTr implementations have not prospered.

The Eclipse QVTd project [Ecld] has provided QVTc and QVTr editors for many years, but it was only in 2016 that preliminary execution functionality was available. Since then the demands of a strongly analyzed implementation have exposed many limitations of the current QVT specification [Obj16], so that current work on Eclipse QVTd necessitates some remedial language design.

Topics such as collection matching, relation overriding, trace synthesis, null matching, multiple roots, when/where semantics, endogeneous/incremental matching have had to be addressed.

In Section 2, we provide the foundations for this paper, first defining the principles of a declarative transformation, then providing our running example. This is followed by the analysis approach that supports the treatment of cycles in Section 3. In Section 4 we consider related work and finally in Section 5 we conclude.

Throughout this paper we use QVTc's term 'mapping' for what is variously called a 'relation' or 'rule' by other approaches. We use the terms 'instance', 'slot' and 'mapping invocation' for the run-time instantiation of their compile-time 'class', 'property' and 'mapping' counterparts. We use 'partition' when emphasizing that we are considering a sub-'mapping' resulting from the break up of a 'mapping'.

2 Foundations

2.1 The Truth

When remedying the language, it is helpful to have some basic principles against which alternative candidate sub-specifications can be assessed. Unfortunately there are no such clear principles in the QVT specification.

The Eclipse QVTd project works with the following principles. A declarative transformation defines a set of sub-truths that hold once the transformation execution has completed. The transformation does not specify how that truth is determined.

Each sub-truth is formulated using an OCL expression [Obj14] as part of a constraint. The transformation language just provides helpful facilities for structuring large numbers of sub-truths as the overall truth.

Once we have captured the execution as an overall truth, we find that re-purposing the courtroom oath 'the truth, the whole truth, and nothing but the truth' as a declarative transformation principle provides the arbitration necessary to select between alternative candidate semantics.

2.2 Example

We will demonstrate the challenges in providing near-optimal code from a declarative exposition by using a very simple but surprisingly awkward example; an isomorphic transformation from a set of `A` instances to a corresponding set of `B` instances. Each instance has a reference to another.

Figure 1 – Example (single class) MMa, MMb Metamodels.

The Class Diagrams in Figure 1 show the 'metamodel' for `A` on the left and the corresponding 'metamodel' for `B` on the right.

The Instance Diagrams in Figure 2 show our test input and output models. Two related `A` instances on the left named `a0` and `a1`, and two similarly related instances of `B` on the right named `b0` and `b1`. The dash-dotted lines indicate the magic performed by the transformation. For this simple, we can synthesize the super-optimal Java-like solution shown in Listing 1.

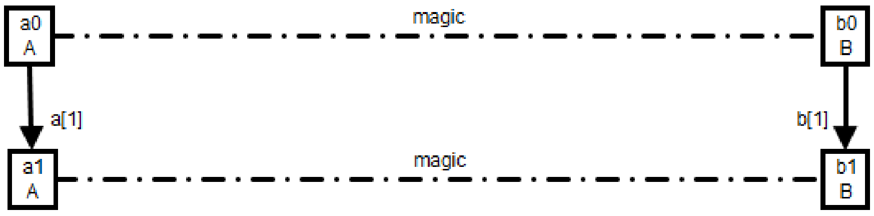


Figure 2 – Example Instances.

```
// load a0, a1
B b0 = new B();
B b1 = new B();
b0.b = b1;
b1.b = b0;
// save b0, b1
```

Listing 1 – Super-Optimal Example Conversion

Ignoring the load/save activities necessary to wrap the in-memory transformation as a practical application, we see two new statements and two assignments. Each new statement creates an output instance of the `B` class. Each assignment populates the slot corresponding to the `B::b` property with a resolved class reference. These necessary actions are performed without any additional control flow overhead.

The super-optimal solution demonstrates that the outputs can be created by interleaving new statements that create new instances, and assignment statements that assign the results of an OCL query to a slot of an instance.

The super-optimal solution, while correct, is of limited use since we need to generate a different super-optimal solution for each source model. We therefore prefer a less efficient solution that abstracts away from the run-time instances and slots to their compile-time classes and properties. This incurs some control flow overheads to accommodate a variety of input models as shown in Listing 2.

```
// load allA
for (A anA : allA) {
    B aB = new B();
    aB.b = lookup(anA.a);
}
// save allB
```

Listing 2 – Idealized Example Conversion Loop

The 'for' loop over all source A elements is an obvious control overhead. The 'lookup' operation is the distinguishing characteristic of model transformations; the ability of one part of a transformation to query conversions made by another part.

2.2.1 Java-like Example

We will look at M2M solutions shortly. For now consider the Bad-Java-like solution shown in Listing 3.

```
Map<A,B> a2b = new Map<A,B>();
// load a2b.keySet()
```

```
for (A anA : a2b.keySet()) {
    B aB = new B();
    a2b.put(anA, aB);
    aB.b = a2b.get(anA.a);
}
// save a2b.values()
```

Listing 3 – Bad-Java-like Example Conversion Loop

A Map is used to correlate each input `A` instance and its corresponding output `B` instance, the keys of the Map are populated by the load. The `lookup` is replaced by a `put` of each created value followed by a `get` of the referenced value.

Unfortunately, the Bad-Java-like Listing 3 code only half-works, since perhaps half of the `get` accesses may occur before the requisite `put`. As is often the case with non-trivial conversions, it is necessary to perform the graph conversion in two passes. A first pass performs a tree traversal to establish the output equivalents and a second pass fills in the graph cross references. This is shown in Listing 4.

```
Map<A,B> a2b = new Map<A,B>();
// load a2b.keySet()
for (A anA : a2b.keySet()) { // tree-pass
    B aB = new B();
    a2b.put(anA, aB);
}
for (A anA : a2b.keySet()) { // graph-pass
    B aB = a2b.get(anA);
    aB.b = a2b.get(anA.a);
}
// save a2b.values()
```

Listing 4 – Ok-Java-like Example Conversion Loop

We can see that the memory overhead for this simple conversion requires a Map of input to output instances, and the control overhead requires two loops, and three Map accesses per element.

2.2.2 Model-to-Model Transformation Solutions

The `lookup` capability is so fundamental to M2M, that all M2M languages provide a dedicated solution so that the Map is maintained internally.

In ATL [Ecla], there is a 'resolveTemp' built-in function to interrogate the Map. In Epsilon [Eclb] it's 'equiv' and in QVTo it's 'resolve'. The behaviour of these magic built-in functions is often poorly specified and programmers discover that they are using the Bad-Java-like solution the hard way. Sometimes the magic function works, sometimes it doesn't. The simplest way to get the transformation to work is to split the conversion into two passes, just like the Ok-Java-like solution; the first pass creates the output elements and populates the hidden Map. The second pass uses the magic built-in function after the hidden Map has been fully populated. This obviously requires the programmer to understand the hazard, to change the exposition to use two passes and to use some form of sequencing statement for the passes.

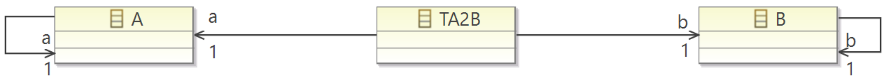


Figure 3 – Example (single class) MMa, Trace, MMb Metamodels.

2.2.3 QVTc Tracing

In QVTc, the magic built-in function is reified by the trace class provided by the additional trace metamodel shown in the middle of Figure 3.

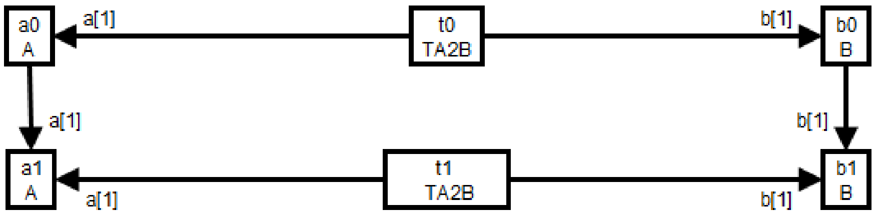


Figure 4 – Example Instances with Traces.

Figure 4 shows how the magic mapping is replaced by a `TA2B` instance with `TA2B::a` and `TA2B::b` slots locating the source and target instances with precisely `[1]` multiplicity. In terms of memory usage, this is almost identical to the `Map::Entry::key` and `Map::Entry::value` fields that lie behind the Java Map implementation. However with a modeled magic function, the magic can be formalized by OCL navigation.

```
 1  // load allA
 2  for (A anA : allA) {
 3      B aB = new B();
 4      TA2B a2b = new TA2B();
 5      a2b.a = anA;
 6      a2b.b = aB;
 7  }
 8  for (A anA : allA) {
 9      anA.TA2B.b = anA.a.TA2B.b;
10  }
11  // save allB
```

Listing 5 – Java-like Example Conversion Loops using the QVTc trace

In Listing 5 we show the use of OCL navigations, but continue to present it in a Java-like style to ease comparison and avoid the unnecessary confusion of a perhaps unfamiliar QVTc syntax.

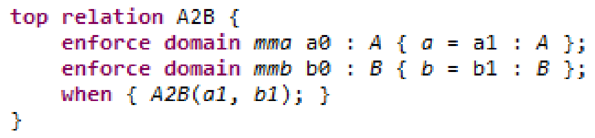
This exploits one of the fundamental characteristics of OCL; all navigations may be traversed in either direction. Navigation directions without explicit names are given implicit names based on the target class names.

Thus the (forward) assignment of `anA` to `a2b.a`, at line 5, also makes the (reverse) assignment of `a2b` to `anA.TA2B` since `A::TA2B` is the implicit opposite role name of `TA2B::a`.

At line 9, the left hand side realizes the top traversal from `a0` to `t0` to `b0` for `lookup(anA)` as `anA.TA2B.b`. Similarly on the right hand side, the bottom traversal `a1` to `t1` to `b1` for `lookup(anA.a)` is reified as `anA.a.TA2B.b`.

2.2.4 QVTr Tracing

Behind the scenes, QVTr re-uses the QVTc trace. The QVTr concrete syntax makes the lookup available in a bidirectional fashion as shown in Figure 5.



```
top relation A2B {
    enforce domain mma a0 : A { a = a1 : A };
    enforce domain mmb b0 : B { b = b1 : B };
    when { A2B(a1, b1); }
}
```

Figure 5 – Example Mapping expressed as a QVTr Relation.

Each `domain`, one for the `mma` and the other for the `mmb` directions, defines the instance pattern shown at the left and right of Figure 4. The the `a0`/`b0` instance at the root of each pattern has a related `a`/`b` slot for an `a1`/`b1` instance.

The overall `relation A2B` relates the `a0`/`b0` root of each pattern and is reified by the `t0` trace shown at the top of Figure 4.

The additional `when` clause requires that `a1` and `b1` are related by `A2B(a1, b1)`. It is reified by the `t1` trace shown at the bottom of Figure 4.

2.3 Timing Hazards

We have seen how in the Bad-Java-like solution the careless use of a Map can lead to malfunctions. This cuts right to the heart of the difference between a fully declarative program exposition and a traditional imperative / functional approach.

2.3.1 Function Hazards

A function `F` has overt/explicit `Inputs(F)` and `Outputs(F)` which a programmer exploits to build a program whose execution can be analyzed by examining the call tree of the functions. However a function also has implicit inputs and outputs in the form of `Gets(F)` and `Puts(F)` for the memory accesses that occur within the function. We can ignore the further side-effects of the form `News(F)` since in practice each new element is put somewhere for use by a subsequent get. It is sufficient to worry about ensuring that each element of `Gets(F1)` is after the appropriate `Puts(F2)`.

More formally, using:

- `Gets(F)` the read accesses of function `F` / mapping `M`
- `Puts(F)` the write accesses of function `F` / mapping `M`
- `Element(A)` the element referenced by an access `A`
- `SSA(A)` the ‘time’ at which an access `A` occurs
- `SSA(E)` the ‘time’ at which the assignment to `E` occurs
- `Ready(A)` whether an access `A` occurs after its assignment
- `Producers(E)` the inverse of `Puts(F)`
- `Consumers(E)` the inverse of `Gets(F)`

We assume a Static Single Assignment formulation of the code. The ‘time’ returned by the `SSA(.)` function has sufficient granularity to enable its values to determine whether one 'time' precedes another. The granularity might be wall-clock time on a single processor or pass numbers on a multi-processor.

The `Ready` helper is therefore true when the 'time' of the access follows the 'time' of the assignment to the accessed element.

$$Ready(A) = SSA(A) > SSA(Element(A)) \tag{1}$$

To avoid malfunction we require that all the gets of all functions are ready.

$$\forall f \ : \ F \ \ \forall e : Gets(f) \mid Ready(e) \tag{2}$$

Conventional languages provide no compile-time or run-time support for `Ready` to ensure that the above is satisfied and so in a large program, the integrity of the design is entirely down to the diligence and skill of the programmer.

In more complicated programs there may be a selective manual reification of `Ready` through the use of `Map::containsKey` or just a `null`/non-`null` variable, However, it remains a manual programming responsibility to ensure that the selective `Ready` disciplines are appropriate and respected.

Sadly, since programmers are human, timing violations occur resulting in program malfunctions and opportunities for exciting debugging sessions. Program maintenance is a particularly perilous undertaking since the maintaining programmers may fail to understand the informal disciplines that the original programmers used to keep problems under control.

2.3.2 Mapping Hazards

A mapping `M` has a similar structuring using `Inputs(M)` and `Outputs(M)`, but for a fully declarative transformation there is no need for a manual guarantee that all `Gets(M1)` follow the appropriate `Puts(M2)`. Rather, every mapping invocation must check that each `Gets(M1)` does follow the appropriate `Puts(M2)`. If the check fails, the mapping invocation must be deferred for retry until the `Puts(M2)` has occurred.

Run-time support for the `Ready` helper is therefore necessary unless the tooling can discover an execution order that guarantees that everything is always ready in time. This is sometimes possible, but certainly not always. In practice the tooling can only look to finding a good execution order that minimizes the mappings/accesses for which the overheads of run-time `Ready` support are necessary.

2.4 Analysis

Neither QVTr nor QVTc Abstract Syntax models are convenient for detailed analysis and so the Eclipse QVTd project has developed a lower level QVTs graphical representation and visualization that is suitable for analysis and schedule preparation. The visualization of the patterns underlying a mapping has many similarities to an instance diagram and consequently a Triple Graph Grammar [LAS14]. However whereas a TGG is a manually developed program, QVTs diagrams are an auto-generated[1] visualization of a declarative model transformation.

[1]The QVTs diagrams in this paper have auto-generated content but manually enhanced layout.

2.4.1 QVTs Tracing

The QVTs visualization of our example is shown in Figure 6. It is a purely declarative exposition of the state of each mapping invocation after the transformation completes. All relation semantics are reified by trace navigation.

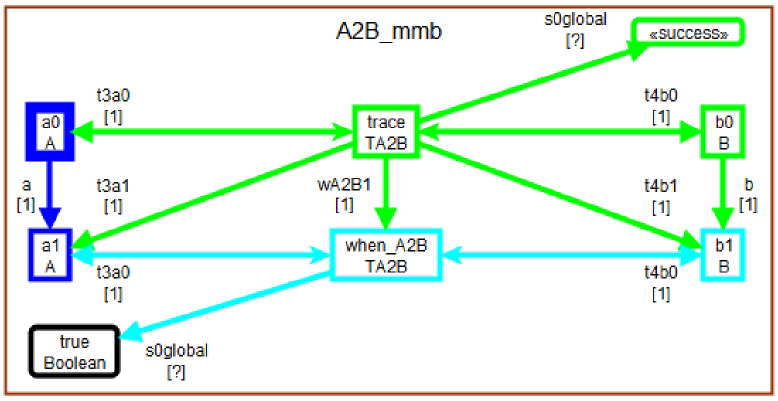


Figure 6 – Example Mapping using QVTs.

Contrasting Figure 6 with Figure 4 the most obvious difference is the (auto-generated) coloring that animates the chosen directional execution towards `mmb`.

Colors The colors indicate when the relevant elements are valid.

- Black elements are all-time constants.
- Blue (loaded) elements are available after the input model is loaded.
- Green (produced) elements are created by successful execution of the mapping.
- Cyan (consumed) elements must be available before execution completes.

Invocation Status Two additional nodes at bottom left and top right reify the three-valued success, failure or not-ready status of an invocation using the `true`, `false` or `null` values of the `s0global` slot.

The upper central row shows the green creation of `trace` and `b0` instances corresponding to the incoming `a0` instance. The `t3a0` and `t4b0` slots are populated so that the `trace` instance captures the invocation. The special value ≪`success`≫ is assigned to the status. If the invocation succeeds all green elements are created/assigned and the success status is `true`. If the the invocation fails, only green trace elements are created/assigned and the status is `false`.

The lower central row shows a topographically similar trace named `when_A2B` from the `a1` to `b1` instance. The coloring is however cyan indicating that the elements are looked-up / re-used from elsewhere. The additional `s0global` status constant at bottom left is a guard condition asserting that the referenced invocation completed and had a successful status.

The use of `t3a1`, `wA2B1` and `t4b1` slots becomes apparent in Figure 8.

Nodes and Shapes Each node identifies a pattern variable bound to a value or an instance by a successful match. The variable may have a name (above), and a type (below) that constrains possible matches.

Each rectangular node instantiates a metamodel `Class`, such as `A`, as a named-variable in the pattern.

Each rounded node instantiates a metamodel `Datatype`, such as `Boolean`, with a value in the pattern.

(Not shown in this example: Ellipses support `Iteration` and `Operation` calls.)

Edges and Arrows Each edge imposes further constraints on the permutations of model instances that can match the pattern.

Each navigation edge instantiates a metamodel `Property` as a constraint on the relationship between its two end nodes. At least one of the role names and multiplicities annotates the edge.

Whereas, in the Figure 4 instance diagram, the arrows show UML navigability, the arrows in Figure 6 show whether the target is related to the source by something-to-zero/one multiplicity. Triangular arrow heads show the easily navigable paths declared in the UML metamodel. Chevron arrow heads show the potentially expensive paths provided by OCL opposites. No-arrow shows a path for which there may be multiple target instances per source instance/value.

(Not shown in this example: Argument edges support Iteration and Operation calls.)

Borders The navigation arrows identify nodes that can uniquely reached from other nodes. The closure of this relationship identifies a minimal set of nodes from which all nodes can be reached uniquely. We call this minimal set the head nodes and use a thicker border for them. In this example, a single node is sufficient, the `a0` node. Therefore given an instance of `A`, all other elements of a successfully matched pattern can be resolved by simple navigation. A practical schedule can loop over all `A` instances and invoke the mapping binding the head to each instance in turn. It is not necessary to perform a mindless two-dimensional search of all possible matches for the two `a0` and `a1` pattern variables.

Occasionally two or more head nodes are required and two or more dimensional instance permutations are then necessary.

2.4.2 Dependency Analysis

The auto-generated QVTs decorations show the fruits of some simple multiplicity-based analysis, the multiplicity variously coming from the metamodel, transformation declarations and fundamental transformation semantics. The cyan and green coloring shows the analyzed consumer/producer behaviors.

Clearly green nodes are produced/created by some mapping and then consumed as cyan nodes by other mappings. These nodes are instances of model elements and may be passed as arguments to functions or mappings or stored in variables or collections.

Less obviously, green edges are produced/assigned by some mapping and consumed as cyan edges by other mappings. These edges are instances of model slots that typically lack a first class run-time representation. They cannot be passed around; rather the instance at one end can be passed together with an ‘instruction’ as to which instance-slot to access. It is therefore tempting to simplify the edge analysis to re-use the source node analysis with a minor addition for the slot ‘instruction’.

However this simplification leads to a serious loss of precision and every loss of precision impairs the ability to discover an efficient schedule. It is often the case that in a relationship such as UML’s `Element::ownedElement`, one or both ends of the relationship is declared in the transformation to be of a more specific type than the `Element` type declared in the metamodel. Without the extra precision we may find that any of perhaps ten possible producers of `Element::ownedElement` can be consumed by any of perhaps ten possible consumers requiring a substantial 10*10 run-time permutation of the options. With the extra precision, we may find that each derived producer of `Element::ownedElement` can be consumed by any of only one or two possible consumers requiring no permutation at all or perhaps just a much reduced two or three way run-time permutation.

We therefore analyze each edge production as a triple {`production-source-type`, `property`, `production-target-type`}. Subsequently determination of the producers for a consumption of a corresponding triple {`consumption-source-type`, `property`, `consumption-target-type`} must consider all possible run-time matches rather than compile-time conformances. We can only reject a match when `production-type` does not conform to `consumption-type` and vice-versa for either source and target types. This either way conformance is not very discriminating, but it is sufficient to distinguish consistent use of derived types.

2.4.3 Operation Analysis

A conventional operation execution is unable to fail and retry when a required slot is not ready. This conflicts with the retry-if-not-ready behavior of the declarative transformation run-time. We must find a solution that ensures that operations execute correctly.

Banning operations is clearly unacceptable and re-implementing operation execution to support retries is very unattractive. We can however extend our dependency analysis to include deep analysis of operation invocations so that the invoking mapping checks the readiness on behalf of the called operation tree. This is tractable since we are dealing with OCL operations whose behavior is amenable to analysis and whose execution is side-effect free.

However a precise deep analysis of every polymorphic operation call tree is not always possible, so we must fall back on a slightly pessimistic analysis of every possible type usage and property access. Improved accuracy of polymorphic operation analysis is obtained by specializing the analysis to the actual types provided by the caller rather than the possible types supported by the operations.

2.4.4 Endogenous Analysis

An endogenous transformation uses the same types for input and output models. Consequently a class/property-based analysis will find that all the early ‘input’ consumptions are in conflict with ‘output’ productions. The dependency analysis concludes that just about everything depends on everything. This difficulty vanishes if each class/property is qualified by the domain direction that it contributes to.

3 Cycles

The naivest declarative execution repeatedly permutes all possible instances against all possible mappings until sufficient progress has been made to constitute completion.

In this paper we seek to discover as close as possible to a fully static schedule in which compile-time analyses determine an appropriate execution order for the mappings so that the run-time iterates over the instances, invokes only relevant mappings exactly once and completely avoids all overheads from maintaining the not-ready status of slots and premature mapping invocations. The auto-discovered static schedule does just what an omniscient manual programmer could have programmed. Of course for non-trivial mappings a real rather than omniscient programmer may accidentally invoke helpers redundantly and more seriously may fail to understand the required execution order and so may produce a buggy program.

3.1 Permutation Cycles

Each naive permutation attempt results in a mapping invocation that may succeed and make a small quantum of progress towards the final result.

The simplest approach to reducing the number of fruitless permutations is to recognise that each mapping has a type signature that restricts the instances that can possibly conform.

A static compile-time analysis of the transformation can identify all the types that contribute to mapping signatures. At load-time, analysis of the input models can distribute the instances to type-specific bins so that during the subsequent execution only type-conformant instances are permuted. Inheritance complicates the picture slightly requiring either that some instances to be distributed to multiple bins or that multiple bins are used for some permutations.

For transformations with many mappings and for metamodels with many types, avoiding the irrelevant permutations may easily lead to an order of magnitude speed-up.

For non-trivial mappings the reduced search space dimensionality through accurate identification of the head node(s) is potentially the most significant optimization since it may give a speed up from cubic or quadratic to linear.

3.1.1 Discriminants

The type-based distribution of instances at load-time is easy for strongly-typed metamodels with access to the type via `eClass()` in Ecore or `xsi:type` in XMI.

Some applications have very weak metamodels with very few types making the simple type-based separation ineffective. However all is not lost since even weakly typed models often have a type discriminant such as a `kind` string. Static analysis of the transformation can first identify candidate discriminants as the attributes that have constant assignments at the time of construction/production. Further examination of the attributes that have constant predicates at the time of consumption reveals which candidates usefully correlate a subset of all possible productions with a subset of all possible consumptions.

The use of discriminants is not limited to weakly-typed metamodels. A derived attribute such as container-is-null may usefully discriminate the root from a non-root element in a tree. This avoids permuting non-root elements to a root-only mapping and vice-versa.

3.1.2 Open Metamodels

Many of the analyses and resulting optimisations in this paper rely on a complete metamodel analysis supporting a strong conclusion that type A either can or cannot conform to type B.

This analysis is unfortunately not safe if the load-time input instances may conform to an extension of the compile-time metamodel. For example, a transformation based on the UML metamodel may optimize to exploit the guarantee that a `Class` instance cannot conform to the `Package` type. Unfortunately, if the transformation is applied to a QVT model whose `Transformation` class extends both `Class` and `Package`. A (derived) `Class` instance may indeed be a (derived) `Package` instance. The optimized transformation may well malfunction.

Examination of the QVT metamodel reveals that the problematic multiple inheritance is a quite disgraceful pragmatism that is easily removed, and indeed is removed in the Eclipse QVTd implementation. We therefore ignore the problems of open metamodels. If a user really requires dirty derived metamodels, the user should arrange to make these dirty metamodels visible at compile time.

3.2 Not-Ready Cycles

Whereas a traditional program malfunctions if some required input slot has not been computed, a declarative transformation fails but arranges to retry until the required slot has been computed. This imposes two undesirable run-time overheads. Firstly each failed mapping invocation must have an ability to be stored for a retry. And each required input slot must have a has-been-computed / not-ready status.

Each retry incurs additional overheads. These can be reduced by only retrying once the cause of the failure has been resolved. Since the failure is associated with a not-ready consumption of a slot, that slot can maintain a queue of mapping invocations waiting to retry once that slot is computed / produced. Correct correlation of production and consumption can be managed automatically by the declarative execution run-time. A corresponding manually maintained correlation in a traditional program is almost unthinkable.

Retrying only when there is a prospect of progress, reduces the number of attempts to typically two, rising only slightly for mappings that have multiple potentially not-ready input slots. This may give an order of magnitude performance improvement over mindless retries.

In the following sections we will examine how determination of a static schedule can guarantee that a mapping invocation only fails because of a predicate violation. The need to support retries is eliminated simplifying the execution dispatcher and eliminating per-slot not-ready status overheads.

3.3 Compile-time Cycles

The complete global analysis of all production/consumption dependencies between all mappings is, in the general case, a directed cyclic graph. Figure 7 shows the dependency graph for our simple example.

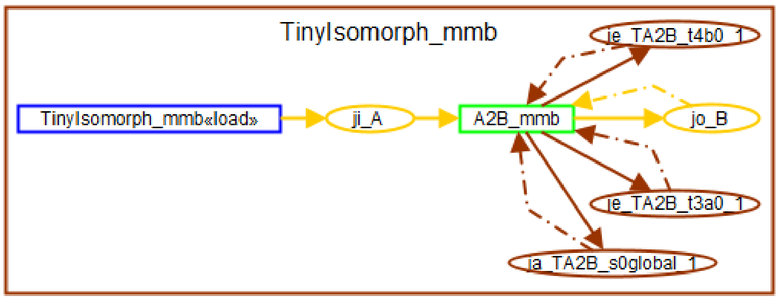


Figure 7 – Example Mapping Dependency Graph prior to Partitioning.

The blue rectangle shows the loader that passes loaded instances of `A` to an orange input connection buffer named `ji_A`. The instances are consumed by the green rectangle for the `A2B` mapping in the `mmb` direction. This mapping passes its produced `B` instances to the output connection buffer named `jo_B`. These output instances are also consumed.

The three brown ellipses support the equivalent production and consumption of edges such as the attribute buffer named `ja_TA2B.s0global_1`. Edge consumptions are hard to realize at run-time and so we would like to optimize them away.

The solid orange lines indicate paths along which instances flow and which can be realized at run-time. The dashed lines indicate consumption with a not-ready hazard.

The four cycles for one node and three edges are not compatible with a simple tree-based schedule, we must therefore resolve. Our simple example contains only one mapping and is 100% cyclic. For out example, we can skip over the considerations in the next section that identify cycles more generally.

3.3.1 Cycle Encapsulation

We can identify the cycles by computing from the green-producer/cyan-consumer analysis of each mapping M as in Figure 6.

- `pre(M)` - predecessors of M, the producers of the elements that M consumes
- `suc(M)` - successors of M, the consumers of the elements that M produces.
- `pre(M)*` - transitive closure of all predecessors of M
- `suc(M)*` - transitive closure of all successors of M
- `cyc(M)` - cycle involving M

We can observe that the mappings that appear in either `pre(M)*` or `suc(M)*` but not both are either transitive predecessors or successors of M and should be scheduled before or after M accordingly to ensure that the producer-consumer timing is unequivocably satisfied.

Conversely every mapping that appears in both `pre(M)*` and `suc(M)*` is part of the cycle involving M. The intersection of `pre(M)*` and `suc(M)*` identifies the `cyc(M)` cycle members.

Mappings for which `cyc(M)` is empty are acyclic; not part of cycles.

A static schedule can be produced for the acyclic mapping by allocating to a first pass all the mappings for which `pre(M)*` is empty, then for subsequent passes allocating the mappings for which `pre(M)*` involves only mappings allocated to earlier passes. This provides a largely parallel schedule since concurrent execution on multiple processors is possible for each mapping in the same pass and each invocation of that mapping. Only passes need sequencing. All overheads in respect of not-ready slots or execution retries can be eliminated.

However in practice cycles do occur. The number of distinct non-empty values for `cyc(M)` identifies the number of cycles.

If `cyc(M1)` and `cyc(M2)` have no common members, `cyc(M1)` and `cyc(M2)` are independent cycles.

If `cyc(M1)` is a subset of `cyc(M2)` then `cyc(M1)` is a nested cycle within `cyc(M2)`.

The third possibility of a non-subset overlap between `cyc(M1)` and `cyc(M2)` cannot arise since all cycle members are both predecessors and successors of each other.

Accommodating cycles within the static schedule proves to be remarkably easy. Each outer cycle has a distinct value of `cyc(M)` that is not a subset of another cycle. The outer cycle can be wrapped in a cyclic-mapping `CM` that is externally acyclic, and only internally cyclic. Since it is externally acyclic, it can be allocated to a pass in just the same was as the inherently acyclic mappings. We just need to ensure that invocations of the `CM` are repeated until the internal cycle is satisfied and ensure that all consumed slots have run-time support for the not-ready that may result from execution of cycle elements in the 'wrong' order.

Producing a static schedule for the members of the cycle is a nested problem. For scheduling we could resort to naive polling, but we can do better inspired by the principles of Structural Induction.

For each mapping M in the cycle we can classify its dependencies as

- `ext(M)` the predecessors of M outside the cycle
- `int(M)` the predecessors of M inside the cycle

A mapping with an empty ext(M) is dependent on some other cycle member and so is a 'recursive' case. Conversely a mapping with a non-empty ext(M) may be able to execute directly and so is a 'basic' case.

The conventional 'base' case is an acyclic preamble. It would have an empty `ext(M)` and `int(M)`. It is not part of the cycle.

The 'basic' cases have a higher probability of successful execution. Their execution attempts can therefore be scheduled first and their dependencies eliminated to allow the remaining 'recursive' cases to be scheduled in acyclic dependency order. An overall surrounding loop repeats for as long as necessary. Extraction of the 'basic' cases improves the initial hit rate and the ordering within the loop avoids premature scheduling of mappings that have no hope of successful execution.

In practice many loops are trivially small or shallow but polymorphically wide limiting the effectiveness of these optimizations.

3.3.2 Partitioning

A typical mapping involves multiple productions and consumptions and may exploit the declarative freedom to intermix inherited attributes, computed top-down, and synthesized attributes, computed bottom-up [ASU86]. This leads to many 'false' cycles, cycles that are an artefact of the convenient declarative exposition rather than a necessary characteristic of the problem. At its simplest a mapping has production that after transitive consumption and production by other members of the cycle is consumed by the original mapping. Since the consumption occurs in the same mapping as the production, the successful consumption is a guard upon the production, but the consumption will fail with a not-ready until the production propagates. Deadlock and since this is exactly what the declarative programmer has specified, it is what we must do. To break the deadlock, we must execute at least one of the productions before their predicating consumptions without changing the overall result.

Fortunately not all consumptions and not all productions are equivalent. Figure 6 is drawn with three columns, the input/left, the trace/middle and the output/right column. No productions occur in the input column. The required externally observable productions occur in the output column. Hidden internal productions occur in the middle column.

In the absence of a complicated facility to roll-back misguided output element creations, we must only perform correct output element creations. However the internal trace provides some flexibility which we exploit by providing the `s0global` status, this is `true` if the trace is good, and `false` if the trace fails or is misguided.

Consumptions of the input model elements typically check for some type/existence constraint. These do not present cyclic problems since the input is available.

Consumptions of the trace model elements typically check that some other execution has happened. This may lead to a cyvle that needs speculation.

Consumptions of output model elements may take two forms, corollaries or post-conditions. A corollary occurs as in Figure 6 when a consumed trace/output object pair is topologcally identical to the trace/output pair in the corresponding producing mapping. If the producer creates the trace and assigns a success, the eventual production of the output is guaranteed to occur. It is therefore sufficient for the consumer to check the consuming trace and its status to determine whther the consuming mapping can be declared successful. The consumed output does not need to exist when success is declared.

Traditionally such usage would require manual separation into distinct passes.

Since the 'false' cycles arise from 'too-many' productions or consumptions per-mapping, we can solve it by emulating the manual partitioning by replacing each too-big mapping by multiple partitions.

The

Production Partitioning Most productions contribute solely to the output from the mapping and so a partitioning to place each production in a distinct partition is possible and in accord with our advocation in [Wil16] to create as many 'atomic' micro-mappings as possible. The cost of too-many micro-mappings was mitigated by a merge of similar micro-mappings as part of the overall partitioning and scheduling.

The combined behavior of the partitions must be identical to the original mapping and so each partition must repeat the predicate checks to ensure that each fails identically. With the same predicates, many of the per-production partitions have identical behavior that can be merged. It is however more effective to avoid over-partitioning. In practice the productions fall into four categories

- ≪`ctor`≫ creation of the unique trace element for the mapping invocation
- ≪`init`≫ creation of the input-side trace edges
- ≪`rest`≫ creation of the output-side nodes and trace edges
- ≪`xtra`≫ creation of the output-side edges

Comsumption Partitioning Production partitioning is fairly straightforward since a production cannot fail. In contrast a consumption can, either temporarily until a slot is ready, or finally if the consumed object is of an unsatisfactory type or value.

Selective Partitioning The cycle analysis applied to the mappings identifies which mappings can be scheduled acyclicly and which need to form part of a cycle. There is no point partitioning acyclic mappings, so we concentrate our efforts on the typically small and sometimes zero number of cyclic mappings. Once we have partitioned the cyclic mappings, we can repeat the cycle analysis and often discover that many of the cycles were 'false' cycles and so the partitions can be statically scheduled. Only the 'true' cycles need incur the extra run-time overheads.

Partition Content Selection Safe, guaranteed to be ready/valid, consumptions provide no benefit when partitioned. Unsafe, possibly not-ready/not-valid consumptions cannot be partitioned since a partial mapping must not succeed when the overall mapping would have failed. Partitioning is therefore restricted to productions.

In [Wil16], we advocated a maximal partitioning into micro-mappings so that each micro-mapping has at most one green creation or assignment element to break many of the 'false' cycles, but this imposes a perhaps three-fold burden on the subsequent analyses that must deal with a much larger number of micro-mappings. Unless a later merge phase is introduced, these extra micro-mappings may incur run-time costs as well. We now advocate the extra effort of a just-right partitioning of as few mappings as possible into as few partitions as possible; green elements with shared consumptions can remain together, but those with distinct dependencies can be separated. This typically results in some or all of:

- a ≪`ctor`≫ partition to create the trace linked to input roots
- a ≪`init`≫ partition to check predicates and assign green trace-to-input edges
- a ≪`loop`≫ partition to speculate and create green trace-to-output edges
- a ≪`rest`≫ partition to assign green output-to-output edges
- a ≪`xtra`≫ partition to assign green output-dependent edges

However partitioning productions is insufficient to break all the 'false' cycles. In Figure 6 we see a very common green production pattern whereby the mapping creates a `trace` trace element an output element `b0` with a linking trace element `trace.t4b0` with a status `trace.s0global` that will be true whenever `trace.t4b0` and `b0` are valid. A similar cyan consumption pattern for `when_A2B`, `b1`, `when_A2B.t4b0` is guarded by a requirement that the status `when_A2B.s0global` is `true`. It is therefore sufficient to check that `when_A2B` exists and that `when_A2B.s0global` is `true` to guarantee that `when_A2B.t4b0` and `b1` are available. We therefore call `b0` a corollary of `A2B`; we can omit `when_A2B.t4b0` and `b1` from partitions provided `when_A2B.s0global` is checked for `true`. Elimination of dependencies on corollaries significantly reduces the number of output dependencies.

Figure 8 shows the content of the four partitions, with two new colors. Red for an not yet initialized trace and orange for elements participating in a speculation. The additional `s0local` status prevents the ≪`loop`≫ satring before ≪`init`≫ has finished.

Figure 9 shows the Dependency Graph for our simple example after it has been partitioned.

There are now four green partitions with the ≪`ctor`≫ partition consuming the `A` input and passing the `TA2B` traces via the `jm_TA2B_1` connection to the other three. All the cycles except that involving the `s0global` status have been broken.

We can now allocate the partitions to passes. Figure 10 shows the Dependency Graph after partitions and connections have been labeled with pass-numbers. and after removal of all edges whose productions are guaranteed to occur in an earlier pass than all its consumptions.

We are left with five passes for the overall ≪`load`≫, and the ≪`ctor`≫, ≪`init`≫, ≪`loop`≫, ≪`rest`≫ partitions of the one mapping. Only one slot requires the ready/not-ready status to be maintained at run-time.

(Folding the ≪ctor≫ partition into its caller is future work.)

(Folding the ≪loop≫ partition into the ≪init≫ partition is future work justified in Section 3.4.2.)

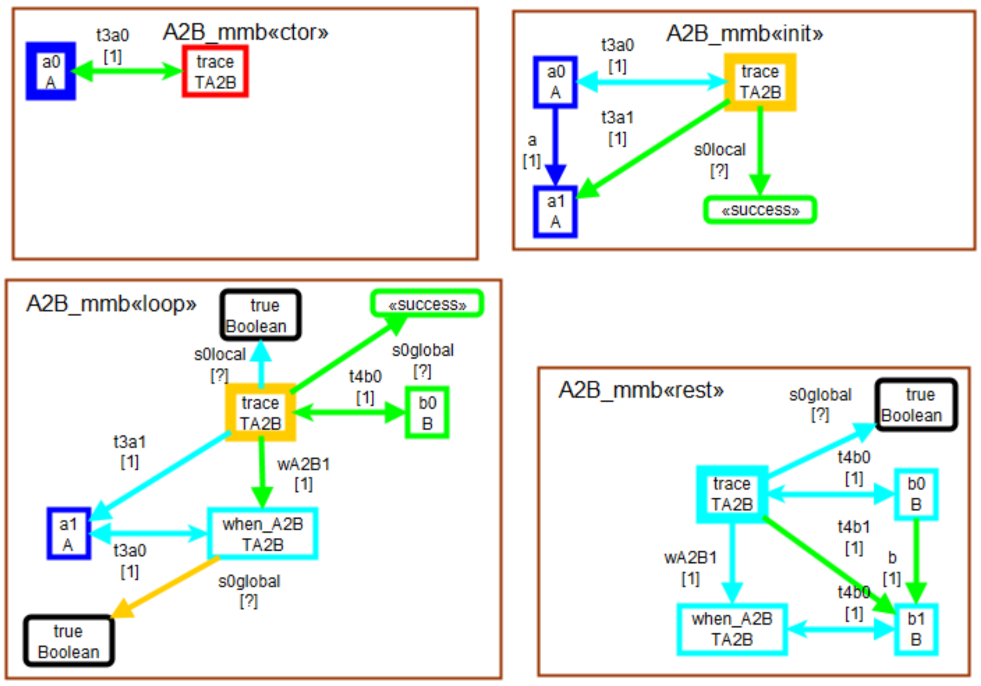


Figure 8 – Partitions for Example Mapping.

3.4 Run-time Cycles

Where our compile-time analysis fails to eliminate a potential cycle, we need the run-time to resolve it.

It is easy to assume that even a naive scheduler that keeps trying all possible permutations of objects and mappings will eventually make progress and produce the required output. This is true of most practical transformations, but not necessarily so because each mapping may have a guard that inhibits execution unless some conditions are satisfied.

3.4.1 Speculation

Consider the very simple game: I'll show you mine, if you show me yours.

There are two stable outcomes; we both show or we both hide. There are also two outcomes where one of us cheats. The possibility of cheating can be eliminated by introducing a mediator. The result can be made deterministic if the mediator directs the play with a bias towards maximum disclosure.

A similar situation may arise in a model transformation. Which of multiple do-nothing or multiple do-something is appropriate? 'the whole truth' principle provides the solution: do-nothing omits a legitimate action, it is not a whole-truth and so we can require maximum transformation. For many cases, this maximum disclosure/execution renders the mediator redundant, since the mediator always directs the players to show.

Let us elaborate our two person game to a multi-person ring: I'll show to the right if the left shows to me. The need for a mediator is now more important, but again redundant if the mediator always directs all players to proceed.

If we differently elaborate our game with a player who declines to gamble on the

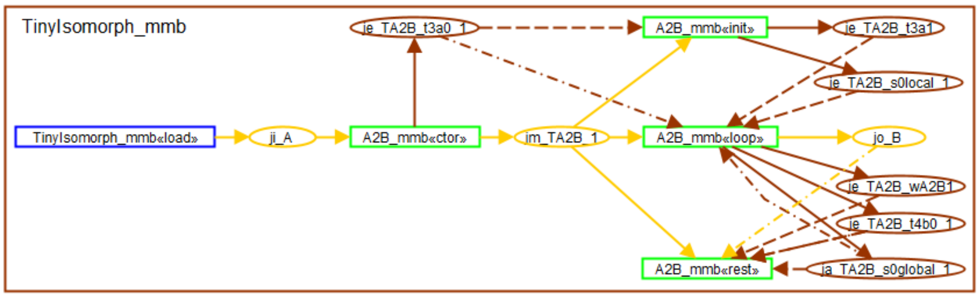


Figure 9 – Example Mapping Dependency Graph after Partitioning.

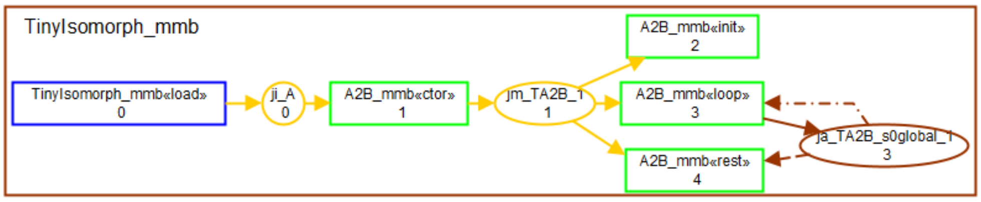


Figure 10 – Example Mapping Dependency Graph with Schedule Passes.

sabbath, the outcome of the game is still deterministic, but we need our mediator to determine whether everyone is prepared to play, before directing play for the consistent outcome. The analogous situation in a model transformation may arise through model or transformation polymorphism; some model elements may behave differently to others.

At run-time the mapping invocation inter-dependencies may form many graphs. The run-time therefore blocks each invocation that needs a speculation verdict from the mediator until all such invocations have been blocked. The mediator can then perform an AND function to determine whether all participants are prepared to play and release the blocked invocations with the appropriate successful/failed speculation.

Now consider a different elaboration of our game with rogue players whose behavior to the right is the opposite of that from the left. With an odd number of rogue players, we have a contradiction around the loop. With an even number of rogue players, there may be multiple outcomes with the maximum disclosure dependent on the relative positioning of the rogue players. There may be no unique maximum solution. This is a bad game. Likewise we must be prepared for the possibility that a transformation may be bad. The rogue player has introduced a NOT to complicate our logic, so that rather than checking the obviously unique hypothesis that all invocations can be successful with an AND gate, we need to search for a unique pattern of successful and failing invocations to satisfy the arbitrary logic that NOT and AND facilitate. Even for our simple example we can see that it is bad. Anything more complicated may require an impractical computation to determine the not-necessarily unique result. It seems better to declare these as bad at the outset.

Consider a resilient persistence transformation that has two alternatives, either to save using a DataBase or to use XML serialization; either may fail. The author may seek to minimize wasted execution by specifying saveInDB XOR saveInXML, but XOR hides an underlying NOT. What is the transformation supposed to do if both saveInDB and saveInXML are successful? It is a bad transformation and should be rejected as such and at the earliest possible opportunity.

3.4.2 Special Cycles

We have identified that acyclic run-time dependencies can be handled without incurring the overheads of speculation support. It is therefore beneficial to identify situations were the run-time dependencies can be guaranteed to be acyclic.

Composite Relationships UML (and Ecore) imposes a constraint that no instance can have more than one container. Consequently any cycle whose elements are interdependent solely because of a container/containment relationship is guaranteed to be acyclic; speculation support is not needed.

Infallible Mappings Some mappings and metamodels are so simple that once constructed with a conformant head node there is no other mechanism for the mapping to fail. The 'mediator' is guaranteed to direct a speculation success. This is the case with our running example. The redundant speculation should be detected, The test for the `s0status` is redundant and so distinct ≪`loop`≫ and ≪`init`≫ partitions are unnecessary. We should have just ≪`init`≫ and ≪`rest`≫ partitions similar to the manual solution.

3.4.3 Overriding Mappings

The QVT specification is somewhat vague as to the precise semantics of mapping overrides. It is therefore left to the Eclipse QVTd project to apply 'the whole truth' principle to specify that as many mapping invocations as possible should be attempted, and the 'nothing but the truth' principle to specify that no mapping invocation should be attempted if an overriding mapping invocation is successful. This amounts to a NOT predicate on each potential override which is not consistent for no-NOTs in a run-time speculation. Fortunately the primary speculation is on the polymorphic invocation which is positive. The individual overrides do not comeinto play until the polymorph has been speculated.

4 Related Work

Existing model to model transformation tools such as ATL, Eclipse QVTo, Epsilon and Henshin [Eclc] do very little if any static analysis or optimization. This work on static analysis and optimization of declarative schedules and metamodels using micromappings and connections appears to be almost completely novel.

In the Triple Graph world, a catalogue of optimizations has been proposed [LAS14]; domain driven applicability, caching/indexing, static analysis, incremental execution. Many of these come for free in the current work.

The explicit middle model imposed by QVTc traceability reifies a trace object so that to-one navigation paths between source and target can be very cheap and optimized to avoid more expensive paths. The trace object is an inherent cache of related information. Indexing is an OCL-level further work optimization.

Although not discussed in this paper, the utility of Connections shown as ellipses in Figures 7, 9, 10 for incremental execution is demonstrated by the Eclipse implementation.

Detailed comparison of the approaches is quite hard, since it is very easy to provide a really bad reference implementation against which almost any sensible implementation will appear fantastic. Erly results demonstarting the performance were presented in [Wil16].

This work diverged from an early Epsilon prototype to exploit the very strong limitations imposed by metamodels and declarative mappings. It therefore uses heuristics to produce a useful schedule relatively quickly, rather than exploring a large number of alternative schedules in an infeasible time [RK16].

5 Conclusion

We have demonstrated how detailed analysis of declarative transformations and metamodels can avoid the potentially horrendous performance of a naive execution.

- Type analysis avoids needless permutations.
- Head node identification minimizes search match dimensionality.
- Producer/consumer analysis guarantees a correctly sequenced execution, enables cycle detection and minimizes not-ready cycling.
- Partitioning breaks many compile-time cycles.
- Efficient run-time support reduces not-ready support and supports the speculation necessary to handle the residual run-time cycles.

Results of the first optimized code generated implementation show that declarative transformations can approach the performance of manually coded transformations [Wil16].

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeff D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

[Ecla] Eclipse ATL Project. https://projects.eclipse.org/projects/modeling.mmt.atl.

[Eclb] Eclipse Epsilon Project. https://projects.eclipse.org/projects/modeling.epsilon.

[Eclc] Eclipse Henshin Project. https://projects.eclipse.org/projects/modeling.emft.henshin.

[Ecld] Eclipse QVT Declarative Project. https://projects.eclipse.org/projects/modeling.mmt.qvtd.

[LAS14] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. A catalogue of optimization techniques for triple graph grammars. In *Modellierung*, 2014.

[Obj08] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification, Version 1.0, April 2008. formal/2008-04-03.

[Obj14] Object Management Group. Object Constraint Language, Version 2.4, February 2014. formal/2014-02-03.

[Obj16] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification, Version 1.3, June 2016. formal/2016-06-03.

[RK16] Horacio Hoyos Rodriguez and Dimitris Kolovos. Declarative model transformation execution planning. In *15th International Workshop on OCL and Textual Modeling*, Saint-Malo, October 2016.

[Wil16] Edward Willink. Local Optimizations in Eclipse QVTc and QVTr using the Micro-Mapping Model of Computation. In *2nd International Workshop on Executable Modeling, Exe 2016*, Saint-Malo, October 2016. http://eclipse.org/mmt/qvt/docs/EXE2016/MicroMappings.pdf.

About the author

Edward D. Willink is the chair of QVT and OCL specification Revision Task Forces at the OMG and project leader for QVTd and OCL at the Eclipse Foundation. Contact him at `ed_at_willink.me.uk`.

Acknowledgments Many thanks to Horacio Hoyos Rodriguez for the graphical perspective and Epsilon prototype and to all Dimitris Kolovos' team for insightful discussions.