Optimized declarative transformation First Eclipse QVTc results

Edward D. Willink[1]

Willink Transformations Ltd., Reading, UK,
ed/at/willink.me.uk,

Abstract. It is over ten years since the first OMG QVT FAS[1] was made available with the aspiration to standardize the fledgling model transformation community. Since then two serious implementations of the operational QVTo language have been made available, but no implementations of the core QVTc language, and only rather preliminary implementations of the QVTr language. No significant optimization of these (or other transformation) languages has been performed. In this paper we present the first results of the new Eclipse QVTc implementation demonstrating scalability and major speedups through the use of metamodel-driven scheduling and direct Java code generation.

Keywords: optimization, declarative transformation, scheduling, code generation, QVTc

1 Introduction

The OMG QVT specification [11] was planned as the standard solution to model transformation problems. The Request for Proposals in 2002 stimulated 8 responses that eventually coalesced into a revised merged submission in 2005 for three different languages.

The QVTo language provides an imperative style of transformation. It stimulated two useful implementations, SmartQVT and Eclipse QVTo.

The QVTr language provides a rich declarative style of transformation. It also stimulated two implementations. ModelMorf never progressed beyond beta releases. Medini QVT had disappointing performance and is not maintained.

The QVTc language provides a much simpler declarative capability that was intended to provide a common core for the other languages. There has never been a QVTc implementation since the Compuware prototype was not updated to track the evolution of the merged QVT submission.

The Eclipse QVTd project [5] extended and enhanced the Eclipse OCL [4] framework to provide QVTc and QVTr editors, but until now provided no execution capability. This paper describes two aspects of the architecture that promises a high performance remedy for this deficiency. In Section 2 we review

[1] Object Management Group Query/View/Transformation Final Adopted Specification.

the efficiency hazards that are addressed in Section 3. Section 4 describes the derivation of an efficient declarative schedule and Section 5 presents preliminary results demonstrating some scalability and code generation speed-ups. Section 6 summarizes some related work and Section 7 concludes.

2 Efficiency

The execution time of each algorithm in a computer program is 'obviously' the product of many factors: $(W.N.R.L.C)^A$

- W - the Workload - the number of data elements to be processed
- N - the Necessary computations per data element
- R - the memory Representation access overhead
- L - the programming Language overhead
- C - the Control overhead
- A - the Algorithmic overhead

We cannot optimize N or W since they represent the Necessary Work.

We have limited choice in regard to L since use of a more efficient Language than Java may incur tooling, portability or cultural difficulties. We have similarly limited choice in Representation since there may be few frameworks to choose from. We should however be aware that Java may cost a factor of two or three and that unthinking use of EMF [3] may incur a factor of ten (Figure 5).

The Control overhead is influenced a bit by programming style, but mostly by the tooling approach, thus an interpreted form of execution may easily incur a ten-fold overhead when compared to a code generated one.

The above costs are all proportionate. Algorithmic cost is however exponential, normally with a unit exponent, but a poor algorithm may involve quadratic or worse costs. It is therefore recognized that the best way to improve performance is to discover a better algorithm, or, more practically, to replace a stupid algorithm that performs repeated or unnecessary work.

For a model transformation using an imperative transformation language, the programmer specifies the algorithms (mappings) and the control structures that invoke them (mapping calls). We hope that the programmer selects good algorithms and that the tooling for the transformation language implements them without disproportionate overheads so that the overall execution incurs only proportionate costs relative to the given program's Necessary Work.

When we use a declarative model transformation, the control structures are no longer defined by the programmer. A control strategy must be discovered by the transformation tooling. This has the potential to give improved performance since a better control strategy may be discovered than that programmed imperatively. On the other hand, inferior declarative transformation tooling may incur Control overheads that result in disproportionate Algorithmic overheads.

In this paper we outline approaches that avoid Algorithmic overheads and since we use code generation to Java, we compound a perhaps five-fold reduction in Representation access and a perhaps five-fold reduction in Control overhead when compared to an interpreted execution using dynamic EMF.

3 Architecture

The Eclipse QVTd architecture for QVTr and QVTc 'solves' the problem of implementing a triple language specification by introducing Yet Another Three QVT Languages[10] : QVTu, QVTm and QVTi. Subsequent scheduler development has introduced two more languages QVTp and QVTs for use in the transformation chain shown in Figure 1.

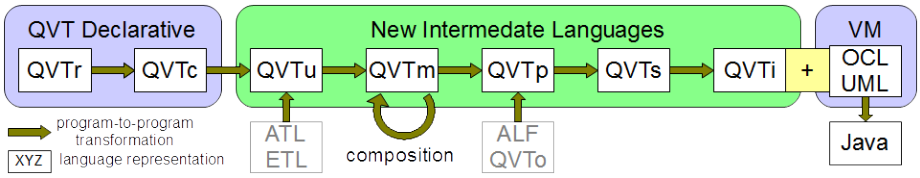

Fig. 1. Progressive transformation approach for Declarative QVT.

QVTr2QVTc implements the RelToCore transformation only partially defined by the QVT specification. This is still a work in progress.

QVTc2QVTu exploits the user's chosen direction to form a Unidirectional transformation without the bloat for the unwanted directions.

QVTu2QVTm normalizes and flattens to form a Minimal transformation free from the complexities of mapping refinement and composition.

QVTm2QVTp Partitions mappings into micro-mappings that are free from deadlock hazards.

The foregoing representations use increasingly restricted semantic subsets of the QVTc language.

QVTp2QVTs creates a graphical representation suitable for dependency analysis enabling an efficient Schedule to be planned.

QVTs2QVTi serializes the graphical representation and schedule into an Imperative extension and variation of the QVTc language. This can be executed directly by the QVTi interpreter that extends the OCL Virtual Machine. Alternatively an extended form of the OCL code generator may be used to produce Java code directly. This bypasses many of the overheads of interpreted execution or dynamic EMF. The generated code comprises one outer class per-transformation. Each mapping is realized as either a nested class or a function depending on whether state is needed to handle repeated or premature execution.

4 Scheduling

4.1 Naive Schedule

A declarative transformation can always execute using the naive polling schedule

- Retry loop - loop until all work done
- Mapping loop - loop over all possible mappings
- Object loops - multi-dimensional loop for all object/argument pairings
- Compatibility guard - if object/argument pairings are type compatible
- Repetition guard - if this is not a repeated execution
- Validity guard - if all input objects are ready
- Execute mapping for given object/argument pairings
- Create a memento of the successful execution

This is hideously inefficient with speculative executions wrapped in at least three loops, guards and mementos when compared to a simple linear 'loop' nest.

The key functionality of a declarative model transformation comprises a suite of OCL expressions that relates output elements and their properties to input elements and their properties using metamodels to define the type system of the elements and properties. The suite of OCL expressions is structured as mappings that provide convenient re-usable modules of control.

4.2 Metamodel-driven planning, micro-mappings

The metamodel type system enables producer/consumer relationships to be established between mappings. These relationships can be used by an intelligent 'Mapping loop' to avoid many of the retries that result from careless attempted execution of consumers before producers. Similarly, considering only type compatible objects in the 'Object loops' eliminates another major naivety.

Analysis of the types alone is of limited value, since each type may have many properties. Each property assignment must be analyzed separately to avoid deadlock hazards. For instance, an attempt to assign both forward and backward references of a circular linked list at once deadlocks somewhere round the loop. Therefore a declarative mapping that appears to assign all properties at once, must be broken up into smaller micro-mappings to avoid a deadlock hazard. In practice fewer than 10% of mappings appear to need partitioning into micro-mappings, and once a valid schedule has been established, some of these can be merged.

The 'Object loops' can be very inefficient even when restricted to type compatible objects, since the number of permutations grows exponentially with the number of inputs. For genuinely independent inputs this cost is fundamental and the only solution is for the programmer to provide a better algorithm that exposes a dependence. In practice many objects have a very strong relationship such as between a parent and a child object. From the parent there may be many child candidates, but from the child there is only zero or one parent object. We

therefore replace the parent as an externally searched input by an internally derived computation. This reduces the order of the input permutation. In practice between 90% and 100% of micro-mappings can be simplified to only one input.

A micro-mapping either executes successfully updating its outputs, or fails completely updating nothing until a later successful re-execution. Failure occurs whenever the computation prematurely navigates from one object to another. We therefore want to identify as many of the objects that a micro-mapping may access in order to provide a static schedule in which consumed objects are produced first.

4.3 Intra-micro-mapping scheduling

Of course at compile-time we have no instances, just types which only give us limited producer-consumer precision. However the constraints in a declarative transformation define patterns that enable us to relate multiple types. Our QVTp2QVTs analysis therefore starts with a transliteration from partitioned QVTc as shown in Figure 2 to graphical form as shown in Figure 3

```
map classToTable in umlRdbms {
    check uml(p : Package, c : Class |) {}
    enforce middle(p2s : PackageToSchema|) {
        realize c2t : ClassToTable
    |}
    enforce rdbms(s : Schema|) {
        realize t : Table, realize pk : Key, realize pc : Column
    |}
    where(p2s.umlPackage = p; p2s.schema = s; c.namespace = p;) {
        c2t.owner := p2s; c2t.umlClass := c; c2t.table := t;
        c2t.name := c.name; c2t.primaryKey := pk; c2t.column := pc;
        t.kind := 'base'; t.schema := s;
        pk.owner := t; pk.kind := 'primary';
        pc.owner := t; pc.keys := Set(SimpleRDBMS::Key){pk}; pc.type := 'NUMBER';
    }
}
```

Fig. 2. Class-to-Table mapping in partitioned QVTc (QVTp).

The example is taken from the ubiquitous UML to RDBMS example and shows the mapping from a Class (child of a Package) to a Table (child of a Schema) and since we are using QVTc, the intermediate trace is programmed explicitly as a ClassToTable. The textual representation in Figure 2 shows a two input (p, c), two output (p2s, s) interface and four creations (c2t, t, pk, pc). The parentheses of the where clause show three pattern matching constraints. The braces of the where clause show thirteen assignments.

The graphical form is inspired in part by Henshin [1] but with additional colors. Black indicates what is constant at compile-time. Blue shows what is loaded from input models. Green indicates what is created. Additionally Cyan identifies what is consumed after it is produced elsewhere.

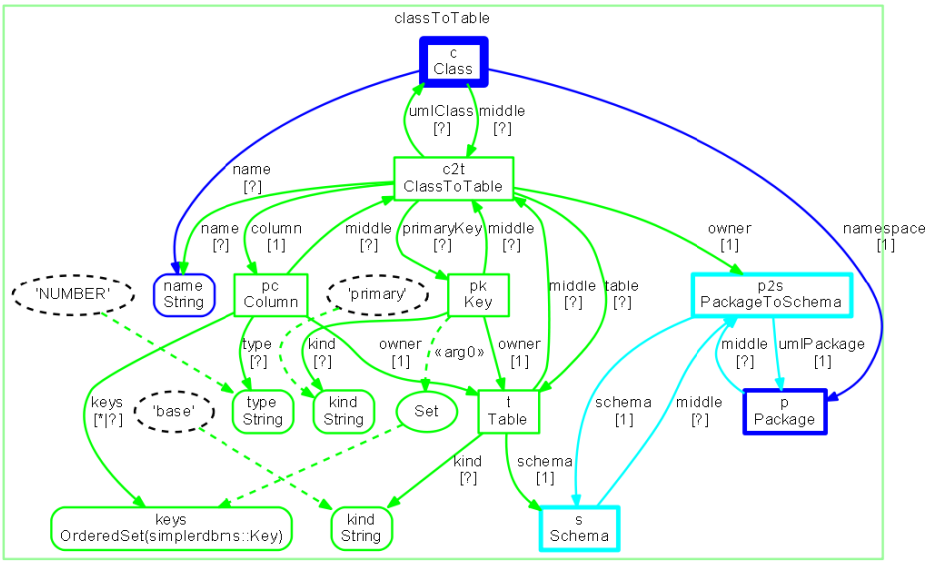

Fig. 3. Class-to-Table mapping in graphical form (QVTs).

Solid edges show something-to-one unidirectional navigation paths between rectangular class instances and their rounded-rectangular attributes. Nodes are annotated with instance name and type, edges with property name and multiplicity. Dashed edges and ellipses show computations rather than navigations.

Bidirectional navigations are shown as pairs of unidirectional edges, but only something-to-one navigations are shown. Consequently any navigation edge may be traversed in its forward direction to the precisely one object that matches the pattern element. Following all the paths in this way identifies that all navigable objects can be reached unambiguously from the c:Class input. This is drawn with a very thick boundary to emphasize its importance. What appeared to be a 4-input mapping in textual form is actually a 1-input mapping, so what appeared to challenge the scheduler with a 4-dimensional search is realizable with a 1-dimensional loop.

The graphical form therefore clearly shows, in blue, that we need to match a c:Class in the loaded input model that has a Package at c.namespace and a String at c.name. Additionally the cyan shows that execution is not possible until p.middle provides p2s and p2s.schema provides s. Once these are all available, the many objects and relationships shown in green can be produced. We therefore have a very simple dependency rule; every cyan node or edge must be produced by a corresponding green node or edge in another micro-mapping.

4.4 Inter-micro-mapping scheduling

In practice it is not quite so simple because we must account for multiple producers and derived types. But in principle we can just join all the micro-mappings up with dependencies to derive the schedule as shown in Figure 4[2].

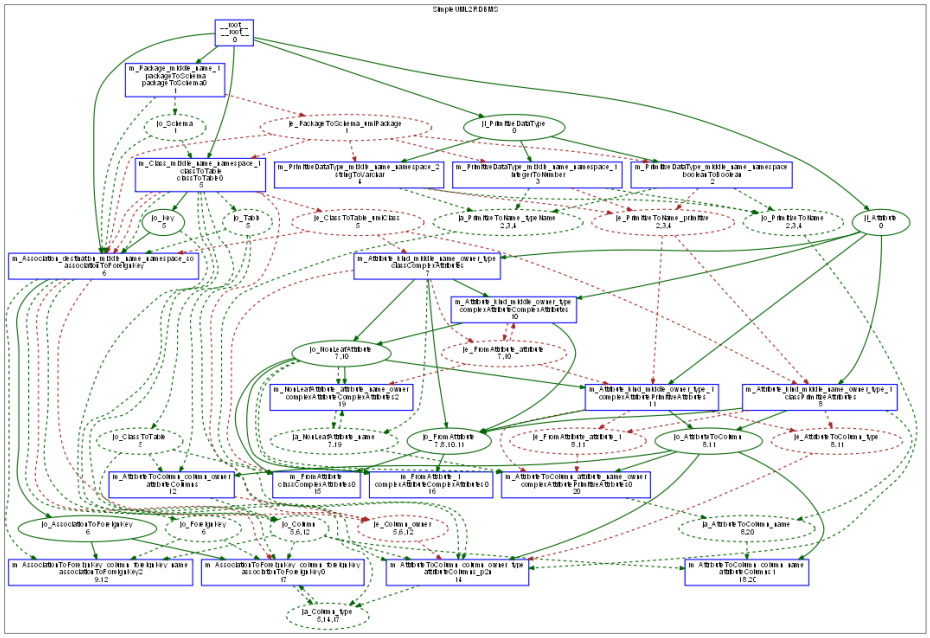

Fig. 4. UML2RDBMS schedule in graphical form (QVTs).

At the top, the root pseudo mapping is a producer of input model elements, triaged as allInstances of interesting types. These are consumed by the 19 rectangles one for each merged micro-mapping of the transformation.

Solid edges pass model elements to each consuming input. Simple connections are drawn directly. Multiple producers or multiple consumers are drawn through an intermediate ellipse which corresponds to a buffer that accumulates inputs from many producers and then makes them available to many consumers.

Further connections are shown using dashed edges and ellipses. These can easily be recomputed by the consumer and so no connection needs to be reified, however the schedule must ensure that the producer produces before the consumer consumes. Thus for the ClassToTable example, a Class is passed via a solid edge; the other three inputs are computed by navigation from the one input. These computations fail until the dashed dependencies have been satisfied. The many (75%) dashed elements shows a useful compile-time optimization.

[2] One rectangle, two ellipses and associated lines have been removed so that the nodes and edges are discernible; the lettering is not relevant.

A sequential schedule is derived by allocating an integer slot index to each micro-mapping in turn such that all its predecessors have smaller indexes. In practice, a perfect static schedule can often be derived, but sometimes a recursion may introduce a cyclic dependency that can only be resolved at run-time. For this small proportion of micro-mappings additional run-time support is activated so that a failed speculative execution blocks according to the failure and retries only once the failure cause has been resolved. Few micro-mappings incur multiple failures and so in practice the retry overhead is less than 50% and only where the data relationships exceed the compile-time analysis capability.

Three of the nineteen UML2RDBMS micro-mappings require two inputs. This incurs a quadratic tooling algorithm inefficiency that shows up when the performance is measured for input models containing associations. For just packages, classes and properties, the execution scales linearly with Workload. Examination of these three mappings reveals that there is a 1:N relationship between a primary and a secondary part of the mapping. Future work includes an optimization to treat the primary input as the one input and derive the secondary input using a local loop. This local loop can iterate only over matches, whereas the current external global loop iterates over many mismatches.

5 Results

As just described, the performance of the QVTc variant of UML2RDBMS scales linearly when the input model contains just packages, classes and properties, but, pending further work, goes quadratic once associations are added.

In this section we plot the performance of the simple Familes2Persons transformation that involves a two-way guarded decision around a copy[3]. The plot demonstrates the scalability and the underlying tooling efficiency.[4]

The top two lines of Figure 5 show the performance of the new interpreted QVTc and contrasts it with the Eclipse QVTo. There is very little difference, except that QVTo has a much higher trace overhead and so fails to execute beyond 2,500,000 elements; an unexpected advantage of the explicit QVTc trace.

The two lines slightly below these show the old and new ATL VMs. Both fail to get close to 10,000,000 elements in the 4GB 64 bit VM.

The next two lines are about 30 times faster and contrast the new code generated QVTc with the standard EcoreUtil copy functionality. QVTc is currently about 20% slower and just fails to reach 10,000,000 elements.

The final line is a manually coded implementation of the copy that bypasses the overheads of dynamic EMF in similar fashion to the code generated QVTc. For large numbers of model elements, this is about twenty times faster indicating that there is considerable scope for further improvement using the current approach. Using a non-EMF model representation can give further improvements and moving to C execution yet more.

[3] Model overheads are reduced by Java code generated by an EMF genmodel.

[4] The plots use no averaging. Single point wobbles may be due to concurrent activity. Multiple point wobbles may be due to fortuitous cache alignment.

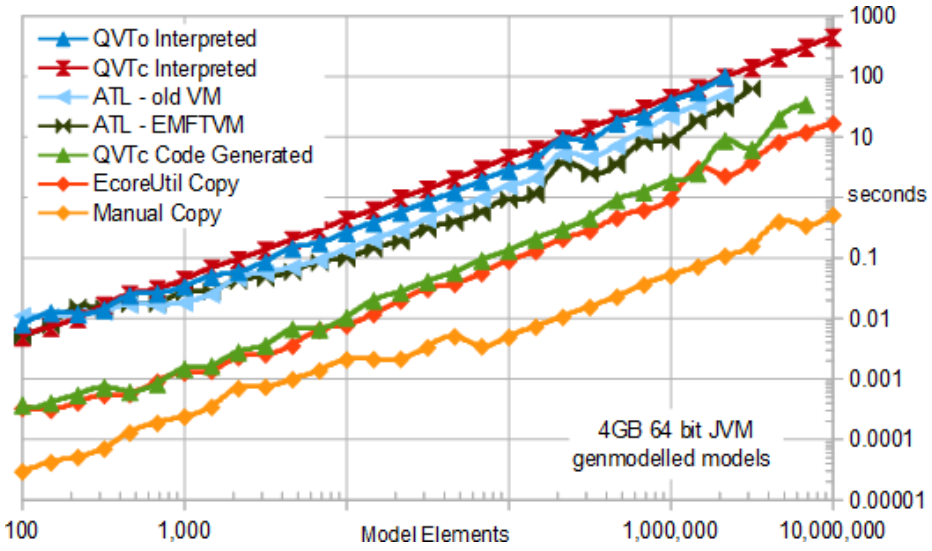

Fig. 5. Performance of the simple Familes2Persons transformation.

6 Related Work

The idea of generalizing the concepts of a Java Virtual Machine to a Modeling Virtual Machine is attractive and has influenced ATL [2]. However while its original representation had small byte-code-like concepts, they were far from byte-sized. The newer EMFTVM is therefore able to do significantly better [6], but retains a transformation structure that supports transformation with OCL expressions as an afterthought. EMFTVM is 80% slower than its EcoreUtil 'optimum'. In contrast the OCL VM uses the OCL Abstract Syntax as its 'byte-code' and supports extension to the QVTi AS so that all AS elements form part of a single executable element hierarchy. The performance reported in the previous section was 20% slower than EcoreUtil but the 'optimum' target for further work is ten times faster than EcoreUtil.

Code generation has not been pursued by other transformation engines, perhaps because OCL code generation is not easy. Superficially there are similarities between OCL and Java and so a number of researchers have performed simple text template mapping [8]. This works sometimes, but fails to handle OCL in general. In contrast the Eclipse OCL Java code generator converts the OCL AS to an intermediate CG AS upon which a number of optimizations and rewrites are performed before Java code is emitted. The code generator supports all OCL constructs and its extensibility is demonstrated by its re-use for QVTi.

The Graph Transformation community has been very active in providing a rigorous foundation for graph mappings. Sadly the QVT specification ignored this important work, preferring instead to define the semantics of the QVTr transformation language using an incomplete exposition of a transformation of

QVTr written in an untested QVTr to another language (QVTc) that has at best informal semantics. The utility and power of the graphical QVTs representation may begin to bridge the gap between these two communities. The coloring in QVTs is inspired by the use of colors to denote create/delete/no-change in endogenous transformations. The reification of the QVTc traceability element mirrors the evolution operators in UMLX [9] for heterogeneous transformations.

Active Operations [7] also reify mappings to persist the state necessary for incremental execution. Micro-mappings similarly support incremental execution, but their primary rationale is to be a deadlock-free unit of computation.

7 Conclusions

We have introduced the first implementation of the QVTc specification.

We have introduced the first direct code generator for model transformations.

We have shown that the direct code generator gives a thirty fold speed-up.

We have shown how a graph presentation of metamodel and dependency analyses tames the naive inefficiencies of a declarative schedule.

We have reported first results and mentioned some future works. Many more optimizations to do, and of course, QVTr.

References

1. Biermann, E., Ermel, C., Schmidt, J., Warning, A.: Visual Modeling of Controlled EMF Model Transformation using HENSHIN Proceedings of the Fourth International Workshop on Graph-Based Tools, GraBaTs 2010.
2. Eclipse ATL Project. https://projects.eclipse.org/projects/modeling.mmt.atl
3. Eclipse EMF Project. https://projects.eclipse.org/projects/modeling.emf.emf
4. Eclipse OCL Project. https://projects.eclipse.org/projects/modeling.mdt.ocl
5. Eclipse QVT Declarative Project. https://projects.eclipse.org/projects/modeling.mmt.qvtd
6. EMFTVM performance. https://wiki.eclipse.org/ATL/EMFTVM#Performance
7. Jouault, F., Beaudoux, O.: On the Use of Active Operations for Incremental Bidirectional Evaluation of OCL 15th International Workshop on OCL and Textual Modeling, Ottawa, 2015
8. Wilke, C.: Java Code Generation for Dresden OCL2 for Eclipse, February 19, 2009. http://claaswilke.de/publications/study/beleg.pdf
9. Willink, E: UMLX : A Graphical Transformation Language for MDA Model Driven Architecture: Foundations and Applications, MDAFA 2003, Twente, June 2003. http://eclipse.org/gmt/umlx/doc/MDAFA2003-4/MDAFA2003-4.pdf
10. Willink, E.: Yet Another Three QVT Languages. ICMT 2013 (2013)
11. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3 Beta. OMG Document Number: ptc/15-10-02, March 2016.