Local Optimizations in Eclipse QVTc and QVTr using the Micro-Mapping Model of Computation

Edward D. Willink
Willink Transformations Ltd.
Reading, United Kingdom
Email: ed_at_willink.me.uk

***Abstract*—The OMG QVT FAS[1] was the result of, perhaps premature, enthusiasm to standardize the fledging model transformation community. The Eclipse implementation of the QVTo language prospers but the initial implementations of the declarative QVTr language had poor performance and have faded away. Perhaps it is time to consign QVTc and QVTr to the dustbin of misguided initiatives. Alternatively, in this paper we show how metamodel-driven analysis and a disciplined Model of Computation support fulfilment of the original aspirations.**

***Index Terms*—Graph Optimization; QVT; QVTc; QVTr; Micro-Mapping; Model of Computation**

I. Introduction

The OMG QVT specification [6] was planned as the standard solution to model transformation problems. The Request for Proposals in 2002 stimulated 8 responses that eventually coalesced as a merged submission in 2005 for three 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 stimulated two implementations. However 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. However 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 [12] extended and enhanced the Eclipse OCL [11] framework to provide QVTc and QVTr editors, but until now provided no execution capability. This paper introduces the Micro-Mapping and its Model of Computation that underpins the production of efficient schedules and describes local Micro-Mapping optimizations.

In Section II we briefly describe the Eclipse QVTd architecture in order to provide the application context for Micro-Mappings. In Section III we consider how a declarative transformation is executed to motivate the Micro-Mapping Model of Computation in Section V. A running example is presented in Section IV and resumed in Section VI. Results from the example appear in Section VII demonstrating some scalability and code generation speed-ups. Section VIII summarizes some related work and Section IX concludes.

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

II. Progressive Transformation Architecture

The Eclipse QVTd architecture 'solves' the problem of implementing a triple language specification by introducing Yet Another Three QVT Languages[8] (QVTu, QVTm and QVTi). Figure 1 shows that a further QVTs is useful.

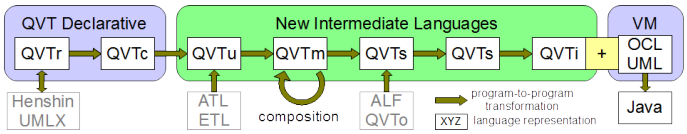

Fig. 1. Progressive transformation approach for Declarative QVT.

1) QVTr2QVTc: the incomplete RelToCore transformation from the QVT specification. This is still a work in progress.

2) QVTc2QVTu: creates a Unidirectional transformation without the bloat for the unwanted directions.

3) QVTu2QVTm: creates a Minimal transformation free from the complexities of mapping refinement and composition.

4) QVTm2QVTs: converts to a graphical form suitable for static Scheduling analyses.

5) QVTs2QVTs: rewrites as Micro-Mappings to avoid live-lock / deadlock hazards and establishes an efficient Schedule.

6) QVTs2QVTi: serializes to an Imperative form for direct execution by the QVTi interpreter that extends the OCL Virtual Machine. Alternatively an extension of the OCL code generator produces Java code with one outer class per transformation and a function or inner class per compound Micro-Mapping. The generated Java code bypasses many of the overheads of interpreted execution or dynamic EMF.

In this paper we concentrate on the local schedule analyses and optimizations that result in the creation of Micro-Mappings at the start of the QVTs2QVTs transformation.

III. Declarative Transformation Execution

An imperative transformation author provides the control strategy that the transformation tooling must use in order to execute the transformation. The performance is governed by the quality of the programmed control strategy and the ability of the transformation tool to implement what it has been instructed to do. The accuracy is totally dependent on unchecked assertions that required function inputs are ready.

In contrast, a declarative transformation just expresses a truth that relates the output models to the input models. The truth can be demonstrated after the transformation completes, but how the constraints that express the truth are executed may not be obvious. A declarative transformation therefore requires less programming[2], but declarative transformation tooling must discover an appropriate control strategy to establish the required truth. This provides an opportunity to provide inherently accurate solutions that are faster than those for imperative transformation languages. But it also requires substantial compilation effort to avoid poor quality solutions.

A. Commit-Actions, Micro-Mappings and Naive Scheduling

The result of a transformation is one or more intermediate or output models each of which may be rendered as a UML Instance Diagram with Class-typed nodes for the model elements and Property-typed edges for their relationships. A diagram may be drawn one node or edge at a time, nodes before edges. A declarative transformation may therefore execute one commit action at a time, where a commit action either creates a node or assigns an edge. The type of object created, or the value assigned by the commit-action is computed from zero or more objects or values that must be ready for use. We therefore wrap the commit-action up inside a primitive Micro-Mapping to include the input parameters, predicates and computations that influence the commit-action. A primitive Micro-Mapping is therefore similar to the mapping or rule or relation of declarative transformation languages, but is constrained to a single commit-action. The correct sequence of primitive Micro-Mapping invocations can be found by a naive polling scheduler executing each primitive Micro-Mapping once after checking that all the objects and values that the Micro-Mapping depends upon are ready and compatible.

```
Retry loop {              – until finished
    Invocation loop {     – all Micro-Mappings
        Object loops {    – all object/parameter pairs
            Compatibility guard
            Repetition guard
            Validity guard
            Execute Micro-Mapping
            Create repetition memento
        }
    }
}
```

This is hideously inefficient. The many executions, wrapped in at least three loops, with guards and mementos contrast poorly to the simple linear 'loop' nests of imperative programs. Fortunately there are many static analyses that we can perform on a declarative transformation in conjunction with its metamodels to tame the naive polling scheduler.

[2]Declarative transformation authors must learn to express the truth of what must happen rather than the mechanism by which it is done.

B. Global Micro-Mapping Optimizations

1) Retries: Most retries can be avoided by executing Micro-Mappings in a sensible order exploiting producer/consumer relationships between mappings. Where retries are necessary, they can be a result of useful progress rather than naive polling.

2) Invocations: Dead Micro-Mappings can be eliminated, but most Micro-Mappings will normally be required. The invocation loop cannot be eliminated, rather it should be effectively sequenced.

3) Objects: Considering all permutations of all objects with respect to all parameters is unnecessary; the metamodel provides a strong type system that should allow only type compatible permutations to be considered. In particular a Micro-Mapping that produces some type can be followed by a Micro-Mapping that consumes that type.

Once Micro-Mappings are scheduled in a deterministic order, many of the guards can be optimized away and many of the invocation mementos eliminated since only a single invocation is possible.

The above optimizations are global. In this paper we concentrate on the utility of Micro-Mappings and local optimizations that facilitate the overall global optimizations. Global optimization will be described in another paper.

C. Local Micro-Mapping Optimizations

Permuting candidate objects and parameters can lead to very poor performance, typically a two parameter quadratic search, but worse for more parameters. In this paper we will see how Micro-Mapping analysis in conjunction with the metamodel relationships enables most Micro-Mappings to be reduced to a single parameter avoiding the very poor performance.

D. Compound Micro-Mappings

Declarative transformation languages do not require each commit-action to be separately programmed, rather mappings (or rules or relations) aggregate one or more commit-actions as an output object pattern related to an input object pattern. These patterns impose dependencies on the availability of source objects, and guard conditions upon their suitability. In the following example we will see how the failure to distinguish between primitive and compound Micro-Mappings causes trouble for some transformation languages and conversely how recognizing the distinction enables Eclipse QVTd to give good performance.

IV. Micro-Mapping Example

We will consider a very simple example that has some interesting difficulties. We will transform a model comprising a DoublyLinkedList that owns a ring of Elements into another DoublyLinkedList that owns a copied ring of Elements with the order of elements in the ring reversed. The metamodel is shown in Figure 2. Since we use models rather than Java Objects, the complexities of maintaining bidirectional linkages are subsumed by the metamodel bidirectional relationships.

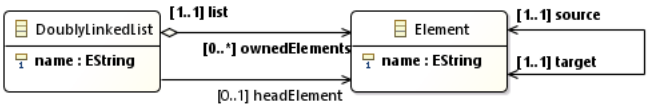

Fig. 2. Example Metamodel - Doubly Linked List.

A. ATL Implementation

The transformation is sufficiently simple to show the full unidirectional implementation in ATL in Figure 3. The transformation requires two rules. list2list declares the mapping from the forwardList input to the reverseList output, populating its name and headElement. element2element populates a reverseElement from a forwardElement. The target to source assignment performs the reversal.

```
module Forward2Reverse;
create OUT : ReverseList from IN : ForwardList;

rule list2list {
  from
    forwardList : ForwardList!DoublyLinkedList
  to
    reverseList : ReverseList!DoublyLinkedList (
      name <- forwardList.name,
      headElement <- forwardList.headElement -- resolveTemp
    )
}

rule element2element {
  from
    forwardElement : ForwardList!Element
  to
    reverseElement : ReverseList!Element (
      name <- forwardElement.name,
      list <- forwardElement.list,          -- resolveTemp
      source <- forwardElement.target       -- resolveTemp
    )
}
```

Fig. 3. Example transformation in ATL.

The exposition is particularly easy in ATL where the implicit resolution of the new-headElement from the old-headElement is done automatically. The commented lines show where an implicit resolution is performed.

Careful study of the transformation reveals that full execution of list2list requires that element2element has previously created the new headElement and that full execution of element2element requires that list2list has previously created the new reverseList. Further study reveals that full execution of element2element requires that another execution of element2element has created its new source which recurses to require that yet another execution of element2element has created the new source's source. The recursion terminates with full execution of element2element requiring that element2element has already created its reverseElement. These dependencies are obscure, cyclic and seemingly insoluble.

B. QVTr Implementation

Figure 4 provides the bidirectional QVTr equivalent of ATL's unidirectional list2list rule. The asymmetric to/from patterns are replaced by symmetric forward/reverse patterns. forwardList.headElement and reverseList.headElement are explicitly co-ordinated by a when clause rather than syntax sugar.

```
top relation list2list {
    enforce domain forward
        forwardList : DoublyLinkedList {
            name = listName : String{},
            headElement = forwardHead : Element{}
        };
    enforce domain reverse
        reverseList : DoublyLinkedList {
            name = listName,
            headElement = reverseHead : Element{}
        };
    when {
        element2element(forwardHead, reverseHead);
    }
}
```

Fig. 4. Example list2list Mapping in QVTr.

C. QVTs Colored Mapping Instance Diagrams

Once a forward execution direction has been selected, the Eclipse QVTr/QVTc tooling provides the UML-like QVTs rendering of the mappings shown in Figures 5 and 6.

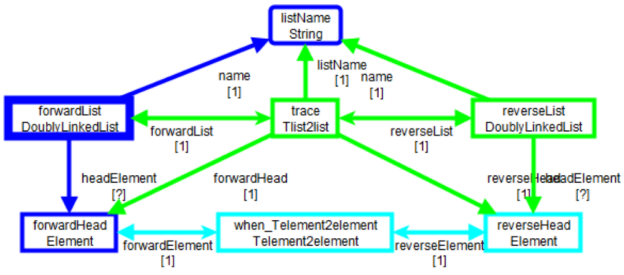

Fig. 5. list2list Mapping in QVTs.

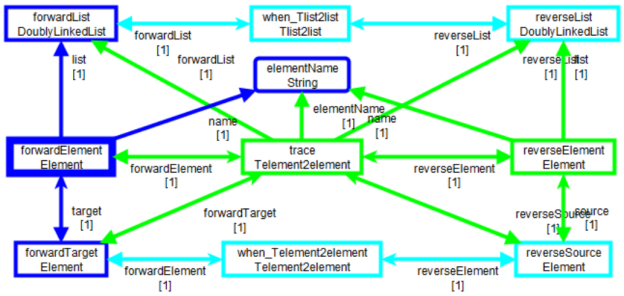

Fig. 6. element2element Mapping in QVTs.

Rectangles represent Class instances with an instance name above a Class name. Rounded rectangles similarly represent shared DataType values.

Edges show directed Properties with name and cardinality. The left hand edge depicts the navigation constraint `forwardHead = forwardList.headElement`. Arrows show something-to-one navigation paths.

Colors show the validity of each part of the mapping.

- BLACK - model elements that are constant

- BLUE - model elements that form part of the input model
- CYAN - model elements required before execution
- GREEN - model elements created by execution

The diagrams are created automatically; only the layout has been enhanced manually. On the left hand side BLUE elements show the input sub-pattern. On the right hand side there is a corresponding output sub-pattern. QVTc and QVTr differ from other transformation languages through the use of explicit model elements to trace the execution of each mapping. There is therefore a column of trace elements directly connected to each other and to the side patterns. Since in OCL, which QVT extends, all unidirectional navigations are bidirectionally navigable, the trace elements are navigable from the sides and so left, trace and right are unified in an overall pattern. In other transformation languages the trace is an implementation detail requiring irregular language constructs to exploit it.

V. The Micro-Mapping Model of Computation

Diagrams or rather graphs of nodes and edges provide a useful way to understand systems and also computations. But as demonstrated by the Ptolemy group, diagrams only become really useful once there is a Model of Computation [5] to define the semantics of information flow along the edges.

A Mapping provides a pattern of constraints that are all satisfied once the Mapping has been executed. It therefore defines the overall truth. In QVTs, colors are used to support intermediate partial truths. The BLACK color identifies compile-time truth. The BLUE is true after input models are loaded. The GREEN is true as a consequence of the truth of BLACK, BLUE and CYAN elements. Partial truths evolve as successively more BLACK then BLUE then CYAN then GREEN elements are resolved.

A Mapping involving more than one GREEN element may not be executable since the multiple GREEN-after-CYAN constraints in one Mapping may deadlock with respect to an inverse ordering in another Mapping.

A. Partitioning, Primitive Micro-Mappings

Primitive Micro-Mappings avoid the deadlock hazard by partitioning the Mapping into multiple primitive Micro-Mappings each with a single GREEN element. This gives a very simple execution semantic; nothing happens until all CYAN elements are available, then execution proceeds uneventfully. When partitioning a Mapping into primitive Micro-Mappings, the chosen GREEN element may depend on other GREEN elements. These other GREEN elements are recolored CYAN since they are prerequisites of the chosen GREEN element. This is rather easy graphically.

B. Local Merging, Compound Micro-Mappings

Partitioning a Mapping such as Figure 5 leads to nine primitive Micro-Mappings to eliminate the deadlock hazards. Primitive Micro-Mappings that share the same dependencies may be merged to form a compound Micro-Mapping without losing the string properties of a primitive Micro-Mapping.

C. Typed Nodes

Each node is typed by a metamodel Class and has an instance name distinguishing its role in the overall pattern.

D. Typed Directed Edges

Each edge is typed by a metamodel Property that defines its name, direction and cardinality, which must be something-to-one. Consequently whenever a source object exists, a target object must form part of the final truth. Wherever the metamodel defines a something-to-one opposite relationship, the opposite edge is added to the Micro-Mapping. Thus at the bottom right of Figure 6 both forwardElement.target and forwardTarget.source are drawn.

E. Inputs and Outputs

The available outputs from the Micro-Mapping are all the GREEN elements (both nodes and edges). A Micro-Mapping has no local knowledge of which GREEN nodes or edges appear in CYAN in another Micro-Mapping.

The potential inputs to the Micro-Mapping are all the BLUE and CYAN elements. A naive implementation may therefore need to invoke the Micro-Mapping for all possible permutations of objects and input nodes. Most invocations will fail through type or connectivity mismatch of a node or edge.

Partial truths involving non-GREEN elements may be falsified when further satisfactory elements are not found. A partial truth involving any GREEN element involves a commit-action; this may not be falsified.

F. Heads

The heads are the smallest set of input (BLUE or CYAN) nodes from which all input nodes can be reached by following directed edges. The heads therefore correspond to the necessary inputs of the Micro-Mapping; all other input elements can be computed from them. In Figure 6 forwardElement is the single head. It is drawn with a thick border to emphasize its importance. We observe that 90% of Micro-Mappings require only a single head and so are amenable to invocation within a loop over compatible input objects. This contrasts with a more naive pattern match that might have attempted a three dimensional search to locate all the compatible forwardList, forwardElement and forwardTarget objects. Use of the metamodel connectivity and cardinality constraints automatically identifies the efficient common sense solution.

G. Computations

This example involves no guards or complicated OCL expressions. For more general purposes, the Micro-Mapping diagram notation is extended with ellipses for iteration or operation calls and computation edges to pass OCL expression results. This ensures that the Micro-Mapping captures all of the declarative mapping. ‘Common subexpression elimination’ occurs for free. OCL operations such as oclIsKindOf, oclAsType and includes are converted directly to edges.

H. Null, Optional and Collection Nodes

The foregoing description stresses the utility of to-one cardinalities. We can generalize this to support to-zero cardinalities by specifying that a node is optional. A null Node may be used to denote the absence of an object. We can also generalize to support to-many cardinalities treating a Collection of objects as a single collected object provided the collection is not dismantled by an OCL computation.

VI. Local Optimization

The graphical Micro-Mapping provides a representation that is much easier to analyze than diverse textual syntaxes. The Model of Computation provides the power to reason about the functionality. GREEN elements identify commit-actions that are performed once the CYAN parts are available to support execution of the commit-actions. CYAN elements therefore inhibit execution until corresponding GREEN elements of other mapping invocations creates them.

It is clear from Figure 5 that list2list has a dependency on an element2element execution since there is a CYAN Telement2element. Similarly element2element in Figure 6 has a dependency on another element2element and a list2list execution. The invocations of element2element can be seen as two instances of Telement2element. One in GREEN named trace represents the successful execution of this invocation. Another in CYAN named when_Telement2element represents the predicate on successful execution of an element2element referenced in a when clause.

A. Speculating

Any dependency-driven attempt to execute the Mappings in Figures 5 and 6 is doomed to fail. list2list cannot execute until element2element has executed for the headElement. element2element cannot execute until list2list has executed. A naive polling scheduler will livelock as it polls in vain for something to execute. A slightly smarter scheduler will deadlock once nothing executes. Yet ATL executes this transformation successfully. How? See Section VI-B.

Our earlier discussion argued that execution of primitive Micro-Mappings with a single commit action is sound and our example demonstrates that Mappings with multiple commit actions (GREEN elements) may be unsound. In this example we have a cyclic dependency that not even primitive Micro-Mappings can avoid.

Figures 7, 8 and 9 show how a sequence of compound Micro-Mappings can execute list2list speculatively and so break the dependency cycle.

1) Speculation Micro-Mapping: The Speculation Micro-Mapping speculates the creation of the trace object wherever the easy BLUE input dependencies are satisfied. This speculated trace object is shown In RED since it has been created without checking its CYAN dependencies.

2) Speculated Micro-Mapping: The Speculated Micro-Mapping has AMBER dependencies for objects that must be provided by Speculation Micro-Mappings and CYAN dependencies for everything else that must be available before the GREEN output-related elements can be created.

Comparing Figure 8 with Figure 5 reveals that a CYAN reverseHead node and associated edges have been omitted at the bottom right. This omission is justified by the observation that reverseHead is a corollary of the element2element Mapping; a successful execution of the AMBER element2element is guaranteed to create the required CYAN element. The GREEN reverseList at the right of Figure 8 is the corresponding list2list corollary.

If the missing CYAN element was added to 8 and its element2element counterpart, the cyclic dependency returns.

3) Edge Micro-Mapping: Once Speculation and Speculated Micro-Mappings have mediated the solution to the dependency cycle, zero or more further Edge Micro-Mappings can then provide the residual GREEN edges once the nodes at their ends are available.

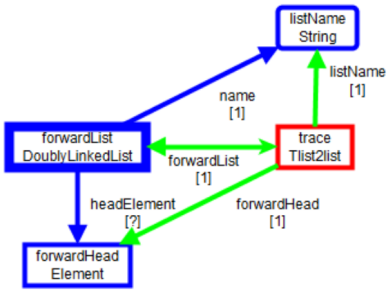

Fig. 7. list2list speculation compound Micro-Mapping in QVTs.

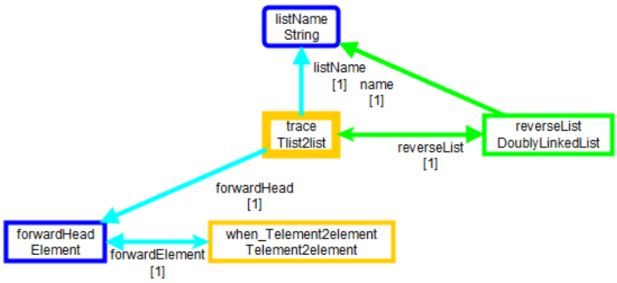

Fig. 8. list2list speculated compound Micro-Mapping in QVTs.

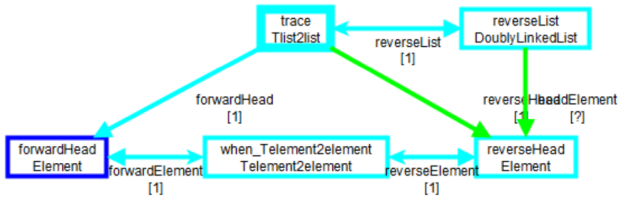

Fig. 9. list2list reverseHead edges compound Micro-Mapping in QVTs.

4) Failure: The speculation may fail, in which case Figures 8 and 9 do not execute and no output model elements are created; the failed speculation is only visible as a still-speculating trace element in the trace model.

B. ATL Execution

We can now understand how ATL successfully executes the example. ATL does not perform a dependency analysis and so does not detect the difficulty. ATL just executes its rules in two stages. First all the new objects are created, then all the inter-object references are populated. ATL has therefore effectively partitioned the two list2list and element2element rules into four mini-rules list2list-create + list2list-assign, and element2element-create + element2element-assign. Executing the create mini-rules before the assign mini-rules works. This is similar to the speculation/speculated partitioning above. However ATL's partitioning is pragmatic and the successful execution fortuitous. If a more complex guard invalidates the premature creation of outputs, ATL is unable to roll-back its invalid creations.

C. More Model of Computation Facilities

Space permits only a very brief summary of the other benefits of the analyses facilitated by the Micro-Mapping Model of Computation.

1) Multiple Heads: Micro-Mappings with multiple heads require multiple inputs and consequently incur invocation difficulties and non-linear execution performance. Analysis of the to-many metamodel relationships allows most multiple head Micro-Mappings to be realized by a single head Micro-Mapping with local rather than global loops for the other heads. This typically gives linear performance with respect to the output model size. Only genuinely Cartesian problems need incur Cartesian costs.

2) Global optimizations: The simple relationship between GREEN creation and CYAN use facilitates powerful global analysis. Some of these were briefly mentioned in Section III-B

3) Static Scheduling: We find that most Micro-Mappings can be statically scheduled and so only a small number incur dynamic scheduling overheads at run-time.

4) Global Merging: The local merging in Section V-B is heavily constrained by the requirement for a mutually shared partial truth. Once a static schedule has been established, merging can be much more aggressive. Micro-Mappings may be merged into their invokers. Predicates guaranteed by an invoker can be pruned from an invoked Micro-Mapping.

5) Incremental Execution: Execution of Micro-Mappings can be persisted as an evaluation graph comprising

- input nodes for each BLUE input element
- invocation nodes for each Micro-Mapping invocation
- BLUE-BLUE dependency edges between invocation nodes and input nodes
- CYAN-GREEN dependency edges between consuming invocation nodes and producing invocation nodes

Each invocation node holds its prevailing GREEN state and is aware of the nodes that consume it. The graph can be selectively re-evaluated to propagate BLUE input changes efficiently.

VII. Results

In Figure 10 we plot[3] the performance of the DoublyLinkedListReversal transformation using a variety of transformation engines and a couple of manual implementations[4]. The plot demonstrates the scalability and the underlying tooling efficiency.

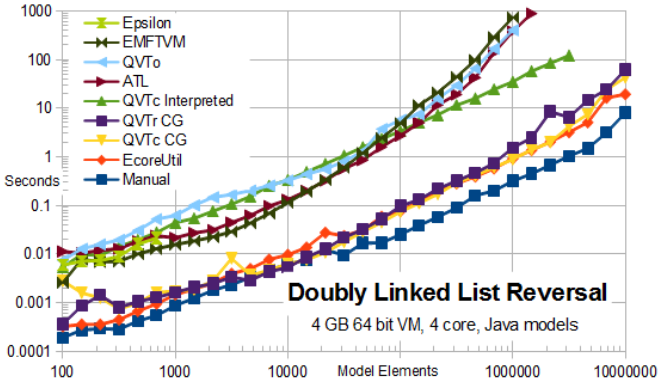

Fig. 10. Performance of the Doubly Linked List Reversal transformation.

Between 1000 and 50000 model elements, the top four and a half lines show superior performance for ATL and EMFTVM (the improved ATL executor) compared to Eclipse QVTo and Interpreted QVTc. However for larger models quadratic performance, probably as a consequence of requiring a linear search of the trace to resolve output/input correspondences affects ATL, EMFTVM and QVTo. Interpreted QVTc remains linear and so at 1000000 model elements, it is 10 times faster. The performance of Epsilon for large models cannot be characterized since its half line terminates abruptly; Epsilon resolves data dependences recursively and so runs out of stack for lists longer than about 1000 elements.

The lower four lines show direct Java results, the upper two for Eclipse QVTc and QVTr, and a further two manually coded reference implementations, one using EMF extensively as a consequence of a EcoreUtil copy, and another using EMF sparingly in manual code.

QVTc and QVTr performance is almost identical since both execute the same schedule for very similar mappings. The QVTr performance is slightly worse than QVTc since the naive auto-generated trace is larger; an optimization of the auto-generated trace model will make QVTr better than QVTc. The performance almost matches EcoreUtil. Further optimizations should get much closer to the manual performance. An improved object representation could enable QVTr to outperform the manual implementation.

(The interpreted QVTc or QVTr performance may improve quite significantly once some of the static analyses for the code generated approach are exploited.)

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

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

The results demonstrate that an efficient declarative schedule can be derived automatically for a difficult dependency problem.

The results also demonstrate, unintentionally, the benefit of QVTc' explicit trace model and the consequent linear performance when an appropriate cache is synthesized for the unnavigable opposite accesses. The quadratic performance of ATL, EMFTVM and QVTo highlights an implementation deficiency that can be remedied at the expense of some extra working memory. Ensuring that the extra memory cost is modest and the additional execution time is small, probably requires similar static compile-time or load-time analyses to those performed by Eclipse QVTd.

The code and results for this example are available in the tests/org.eclipse.qvtd.doc.exe2016.tests plugin of the https://git.eclipse.org/r/mmt/org.eclipse.qvtd GIT repository. Significant bugs in Eclipse QVTd were fixed to support this example. These fixes should be available in the Oxygen M2 milestone build in mid September 2016.

VIII. Related Work

Scheduling and particularly static scheduling has been a rich research topic with provision of optimized schedules recognized as a computationally hard problem. The many works of the Ptolemy group [1] that build upon [4] has been a strong background influence. However the appreciation that metamodels impose such strong constraints that sensible schedules can be produced rapidly for declarative transformations appears to be novel.

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 QVTs graphical Micro-Mapping and its Model of Computation may begin to bridge the gap between these two communities. The automated coloring in QVTs is inspired by Henshin's [2] manual 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 [7] for heterogeneous transformations.

Active Operations [3] 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.

IX. Conclusion

We have introduced the Micro-Mapping Model of Computation and shown how it supports efficient declarative schedules for Eclipse QVTc and QVTr.

We used the Micro-Mapping Model of Computation to demonstrate the need for speculative creation of trace objects.

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

We have introduced the first implementation of the QVTc specification.

We have presented the first results for a QVTr implementation using a direct code generator.

We have mentioned some future works. Many more optimizations to do.

Acknowledgment

The authors would like to thank Adolfo Sanchez-Barbudo Herrera, Horacio Hoyos Rodriguez, Dimitris Kolovos and Richard Paige for helpful discussions about declarative scheduling approaches. Horacio prepared some of the results and prototyped some of the scheduler algorithms.

References

[1] Bhattacharyya, S., Murthy, P., Lee, E.: Software synthesis from dataflow graphs, Kluwer Academic Press, Norwell, MA, 1996

[2] 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.

[3] 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

[4] Lee, E., Messerschmitt, D.: Synchronous data flow, Proceedings of the IEEE, 1987

[5] Lee, E., Sangiovanni-Vincentelli, A.: Comparing models of computation, Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design

[6] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3. OMG Document Number: ptc/16-06-03, June 2016.

[7] 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

[8] Willink, E.: Yet Another Three QVT Languages. ICMT 2013 (2013)

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

[10] Eclipse EMF Project. https://projects.eclipse.org/projects/modeling.emf.emf

[11] Eclipse OCL Project. https://projects.eclipse.org/projects/modeling.mdt.ocl

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