QVTs: A TGG-like graphical representation for efficient Declarative Model to Model Transformation scheduling.

Edward D. Willink

Willink Transformations Ltd., Reading, UK,
ed _at_ willink.me.uk

Abstract. Graph Grammars have been providing an increasingly sound foundation for Graph Transformation for nearly 50 years. 20 years ago, the enthusiasm surrounding the Model Driven Architecture motivated the OMG to start standardization of the QVT Model to Model transformation language. Sadly, the M2M work ignored the considerable overlap with GT and is much the worse for it. This paper shows how a graphical approach can resolve many of the M2M challenges and hopefully start to bridge the gap between the GT and M2M communities.

Keywords: M2M, Model to Model Transformation, Graph Transformation, QVT, Scheduling, TGG

1 Introduction

Within the academic community, the work on Single Push Out and Double Push Out established the principles of Graph Transformation [1]. Triple Graph Grammars [2] leverage this foundation to provide a formulation that is more practical for model transformation. A TGG implementation is available via the Eclipse Henshin project [13].

Within the industrial community, the utility of models was recognized leading to the Object Management Group's Model Driven Architecture [4]. Model transformation was recognized as important and so the OMG requested proposals for a model Query/View/Transformation language. The eight initial submissions eventually merged into a triple language specification [5]; one imperative language, QVTo (Operational Mappings), and two declarative languages, QVTc (Core) and QVTr (Relations). 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 original QVTr implementations, ModelMorf and Medini QVT have remained proprietary and have not prospered. The Eclipse QVTd project [14] has provided QVTc and QVTr editors for many years, but it is only relatively recently[1] that any execution functionality has been available.

[1] As part of the Eclipse Neon release in June 2016.

Unfortunately the industrial community ignored the significant overlaps between QVTr and TGG. The lack of accurate or detailed QVTr publications has made it hard for the academic community to investigate the overlap in detail.

In this paper we outline the QVTs[2] graphical representation that facilitates visualization of the Eclipse QVTd schedule synthesis. In Section 2, we introduce a running example and highlight characteristics of navigable opposites and declarative execution. The QVTs representation is described in Section 3 and exploited in for static scheduling in Section 4. Section 5 presents related work and Section 6 concludes.

2 Background

2.1 Running Example

For most of the examples in this paper we will re-use [11] a very simple, but surprisingly troublesome, transformation that copies of a cyclic doubly linked list with reversed element order. The UML metamodel is shown in Fig 1.

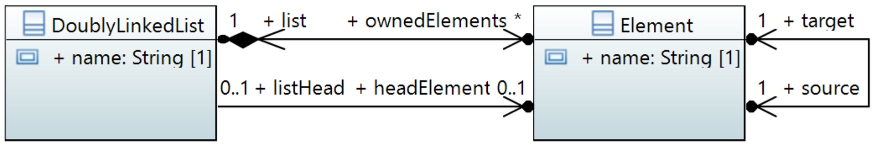

Fig. 1. Doubly Linked List UML Metamodel.

There are just two classes, a `DoublyLinkedList` that has many `Elements`, composed by the `DoublyLinkedList::ownedElements` relationship. A doubly-linked-list of `Element` is established by the `Element::source`/`Element::target` bidirectional relationship. A particular `Element` is distinguished as the list head via the `DoublyLinkedList::headElement` relationship.

2.2 Navigation and Opposites

The Eclipse QVTd tooling leverages a little known capability of OCL; all associations are unconditionally navigable at analysis time. In UML, the role names on the ends of Associations distinguish alternate navigations from a common source. The arrows on the ends of Associations identify which ends are navigable at run-time. OCL is specified against the metamodel and so all role names are accessible in OCL. OCL provides mechanisms for defining and disambiguating omitted role names. As an extension of OCL, QVT can navigate a relationship such as `DoublyLinkedList::headElement` in either direction.

[2] QVTs is not part of the OMG specification

2.3 Declarative Transformation Execution Schedule

For a traditional functional/imperative/procedural program, a call tree specifies the order in which each stage of execution occurs. This should ensure that the inputs of each stage are provided by a previous stage. Unfortunately in complex programs, particularly those with global state or multiple threads, this does not always happen. The result provides for exciting debugging opportunities.

A declarative transformation eliminates these hazards by only specifying what must happen in each stage[3]. It is the responsibility of the execution engine to ensure that stages execute in an appropriate order. Naively, each stage may be attempted for arbitrary elements in an arbitrary order. If inputs are not available, the attempt must be aborted and retried later. Fig 2 shows this.

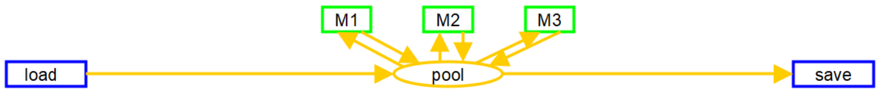

Fig. 2. Naive Declarative Transformation Execution.

A `load` stage loads all the input model elements into a `pool`, from which attempted executions of the `M1`, `M2`, `M3` stages are repeated until execution completes. Then the `save` stage executes. The naive repetition of stages and permutation of input elements may easily lead to quadratic or worse execution performance. This performance is aggravated by the costs of keeping track of which inputs are ready, and which executions need a retry.

The strong typing of a declarative transformation provides ample opportunities for analysis so that we can hope to find an efficient schedule such as Fig 3.

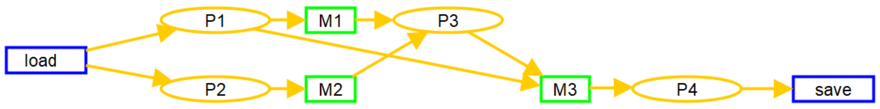

Fig. 3. Smart Declarative Transformation Execution.

The `load` now separates input elements into distinct `P1`, `P2` pools whose element types suit the downstream `M1` and `M2` consumers. These produce to a distinct `P3` pool for `M3`.

The smart schedule is a directed acyclic graph amenable to efficient scheduling. Given an accurate analysis, all `M1` and `M2` executions occur before any `M3`; no time need be wasted attempting `M3` before `M1` and `M2` have completed. Consequently no time need be consumed testing whether `M1` and `M2` have completed.

[3] a stage is variously a mapping, partition, relation or rule

Finding the smart schedule is the motivation for the QVTs representation described in this paper.

The need to find a smart schedule is particularly important for QVTr, since QVTr lacks the imperative workarounds that some declarative languages use to solve too-hard problems. For complex transformations, a first pass may create the output elements in a tree structure and then a second pass may elaborate the tree to form a more complex graph structure. Sequencing these passes requires the programmer to understand and use some limited imperative capabilities within the declarative transformation language. QVTr is more powerful; the programmer is not required to understand the practicalities of the execution schedule, rather the QVTr analysis or run-time must solve the problem.

2.4 Eclipse QVTd Tool Chain

A simple tool using a naive execution schedule could attempt an interpreted execution using little more than a transformation language loader and interpreter. Eclipse QVTd provides an efficient execution using the tool chain shown in Fig 4.

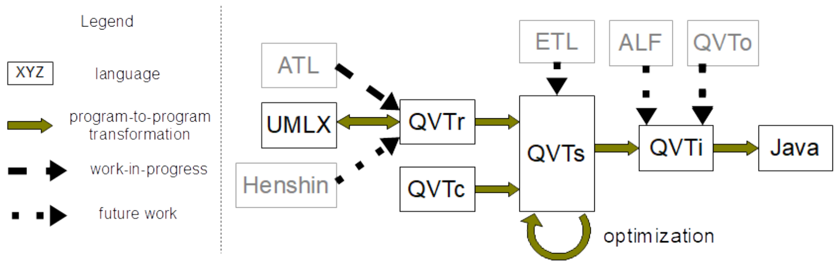

Fig. 4. QVTd Tool Chain.

The QVTr language support is followed by a transformation to the QVTs graphical representation. After optimization this is converted to an assembler-like QVTi imperative representation. This can be executed directly by an interpreter or code generated using an extension of the Eclipse OCL Java code generator to give pure Java code. The tool chain is open for re-use by other languages. UMLX [10] is fully supported, ATL [12] partially.

3 The QVTs representation

In this paper we elaborate the auto-generated QVTs graphical representation and its many similarities to TGG. The most obvious similarities between TGG rules and QVTs mappings are:

– diagrams based on UML Object Diagrams

– patterns with nodes for Class-typed variables, edges for typed-relationships
– three swim lanes: left : correspondence(TGG), middle/trace(QVT) : right
– metamodel typing for all three lanes, with accurate multiplicities
– no deletion
– green coloring for additions (also ++ annotation for TGG)
– positive Constraints, rather than Negative Application Conditions

The main practical difference is that TGG Rules are manually created programming artefacts, whereas QVTs mappings are auto-generated visualizations[4] derived from QVTr Relations.

The most fundamental difference is in the use of OCL. A TGG Assignment or Constraint node has an opaque embedding of an OCL expression. QVTs has nodes for the variables, and edges for the navigation, of an OCL expression AST. QVTs reifies an Attribute as a DataType-typed node and a Property-typed edge.

Other differences are:

– TGG has a shortform for endogenous (often in-place) transformations
– QVTs resolves in-place[5] by a post-synthesis optimization / synchronization
– TGG rules can be un-directional, QVTs mappings are for a chosen direction
– A negative link in TGG is a (positive) link to a `null` constant in QVTs
– QVTr auto-generates the middle metamodel with accurate opposites

Since the QVTs is auto-generated, a variety of rendering capabilities can be used to provide an enhanced visualization.

– QVTs uses multiple colors to show lifecycle
– QVTs arrows show to-one(-or-zero) navigability
– QVTs line styles show nullability and distinguish matches/expressions

3.1 QVTs Synthesis - QVTr to QVTs conversion

Space limitations permit only the observation that the conversion from QVTr to QVTs normalizes QVTr high level facilities and syntax sugar to a simpler form.

– Inherited Class matching
– Inherited Property matching
– Relation Overriding
– Exogeneous and Endogeneous Transformations
– Top and Non-Top Relations
– Keys
– Trace Metamodel Synthesis

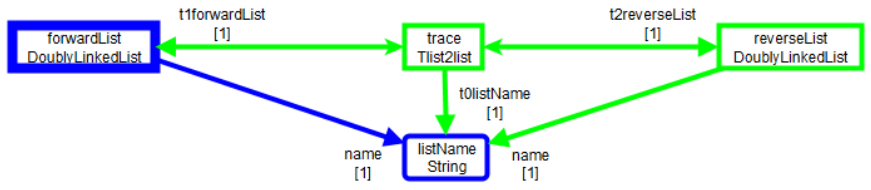

Fig. 5. Trimmed list2list Mapping.

3.2 Trimmed list2list Mapping

The two mappings for our example are shown in Fig 6 and Fig 7. The `list2list` mapping is the TGG-Axiom. It is simplified in Fig 5 to ease discussion.

The three columns and top row are essentially identical to TGG. On the left an input pattern variable named `forwardList` of type `DoublyLinkedList` is transformed, via a middle variable named `trace` of type `TList2List`, to an output pattern variable named `reverseList` of type `DoublyLinkedList`.

The green coloring identifies elements created by the mapping in similar way to green coloring and ++ annotations in TGG. The blue coloring identifies elements that are present in the input model.

The labels and multiplicities on the arrows show the role names and multiplicities of the relationships in the metamodels.

QVTs arrows are different to UML and TGG. A QVTs arrow head identifies a direction in which the navigation yields precisely zero or one element. A triangular arrow as from `trace` to `forwardList` identifies a direction that is available at run-time. The curved arrow as from `forwardList` to `trace` identifies an opposite navigation that, if used, may require additional run-time support.

Below the top row, we have an additional DataType-typed variable named `listName` and of type `String`. It is an Attribute. It is drawn as a rounded rectangle to distinguish it from a Class-typed rectangle. DataTypes exist by value, rather than as objects and so may be shared/duplicated at the convenience of the implementation. We therefore draw navigation arrows from each of the objects for which `listName` is their name; there is no need for a separate constraint node such as `reverseList.name = forwardList.name`. Navigation from a DataType-typed node is of course impossible.

3.3 Full list2list Mapping

The full mapping for `list2list` is shown in Fig 6.

The additional bottom row transforms the `forwardList.headElement` to the `reverseList.headElement`. The `headElement` is optional, and so may be `null`. This is denoted by the `[?]` multiplicities, dashed edges and borders. Execution of a mapping requires solid nodes and edges to match; dashed nodes and edges

[4] the content is auto-generated, the aesthetic layout is currently manual
[5] in-place support is future work

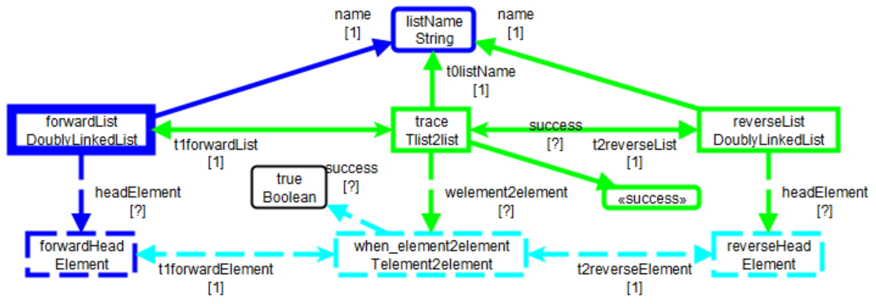

Fig. 6. Full list2list Mapping.

are ignored for matching purposes, but are used by additional predicates and result computations.

Assignment of `reverseList.headElement` is not possible until `reverseHead` Element has been created by the `element2element` mapping whose execution is reified by the `when_element2element` variable of type `TElement2Element`. The cyan coloring identifies nodes and edges that must be available from somewhere else before this mapping can execute.

The full mapping also shows the bookkeeping `success` variable. The green arrow to the green `«success»` node causes a `true` or `false` value to be assigned to `trace.success` according to the successful execution of the mapping. `«success»` is unusual in that it is the only green element that it is assigned in the case of a failure. This enables other mappings to react to the success or failure. This can be seen in the `when_element2element.success` cyan navigation to the `true` black compile-time constant; the `list2list` mapping fails if the `when_element2element` fails.

3.4 Full element2element Mapping

The full mapping for `element2element` is shown in Fig 7.

It is structurally very similar to `list2list`. `forwardElement` is copied to `reverseElement` via a `trace` element of type `TElement2Element`. Copies of the `forwardElement.list` and `forwardElement.target` elements are assigned to `reverseElement.list` and `reverseElement.source`. In each case, the cyan copy created elsewhere is located by an opposite navigation from the blue source to its cyan trace and then by forward navigation to the cyan copy. Only the success of `when_element2element` needs checking since `list2list` has no independent failure mechanism. Note that since `DoublyLinkedList::ownedElements` is a one-to-many relationship, it is not shown on the diagram. A unique set of object matches is available by traversal from the `forwardElement`. There is no such similar unique set available from the `forwardList`.

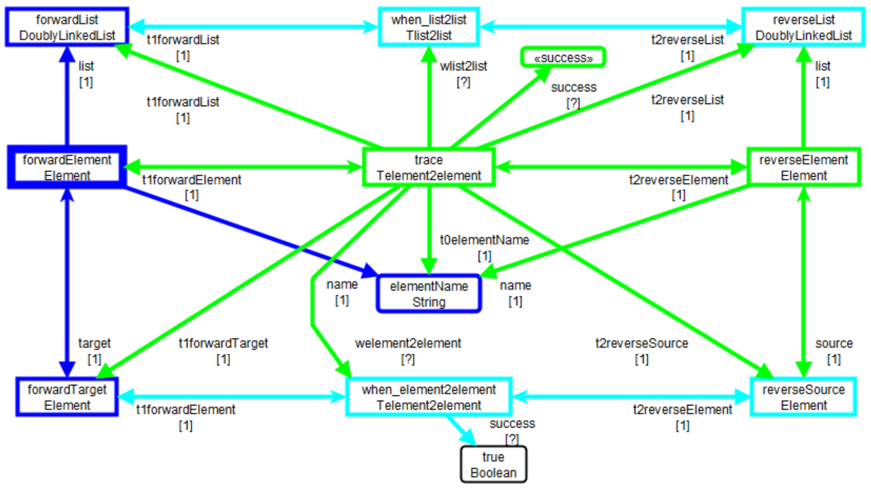

Fig. 7. element2element Mapping.

3.5 Local Mapping Analysis

The line styles and colors show the results of some minor analyses.

Heads Analysis The result of the more significant heads analysis is shown by the thick border of `forwardList` in Fig 6 and `forwardElement` in Fig 7. The set of heads of each mapping is the smallest set of non-green nodes from which all nodes can be reached by traversing to-one(-or-zero) relationships. For nearly all mappings, just like the examples, a set of one node is sufficient.

A single head provides the start of a one-dimensional matching search strategy; for each `Element` model element we must attempt the `element2element` mapping. The full pattern match can be found in constant time following the to-one(-or-zero) relationships. This is clearly much more efficient than a super-naive attempt at a seven-dimensional permutation of all possible model elements against the seven blue and cyan nodes in `element2element`.

The diagram therefore shows that we can navigate from the `forwardElement` to precisely one `trace` using an inferior navigation path and from there to precisely one `reverseElement` using a good navigation path.

Producer/Consumer Analysis The green coloring clearly identifies nodes and edges that are created/produced by the mapping execution. Similarly blue and cyan coloring identifies nodes and edges that are consumed. The blue elements come from the input model and are therefore available just as soon as the input model loading completes. The cyan elements are more troublesome; they come form somewhere else. This gives the simple scheduling policy that consumed cyan elements must be computed before green elements can be produced.

The `list2list` mapping produces 2 Class-typed nodes, 1 DataType-typed node and 7 edges (green). It consumes 2 Class-typed nodes and 3 edges (cyan).

Looking at `list2list` we see that there is a green `TList2List` and a cyan `TElement2Element`, therefore `list2list` cannot execute until the appropriate `element2element` has executed. Similarly `element2element` has a green `TElement2Element` and a cyan `TList2List`, therefore `element2element` cannot execute until the appropriate `list2list` has executed. At compile-time, without knowledge of the appropriate instance, the static schedule must wait for all instances. There is a cyclic dependency; a deadlock.

Partitioning The `list2list` / `element2element` deadlock is easily broken by using multiple passes. A first pass creates the new `DoublyLinkedList` and `Element` elements, a second pass assigns the `DoublyLinkedList.headElement`. We can minimize the likelihood of deadlocks by partitioning each mapping into atomic micromappings with a single green element per micromapping. This proves to be excessive and so we partition into a number of partitions each tackling a separate 'theme':

- activator: to create the green trace element and green edges to input elements
- local: predicate to verify that cyan elements are locally satisfied
- global: predicate to verify that cyan elements are globally satisfied
- speculated: to create the green output node(s) and assign trace success
- per-edge: for each remaining green edge to a green output node

OCL Expression Analysis The list example is too simple to demonstrate the use of OCL within QVTs. We therefore use a mapping from the Families2Persons example to show OCL expressions in Fig 8.

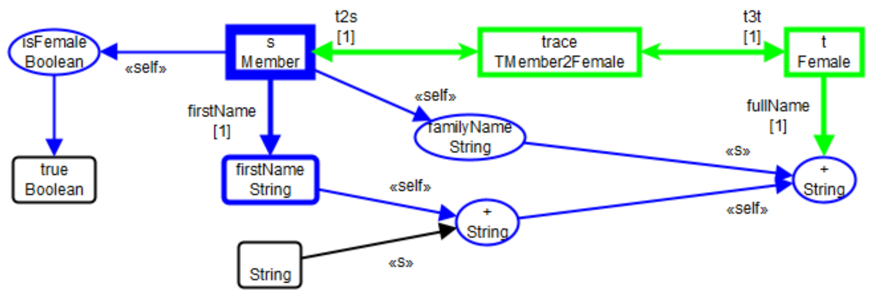

Fig. 8. Simple Expressions Example.

The matched part of the mapping is shown by the creation of two green `trace` and `t` nodes from the blue `s` node.

At the left an operation call to the `isFemale` query using `s` as its «`self`» input is shown using an ellipse. The output is consumed by the `true` constant. The overall truth of the mapping is only satisfied when everything is true, therefore if `isFemale` returns false the match fails. The graphics shows the constraint `s.isFemale() = true`.

In the lower part of the figure we see the string concatenation for the assignment `t.fullName = s.firstName + ' ' + s.familyName()`.

OperationNodes and ArgumentEdges are shown with thin line-styles. Black coloring is used for compile time constants. Blue coloring is used for computations that can be made as soon as the input model is loaded. Only the actual assignment edge for `t.fullName = ...` is green. The ArgumentEdges are annotated with the receiving parameter name such as «`self`» to eliminate ambiguity. These nodes and edges are really just a visualization of the OCL AST. The part of the ATL2QVTr transformation shown in Fig 9 has many more complex usages.

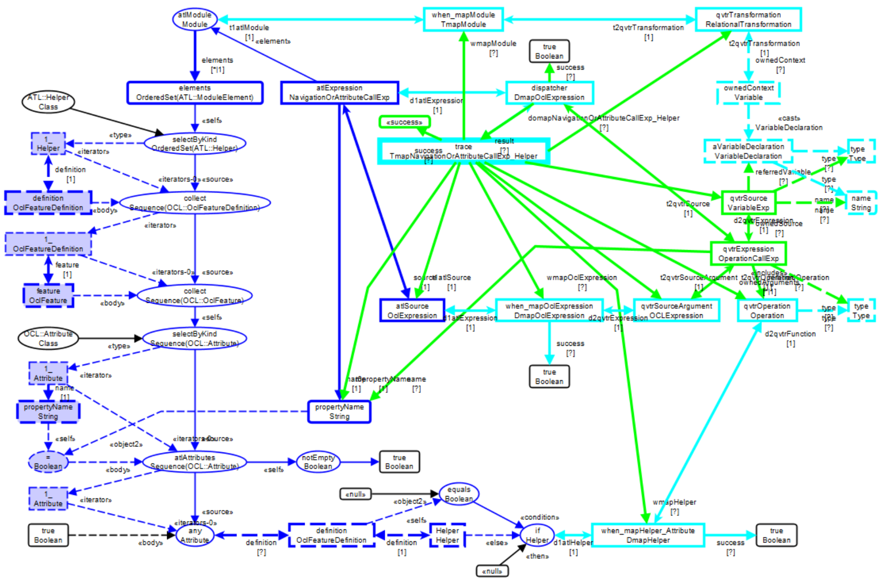

Fig. 9. Complex Expressions Example.

The upper part of the left hand two columns evaluate

```
atlAttributes = atlModule.elements
  ->selectByKind(atlMM::Helper)._definition.feature
  ->selectByKind(atloclMM::Attribute)->select(name=propertyName)
```

This demonstrates the use of DataType-typed Collection nodes and IteratorNode variables in body expressions. It also demonstrates that node sharing in the graphical representation corresponds to Common Subexpression Elimination in traditional tooling.

3.6 Operation Analysis

OCL expressions may call operations, which may involve navigations and consequently an operation may fail if its execution is premature. Operations have no mechanism for aborting and retrying later and so it is the callers responsibility to anticipate the navigations and guarantee that the execution is not premature. This guarantee requires a deep analysis of each operation call. Additional consumer dependencies capture the analysis result.

A straightforward analysis of the potential dependencies of each OCL navigation must take a very pessimistic view of the dependencies for utility functions whose argument types are high up polymorphic inheritance trees. These pessimistic dependencies can lead to unnecessary dependency cycles. A more precise analysis specializing each OCL call expression to the known calling types can avoid these cycles.

4 QVTs Exploitation - Static Scheduling

The QVTs representation analysis supports derivation of an efficient schedule.

4.1 Global Analysis

The cyan consumer analysis identifies which nodes and edges are consumed by each partition. These can be joined to the corresponding green producers by a pool for the type of each node or edge as shown in Fig 3. Additionally the heads analysis identifies which consumers require the corresponding node to be passed-by-value. All other consumptions can be computed by navigation from a head. We therefore refer to these additional consumptions as passed-by-existence.

The global consumer/producer analysis for our simple two mapping example is shown in Fig 10.

The blue mapping at the top is the auto-generated loader responsible for loading the input model and distributing the relevant model elements to suitable pools.

The two orange ellipses are the input pools, one for the `DoublyLinkedList` element and the other for the `Element` elements. The incoming arrows show that the elements are produced by the loader.

The two green rectangles are the two unidirectional mappings[6] each consume a `DoublyLinkedList` or `Element` element via a solid incoming orange arrow.

In addition to consuming model element node instances from the input model, each mapping execution also consumes edge instances. The pools of

[6] the *reverse* suffix denotes the chosen direction of QVTr execution

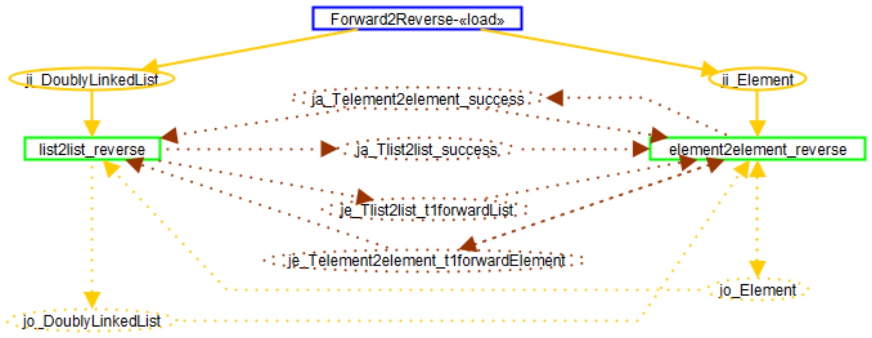

Fig. 10. Raw Producer/Consumer Dependency Graph.

consumed/produced edges are shown using brown ellipses. The top brown ellipse shows that the `Telement2element::success` property is produced by the `element2element` mapping and consumed by both mappings.

The `list2list` mapping has four incoming arrows indicating that an element must be available in each of four source pools to support a successful execution. The solid arrow passes the `DoublyLinkedList` element by value. The remaining arrows are dotted to indicate pass-by-existence.

The diagram shows many cycles between the two mappings. An efficient purely static schedule is not possible.

4.2 Partitioned Schedule

Fig 11 shows the more complex dependency graph that results after partitioning. The graph is topologically similar to Fig 10 with the partitioned `list2list` mapping replaced by 5 partitions on the left hand side and the partitioned `element2element` mapping replaced by 6 partitions on the right hand side.

At the top we see the activators have consumed their input model element and produced a corresponding trace element which is consumed by the local and other partitions.

At the bottom we see the speculated and edge partitions with inputs, outputs and no problematic cycles. Introducing a separate partition, that can be performed once everything else is ready, has resolved the cycle involving the assignment of `DoublyLinkedList::headElement`.

Many of the dependencies use a dashed style. This is a much stronger version of pass-by-existence. The value is guaranteed to exist, the consumer can therefore use it without checking. In contrast the dotted line only indicates that the value may-be available; the consumer must test for readiness and arrange to retry until the required value is ready. Dashed lines are therefore efficient at run-time. Dotted lines incur run-time overheads for retries. For both dotted and dashed pools, no value is actually passed and so there is no need to provide run-time

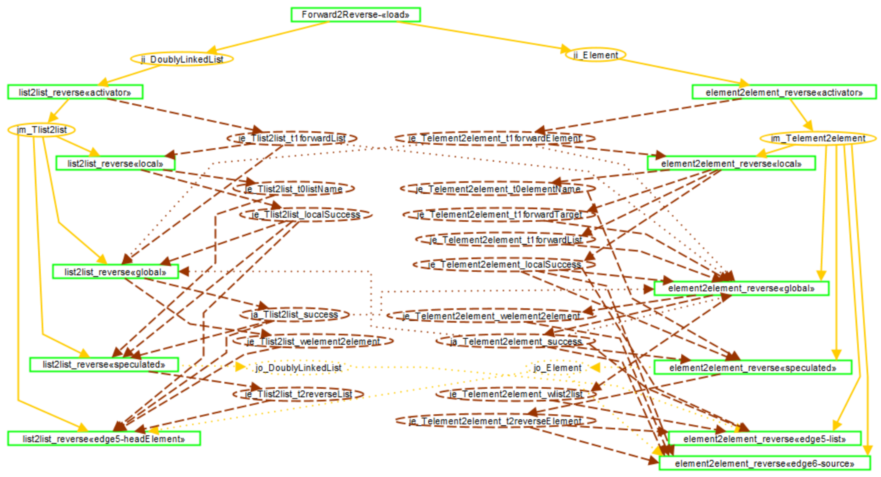
Forward2Reverse-«load»
ji_DoublyLinkedList
ji_Element
list2list_reverse«activator»
element2element_reverse«activator»
jm_Tlist2list
je_Tlist2list_t1forwardList
je_Telement2element_t1forwardElement
jm_Telement2element
list2list_reverse«local»
element2element_reverse«local»
je_Tlist2list_t0listName
je_Telement2element_t0elementName
je_Tlist2list_localSuccess
je_Telement2element_t1forwardTarget
je_Telement2element_t1forwardList
je_Telement2element_localSuccess
list2list_reverse«global»
element2element_reverse«global»
ja_Tlist2list_success
je_Telement2element_welement2element
je_Tlist2list_welement2element
ja_Telement2element_success
list2list_reverse«speculated»
jo_DoublyLinkedList
jo_Element
element2element_reverse«speculated»
je_Tlist2list_t2reverseList
je_Telement2element_wlist2list
je_Telement2element_t2reverseElement
list2list_reverse«edge5-headElement»
element2element_reverse«edge5-list»
element2element_reverse«edge6-source»

Fig. 11. Partitioned Producer/Consumer Dependency Graph.

support; it is a virtual pool. It is the consumer that computes the passed-by-existence value and must check whether the computed value is ready.

4.3 Speculation

Further examination of Fig 11 shows that we still have a cycle involving the global partitions and the success pools. This is because the cyclic list in this simple example is really troublesome; the transformation can only proceed if the mapping executions for each list element collaborate. When acyclic structures such as composition trees are involved, collaboration can be guaranteed. For genuinely cyclic structures a global mediator must be invoked at run-time.

4.4 Cycle Elimination

The dependency graph is only mostly acyclic, so we need a solution for the cyclic parts. This proves to be rather simple. Cycles are easily identified as the intersections of transitive predecessors and successors. Each cycle can be wrapped up rather like a Russian doll. The result is acyclic with respect to the dependency edges; cycles are localized to the Russian doll node that may have a further acyclic dependency graph inside.

4.5 Pass Allocation

The acyclic directed dependency graph provides an ordering of the partitions so that we may allocate the partitions to sequential passes. Typically pass 0 for the loader, pass 1 for the activators, pass 2 for ... One or more pass numbers are allocated to each pool in accordance with the pass numbers in which each producer may add content.

These pass numbers are shown in the the Fig 12 which also shows the results of some optimizations.

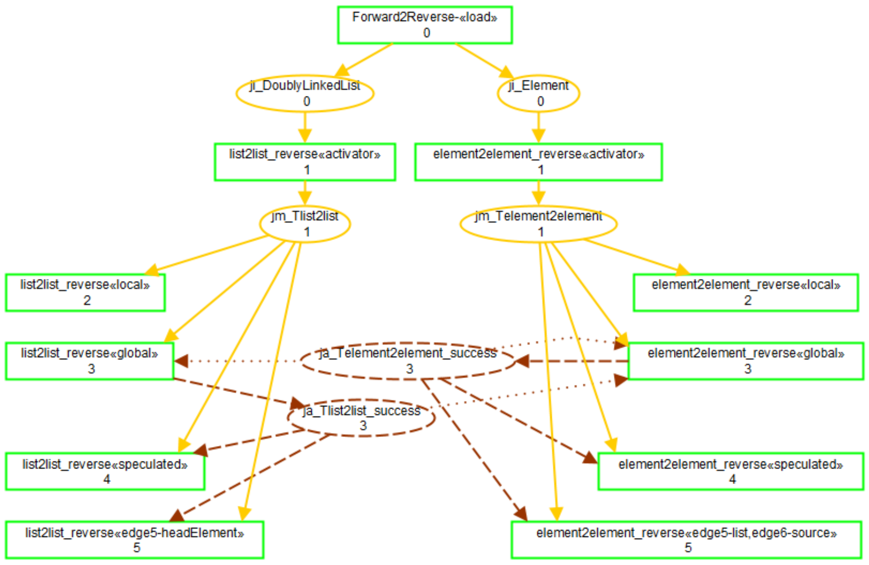
Forward2Reverse-«load»
0
ji_DoublyLinkedList
0
ji_Element
0
list2list_reverse«activator»
1
element2element_reverse«activator»
1
jm_Tlist2list
1
jm_Telement2element
1
list2list_reverse«local»
2
element2element_reverse«local»
2
list2list_reverse«global»
3
ja_Telement2element_success
3
element2element_reverse«global»
3
ja_Tlist2list_success
3
list2list_reverse«speculated»
4
element2element_reverse«speculated»
4
list2list_reverse«edge5-headElement»
5
element2element_reverse«edge5-list,edge6-source»
5

Fig. 12. Final Producer/Consumer Dependency Graph/Schedule.

Virtual pools whose pass numbers are smaller than all the pass numbers of all their consumers are inherently safe and are removed. Only pass 3 involving the global speculation remains indicating a need for run-time support.

Sibling partitions sharing the same failure mechanisms are merged.

We also see that the solid orange arrows provide a very simple call-tree in which each partition has a single input avoiding the need for any complex multi-dimensional dispatch. The partitions for each pass can be concurrently executed one pass at a time, with the exception of pass 3 that requires a global mediation to resolve its instance cycle.

5 Related Work

A catalog of optimizations has been proposed [3] for Triple Graphs; domain driven applicability, caching/indexing, static analysis, incremental execution. Many of these come for free in the current work.

Many papers refer to QVT, but few provide insight because the only exposition of QVT is its flawed specification. Too many authors have tended to regard an OMG specification as the work of the gods, whereas in practice a specification is the best endeavors of a small, sometimes very small, team of busy, underfunded, not always omniscient, contributors. Revised specifications are necessary to remedy defects as well as accommodate unforeseen use cases. The QVT specification is unique. It is the first time the OMG has standardized research work; it really should have started at version 0.0 to emphasize this.

The QVTr language is 'formalized' by RelToCore, a hard to read transformation written in QVTr from QVTr to QVTc. QVTc is 'formalized' using semi-formal language. Unfortunately the QVTc semi-formalization is incomplete and RelToCore neglects important concepts such as Collection matches and Relation overriding. RelToCore comprises over 1000 lines of the new QVTr language. RelToCore was written without the aid of language support tooling. It is therefore not surprising that when Eclipse QVTd started to provide QVTr editors, many syntactic and semantic problems were uncovered. Clearly the 1000 lines of QVTr that formalize QVTr have never been exercised.

Authors such as Stevens [8] have endeavored to understand RelToCore and the QVTc semi-formalization and concluded that QVTr and QVTc are inconsistent despite QVTr being just a transformation away from QVTc. Westfechtel [9] has endeavored to provide a bidirectional implementation of Families-to-Person and found that QVTr's check-mode[7] semantics are inconsistent with enforce-mode. This mandated some heroic workarounds to make check-mode usable. Both of these not-incorrect observations stem from inconsistencies in the specification which need investigation. There is an inconsistency between the detailed exposition of check-mode semantics and the philosophical statement that check-mode is semantically equivalent to enforce-mode and comparison. QVTr can only be a useful language if its details correspond to some clear sensible philosophical principles. The two pages of details are therfore wrong; the one sentence of principle is correct. Given Stevens' observation [7] that a transformation should involve minimal change, it is unclear that a detailed exposition of a simplified check-mode that is consistent with enforce-mode is even possible for the general case.

QVTr provides a very powerful declarative exposition which is very strongly typed by its input, trace and output metamodels. QVTr leverages OCL whose expressions are side effect free, to introduce a very controlled form of model mutation. This limited mutation is amenable to the global analysis outlined above. Early results of the QVTr scheduler suggest that the performance of declarative transformations can approach that of manual code; a typically 40-fold speed up over tools such as ATL and QVTo that lack schedule optimization or Java code generation [11].

The Eclipse QVTd work diverged from an early Epsilon prototype to exploit the very strong limitations imposed by metamodels and declarative mappings. It therefore uses mostly analysis and a few heuristics to produce a useful schedule relatively quickly, rather than exploring a large number of alternative schedules in an infeasible time [6].

[7] a check-mode execution precedes an enforce-mode to avoid the cost of an enforce

Acknowledgements Many thanks to Horacio Hoyos Rodriguez who pioneered the invaluable DOT and yEd work on visualization of the QVTr scheduling.

6 Conclusion

We have described the auto-generated QVTs representation and identified similarities to, and differences from, TGG.

We have shown how the full integration of OCL supports a narrowly pessimistic analysis leading to an efficient static schedule and effective generation of a direct Java code realization of a QVTr transformation.

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

References

1. Ehrig, H:, Pfender, M:, Schneider, H: Graph-grammars: An algebraic approach, Switching and Automata Theory, 1973.
2. Kindler, E., Wagner, R.: Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical Report tr-ri-07-284, Department of Computer Science, University of Paderborn, Germany (June 2007)
3. Leblebici, E:, Anjorin, A:, Schürr, A: A Catalogue of Optimization Techniques for Triple Graph Grammars, In: Fill, H.G., Karagiannis, D., Reimer, U. (eds.) Modellierung 14. LNI, vol. 225, pp. 225–240. GI (2014)
4. OMG. MDA Guide Version 1.0. OMG Document Number: omg/2003-06-01, June 2003.
5. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0. OMG Document Number: formal/2008-04-03, April 2008.
6. Rodriguez, H.H:, Kolovos, D: Declarative Model Transformation Execution Planning, 15th International Workshop on OCL and Textual Modeling, Saint-Malo, October 2016
7. Stevens, P : Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions,MoDELS 2007: 1-15
8. Stevens, P : A simple game-theoretic approach to checkonly QVT-Relations, Software and Systems Modeling 12,January 2009, pp. 165–180
9. Westfechtel, B.: Incremental Bidirectional Transformations: Applying QVT Relations to the Families to Persons Benchmark. ENASE, 2018.
10. Willink, E: UMLX : A Graphical Transformation Language for MDA, Model Driven Architecture: Foundations and Applications, MDA 2003, Twente, June 2003.
11. Willink, E: The Micromapping Model of Computation; the Foundation for Optimized Execution of Eclipse QVTc/QVTr/UMLX, 10th International Conference on Model Transformation (ICMT2017), July 2017
12. Eclipse ATL Project.
https://projects.eclipse.org/projects/modeling.mmt.atl
13. Eclipse Henshin Project.
https://projects.eclipse.org/projects/modeling.emft.henshin
14. Eclipse QVT Declarative Project.
https://projects.eclipse.org/projects/modeling.mmt.qvtd