QVT Imperative - A practical foundation for declarative transformation execution

Horacio Hoyos
Department of Computer Science
The University of York
York, UK
Email: horacio.hoyos.rodriguez@ieee.org

Dimitris Kolovos
Department of Computer Science
The University of York
York, UK
Email: dimitris.kolovos@york.ac.uk

Edward Willink
Willink Transformations Ltd.
Reading, UK
Email: ed@willinktransformations.co.uk

***Abstract*—The early enthusiasm, in 2002, for model to model transformation languages led to eight submissions for an OMG standard comprising three languages, yet no commercial products have appeared. The QVT Core language was intended as the foundation for QVT Relations but the available implementations have ignored the core language. Rather than ignoring the core language, we take the opposite approach and introduce three more core languages. Progressive semantic simplification through these core language terminates in an imperative unidirectional language that facilitates implementation.**

I. Introduction

The importance and benefits of standardization are widely recognized in all engineering disciplines. In the domain of Model Driven Engineering, the Object Management Group (OMG) has provided a set of standards related to modeling and model management. One of these standards is the Query/View/Transformation (QVT) specification[1] that addresses the task of model transformation. Although the QVT Request For Proposals (RFP)[2] in 2002 attracted 8 submissions, ten years later the initial enthusiasm has failed to mature into any commercial implementations.

The RFP called for one transformation language, but the submitters could not agree whether an imperative or declarative approach should be used, and so a compromise between the two viewpoints was resolved by specifying three languages. The QVT Operational Mappings language (QVTo) supports an imperative form of model transformation. The QVT Relations (QVTr) language provides powerful multi-directional declarative transformation capabilities and the QVT Core (QVTc) language provides much simpler multi-directional declarative capabilities.

Unfortunately the available QVTr tools are limited. Medini QVT is Open Source, provides a partial implementation but does not appear to be progressing. Performance results have been very disappointing [3]. ModelMorf is proprietary but the available Beta releases have not matured into a product.

For QVTc the situation is worse. The internal submission prototype at Compuware was never updated to match the specification and so QVTc has never had an implementation.

At Eclipse, the QVT Declarative project provides editors, parsers and models for QVTr and QVTc but no execution.

This paper describes ongoing activity towards remedying these execution deficiencies. Our work is aligned with the compromise of multiple languages and argues for three more intermediate languages in order to provide an implementation that supports both QVTc and QVTr.

It should be noted that the QVT 1.1 specification failed to address any of the issues raised against QVTc in the QVT draft or 1.0 specification, and since no prototype of QVTc has been produced anywhere, we have to treat the precise wording of the QVT specification with a little skepticism. We therefore work to what we perceive to be the spirit rather than the letter of the specification.

In this paper, section II presents the motivation for three new QVTc subset languages and the progressive transformation to QVTi, then section III provides an overview of QVTc. The details of QVTi are presented in section IV, with related work presented in section V. Finally, section VI concludes.

II. Motivation

A *model transformation* is the automated maintenance of a set of target models from a set of source models [4]. The models comprise Objects that are instances of Types within Packages, all defined by metamodels. As shown in Figure 1, a declarative transformation defines relationships between the models by specifying mappings (or rules) between the left and right metamodel elements. Each mapping relates a Pattern of left Objects to a Pattern of right Objects. When the transformation executes, the Patterns are instantiated as Bindings in which each Binding identifies the Object in use for each Pattern element. The set of all possible distinct Bindings forms the execution trace.

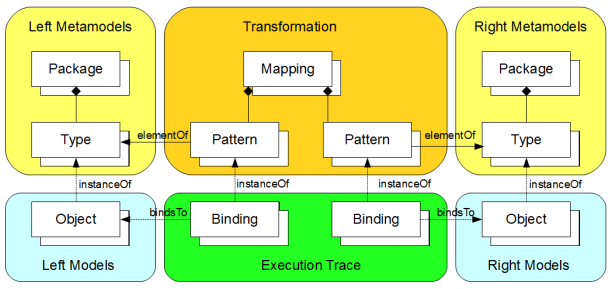

Fig. 1. Transformation Anatomy.

Since a declarative transformation expresses relationships to be satisfied, a transformation may be bi- or even multi-directional. The multi-directional capability allows a single transformation program to specify many model transformations, and to avoid the inconsistencies that may arise through writing independent programs for each direction and enforcement mode[1].

In contrast an imperative transformation specifies the steps to obtain the target model from the source model for a single direction and enforcement[5].

The high level of abstraction of QVTr presents many specification and implementation challenges which the QVT [1] specification resolves with a QVTr to QVTc transformation. However although simpler, QVTc remains as powerful as QVTr but more verbose. The flexibilities of multi-directionality and the lack of an obvious execution schedule remain.

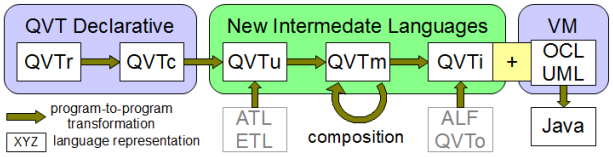

Fig. 2. Overview of the proposed QVT 6 languages architecture.

For efficient execution and more feasible implementation we propose a highly optimized unidirectional imperative language that we call QVT Imperative (QVTi). As with QVTr being realized by QVTc, QVTc can be realized by QVTi. Further, to facilitate this realization, we propose two additional languages QVT unidirectional (QVTu) and QVT minimal (QVTm) as presented in Figure 2. We propose a progressive program to program[2] transformation chain

- QVTc to QVT unidirectional (QVTu); to align the transformation to the user's invocation context and eliminate the redundant multi-directional and enforcement flexibilities.
- QVTu to QVT minimal (QVTm); to normalize the transformation to eliminate syntactic sugar and alternate representation flexibilities.
- QVTm to QVT Imperative (QVTi); to discard declarative flexibilities and synthesize a multi-pass imperative search schedule that can be executed easily by a model-friendly Virtual Machine.

QVTi, QVTm and QVTu are simplifications of QVTc and so their meta-models are extensions of the OCL meta-model. An OCL execution facility such as the Eclipse OCL VM[6] is therefore easily extended to support QVTi.

These new languages also offer important interchange points for other transformation technologies to exploit and so share the tool chain in the future:

- QVTu provides a high level interchange point for other uni-directional declarative transformation languages such as ATL[7] or ETL[8].
- QVTm provides a normalized representation at which declarative transformation composition and optimization can be applied.
- QVTi provides a low level interchange point that imperative transformation languages such as QVTo, ALF[9] or EOL[8] may exploit.

III. QVT Core

QVT Core (QVTc) is a multi-directional, multi-input, multi-output declarative model transformation language. For simplicity, in the following description, we defer consideration of more than just a simple left to right creation until SectionIV-B.

A significant challenge for model transformation arises in specifying how the overlap of target Bindings is to be handled. A common solution provides specialized constructs to interrogate the execution trace and so allow the instantiation of one Pattern to interact with the instantiation of another. These specialized constructs are often rather obscure. QVTc is unusual in making the traceability model explicit; it is called the Middle model.

Comparison of Figure 3 with Figure 1 shows the distinctive Middle Metamodel. A QVTc transformation is therefore specified as a set of mappings with constraints defined in a domain for each left, right and middle model. Since the execution traceability must be explicitly defined, a *Middle Model* (and metamodel) are used to define and instantiate the trace classes. It is important to note that it is the transformation author's responsibility to design the Middle metamodel.

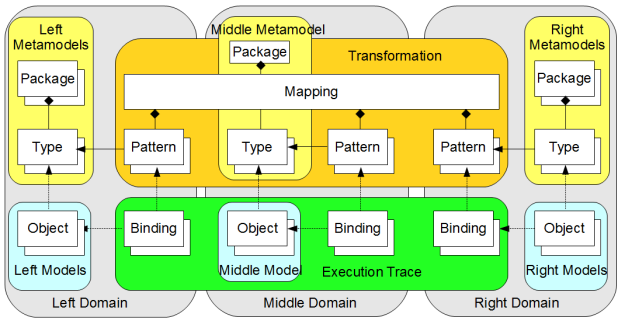

Fig. 3. QVT Core Anatomy.

The additional Middle model may be quite simple and the Middle Bindings may be free from overlap. This may significantly simplify the transformation exposition since with the aid of the intermediate, a direct N:M mapping from left-to-right may be expressed as a two-pass transformation comprising an N:1 left-to-middle pass and a 1:M middle-to-right pass. Any information that needs to be gleaned from the left can be cached in the middle model during left-to-middle pass so that it is readily available for use during the middle-to-right pass. Very complex transformations may specify additional passes that operate from Middle model to

[1] check/create/update execution

[2] We use program to program rather than model to model to stress the distinction between compile-time transformation of program models and run-time transformation of user models.

Middle model. QVTc supports reuse of mappings within a transformation by refinement, and reuse of transformations by inheritance.

The simpler semantics of QVTc make a QVTc implementation of a declarative transformation more tractable. QVTc makes only small abstract syntax extensions to EMOF and OCL. A QVTc implementation is therefore an attractive intermediate implementation approach for QVTr, as suggested in the QVT specification.

Within a Mapping, the QVTc semantics are simple; each Domain comprises a GuardPattern and a BottomPattern. The GuardPattern is responsible for the matching, and the BottomPattern for the model mutation.

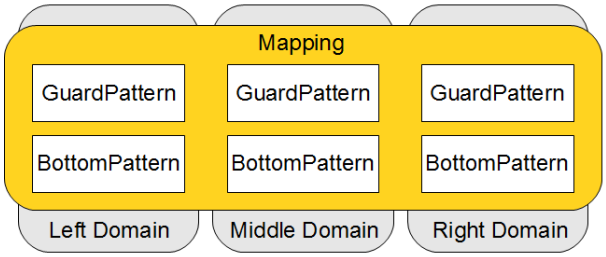

Fig. 4. QVT Core Areas.

The two-dimensional layout shown in Figure 4 is difficult to achieve in a text file and so the concrete syntax is

```
map mapping-name in tx-name {
  left   ( left-guard-pattern-variables
         | left-guard-pattern-constraints )
         { left-bottom-pattern-variables
         | left-bottom-pattern-constraints }
  right  ( right-guard-pattern-variables
         | right-guard-pattern-constraints )
         { right-bottom-pattern-variables
         | right-bottom-pattern-constraints }
  where  ( middle-guard-pattern-variables
          | middle-guard-pattern-constraints )
         { middle-bottom-pattern-variables
         | middle-bottom-pattern-constraints }
}
```

Let us consider as complex an example of a bidirectional transformation as space permits; transformation of colored Node trees. The trees demonstrate recursive hierarchy and edges. The alternate color representations demonstrate attributes; HSV (hue, saturation, value) on the left, HLS (hue, lightness, saturation) on the right and RGB (red, green, blue) as a middle intermediate. Figure 5 shows the three metamodels with the additional traceability references from middle metamodel to the external metamodels. Listing 1 presents the QVTc transformation for this example.

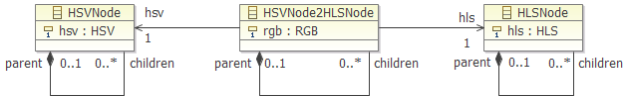

Fig. 5. Simple Tree metamodels; Left(HSV), Middle(RGB), Right(HLS).

Listing 1. QVTc transformation of colored nodes.

```
1  -- declare the ColorChange transformation
2  transformation ColorChange {
3      hsv imports HSVTree;
4      hls imports HLSTree;
5      imports HSV2HLS; }
6
7  -- utility queries for color conversions
8  query ColorChange::hls2rgb(color:HLSTree::HLS):HSV2HLS::RGB;
9  query ColorChange::hsv2rgb(color:HSVTree::HSV):HSV2HLS::RGB;
10 query ColorChange::rgb2hls(color:HSV2HLS::RGB):HLSTree::HLS;
11 query ColorChange::rgb2hsv(color:HSV2HLS::RGB):HSVTree::HSV;
12 -- abstract mapping of a Node
13 map Node2Node in ColorChange {
14     enforce hsv() { realize hsvNode : HSVNode | }
15     enforce hls() { realize hlsNode : HLSNode | }
16     where() {
17         realize middleNode : HSVNode2HLSNode |
18         middleNode.hsv := hsvNode; -- middle to hsv trace
19         middleNode.hls := hlsNode; -- middle to hls trace
20         -- hsv to rgb and rgb to hls conversions
21         middleNode.rgb := hsv2rgb(hsvNode.hsv);
22         middleNode.rgb := hls2rgb(hlsNode.hls);
23         hsvNode.hsv := rgb2hsv(middleNode.rgb);
24         hlsNode.hls := rgb2hls(middleNode.rgb);}
25 }
26 -- refined for a root, parent must be null
27 map Root2Root in ColorChange refines Node2Node {
28     enforce hsv() {
29         hsvNode.parent := null; }
30     enforce hls() {
31         hlsNode.parent := null; }
32     where () {
33         middleNode.parent := null;}
34 }
35 -- refined for a child, node's parent is parent
36 map Child2Child in ColorChange refines Node2Node {
37     enforce hsv(hsvParent : HSVNode | ) {
38         hsvNode.parent := hsvParent; }
39     enforce hls(hlsParent : HLSNode | ) {
40         hlsNode.parent := hlsParent; }
41     where (middleParent : HSVNode2HLSNode | ) {
42         middleNode.parent := middleParent;}
43 }
```

Lines 2-5 express the `transformation` declaration including our proposal of using an unnamed *TypedModel* for the middle model. The bodies of the the color converter queries (lines 8-11) are omitted for space reasons.

Lines 13 to 25 provide the *Node2Node* mapping, without any guard variables or constraints; the mapping is therefore unbound. Each domain realizes a *Node*, and so requires that where that node exists in a source domain, the corresponding nodes are created or updated in the middle and target domains. Lines 21 to 24 initialize the middle node from whichever of *hsv* or *hls* is the source.

Lines 27 to 34 refine the *Node2Node* mapping to enforce consistency at the root so that all root nodes have no parent. Lines 36 to 43 refine the *Node2Node* mapping to enforce consistency of parent-child relationships. The guard pattern introduces an additional parent node variable for each domain and requires that this is indeed the parent of each node inherited from *Node2Node*. For the source domain, the parent-child relationship is interpreted as a guard, whereas for the middle and target domains, the parent-child relationship is enforced.

The foregoing quick summaries demonstrate how symmetrical definition of multi-directional transformations is supported by the combination of the pattern variable declarations, some of which can be realized while others must exist, and by the assignments of OCL queries to properties. Some declarations are distinct for each direction, while others adopt distinct pred-

icate or assignment semantics according to the transformation direction.

IV. The QVT Imperative Language

The QVT Imperative (QVTi) language re-uses the principles and syntax of QVTc to provide an easily executable semantics that can be targeted by the program-to-program transformations shown in Figure 2, i.e., QVTi is intended to be a machine generated language rather than written by a programmer. The major simplifications of QVTi in comparison to QVTc are:

- Uni-directional (not multi-directional)
- Specific creation/update/check behavior (no check/enforce flexibility)
- No complex syntax such as refinement, or inheritance (no syntax sugar)
- No mappings have both source and target domains (no complex dataflow)
- Multiple mappings are executed sequentially (rather than declaratively)
- Nested mappings may be invoked directly (rather than declaratively)

These simplifications combine to support a simple mode of execution in which mappings are executed in sequence within simple loops. Nested mappings nest in a manner that allows a conventional execution stack to maintain the prevailing state of each search variable. The overall transformation is executed as a guarded depth-first search of the source and then middle model spaces. Where the guarded search matches, either the middle model element temporarily persists the context of the match, or a target model element is updated.

The search strategy is defined by the program-to-program transformation that produces the QVTi program. As a minimum this producer must serialize mappings with multi-variable patterns so that sub-mappings re-use rather rediscover parent bindings. This serialization offers significant opportunities for optimization through the use of the known metamodels and optionally through profiling as well. It is very desirable for the search to iterate over easy navigation paths such as compositions and forward references. It is also desirable to search first over model elements that have strict guard conditions since these may result in early pruning of the search space. It is highly undesirable to perform whole model searches or traverse associations in an unnavigable direction.

We will demonstrate the simplified QVTi semantics by reworking the Listing 1 example in accordance with a user requirement to create an HLS model from an HSV model. The resulting mappings, shown in Listing 2, are manually produced pending future work on the program-to-program transformation chain.

Listing 2. QVTi mappings of colored nodes.

```
1  -- Mapping root nodes Left to Middle
2  map HSV2MiddleRoot in ColorChanger {
3      hsv() { hsvRoot : HSVNode | hsvRoot.parent = null;}
4      where() {
5          realize middleRoot : HSVNode2HLSNode |
6          middleRoot.hsv := hsvRoot;
7          middleRoot.rgb := hsv2rgb(hsvRoot.hsv);
8      }
9      -- recursive call to visit children
10     map HSV2MiddleRecursion {
11         hsvNode <= hsvRoot.children;
12         middleParent := middleRoot;
13     }
14     -- invoke middle to output mapping
15     map Middle2HLSRoot {
16         middleNode := middleRoot;
17     }
18 }
19 -- Mapping child nodes Left to Middle
20 map HSV2MiddleRecursion in ColorChanger {
21     hsv(hsvNode : HSVNode | ) {}
22     where(middleParent : HSVNode2HLSNode | ) {
23         realize middleNode : HSVNode2HLSNode |
24         middleNode.parent := middleParent;
25         middleNode.hsv := hsvNode;
26         middleNode.rgb := hsv2rgb(hsvNode.hsv);
27     }
28     -- recursive call to visit children
29     map HSV2MiddleRecursion {
30         hsvNode <= hsvNode.children;
31         middleParent := middleNode;
32     }
33 }
34 -- Mapping root nodes Middle to Right
35 map Middle2HLSRoot in ColorChanger {
36     enforce hls() { realize hlsNode : HLSNode | }
37     where(middleNode : HSVNode2HLSNode) {
38         middleNode.hls := hlsNode;
39         hlsNode.parent := null;
40         hlsNode.hls := rgb2hls(middleNode.rgb);
41     }
42     -- recursive call to visit children
43     map Middle2HLSRecursion {
44         middleNode <= middleNode.children;
45     }
46 }
47 -- Mapping child nodes Middle to Right
48 map Middle2HLSRecursion in ColorChanger {
49     enforce hls() { realize hlsNode : HLSNode | }
50     where(middleNode : HSVNode2HLSNode |) {
51         middleNode.hls := hlsNode;
52         hlsNode.parent := middleNode.parent.hls;
53         hlsNode.hls := rgb2hls(middleNode.rgb);
54     }
55     -- recursive call to visit children
56     map Middle2HLSRecursion {
57         middleNode <= middleNode.children;
58     }
59 }
```

The transformation starts with a potentially declarative match of the guard variable(s) of the first mapping to the 'whole' model, from which The *GuardPattern* on line 2 of HSV2MiddleRoot selects the *HSVNode* source without a parent. Where this match is found, lines 5-7 realize an *HSVNode2HLSNode* middle node and populate it with the source context and computed RGB value.

The two nested mapping calls on lines 10-17 are then executed sequentially. Explicit mappings calls are the sole extension in QVTi. The target mapping is invoked with the guard variable to the left of each := assignment bound to the value of an OCL expression, and the guard variable to the left of each <= assignment looping over each value of the collection-valued OCL expression.

The *HSV2MiddleRecursion* mapping is therefore invoked with its *hsvNode* guard variable bound to each of the original source node's children, and its *middleParent* bound to the realized middle node. The *HSV2MiddleRecursion* realizes a corresponding middle node for each source node and recurses down the source tree.

Once the *HSV2MiddleRecursion* completes, the *HSV2MiddleRoot* mapping resumes and invokes the *Middle2HLSRoot* mapping, binding its *middleNode* guard variable to the *middleRoot* node. The *Middle2HLSRoot* behaves in a very similar fashion realizing a target node and using the *Middle2HLSRecursion* to build the target tree.

This simple example demonstrates the simplifications underlying QVTi and the one syntax extension to QVTc. With these reduced semantics QVTi still has the power to express important programming idioms.

1) Sequencing and Passes: Sequential execution of multiple patterns within one pass or of multiple passes can be expressed by sequential nested mappings as in the *HSV2MiddleRecursion* then *HSV2MiddleRoot* sequencing.

2) Iteration and Recursion: Looping over multiple model elements is supported by a *GuardPattern* variable bound to each element in turn of a collection of model elements as in the *HSV2MiddleRecursion* over the children. Simple iteration loops may use nested *mappings*. Recursive loops exploit a nested mapping that invokes a named mapping with bindings. This syntax extension short-circuits the total model search associated with a declarative exposition.

3) Conditional Execution: Arbitrary OCL constraints may be used in the guards for each step of each iteration. The example is very regular so there is only a single 'at the root' guard for the *HSV2MiddleRoot* mapping.

4) Model Mutation: Model elements are created by the realized variable declarations in the bottom patterns. Arbitrary OCL queries define the value to be assigned to each model element, or the iteration domain of a nested mapping. These expressions appear to the right of `:=` or `<=` operators in the example.

5) Traceability: Traceability is provided by the middle model. This is user-defined and so allows the user to control how much information relating source and target is maintained. In a more complex example OCL expressions may navigate from source or target models to the middle model using the standard UML opposite navigation semantics.

6) Timing: Preparation of the optimized QVTi transformation as AST or compiled Java is a compile-time activity that may be performed ahead of time whenever the desired execution direction mode is also known ahead of time.

A. Implementation

Some simple QVTi transformations have been implemented using the Eclipse QVTi editor and parser and the Eclipse OCL VM[6].

The OCL VM offers two modes of execution, the simplest of which is interpreted. It comprises a simple tree-walking evaluator over the OCL AST. This evaluator is realized by an extensible EvaluationVisitor and so, since the QVTi AST is an extension of the OCL AST, it is sufficient to extend the OCL EvaluationVisitor to support the additional QVTi AST nodes.

This proved to be surprisingly easy. It was not even necessary to add extensions for model mutation since the Eclipse OCL VM has a prototype type constructor implementation.[3]

The API provides only two methods to create objects and to initialize fields. These were sufficient for the disciplined form of model mutation in QVTi.

The Eclipse OCL VM also offers a tree-walking code generator that produces a direct Java realization of OCL. This has been extended to support the additional QVTi AST nodes and so generates direct Java code for a QVTi transformation.

B. Future Work

We will now discuss a number of complexities that we have glossed over so far.

1) Bootstrapping: The QVT specification assumes the existence of a QVTr capability in order to realize the QVTr to QVTc transformation. We propose to provide partial implementations of the QVTr to QVTi chain using ETL/Flock[8]. These bootstraps will be promoted to QVTu using a further ETL/Flock to QVTu transformation (also using ETL/Flock). The final promotion from QVTu to QVTc or QVTr will be manual.

2) Update/Check: A transformation may create or update target models or just check source models. These distinct modes of execution are resolved at the same time as multi-directionality; the transformation relationships are adjusted to reflect that user's requirements. In the case of a check or update transformation a reconciliation of multiple source models is synthesized. The QVTi implementation is simplified by only needing to execute the single required mode of execution.

3) In-Place Update: A QVTc transformation is amenable to in place execution since the middle model separates the source and target model accesses. It is only necessary to ensure that the generated QVTi schedule caches source accesses in the middle model before any target updates introduce conflicts.

4) Generality: Declarative rule matches provide arbitrary flexibility without efficiency, whereas the imperative QVTi schedule provides efficient realization only of metamodel guided rule sequences. However QVTi retains the ability to perform a brute force recursion over all possible rules, so there is no loss of generality. The ongoing research goal is to maximize the ability to exploit the metamodel.

5) Performance: Once the QVTr to QVTc to QVTi program-to-program transformations are in-place, we can look forward to a high performance direct Java realization of QVTr. And with QVTr in-place, the slightly verbose expositions of the QVTc and its subset languages can be ignored by users. Only transformation toolsmiths need use them to exploit their interchange opportunities.

V. Related Work

The Kermeta model transformation language development tools include code generators that can transform Kermeta transformations into Java and Scala code which can then be executed against a Java Virtual Machine (JVM) for more

[3]Type constructors extend the Tuple syntax to allow construction of fully initialized user objects as e.g. `Person{name:='Me'; age:=20;}`.

efficient execution [10]. The Epsilon[8] platform of model management languages also features a layered approach where all model task-specific languages extend a common expression language (EOL).

Following the paradigm of VM-based programming language architectures (e.g. JVM), the Atlas Transformation Language (ATL) features a layered architecture in which transformations are compiled to XML-based byte-code, which is then executed by a virtual machine [7]. The architecture of ATL enables the substitution of the default VM with custom VM implementations. Beyond the default generic VM, ATL ships with an additional optimized VM for transforming EMF-based models. The ATL VM is based upon the ATL language and so has limited integration with its OCL implementation. The similarities of the QVT and ATL architectures provide interesting points for interoperability[11]. In contrast our approach exploits the inherent tree structure of the OCL AST to extend the Eclipse OCL VM's tree-walking interpreter and code generator for the extended QVTi AST.

Building on the idea of a 2-stage execution of model transformations, in [12], the authors present a generic transformation engine VM (EMFTVM). Similarly to the ATL VM, EMFTVM also executes byte-code but unlike the ATL VM where byte-code is represented using proprietary XML, in EMFTVM byte-code conforms to an Ecore metamodel – and as such it is easier for higher-level transformation languages to compile down to it using higher-order transformations. The aim of EMFTVM is to serve as an underlying VM for additional transformation languages and as a proof of concept, the developer of the EMFTVM has implemented higher-order model transformations that map transformations expressed both in ATL and in a simple graph transformation language[4] to VM byte-code. The concept of a reusable EMF-based VM that can act as a compilation target for higher-level languages is similar to the approach proposed in this paper. However, in our approach we envision multiple hook points which will allow transformation languages operating at different levels of abstraction to integrate with the proposed architecture with reduced duplication.

The most similar approach to the one proposed in this paper, is the work presented in [13] where the authors propose a layered architecture for implementing QVTc and QVTo. More specifically, the authors propose a program-to-program transformation to compile QVTc and QVTo transformations into a low-level imperative transformation language called Atomic Transformation Code (ATC), and then compile ATC code to byte-code that can be executed by a proprietary virtual machine called Virtual Transformation Engine (VTE). The architecture of our approach is similar, but we propose to eliminate dependencies on proprietary components (ATC), and to further decompose the compilation process by introducing additional intermediate QVTx languages.

[4]http://soft.vub.ac.be/soft/research/mdd/simplegt

VI. Conclusions

We have proposed a simple unidirectional imperative language, QVTi, that provides a feasible implementation. Further, we propose two more languages (QVTu and QVTm) and a progressive program-to-program transformation chain from QVTr to QVTc to QVTu to QVTm to QVTi, in order to provide a practical execution semantics for QVTc and QVTr. We have introduced the QVT Imperative (QVTi) language and presented its syntax and semantics. We have demonstrated that QVTi may retain the basic QVTc concrete syntax and yet support useful idioms for optimized pattern matching strategies and multiple passes. Our preliminary QVTi implementations demonstrate that the Eclipse OCL VM interpreter is easily extended for QVTi. We can therefore look to exploit the Eclipse OCL VM's Java code generator to provide good quality compiled Java code and then concentrate on the QVTr to QVTi program-to-program transformations to provide effective execution strategies.

References

[1] OMG, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification," http://www.omg.org/spec/QVT/1.1/, January 2011, version 1.1.

[2] OMG, "Request for Proposal: MOF 2.0 Query / Views / Transformations RFP," OMG Document: ad/2002-04-10, revised on: April 24, 2002.

[3] S. Bosems, "A Performance Analysis of Model Transformations and Tools," Master's thesis, Department of Electrical Engineering Mathematics and Computer Science, University of Twente, March 2011, http://www.utwente.nl/ewi/trese/graduation_projects/2011/004.pdf.

[4] A. G. Kleppe, J. B. Warmer, and W. Bast, *MDA explained, the model driven architecture: Practice and promise*. Addison-Wesley Professional, 2003.

[5] T. Mens and P. V. Gorp, "A Taxonomy of Model Transformation," *Electronic Notes in Theoretical Computer Science*, vol. 152, no. 0, pp. 125–142, 2006.

[6] E. Willink, "An extensible OCL virtual machine and code generator," in *Proceedings of the 12th Workshop on OCL and Textual Modelling*, ser. OCL '12. ACM, 2012, pp. 13–18.

[7] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 12, pp. 31 – 39, 2008.

[8] R. Paige, D. Kolovos, L. Rose, N. Drivalos, and F. Polack, "The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering," in *Proc. 14th IEEE International Conference on Engineering of Complex Computer Systems*, 2009, pp. 162 –171.

[9] I. Perseil, "ALF formal," *Innovations in Systems and Software Engineering*, vol. 7, no. 4, pp. 325–326, 2011.

[10] F. Fouquet, O. Barais, and J.-M. Jézéquel, "Building a Kermeta Compiler using Scala: an Experience Report," in *Proc. Scala Days 2010*, 2010.

[11] F. Jouault and I. Kurtev, "On the architectural alignment of ATL and QVT," in *Proceedings of the 2006 ACM symposium on Applied computing*, ser. SAC '06. ACM, 2006, pp. 1188–1195.

[12] D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault, "Towards a general composition semantics for rule-based model transformation," in *Proceedings of the 14th international conference on Model driven engineering languages and systems*, ser. MODELS'11. Springer-Verlag, 2011, pp. 623–637.

[13] A. Sánchez-Barbudo, E. V. Sánchez, V. Roldán, A. Estévez, and J. L. Roda, "Providing an open virtual-machine-based QVT implementation," in *Actas de los Talleres de las Jornadas de Ingeniera del Software y Bases de Datos*, vol. 2, no. 3, 2008, pp. 42–51.