QVT Imperative - A practical semantics for declarative transformations

Horacio Hoyos, Dimitris Kolovos[1], and Edward Willink[2]

[1] The University of York, York, UK,
horacio.hoyos.rodriguez@ieee.org, dimitris.kolovos@york.ac.uk,
[2] Willink Transformations Ltd., Reading, UK
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 program-to-program transformation through these core languages terminates in an easily implemented imperative language that supports declarative transformations.

Keywords: QVT, OCL, virtual machine, program transformation, declarative transformation, progressive transformation, transformation chain

1 Introduction

The importance and benefits of standardisation are widely recognised in all engineering disciplines. In the domain of Model Driven Engineering, the OMG has provided a set of standards related to modelling and model management. One of these standards is the Query/View/Transformation (QVT) specification[7] that addresses the task of model transformation. Although the QVT Request For Proposals (RFP)[6] 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. Three languages might seem like a typical committee outcome, however in this paper we run with this compromise and start to argue for six languages.

The QVT Operational Mappings language (QVTo) supports an imperative form of model transformation and is the most successful with two Open Source implementations available; SmartQVT and the QVT Operational project at Eclipse. The most recent release of SmartQVT was in 2008. Eclipse QVTo was originally developed by Borland, and after a three year lull is again under active maintenance and development.

The QVT Relations (QVTr) language provides powerful multi-directional declarative transformation capabilities. It has two implementations. Medini QVT is Open Source, provides a partial implementation but does not appear to be progressing. Performance results have been very disappointing [2]. ModelMorf is proprietary but the freely available Beta releases have not matured into a product.

The QVT Core (QVTc) language provides much simpler multi-directional declarative capabilities. 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 has provided editors, parsers and models for QVTr and QVTc but no execution capability. This paper describes ongoing activity towards remedying the execution deficiencies.

The two-level declarative approach adopted by the QVT specification provides powerful abstractions but no obvious way to realize them. For efficient execution we want a highly optimized imperative representation that we call QVT Imperative (QVTi). Following the QVT specification's suggestion that QVTr should be realized by a program-to-program transformation to QVTc, we propose a program-to-program transformation from QVTc to QVTi. This transformation will be realized as a chain of three program-to-program transformations from QVTc via QVT Unidirectional (QVTu) and QVT Minimal (QVTm) to QVTi. In a future paper, we will discuss the QVTc to QVTi transformations and the QVTu and QVTm languages. In this paper, we concentrate on the QVTi semantics, its execution and the reuse of QVTc concrete syntax for QVTi (and QVTm and QVTu).

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 scepticism. We therefore work to what we perceive to be the spirit rather than the letter of the specification. Our tooling introduces a QVTo-like import statement for the QVTc metamodels, including the middle metamodel.

In this paper we present the motivation for the three new QVTc subset languages in section 2 and an overview of QVTc in section 3. The details of QVTi are presented in section 4, with related work presented in section 5. Finally, section 6 concludes.

2 Motivation

In QVTr, a model transformation is defined using powerful abstractions. A set of relations that "declare constraints that must be satisfied by the elements of the candidate models"[7].

- Constraints are defined by matching properties of elements in the candidate models.

- Property matching uses expressions written in OCL and grouped in domains.
- Each domain represents a candidate model.
- Constraints can be specialized to check models (checking semantics).
- Constraints can be specialized to modify models (enforcement semantics).
- Constraints semantics varies with the chosen transformation direction.

The complexity of the language semantics, and the underlying abstractions of pattern matching, constraints and OCL make the specification and implementation of QVTr a complex and daunting task. The QVT specification uses an almost unreadable and untested QVTr to QVTc transformation to 'solve' the problem.

QVTc is "as powerful as the Relations language, though simpler. Consequently, the semantics of the Core Language can be defined more simply, though transformations described using the Core are more verbose"[7]. Since in QVTc the trace models must be explicitly defined, constraints are now defined by matching properties of elements in the candidate models and the trace models. Checking and enforcement provide the same functionality as in QVTr.

A QVTc transformation is specified as a set of mappings with constraints defined in a domain for each candidate and trace model. The simpler semantics of QVTc make a QVTc implementation 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. This approach may use a debugged version of the QVTr to QVTc transformation from the specification.

However, multi-directional transformations still present several challenges in a number of domains and disciplines [3], some of which translate to QVT[10]. Declarative transformations present a challenge from a rule schedulability point of view. In the specific case of QVT this translates to an implementation in which the execution of rules can be partial, interrupted or postponed until matches in other rules provide the required variable values (bindings), eventually requiring multiple passes. These challenges can be overcome by removing multi-directionality, normalizing constraints and defining an imperative semantics.

However, the semantic gap between <multi-directional + declarative> and <uni-directional + imperative> makes it difficult to tackle the aforementioned aspects in a one-step mapping. Moreover, there is a risk that what was gained from having to implement a simpler semantics would be lost by the work needed to realize the required mappings.

Figure 1 presents our progressive transformation solution to realizing QVTr on an OCL Virtual Machine[12]. At the top we have the two QVT Declarative languages, with QVTr realized by a QVTr to QVTc program-to-program transformation. Our three new languages, QVTu, QVTm and QVTi are syntactic and semantic simplifications of QVTc.

- For QVT Uni-directional (QVTu), we align the transformation to the user's invocation context and eliminate the redundant multi-directional and enforcement flexibilities.

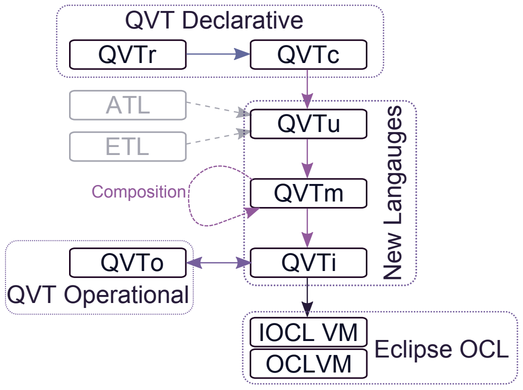

Fig. 1. Overview of the proposed QVT 6 languages architecture.

- For QVT Minimal (QVTm) we normalize to eliminate syntactic sugar and alternate representation flexibilities.
- For QVT Imperative (QVTi) we discard declarative flexibilities and synthesize a multi-pass imperative search schedule that can be executed easily by a model-friendly Virtual Machine.

These new languages are not just a convenience for realizing QVTc, they also offer important interchange points for other transformation technologies to exploit and so share the tool chain.

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

3 The QVT Core Language

QVT Core (QVTc) is a surprisingly simple multi-directional, multi-input, multi-output declarative model transformation language. The complexities of multi-versatility are considered in 3.4. In the following description, we therefore consider just a simple left to right transformation.

A declarative model transformation language declares the many transformation relationships that all the objects in all the input models and in all the output models satisfy on completion of the transformation; it does not necessarily specify how this is achieved. The complexity of the many relationships is managed by exploiting the familiar metamodels, shown at the left and right hand sides of Figure 2. These comprise packages and types to categorize the different kinds of

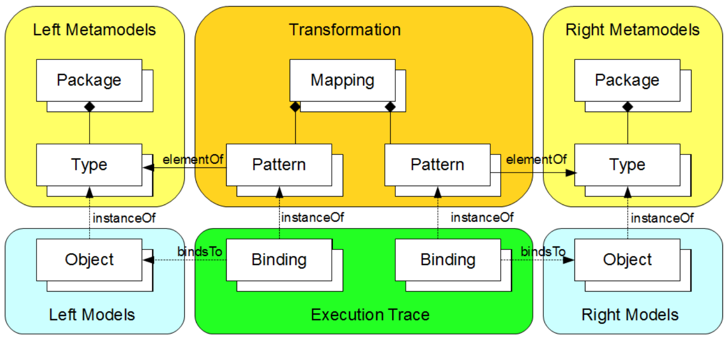

Fig. 2. Transformation Anatomy.

objects in a model. The Transformation adds Mappings and Patterns to organize the different kinds of relationship to be satisfied.

Figure 3 is an Object Diagram showing an example Pattern describing a parent-child match. The pattern involves two pattern variables *theParent* and the *theChild* each of which may be bound to a *Node* object in a user model. The pattern imposes the additional constraint that the objects bound to *theParent* and *theChild* variables must lie at each end of a *parent-children* Association. The *theParent* lies at the composing (diamond) end of an optional (?) multiplicity. The *theChild* lies at the end of an arbitrary (*) multiplicity.

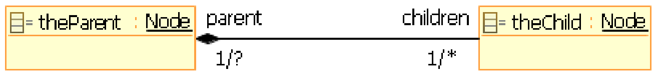

Fig. 3. Example Parent-Child Pattern.

When the transformation relationships are satisfied, the models have been partitioned into groups of objects and every object is a member of at least one group. Each group is identified by a Binding in which the Type of each object conforms to the Type of the Pattern element to which the object is bound. The interrelationships between the bound objects similarly conform to the interrelationships between the pattern elements. For the example pattern, each Binding comprises a pair of *Node*s one bound to *theParent* and the other bound to *theChild*. A Binding exists for every possible pair of *Node* objects that match the Pattern. The transformation between the Bindings is defined by Mappings, each of which defines the interrelationships between one left Pattern and one right Pattern.

The foregoing principles are common to many declarative and some imperative transformation languages. It is in the way in which mappings are structured that transformation languages vary.

3.1 Traceability and the Middle Model

A significant challenge for model transformation arises in specifying how the overlap of output 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. Figure 4 shows how for QVTc there are three Domains, Left, Middle and Right, each of which comprises Models and Metamodels. Associated with each Domain are the Bindings and Patterns.

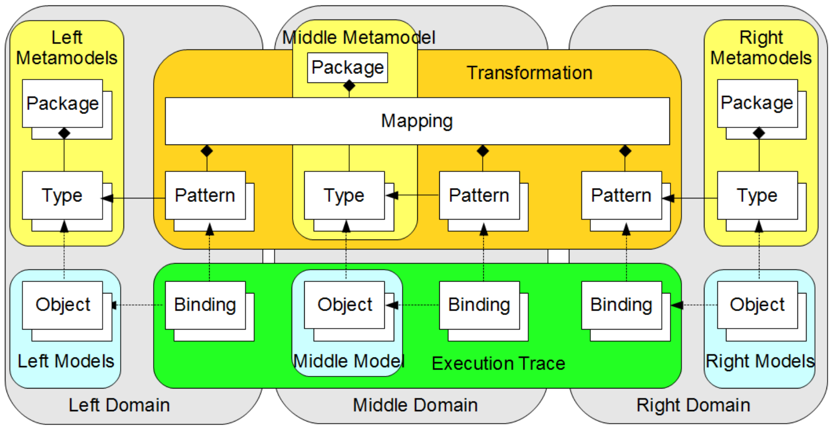

Fig. 4. QVT Core Anatomy.

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 Middle model.

The Middle model of course conforms to its metamodel and, for QVTc, it is the transformation author's responsibility to design the Middle metamodel so that overlaps can be resolved and information cached.

3.2 Intra-Mapping Semantics

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.

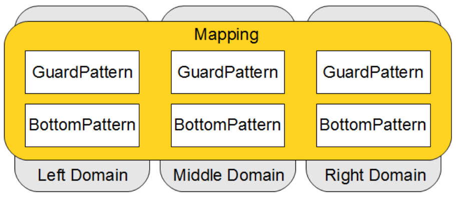

Fig. 5. QVT Core Areas.

The two-dimensional layout shown in Figure 5 is difficult to achieve in a text file and so the concrete syntax is

```
map {
  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 a very simple example of a bidirectional transformation between colored Node trees with different color representations; HSV (hue, saturation, value) on the left, HLS (hue, lightness, saturation) on the right and RGB (red, green, blue) as a middle intermediate. Figure 6 shows the three metamodels with the additional traceability references from middle metamodel to the external metamodels. Listing 1.1 presents the QVTc transformation for this example.

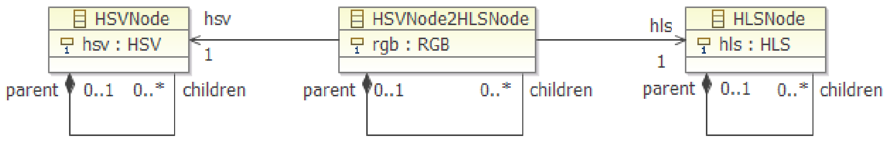

Fig. 6. Simple colored Tree metamodels; Left(HSV), Middle(RGB), Right(HLS).

Listing 1.1. QVTc transformation of colored nodes.

```
import 'HSVTree.ecore';              -- import the HSVTree package
import 'HLSTree.ecore';              -- import the HLSTree package
import 'HSV2HLS.ecore';              -- import the HSV2HLS package

transformation ColorChange {         -- declare the ColorChange transformation
    hsv imports HSVTree;             -- hsv TypedModel uses HSVTree package
    hls imports HLSTree;             -- hls TypedModel uses HLSTree package
    imports HSV2HLS;                 -- middle TypedModel uses HSV2HLS package
}
                                     -- utility queries for color conversions
query ColorChange::hls2rgb(color : HLSTree::HLS) : HSV2HLS::RGB;
query ColorChange::hsv2rgb(color : HSVTree::HSV) : HSV2HLS::RGB;
query ColorChange::rgb2hls(color : HSV2HLS::RGB) : HLSTree::HLS;
query ColorChange::rgb2hsv(color : HSV2HLS::RGB) : HSVTree::HSV;
```

```
15
16 map Node2Node in ColorChange {        -- abstract mapping of a Node
17     enforce hsv() { realize hsvNode : HSVNode | } -- hsv node exists
18     enforce hls() { realize hlsNode : HLSNode | } -- hls node exists
19     where() {
20         realize middleNode : HSVNode2HLSNode |   -- middle node exists
21         middleNode.hsv := hsvNode;               -- middle to hsv trace
22         middleNode.hls := hlsNode;               -- middle to hls trace
23         middleNode.rgb := hsv2rgb(hsvNode.hsv); -- hsv to middle color
24         middleNode.rgb := hls2rgb(hlsNode.hls); -- hls to middle color
25         hsvNode.hsv := rgb2hsv(middleNode.rgb); -- middle to hsv color
26         hlsNode.hls := rgb2hls(middleNode.rgb); -- middle to hls color
27     }
28 }
29
30 map Root2Root in ColorChange refines Node2Node { -- refined for a root
31     enforce hsv() {                    -- hsv can be an output
32         hsvNode.parent := null; }     -- parent must be null
33     enforce hls() {                    -- hls can be an output
34         hlsNode.parent := null; }     -- parent must be null
35     where() {                          -- middle is always the middle
36         middleNode.parent := null;    -- parent must be null
37     }
38 }
39
40 map Child2Child in ColorChange refines Node2Node { -- refined for a child
41     enforce hsv(hsvParent : HSVNode | ) {    -- additional parent variable
42         hsvNode.parent := hsvParent; }       -- node's parent is parent
43     enforce hls(hlsParent : HLSNode | ) {    -- additional parent variable
44         hlsNode.parent := hlsParent; }       -- node's parent is parent
45     where(middleParent : HSVNode2HLSNode | ) { -- additional parent var
46         middleNode.parent := middleParent; } -- node's parent is parent
47 }
```

Lines 1 to 15 provide some boilerplate; the `import` statements of the actual metamodels whose package names are the same as the file name; the `transformation` declaration including an unnamed *TypedModel* for the middle model; declarations of the color converter queries whose definition is omitted for space reasons.

Lines 16 to 28 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 an input domain, the corresponding nodes are created or updated in the middle and output domains. Lines 21 to 24 initialize the middle node from whichever of *hsv* or *hls* is the input and lines 25 to 26 initialize the color value of the *hsv* or *hls* nodes from the middle node.

Lines 30 to 38 refine the *Node2Node* mapping to enforce consistency at the root so that all root nodes have no parent. Lines 40 to 47 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 input domain, the parent-child relationship is interpreted as a guard, whereas for the middle and output domains, the parent-child relationship is enforced.

The foregoing quick summaries demonstrate how the combination of the pattern variable declarations, some of which can be realized while others must exist, and the assignments of OCL queries to properties support symmetrical definition of multi-directional transformations. Some declarations are distinct for each direction, others adopt distinct predicate or assignment semantics according to the transformation direction.

3.3 Inter-Mapping Semantics

A single mapping is of limited utility. Practical transformations require many mappings and an ability to share context between mappings. Consider a mapping from a hierarchical metamodel such as UML. A high level mapping may transform packages, and the transformed package is then needed as context for a mapping involving classes. QVTc supports independent mappings by declaring each as a top level mapping as in Listing 1.1. Shared context is supported by declaring the dependent mappings as nested within the mapping whose bindings the nested mapping shares as in Listing 1.2.

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

3.4 Execution Modes

Direction The multi-directional capability allows a single QVTc transformation program to specify many model transformations, and to avoid the inconsistencies that may arise through writing independent programs for each direction and enforcement.

A multi-directional transformation does not have unambiguous inputs or outputs and so a QVTc transformation is specified between TypedModels. In practice only one direction will be of interest at any one time and so once the invocation identifies which TypedModels are inputs and which are outputs, a practical QVTc tool may optimize away the redundant declarations for unwanted directions. Since all transformations are not reversible, QVTc allows the programmer to restrict particular Domains to input or output by using the `check` or `enforce` keywords.

Enforcement A transformation may be used for more than one purpose: *check*, *update* or *create*. In the more common case, a transformation *creates* output models corresponding to input models. A transformation may also be used to *check* that existing output models are consistent with input models, or to *update* existing output models to be consistent with input models. In the case of an update, for many important system applications the update may need to occur in-place. For this use case, the declarative QVTc exposition enables the QVTc tooling to ensure that the update occurs in a coherent fashion so that all input locations are read before any co-located output locations are written.

4 The QVT Imperative Language

The new QVT Imperative (QVTi) language re-uses the principles and syntax of QVTc to provide an easily executable semantics that can be targetted by program-to-program transformations shown in Figure 1. The major simplifications of QVTi in comparison to QVTc are:

– Uni-directional (not multi-directional)
– Specific creation/update/check behaviour (no check/enforce flexibility)
– No complex syntax such as refinement, or inheritance (no syntax sugar)
– No mappings have both input and output domains (no complex dataflow)
– Each mapping has one unbound variable (no multi-variable patterns)
– 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 one-dimensional 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 input and then middle model spaces. Where the guarded search matches, either the middle model element temporarily persists the context of the match, or an output 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 match just one variable at a time. This serialization offers significant opportunities for optimization through 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 1.1 example in accordance with a user requirement to create an HLS model from an HSV model. The resulting mappings, shown in Listing 1.2, are manually produced pending future work on the program-to-program transformation chain.

Listing 1.2. QVTi mappings of colored nodes.

```
map HSV2MiddleRoot in ColorChanger {     -- Mapping root nodes L to M
    hsv() { hsvRoot : HSVNode | hsvRoot.parent = null;}
    where() {
        realize middleRoot : HSVNode2HLSNode |
        middleRoot.hsv := hsvRoot;
        middleRoot.rgb := hsv2rgb(hsvRoot.hsv);
    }
    map HSV2MiddleRecursion {          -- recursive call to visit children
        hsvNode := hsvRoot.children;
        middleParent := middleRoot;
    }
    map Middle2HLSRoot {           -- invoke middle to output mapping
        middleNode := middleRoot;
    }
}

map HSV2MiddleRecursion in ColorChanger { -- Mapping child nodes L to M
    hsv(hsvNode : HSVNode | ) {}
    where(middleParent : HSVNode2HLSNode | ) {
```

```
20          realize middleNode : HSVNode2HLSNode |
21          middleNode.parent := middleParent;
22          middleNode.hsv := hsvNode;
23          middleNode.rgb := hsv2rgb(hsvNode.hsv);
24      }
25      map HSV2MiddleRecursion {       -- recursive call to visit children
26          hsvNode := hsvNode.children;
27          middleParent := middleNode;
28      }
29  }
30
31  map Middle2HLSRoot in ColorChanger { -- Mapping root nodes M to R
32      enforce hls() { realize hlsNode : HLSNode | }
33      where(middleNode : HSVNode2HLSNode) {
34          middleNode.hls := hlsNode;
35          hlsNode.parent := null;
36          hlsNode.hls := rgb2hls(middleNode.rgb);
37      }
38      map Middle2HLSRecursion {       -- recursive call to visit children
39          middleNode := middleNode.children;
40      }
41  }
42
43  map Middle2HLSRecursion in ColorChanger { -- Mapping child nodes M to R
44      enforce hls() { realize hlsNode : HLSNode | }
45      where(middleNode : HSVNode2HLSNode |) {
46          middleNode.hls := hlsNode;
47          hlsNode.parent := middleNode.parent.hls;
48          hlsNode.hls := rgb2hls(middleNode.rgb);
49      }
50      map Middle2HLSRecursion {       -- recursive call to visit children
51          middleNode := middleNode.children;
52      }
53  }
```

The transformation starts with the HSV2MiddleRoot mapping whose *Guard-Pattern* on line 2 searches for an *HSVNode* input without a parent. Since the transformation is now solely executed from *hsv* to *hsl*, the `check` and `enforce` keywords have been removed. Wherever a match is found, lines 4-6 realize an *HSVNode2HLSNode* middle node and populate it with the input context and computed RGB value.

The two nested mapping calls on lines 8-14 are then executed sequentially. The *HSV2MiddleRecursion* mapping is invoked with its *hsvNode* guard variable bound to each of the original input nodes children, and its *middleParent* bound to the realized middle node. The *HSV2MiddleRecursion* realizes a corresponding middle node for each input node and recurses down the input 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 an output node and using the *Middle2HLSRecursion* to build the output tree. Notice in this case that since *hls* is an output domain, the `enforce` keyword has been preserved.

This simple example demonstrates the simplifications underlying QVTi and the one syntax extension to QVTc; an explicit mapping call. In order to define this as a QVTc, rather than QVTi extension, we define the invocation on line 8, as providing bound search domains for the two guard variables of the *HSV2MiddleRecursion* mapping named *hsvNode* and *middleParent*. Where collections are provided as the bound domains for non-collection guard variables, a distinct `mapping` invocation occurs for the Cartesian product of each variable from each bound domain. Guard variables not bound by the invocation are bound one element at a time to the whole model. This is a natural extension restricting the declarative search space of QVTc whereby every guard variable is bound to the whole model. For QVTi, we impose the reduced semantics that at most one bound domain may be a collection and no guard variables may be left unbound, thereby ensuring that the invocation involves at most a simple loop.

With these reduced semantics QVTi still has the power to express important programming idioms.

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.

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 the declarative exposition.

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.

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 *assignment* (`:=`) operators in the example.

Traceability Traceability is provided by the middle model. This is user-defined and so allows the user to control how much information relating input and output is maintained.

4.1 Implementation

Some simple QVTi transformations have been implemented using the Eclipse QVTc editor and parser and the Eclipse OCL VM[12].

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 QVTc

AST is an extension of the OCL AST, it is sufficient to extend the OCL EvaluationVisitor to support the additional QVTc 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 implementation of type constructors.

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

The API for type constructors provides `Type.createInstance()` and `Property.initValue()` methods. These were sufficient for the disciplined form of model mutation in QVTi. The extended IOCL VM with Imperative OCL functionality shown in Figure 1 has not been necessary for QVTi; it may yet prove necessary to fully integrate QVTo.

4.2 Future Work

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

We have glossed over the complexities of update transformations by choosing a simple model creation as our example. We can justify this because QVTi is intended to have simple imperative semantics. From this perspective the many challenges of an update transformation are handled by a reconciliation between input and output models for which the QVTc to QVTu transformation synthesizes a solution. The additional complexities of in-place updates are respected if the synthesis ensures that all input/output state is read from output models and cached in the middle model before any potential corruption by an update. Similarly a check mode of operation again requires an input/output reconciliation followed by a simpler transformation that generates a report model.

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.

5 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 efficient execution [4]. 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 [1]. 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[5]. 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 [11], 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[3] 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.

Most similar to the approach proposed in this paper, is the work presented in [9] 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.

6 Conclusions

We have proposed a progressive program-to-program transformation chain from QVTr to QVTc to QVTu to QVTm to QVTi, in order to provide a simple unidirectional imperative language that provides a practical execution semantics for

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

QVTc and QVTr. We have introduced the QVT Imperative (QVTi) language and presented its simple semantics and basic syntax. 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 QVTc to QVTi program-to-program transformations to provide effective execution strategies.

References

1. ATLAS group, L..I.N.: Specification of the ATL Virtual Machine, http://www.eclipse.org/atl/documentation/old/ATL_VMSpecification[v00.01].pdf
2. Bosems, S.: 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
3. Czarnecki, K., Foster, J., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.: Bidirectional transformations: A cross-discipline perspective. Theory and Practice of Model Transformations pp. 260–283 (2009)
4. Fouquet, F., Barais, O., Jéz'equel, J.M.: Building a kermeta compiler using scala: an experience report. In: Proc. Scala Days 2010 (2010)
5. Jouault, F., Kurtev, I.: On the architectural alignment of atl and qvt. In: Proceedings of the 2006 ACM symposium on Applied computing. pp. 1188–1195. SAC '06, ACM (2006)
6. OMG: Request for Proposal: MOF 2.0 Query / Views / Transformations RFP. OMG Document: ad/2002-04-10, revised on: April 24, 2002
7. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. http://www.omg.org/spec/QVT/1.1/ (January 2011), version 1.1
8. Paige, R., Kolovos, D., Rose, L., Drivalos, N., Polack, F.: 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. pp. 162 –171 (2009)
9. Sánchez-Barbudo, A., Sánchez, E.V., Roldán, V., Estévez, A., Roda, J.L.: 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, pp. 42–51 (2008)
10. Stevens, P.: Bidirectional model transformations in QVT: semantic issues and open questions. Software & Systems Modeling 9, 7–20 (2010)
11. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a general composition semantics for rule-based model transformation. In: Proceedings of the 14th international conference on Model driven engineering languages and systems. pp. 623–637. MODELS'11, Springer-Verlag (2011)
12. Willink, E.D.: An extensible ocl virtual machine and code generator. In: Proceedings of the 12th Workshop on OCL and Textual Modelling. pp. 13–18. OCL '12, ACM (2012)