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

Edward D. Willink
Willink Transformations Ltd
Eclipse Foundation
MMT Component co-Lead
OCL Project Lead
QVTd Project Lead
QVTo Committer

OMG (Model Driven Solutions)
OCL 2.3, 2.4, 2.5 RTF Chair
QVT 1.2, 1.3, 1.4 RTF Chair

EXE 2016 @ MODELS 2016
3rd October 2016
Overview

- QVT Background
- Eclipse QVtD architecture
- Do things in the right order
  - imperative/declarative
- Intra-mapping scheduling
- Example results
- Eclipse QVTc/QVTr status
- Conclusion
QVT Background
Query/View/Transformation

2002: standard transformation language RFP
- OMG specification - slow to mature
- ATL took a pragmatic short cut

2005: Three language compromise
- QVTo (Operational Mappings) - Imperative
  - 2 1 good implementations: SmartQVT, Eclipse QVTo
- QVTr (Relational) - Declarative, rich
  - 2 0 poor implementations: ModelMorf, Medini QVT
- QVTc (Core) - Declarative, simple
  - notional common core, no implementations

Eclipse QVTd: QVTc/QVTr editors
Eclipse QVTd Tx Chain Architecture

- QVTr2QVTc - nominally as in QVT specification
- QVTc2QVTu $\Rightarrow$ Unidirectional (remove reverse bloat)
- QVTu2QVTm $\Rightarrow$ Minimal (remove refinement etc)
- QVTm2QVTs $\Rightarrow$ Create graphical form
- QVTs2QVTs $\Rightarrow$ Optimize/schedule graphical form
- QVTs2QVTi $\Rightarrow$ Imperative executable form
Correct Execution 1

- No global state => Object Orientation
- No naughty writes => Static Single Assignment
  - impractical in the large
- No naughty writes => Functional Programming
  - new system, inefficient in the large

But
- multiple threads
- complex object state
- evolving object state
Correct Execution 2

- No naughty reads
  - every property read occurs after its property write

Functions - f(a,b,c) { return a.x + g(b.y.z, c); }
  - parameters easy to analyze - a, b, c
  - references hard to analyze - g(b.y.z, c)
  - => secret undeclared inputs, manual discipline

- Declarative Mappings/Relations/Rules
  - same problem; global analysis necessary/possible
Imperative Transformations

- Explicit control statements
  - hopefully good
  - may be bad

- Manual programming
  - hopefully good
  - may be bad

- Tooling
  - hopefully good
  - may be bad
Declarative Transformations

- No control statements

- Manual programming
  - different approach, may be good/bad

- Tooling
  - must discover a control strategy
  - hopefully good
  - may be VERY BAD
Naive Polling Schedule

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

- Works for any declarative transformation
- Hideously inefficient - VERY VERY BAD
- Optimization goal - a statically ordered schedule
Doubly Linked List Reversal Example
module Forward2Reverse;
create OUT : ReverseList from IN : ForwardList;

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

rule element2element {
    from
        forwardElement : ForwardList!Element
    to
        reverseElement : ReverseList!Element ( 
            name <- forwardElement.name, 
            list <- forwardElement.list, 
            source <- forwardElement.target -- resolveTemp
        )
}
top relation list2list {
  enforce domain forward
  forwardList : DoublyLinkedList {
    name = listName : String{},
    headElement = forwardHead : Element{}
  };
  enforce domain reverse
  reverseList : DoublyLinkedList {
    name = listName,
    headElement = reverseHead : Element{}
  };
  when {
    element2element(forwardHead, reverseHead);
  }
}

top relation element2element {
  domain forward forwardElement : Element {
    list = forwardList : DoublyLinkedList{},
    name = elementName : String{},
    target = forwardTarget : Element{}
  };
  enforce domain reverse reverseElement : Element {
    list = reverseList : DoublyLinkedList{},
    name = elementName,
    source = reverseSource : Element{}
  };
  when {
    list2list(forwardList, reverseList);
    element2element(forwardTarget, reverseSource);
  }
}
Underlying (ATL) functionality

```plaintext
rule list2list {
  from
    forwardList : ForwardList!DoublyLinkedList
  to
    reverseList : ReverseList!DoublyLinkedList
    name <- forwardList.name,
    headElement <- forwardList.headElement -- resolveTemp
}
```
Underlying QVTr in QVTC functionality
Mapping Diagram Artefacts

Diagram showing different types of navigation in a mapping context:
- **to-1 navigation**
- **1-to-1 navigation**
- **to-? navigation**

Artefacts: d (DataType), c (Class), e (OCL), HEAD

States: CONSTANT, LOADED, SPECULATION, SPECULATED, PREDICATED, REALIZED

Date: 3-October-2016

Made available under EPL 1.0
Mapping MoC

Truth - after execution

- to-1 relationships
  - => 1:1 group of objects
  - => HEAD from which all 1:1 objects can be reached
Dependency Conflicts

REALIZE before PREDICATE
Declarative Transformation Execution

- Transformation specifies numerous 'final' truths
  - relationships between output and input model elements

- Execution must proceed step by step
  - permutations of input objects that match mappings
  - compute step sequence at compile time

- Mapping
  - good / useful unit of programming
    - relevant relationships for a few types
  - bad execution step
    - deadlocks between relationships
Micro-Mapping

Executable step in a declarative execution
- no deadlocks between steps

Primitive Micro-Mapping
- many dependencies to be satisfied
- single action - object creation / property assignment

Composite Micro-Mapping
- merge primitives with identical dependencies
- multiple actions
Primitive Micro-Mappings

- One **GREEN** action at a time
- Once **CYAN** predicates satisfied
Speculation

- All Primitive Micro-Mappings share predicate
- Acyclic dependency resolveable at run-time
- Cyclic dependency insoluble
  - need to speculate
- defer predicates
  - ATL ignores inter-mapping predicates
    - works for typical transformations
  - Eclipse QVTd ignores predicates wrt trace creation
    - checks predicates wrt output objects/properties
Speculation Partitioning

Overall

1: Speculating

2: Speculated

3: Residue
Mapping with all dependencies
Scheduled Mapping, pruned dependencies
Overview static schedule

- 2 Mappings
  - => 1 Root, 4 communication buffers, 8 Micro-Mappings,
  - TODO post-scheduling merge
Doubly Linked List Reversal Results

4 GB 64 bit VM, 4 core, Java models
Eclipse QVTD Status

0.12.0 (Mars - June 2015)
- QVTi execution (code generated or interpreted)

0.13.0 (Neon - June 2016)
- preliminary QVTc / QVTr execution
  - low quality - research only
- no incremental / check / in-place facilities
- no debugger
- minimal documentation / examples

1.0.0 (Oxygen - June 2017)
- first release functionality (? with UMLX ?)
Conclusion

- Do things in the right order
  - Mappings declare the order
  - Micro-Mappings can be ordered (graphically)
- First implementation of the QVTC specification.
- First optimized implementation of QVTR.
- First direct code generator for model transformations.
- Thirty fold speed-up.
- Many more optimizations to do.