A concrete UML-based graphical transformation syntax:

The UML to RDBMS example in UMLX

E.D. Willink, EdWillink@iee.org

GMT Consortium, www.eclipse.org/gmt
(Thales Research and Technology (UK) Limited, Reading, England)

Metamodelling for MDA Workshop

M4M’03, York
25 November 2003
Overview

Different QVT perspective
Signal Processing Background
Graphical Exposition

UML and QVT perspective
Use UMLX in less obvious ways

Transformation as a Composable Component
The importance of the Mark Model to MDA

Summary
Transformation Syntaxes

1 Abstract Transformation Syntax (QVT)
1+ Concrete Textual Syntax
1+ Concrete Graphical Syntax

Transformations
to QVT (extract information content)
between formats via XMI or XMID
Textual/Graphical Syntaxes

Text
One Dimensional programming medium
good for precise detail
essential for leaf algorithmic definitions

Graphics
Two Dimensional programming medium
good for higher level context
easier to grasp with appropriate ‘icons’/metaphors

Use of text or graphics should be a user preference
not a vendor/process imposition
UMLX Graphical Syntax Principles

(Unadoption Model Language for Transformations)

Visually: as like UML as possible
- no co-text
  - just embedded OCL
- instance diagram to instance diagram
- multiplicities

Textually: no UML to emulate
- serious limitation in UML
  - OCL refines but doesn’t define
- UMLX may adopt a QVT concrete textual syntax
Transformation

From LHS

To RHS

LHS and RHS are meta-model instances
MDA Models

Platform Independent Model
specification of what needs to be done
too abstract for implementation

Platform (Description) Model
physical, virtual, intellectual resources
available to implement solution

Platform Specific Model
a particular implementation

Mark Model
PIM + PDM specific configuration
Mark Model

**Problem:**
- deploy a reusable PIM on a reusable PM

**Corollary:**
- cannot edit PIM or PM
- all strategic customisation must be in Mark Model

**Customisation:**
- simple parameterisation
- restructure PIM to exploit some Intellectual Property
- restructure PIM to use a smarter algorithm
- restructure PM to configure programmable hardware
- allocate PIM functionality to selected PM elements

**Mark Model is a referential PIM+PM super-set**
MDA Meta-Models

Every Model is an instance of some Meta-Model

traditionally the meta-model has been undefined but implicit
with MDA we model it

UMLX requires and re-uses it
Transform as a Model

- Transform may also be a model
  - traditionally just Java code
  - but even Java code is an instance of the Java meta-model

Closed perspective - frozen executable
- custom code - solves a problem

Open perspective - generic engine, re-usable model
Transform between Meta-Models

Every model instantiates a meta-model
Transformation meta-model reuses input/output meta-model
UMLX extends UML for transformations
UMLX Graphical Syntax Semantics

LHS and RHS are meta-model instance patterns
  association, composition (like QVT partners, GReAT, Gmorph)

Graphical Constraints
  multiplicity (like GReAT, Gmorph)
  OCL constraints (like text syntaxes)
  predicate transformations (like helpers)
  multi-instances
  transformation inheritance

LHS to RHS Transformation operators
  port interfaces (like GReAT, UML 2)
  nested, recursive, concurrent, sequential invocation
  preservation/removal
  evolution (with identity like DSTC tracking)
The UML to RDBMS example in UMLX

LHS - SimpleUML

(from xxxx) identifies package path xxxx.

<<primitive>> stereotype for built-in String used for all attribute values.
The UML to RDBMS example in UMLX

**Instance diagrams**

root: Package
(from SimpleUML)

root.classes.From, root.classes.To are instances underlined
each classes.member is distinct with unit multiplicity

Instance diagram defines one particular instance configuration
Class diagram defines the universe of instance configurations
The UML to RDBMS example in UMLX

UMLX syntax

Transformation Invocation

Transformation Input and Output Interfaces

Transformation Operators
model, database determined by invocation context

LHS formals are:
\{package, class\}

LHS constraints are

- `package` is an instance of `Package`
- `class` is an instance of `Class`
- `class` is a `package.member`

if the `package` is of base type `Package`

for each `package.member` of base type `Class`

evolve a Table as `database.table`
UMLX requires an input and output context

Package and Database added to example

Tables get a hierarchical name

t_package1_package2_class

prefix seeds the name building
The UML to RDBMS example in UMLX

Uml2Rdbms.PackageToDataBase

Delegates for each sub-problem
- nested packages
- nested classes
- associations
Uml2Rdbms.PackagesToDataBase

Recurses for each nested package
prefix name grows, using OCL syntax

Nested transformation for nested match
existence of a child match does not affect parent match
Table and primary Key for each persistent Class
Delegates again for the nested attribute problem
Delegates the conversion of each attribute
to all alternative strategies
Inheritance

Uml2Rdbms.\textit{AttrToColumn} Interface

- prefix
- attribute
- rootClass

prefix : String $\ll\text{primitive}\gg$
\hspace{1cm} (from SimpleUML)

attribute : Attribute
\hspace{1cm} (from SimpleUML)

rootClass : Class
\hspace{1cm} (from SimpleUML)

Semantics

each set of instance matches
therefore ALL leaf transformations invoked
static analysis can eliminate most invocations
for-each package.association

with a persistent source and target Class,

evolve a correlated foreign key ...
Evolution Identity Semantics

\textbf{q evolves}

via the conversions \{a, b, c\}

from the actual LHS instances \{a: \{x\}, b: \{y, z\}, c: \{#\}\}

# is unique for each transform invocation

\{,,\} is an unordered set

\textbf{Same} \{a: \{x\}, b: \{y, z\}, c: \{#\}\} \textbf{is same} q
Evolution Identity Example

Identities:

{tableForClass:source}

{keyForClass:target}
Match order

Search Problem:
find each \{ \textit{package}, \textit{association}, \textit{source}, \textit{target} \}
satisfying the (UML graphical) constraints
Naively a 4D total model search.
But

\textit{package} is given,
1D iteration over \textit{package}.member
then \textit{source}, \textit{target} known (unit multiplicity)

Efficient matching can be derived from input meta-model
automatic ‘loop hoisting’

Imperative loop becomes Declarative existence multiplicity
Imperative test becomes Declarative constraint
Auto-transformation sequencing

Edit-compile-link-run
   easy to remember, but not foolproof
100 or even a 1000 transformations
   cannot be sequenced manually
Meta-models provide the answer
PIM makes use of a variety of modelling concepts
PM defines the concepts acceptable to the target platform
Each transformation defines its input and output concepts
At each node in a valid transformation sequence:
   concepts-in-use ⊆ acceptable-concepts
Observations

Somewhat verbose
- visually clear, 2D rather than 1D source problem exposition
- precise
- handles irregularities, recursions

Small basic syntax for now
- more compact syntax can be added
  mapped to basic syntax
Scalability

Graphical syntax can scale badly

UMLX diagrams become cluttered
  if too much happening

If too much happening
  factor out partial patterns as predicates
    re-usable, readable
  factor out multiple activities as concurrent transforms
    reusable, avoids multiplicity interactions
  introduce pivot models and sequential transforms
    simpler, reusable, eliminate ordering subtleties
Composition

Re-usable black boxes

Encapsulation
- internal data, internal behaviour

Defined Interface
- external data, external behaviour

Ports define where the external data flows
Sequential Composition Scheduling

‘Data’ flow defines when transformation occurs
when the input model(s) exist, then the output(s) get created

cf. Token flow for UML 1.5 Action Semantics

cf. Token flow for UML 2 Activity diagram

cf. (Unit Rate) Data Flow for Signal Processing

cf. Petri Nets
Concurrent Composition Scheduling

Deterministic with respect to input
- Shared, read-only LHS

Deterministic with respect to output
- Shared, non-conflicting write-only RHS

Deterministic with respect to output merge
- Evolution identities define how multiple concurrent match results are combined
- Multiple concurrent transform results are combined
- May have \{ordered\} collections

Deterministic with respect to in-place update
- Declarative definition, no read-after-write
Transformation in Program Context

Start state comprises the input multi-model
End state comprises the output multi-model

For QVT:
Transformation hierarchy is the full program from external input multi-model to extern output multi-model

More generally:
Transformation is a more powerful form of state transition from source state multi-model to target state multi-model
QVT as the missing Programming Link

Class Diagrams etc
    static definition of leaf and nested structure

Statecharts
    dynamic (OR) composition of nested behaviour

Block Diagrams (UML 2)
    static (AND) composition of nested behaviour

Transformations (QVT) Diagrams (UMLX)
    dynamic definition of behaviour

User-level Transformation
    portable dynamic program specification

Compiler-level Transformation
    standard and custom DSL, MDA, compiler transformations
UMLX Status

**UMLX editor configured within GME**
by defining a UMLX editing meta-model

**UMLX compiler designed in UMLX**
over 200 diagrams so far

**UMLX execution engine designed in UMLX**
over 100 diagrams so far

**Bootstrap by manual transliteration to NiceXSL**
human-friendly form of XSLT

**Prototype generates XSLT code**
currently for concurrent and sequential transforms
(OCL complexity limited to \texttt{name = ‘literal’})

\[\Rightarrow \text{Fast, configurable, custom compiler-compiler}\]
Generative Model Transformer (GMT)

www.eclipse.org/gmt

Open Source, Flexible, Extensible MDA tool

Fit for use within industry

So far:

project established, UMLX and Fuutje prototypes

Soon:

- Eclipse workflow framework
- Arbitrary tools as (workflow) transformations
  - wrappers for non-standard interfaces (typically via XMI)
- Arbitrary model storage between (workflow) transformations
  - memory or disk location
  - translation between standard meta-model formats
Summary

**UMLX is a natural extension of UML**
- opportunities for more elaborate/useful syntaxes

**UMLX exhibits good composition properties**
- declarative transform sequencing
  - can degenerate to imperative
- declarative transform concurrency
  - can be \{ordered\}
- arbitrary OCL at innermost level

- influence the QVT abstract syntax

**UMLX and GMT progressing**