UMLX : A graphical transformation language for MDA

Edward D. Willink, EdWillink@iee.org
GMT Consortium
www.eclipse.org/gmt
4 September 2003

Abstract

With the increased use of modelling techniques has come the desire to use models as a programming language as part of a Model Driven Architecture. This desire can now be satisfied by exploiting XMI for model interchange and XSLT for model transformation. However the current transformation techniques are far removed from modelling techniques. We therefore describe a graphical transformation language, which involves only minor extensions to UML but constitutes a high level language for transformations.

1 Introduction

The Object Management Group (OMG) has issued a Request For Proposal [16] for a Query / Views / Transformations (QVT) language to exploit the Meta-Object Facility (MOF) [15], which as from version 2.0 should share common core concepts with the Unified Modeling Language (UML) 2.0 [17]. The initial submissions of 8 and first revised submissions of 5 consortia[1] have been made, and somewhat surprisingly, only one of them [18] uses a partial graphical representation and another [11] just a graphical context for their language.

This paper describes independent work to provide an Open Source tool to support the OMG's Model Driven Architecture (MDA) initiative [13]. A primarily graphical transformation language is described that extends UML through the use of a transformation diagram to define how an input model is to be transformed into an output model. This work has much in common with two of the QVT proposals [12] [18], and it is hoped that it is not too late for some of the ideas in UMLX to influence revised QVT proposals.

Proprietary file formats once created significant impediments to sharing or re-use of information between software tools. Fortunately, the advent of XML [20] is steadily eroding these barriers, and the ease with which XML can be read and written using the Java DOM [19] support makes XML a natural choice for new applications. When information is stored in an XML format, albeit with a proprietary schema, it is possible to deduce the schema and gain access to the information content. The advent of XSLT [21] has made transformation between XML formats relatively easy, so that as standard XML formats are agreed, proprietary or legacy formats can be accommodated by translators.

Both XML and XSLT, which is an XML dialect, are effective compromises between man and machine intelligibility, but as compromises they leave plenty of scope for more user-friendly representations. There are therefore a variety of data modelling tools, many based on UML, that provide greater rigour in the use of an underlying XML representation, increasingly exploiting the stronger disciplines of XMI [14]. There are, however, few tools that hide the impressively compact and sometimes dangerously terse underlying XPath representation. We will now describe one such tool and its associated language: UMLX.

Before we review the MDA and its supporting transformations in Section 3, we introduce the UMLX concepts in Section 2, where a simple example demonstrates that transformations are applicable for specification of conventional program execution as well as for program compilation. In Section 4, we go a stage further and discuss the use of transformations to specify the UMLX compiler compilation. We then summarize the current compiler status and future plans in Section 5. Finally we discuss related work.

2 UMLX

UMLX uses standard UML class diagrams to define information schema and their instances, and extends the class diagram to define inter-schema transformations. We will therefore use as much standard practice as possible in our introduction of the additional UMLX transformation concepts. We describe an address book with email and telephone contact details as a running example, as we move from the hopefully familiar territory of information modelling, on through transformations at the program-level, via the compiler-level to the compiler-compiler-level.

[1] The author is not directly associated with the OpenQVT consortium of which his employer, Thales Research and Technology (UK) Limited as part of Thales, is a member.

In this section we just introduce the UMLX extensions to UML. The very important area of compiler transformations is deferred until Section 3 and the reader is referred to [9] for a presentation of the standard UML to RDBMS example using UMLX.

2.1 Schema Definitions

An information model is defined using a schema, which uses the sub-set of UML class diagram syntax appropriate to information modelling. This syntax is summarized in Figure 2.1.

In Figure 2.2 we model an `AddressBook` with many `Contacts` for each of many `Entrys` using composition relationships. The two types of `Contact` are modelled using inheritance of `Phone` and `Email` from the abstract `Contact`. The national, regional and local parts of a phone number are modelled using distinct objects. The national dialling codes are modelled using one object per country within the `AddressBook`, so that the phone number references the appropriate country object using a navigable association. All other information, such as the name of the `AddressBook` owner, is modelled using attributes.

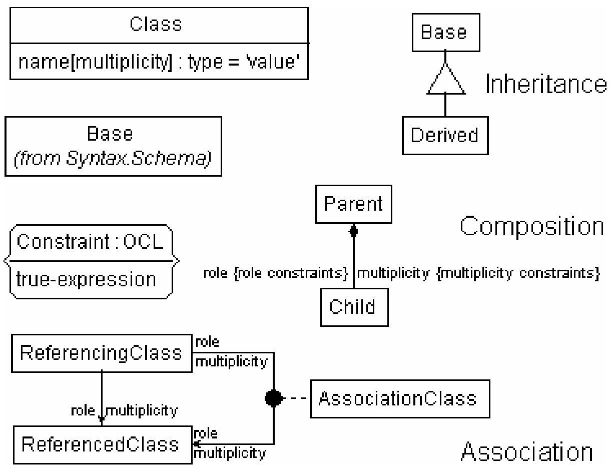

Figure 2.1 Schema Syntax

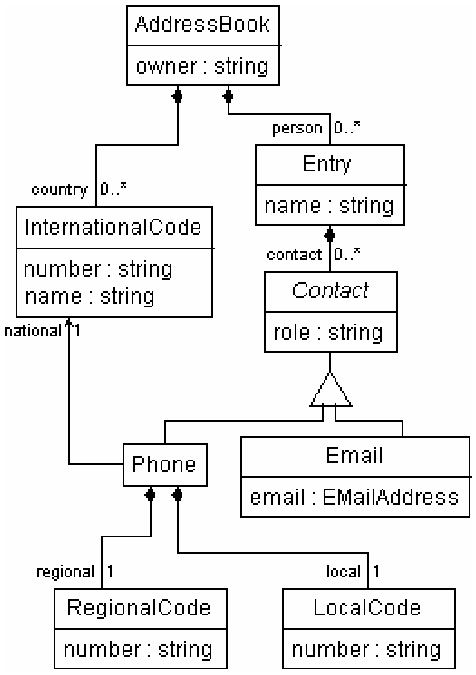

Figure 2.2 Example Schema

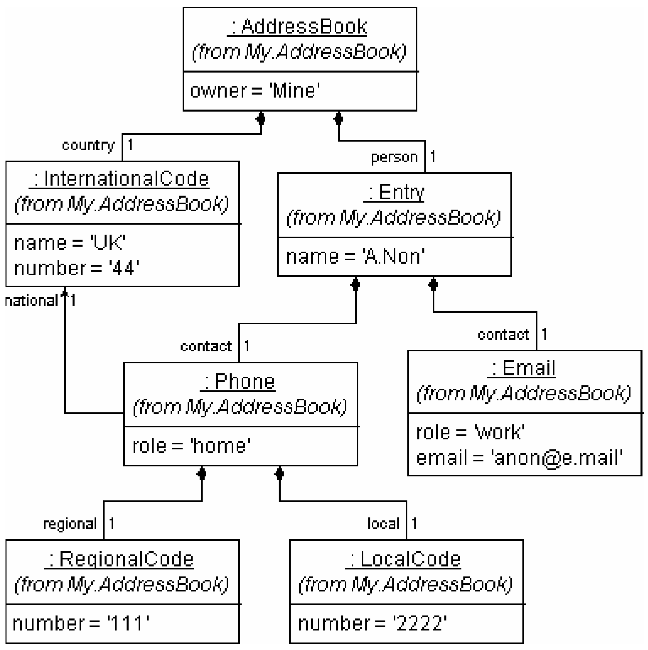

Figure 2.3 Example Instance

2.2 Schema Usage

A schema defines all possible information sets that comply with the information model. A specific instantiation or usage of a schema may be defined graphically using instances. Figure 2.3 shows an address book instance with just one entry with two contacts.

2.3 Schema to Schema Transformation

Instances of schemas may be maintained by a wide variety of often proprietary tools, which provide internal capabilities to transform to external formats, but generally prevent direct access to the internal representation. However, when a standard XML format is used we may look to start applying custom transformations.

The UMLX extensions that support transformations are summarized in Figure 2.4.

It is convenient, though not necessary, to draw transformations with inputs on the left and outputs on the right so that we can refer to the left hand side (LHS) as the pre-transformation or input context and the right hand side (RHS) as the post-transformation or output context.

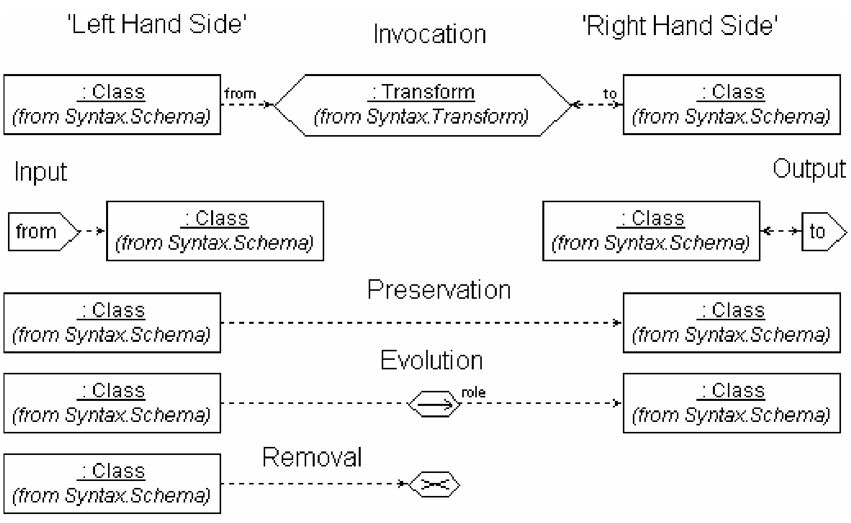
'Left Hand Side'
Invocation
'Right Hand Side'
: Class (from Syntax.Schema)
from
: Transform (from Syntax.Transform)
to
: Class (from Syntax.Schema)
Input
Output
from
: Class (from Syntax.Schema)
: Class (from Syntax.Schema)
to
Preservation
: Class (from Syntax.Schema)
: Class (from Syntax.Schema)
Evolution
: Class (from Syntax.Schema)
role
: Class (from Syntax.Schema)
Removal
: Class (from Syntax.Schema)

Figure 2.4 Transformation Syntax

A hierarchical transform has an *Invocation* context established by binding LHS contexts to input ports and RHS contexts to output ports. The *Input* and *Output* syntaxes define how these ports are in turn associated with the internal LHS and RHS contexts. The *Preservation*, *Evolution* and *Removal* syntaxes define the contribution of an LHS construct to the RHS. The LHS is always unchanged. We hope that the usage of these syntaxes will adequately correspond to intuition as we progress our running example. More details are given in [8].

A too frequent problem with address books is the requirement to update phone numbers to adjust to the changed policies of the phone companies. A transformation to change all UK numbers with the regional code 111 to 10111 may be defined in UMLX as shown in Figure 2.5.

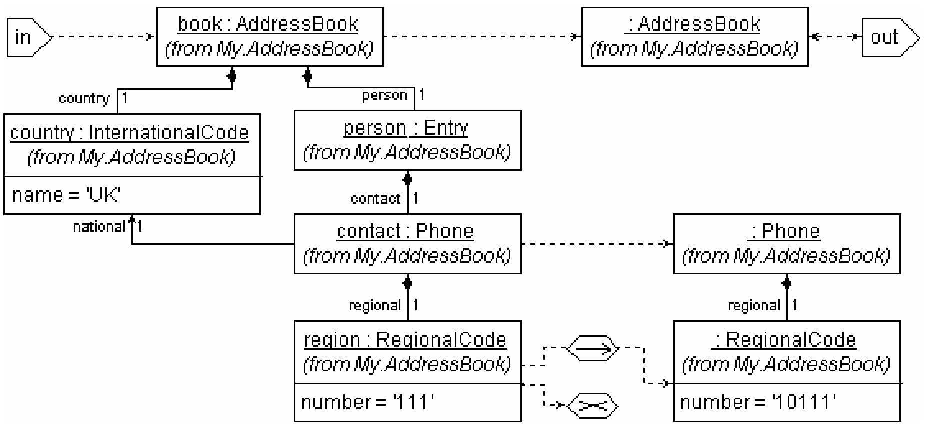
in
book : AddressBook (from My.AddressBook)
: AddressBook (from My.AddressBook)
out
country 1
person 1
country : InternationalCode (from My.AddressBook)
name = 'UK'
person : Entry (from My.AddressBook)
contact 1
national 1
contact : Phone (from My.AddressBook)
: Phone (from My.AddressBook)
regional 1
regional 1
region : RegionalCode (from My.AddressBook)
number = '111'
: RegionalCode (from My.AddressBook)
number = '10111'

Figure 2.5 Explicit Phone Number Transformation

At the extreme top left and right hand side of the diagram are two port icons that define the external interface of the transformation. The `in` port accepts an `AddressBook`, or an instance of some class derived from `AddressBook`, with the unidirectional arrow providing a visual indication that the source may be read but not updated. The `out` port similarly handles an `AddressBook`, and the bidirectional arrow provides a visual reminder that the write-only result is shared and so may be updated by concurrent non-conflicting matches of this and other transformations.

The diagram comprises two schema instantiations, the one on the left connected to the input defines a structure to be discovered in the input model, each of whose instances trigger the transformation to produce the schema instantiation on the right hand side. A structure is an arrangement of objects with connectivity, multiplicity, type, value and other constraints. (We refer to structures rather than patterns to avoid confusion with the more abstract concept used in the pattern community. The concepts are closely related; structures form part of the specification of a re-usable solution to a problem in the transformation context, whereas a pattern concerns a recurring problem in a more general context.)

A distinct structure match is detected for each matching set of objects in the input model, which in the example means a match for each contact whose national name is UK, and whose regional number is 111.

The explicit preservation between the two `AddressBook`s provides an implicit preservation of its composed contents. Implicit preservation or removal recurses until an explicit transformation operator dominates, as for the matched `Phone` contact. The old regional code is excluded from the output by the removal, and replaced by the evolution of a new regional code with the changed value.

The interpretation of cardinalities in a transformation deserves clarification. In a schema, cardinalities define the bounds against which the application elements of a particular schema instance may be validated. In a transformation, the cardinalities represent the multiplicities that must be satisfied by each match; there is a distinct match for each possible set of correspondences between transform and application elements for which all cardinalities are fully and maximally satisfied.

We may therefore interpret the left hand side match as

```
given a book of base-type AddressBook
 for-each person of base-type Entry in book.person
  for-each contact of base-type Phone in person.contact
   for-each region of base-type RegionalCode in contact.regional
    for-each country of base-type InternationalCode in book.country
     if country is referenced by contact.national
      if country.name is 'UK'
       if region.number is '111'
        <match found at book,person,contact,region,country>
```

and the transformation action in the context of each match as

```
within the context of the preserved matched contact
 remove the old regional child object
 create a new regional child object of type RegionalCode
  with number set to '10111'
```

When a model is known to comply with its schema, there are many optimizations that can be applied:

- Only the `Phone` type needs validation, since it is the only derived type in the transformation.
- The region loop is redundant since its multiplicity is exactly one.

The sequencing of the loops is also subject to optimization. For instance, if an implementation has a fast look-up key for country, that loop and its conditional could be performed first. Conversely, we may analyze this and other transforms and choose to synthesize an implementation in which there is a fast look-up for country.

The relationship multiplicities determine the complexity of the matching, and in this example all multiplicities have been unity. Other multiplicities, such as zero, which requires an absence of matches, or more than one which may match combinatorially, are discussed in [8]. When multiplicities are applied to hierarchical transformations, predicates and sub-matches are supported.

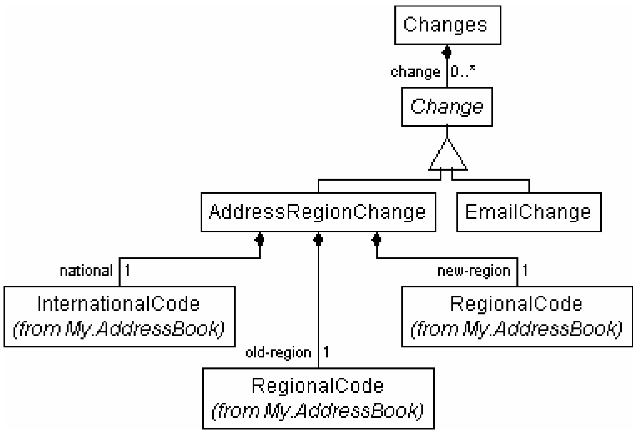

Figure 2.6 Contact Changes Schema

Consistent resolution of overlapping matches between concurrent transforms is made possible by the concept of an evolution identity. Each evolved RHS transformation entity has a formal signature determined by the set of evolutions from which it evolves. Each of these evolutions may in turn be associated with a set of LHS entities. The correspondence of actual LHS instances in the discovered match to the LHS instances in the formal signature defines the identity of the actual RHS entity. Again more details may be found in [8].

The example application can be made a little more useful by defining the additional schema for changes shown in Figure 2.6. We may now seek to apply a batch of `AddressRegionChange`s and/or `EmailChange`s. An individual `AddressRegionChange` may be realized by the slightly changed transformation, shown in Figure 2.7 that now takes two inputs, and matches where common `<<primitive>> String` values are found. A match therefore occurs for the combination of each `Phone` contact that matches an `AddressRegionChange`.

This transformation may be invoked hierarchically as shown in Figure 2.8. The two incoming models, an address book and a change list, are merged to produce an updated address book.

The outer transform finds a distinct match for each `Change`, and attempts to apply both the `ApplyAddressRegionChange` and the `ApplyEmailChange` sub-transformations. However the use of

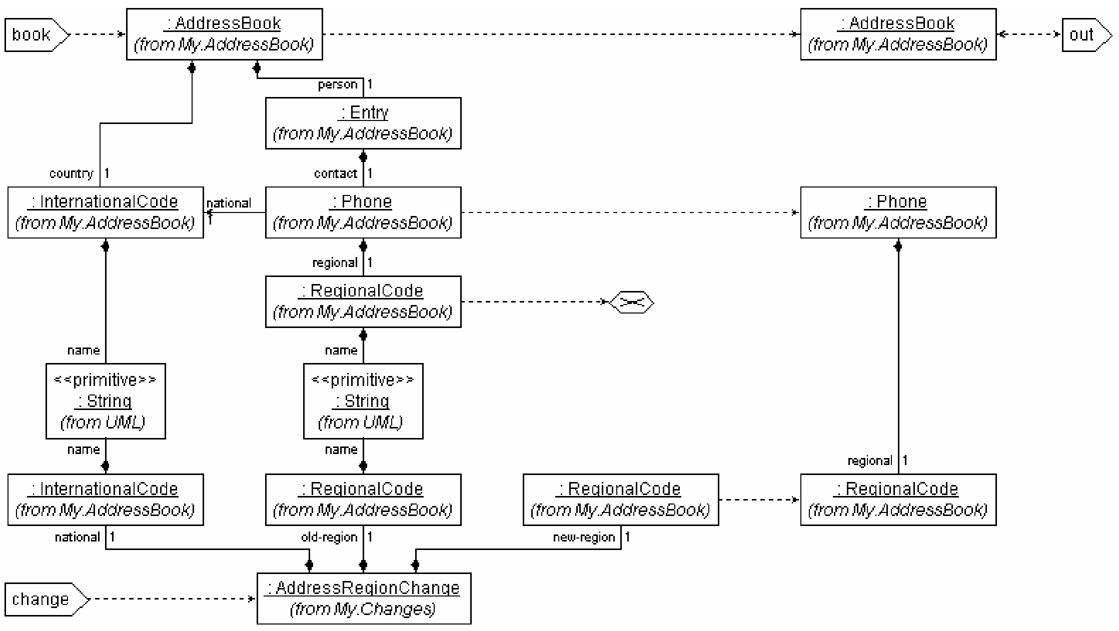
book
: AddressBook (from My.AddressBook)
: AddressBook (from My.AddressBook)
out
person 1
: Entry (from My.AddressBook)
country 1
contact 1
: InternationalCode (from My.AddressBook)
national
: Phone (from My.AddressBook)
: Phone (from My.AddressBook)
regional 1
: RegionalCode (from My.AddressBook)
name
name
<<primitive>> : String (from UML)
<<primitive>> : String (from UML)
name
name
regional 1
: InternationalCode (from My.AddressBook)
: RegionalCode (from My.AddressBook)
: RegionalCode (from My.AddressBook)
: RegionalCode (from My.AddressBook)
national 1
old-region 1
new-region 1
change
: AddressRegionChange (from My.Changes)

Figure 2.7 ApplyAddressRegionChange Transformation

a derived input type, `AddressRegionChange`, in `ApplyAddressRegionChange` ensures that only the transformation applicable to the actual `Change` progresses.

Having modelled an address book and a mechanism for automating changes, we can adapt to changes more easily. When a standard AddressChangeML dialect emerges, the update code can be remodelled on the new standard, or the change mechanism revised to define a transformation from the new change standard. XML files can then supersede or at least augment informal change of address emails or cards for address change notification. A transform may similarly be created to rescue our address book following an upgrade to a tool that uses a new standard AddressBookML.

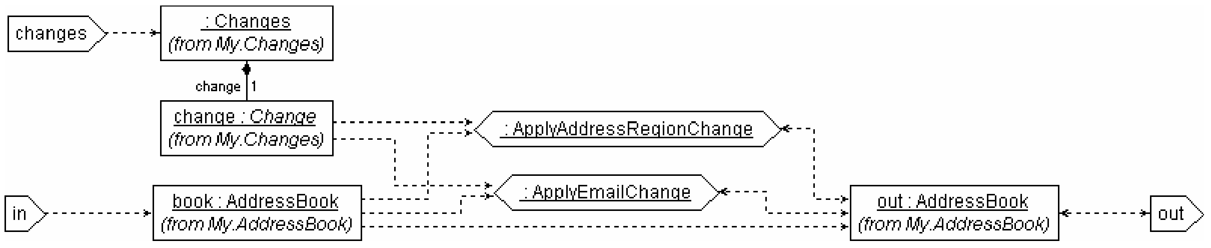
changes
: Changes (from My.Changes)
change 1
change : Change (from My.Changes)
: ApplyAddressRegionChange
: ApplyEmailChange
in
book : AddressBook (from My.AddressBook)
out : AddressBook (from My.AddressBook)
out

Figure 2.8 Compound Transformation

3 The Model Driven Architecture

In support of the MDA, we want to transform portable Platform Independent Models (PIM) into efficient Platform Specific Models (PSM).

3.1 PIMs, PSMs and PDMs

The models presented so far have been PIMs; they only specify required functionality. Their persistent representation may use a database, an XML file, or a proprietary format. Their functional implementation may use database queries, XML transformations or proprietary code. They may therefore be simulated, using whatever representation and implementation a simulator chooses. However, to produce a practical application, they must be converted to PSMs that incorporate extra platform information. It has been common practice to either produce PSMs in the first place and thereby lose portability, or to redraw the PIMs as PSMs and thereby create a disconnect between design and implementation models.

The additional context is provided by a Platform Description Model (PDM), which should also be re-usable since we may require many different PIMs to operate in the same context. The PDM may comprise descriptions of

- component/Operating System/instruction set capabilities
- language/assembler type systems
- driver/hardware interfaces
- communication protocols
- network connectivity
- library resources
- tools

Neither PIM nor PDM should be modified to field a specific PIM to a specific PDM, so we need a further model in which to capture the mapping requirements unique to a particular PIM and PDM combination. The transformation language that can implement the merge of PIM and PDM to produce a PSM is a sufficient topic for this article, so we will just suggest that after some elaboration, a hierarchy of UML Deployment Diagrams may fulfil this role as shown in Figure 3.1.

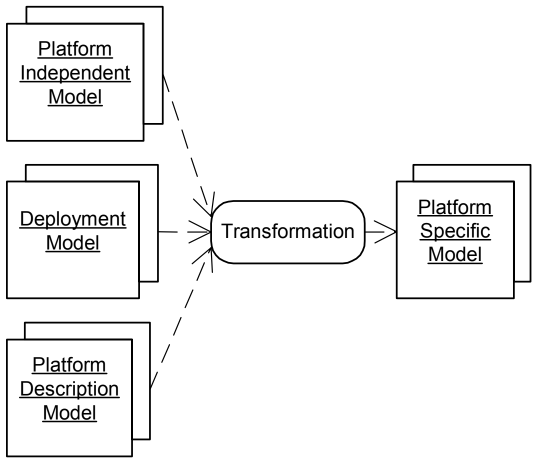

Figure 3.1 MDA Models

Two re-usable models must be merged and transformed under control of the third. There are many different problems that must be resolved:

- alignment of specification attribute types to the implementation type system
- reification of compositions and associations
- selection, parameterization and interfacing of library elements
- partitioning of specification regions to execution units
  (classes or components or processes or processors or …)
- selection of communication policies between execution units
- selection of scheduling policies for execution units
- activation of communication policies between execution units
- activation of scheduling policies for execution units

Code generation may then be performed in favoured language(s).

The above list is incomplete for conventional applications, and we should add any aspects that may be of concern to particular applications

- establishment of an error handling policy
- introduction of fault tolerant redundancy
- distributing persistence
- validation of throughput capabilities
- dynamic or static load balancing
- array distribution and cache coherence

There are clearly far too many different problems to solve all at once, so a practical tool must modularize them so that they are resolved one at a time, with as little interaction as possible. What is shown in Figure 3.1 as a single transformation is in practice a compound transformation, comprising many sequential, concurrent and hierarchical sub-transformations with many intermediate pivot models.

Some of these problems may be resolved by patterns, with a transformation library supplying established solutions that can be applied in response to the problem primarily defined by the PIM, the context primarily defined by the PDM and extra forces perhaps in the Deployment Model. The compound transformation must therefore adapt, in some cases through automatic recognition of PIM concepts that must be eliminated. However, it is unlikely that a satisfactorily efficient conversion can be fully automated, so iteration to allow elaboration of the Deployment Model with sufficient guidance will be essential. This will require dynamic activation of analysis and diagnostic transforms to assist the system designer.

3.2 Compiler Transformation

A number of categories of compiler transformations have been listed above. Some rather more obvious examples are provided by reification of specific UML concepts.

- Translation of Abstract classes to Interfaces in Java or pure virtuals in C++
- Translation of State Machine States and Events into Classes
- Translation of UML relationships into Operations and Attributes
- Translation of a UML model to an RDBMS model [9]

We will give one simple example. UML diagrams comprise boxes for concepts with lines for relationships between them. A few boxes correspond directly to language constructs such as classes, but for the other boxes and lines, it is necessary to find an appropriate implementation approach. An example of one form of relationship without a direct language counterpart is a composition.

Whereas in the earlier examples we were considering the program level where execution involves instances of application concepts such as `AddressBook` and `Contact`, we now consider the compiler level where compilation involves instances of language concepts such as `Class` and `Composition`.

The composition shown in Figure 3.2 may be transformed into a `Sequence(Contact)` member variable in `AddressBook`, where `Sequence()` is the OCL collection type. Further transformations, close to code generation, are required to convert it to the appropriate Java, C++ or VHDL equivalent.

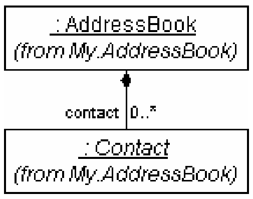

Figure 3.2 UML Composition

We do not normally want to specify a separate transformation for each composition individually; rather the same transformation policy may form part of a package of Object Oriented transforms and can be applied to all compositions, so we specify a transformation on the UML meta-model:[2]

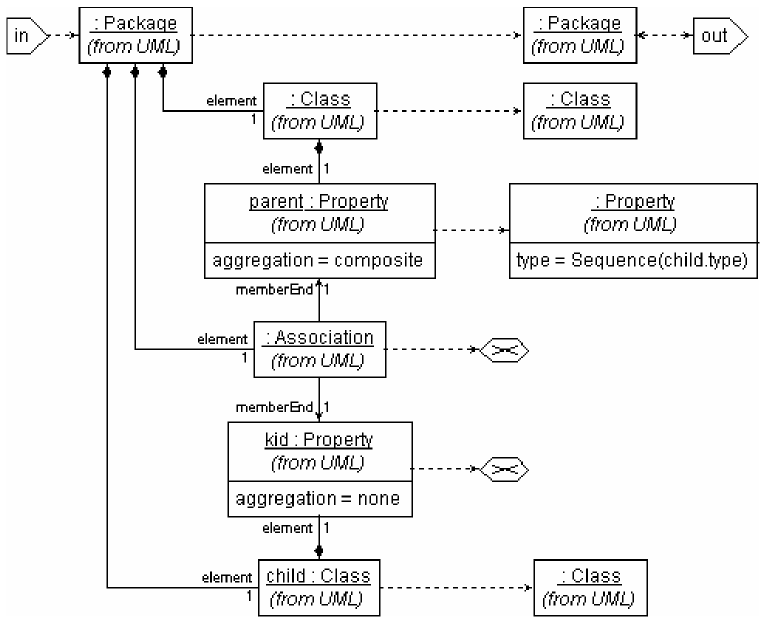

Figure 3.3 Composition Transformation

Here the left-hand side defines the structure involving the `Association` that represents the composition line, the two `Property`s that represent each line end and the two `Classes` within a `Package`. In UML, the distinction between associations and compositions is made by the graphical attributes on the line ends, so we apply constraints to indicate that one end should have a `composite` diamond and the other `none` as its decoration.

Wherever this structure is found, the transformation specifies that the `Association` and `kid Property` be removed, and that the type of the `parent` property be changed to `Sequence` of the `child` type.

This transform is far from complete, since any practical implementation of a composition must also provide support code to ensure that the child and its contents are appropriately constructed and maintained.

[2] Specific classes or compositions may be transformed by additional constraints on the class names.

The transform is therefore just a part of the support necessary for the UML Composition concept, which is just one of many concepts in need of transformation. It also forms just one of a number of passes; further transformations will be necessary to progress the OCL concepts into a particular programming language.

4 UMLX Compiler

Use of UMLX enables the problem of model transformation that lies at the heart of MDA to be expressed using modelling technology. The meta-models for each of the input and output models are used by the meta-model of the transformation model. This transformation meta-model is compiled to produce the transformation model (or program) that is executed by some transformation engine to realise the required transformation.

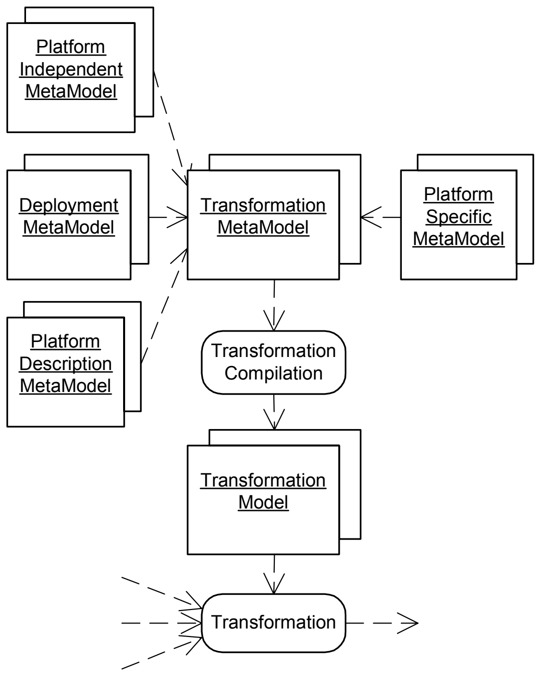
Platform Independent MetaModel
Deployment MetaModel
Transformation MetaModel
Platform Specific MetaModel
Platform Description MetaModel
Transformation Compilation
Transformation Model
Transformation

Figure 4.1 MDA Meta-Models

XML technology is appropriate for transfer of models between tools and transformation engines. The transformation engine may then be realised using any technology for which XML import and export are supported, and a transform compiler is available.

An editor for UMLX has been implemented using the GME [7] meta-modelling tool, and a capability has been added to support XMI export.

A compiler for UMLX is being defined using UMLX meta-models, and manually implemented using XSLT, or rather NiceXSL[3]. Once this implementation is operational, it should be possible to use it as a bootstrap to regenerate itself from the UMLX meta-models.

Implementation of the compiler using XSLT is particularly straightforward since XML models can be imported directly; a remarkably small transformation engine just interprets the activities encoded in the transformation model.

Once code generators for C++ or Java have been modelled, the transformation compiler can generate more efficient or more portable code to implement transformations. And since the UMLX compiler is defined in UMLX, the transformation compilation can be regenerated with similar benefits.

Thereafter, development of optimizations described in Section 5 may proceed in UMLX, without further recourse to XSLT. Once these optimizations are in place, it will be possible to build an efficient custom compiler from a set of user-selected transformations. This custom compiler need only be regenerated when the meta-models are changed.

More significantly, it will be possible for a custom compiler to be developed, based on the re-usable transformations, and adapted for unique application requirements. This requires a transformation library for the various stages of transformation from UML specification to implementation.

Looking further ahead, if compilers and synthesis tools are restructured as transformation engines, the increasingly powerful but specialized algorithms that they contain may be exposed for similar intervention.

It is the aim of the Generative Model Transformer (GMT) project at Eclipse to provide an Open Source transformation tool and a library of transformations so that this all becomes possible.

The UMLX compiler, in its current bootstrap state, comprises over 100 diagrams, so it is clearly inappropriate to present them all here. We will therefore just present two diagrams to demonstrate the need to support multiple inputs, multiple schema, workflow and hierarchical transformations, and to introduce the UMLX usage of multi-objects and inheritance.

[3] NiceXSL is a more conventional textual representation for XSLT. Translators to and from XSLT are available from http://www.gigascale.org/caltrop/NiceXSL.

4.1 Transformation Compilation Example

The top level diagram of the UMLX compiler, shown in Figure 4.2, demonstrates the need for multiple inputs with intermediate pivot models. The two inputs comprise the schema defining the data meta-models, and the transformations between these meta-models. Two intermediate pivot models are produced by annotating the input models with derived information to simplify the subsequent compilation activity.

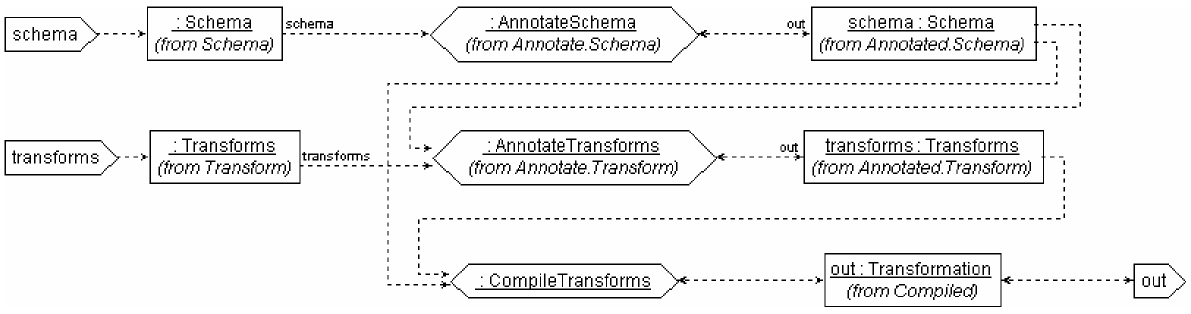

Figure 4.2 Transformation Compilation

One important part of the schema annotation pass is to attach a transitive list of all base classes to each class, so that subsequent tests for inheritance and name visibility can be made without repeated traversal and resolution of name occlusion within the inheritance hierarchy.

The complexities of multiple inheritance require the use of two passes, first to identify the base classes and then generating the annotations describing them. Figure 4.3 shows the main recursion to gather another layer of base classes with respect to a `new` work-list of as-yet-unanalysed base classes, and an `old` work-list of already-analysed base classes, all within the context of a particular `schema`. (The recursion is started with an empty `old` list, and a `new` list containing just the class to be analysed.)

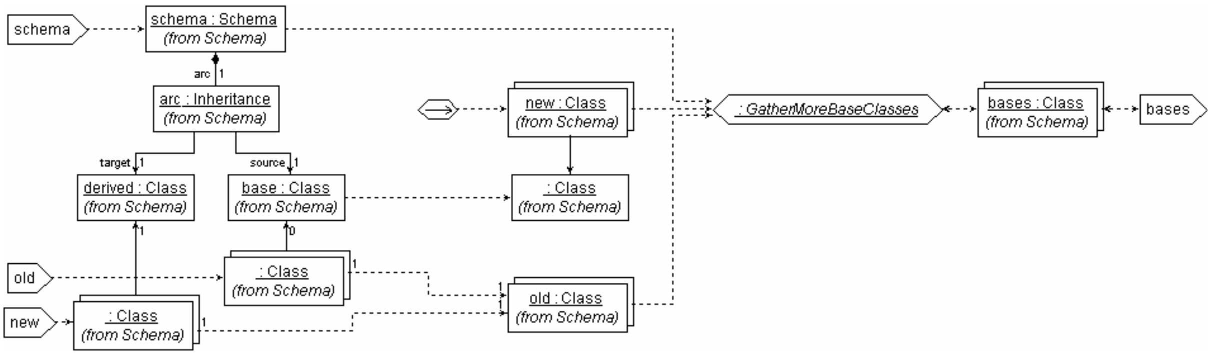

Figure 4.3 GatherYetMoreBaseClasses

UML multi-objects are used for the work-lists, and so the left hand side may be read as

```
for each arc of base type Inheritance in schema
  for which base at arc.source has not yet been analysed
  and for which derived at arc.target has yet to be analysed
    <generate a match response for base>
```

The 0 multiplicity on elements of the incoming `old` work-list requires the `base` match to be absent from the already-analysed work-list, and so avoids revisiting multiply inherited bases.

The 1 multiplicity on elements of the incoming `new` work-list requires the `target` match to be present in the as-yet-unanalysed work-list.

This is a multi-pass transformation, and so the centre of the diagram, which we may refer to as the middle side acts as the right hand side of the first pass, and the left hand side of the second. The middle side directs that a new `new` work-list be evolved (and shared across all concurrent matches) so that each match contributes its newly found inheritance `source` to the work-list. The preservation of incoming `old` and `new` work-lists as the middle `old` work-list, with redundantly explicit 1 to 1 multiplicities, directs that for each incoming work-list each element is preserved in the resulting work-list, and that the resulting work-list contains exactly one copy of each incoming element. There is therefore a merge, with removal of duplicates.

Once the left hand side to middle side sub-transformation is complete, the middle side to right hand side can progress by invoking a recursive transform. The recursion involves two distinct cases and so it is convenient to define and invoke an abstract transformation from which the distinct cases derive.

Invocation of the abstract transformation invokes all the derived transformations shown in Figure 4.4. We have just seen that `GatherYetMoreBaseClasses` matches all additional base classes. However, if there are no base classes there will be no LHS matches and consequently no middle side from which to recurse. It is therefore necessary for `GatherNoMoreBaseClasses` to match at the end of the recursion. This simpler case differs from Figure 4.3 through specification of zero for the `Schema` to `Inheritance` multiplicity, and use of the merged work-lists as the `bases` result without recursion.

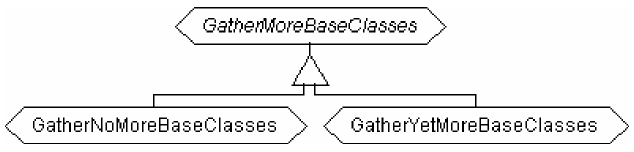

Figure 4.4 Transformation Inheritance

The example just presented is a compromise between identifying a simple enough part of the compiler to be presented in isolation, while showing sufficient complexity to reflect real problems. There are parts of the compiler that are significantly more complicated and the author regrets that he was unable to program these areas correctly first time in XSLT, and as a result has engaged in a distressing amount of empirical development. As the UMLX compiler has progressed, it has been possible to exploit visual symmetries within UMLX diagrams to make complex relations easier to understand, and with the ability to machine check the validity of each drawn relationship against a meta-model, UMLX now enables subtle XPath expressions to be constructed with far greater predictability and will soon support fully automated generation. The full set of UMLX diagrams for the UMLX compiler and executor may be found in [10].

5 Current Status and Future Work

An editor for UMLX has been configured and most of a bootstrap compiler has been designed using UMLX and implemented in NiceXSL. This already shows useful ability to validate UMLX designs and generates XSLT that successfully applies a concurrent and sequential transform hierarchy to models provided that any OCL expressions are kept simple.

The main priority is to raise the functionality level to the point where the bootstrap compiles its own design to produce a viable compiler. Progress can then be made on the compiler and on a library of standard transformations to support at least MDA. Compiler work will involve

- single transform optimizations that exploit properties of schemas to improve the speed of structure matches, and generate code more efficiently
- concurrent transform optimizations that sequence matches to maximize the sharing of partial match contexts exploiting fast indexing approaches
- sequential transform optimizations to eliminate overheads by combining transforms and sharing intermediates
- code generation to Java, C++ to improve the speed of structure matches, and generate code more efficiently.

This should produce an increasingly viable compiler for XMI to XMI or text transformations that may then be integrated as an additional code generator behind configurable UML tools.

In parallel with this, work on the basic MDA tool box is needed to support

- type resolution
- processor allocation
- component configuration
- performance assessment
- common patterns
- etc. etc.
- code generation to various implementation languages (C++, Java, SQL, XML, VHDL, …).

It is hoped that UMLX can provide a graphical presentation and a QVT framework in which research teams can make their unique contributions by complementing rather than competing with the achievements of others.

6 Related Work

Presenting transformations in a graphical style highlights the very close relationship with work on Graph Transformations [4][5]. The UMLX concepts of Preserve, Evolve, Remove correspond directly to Keep, Add and Delete, and so Single or Double Push Out representations of UMLX diagrams are easily derived, where they exist. The Graph Theory work provides a solid foundation upon which proofs of transformation optimisations can be based, and an identification of the conditions that must be satisfied by reversible transformations. It is clear that potentially useful transformations cannot be reversed if they involve many to one mappings or if they destroy cross-linking edges as a graph is transformed into a tree. Characterising UMLX as either SPO or DPO is a matter for further research, since the discipline of evolution identities may avoid some limitations of SPO, but the second class treatment of arcs prohibits categorisation as DPO.

Gerber et al [6] have experimented with a variety of different transformation languages, and while favouring XSLT, they clearly have their reservations as their code became unreadable. Their experiences have influenced their QVT proposal [12], which we feel is not dissimilar to a textual representation of UMLX. Their concept of tracking before/after instances to correlate multiple transformations is subsumed by an evolution identity in UMLX; the latter is a natural consequence of the graphical syntax, whereas the former is a little untidy.

The QVT partners' submission [18] draws an interesting distinction between bi-directional mappings and uni-directional transformations. Their LHS and RHS graphics is similar to UMLX, but without the multiplicities, and they rely on text to define the relationship between LHS and RHS.

The Compuware and Sun joint submission [11] uses graphics to show the context of their transformation, which is then defined textually in a very declarative and reversible style. It is not clear how irreversible transformations can be represented.

The ISIS group at Vanderbilt has pursued the concepts of meta-modelling through the GME tool [7]. A preliminary paper on a Graphical Rewrite Engine [1] inspired the development of UMLX. The evolution to GReAT is described in [2] together with a good discussion on the significance of cardinalities in a UML context. GReAT is similar to UMLX, but lacks a clear distinction between LHS and RHS. Perhaps the main difference is one of emphasis. GReAT is concerned with simple compilation to an efficient transformation, with transformation compilation and implementation directly implemented in C++. UMLX is more concerned with specifying the required transformation, using UMLX to specify both the compilation and the execution of the transformation. UMLX will therefore be very slow until the UMLX specifications for C++ code generation and optimization are in place. Since UMLX is declarative, with a clear LHS/RHS distinction, there should be greater scope for inter-transform composition and optimization of UMLX.

The underlying philosophy of UMLX is identical to ATL [3]. Both seek to provide a concrete syntax for a consistently meta-modelled abstract syntax that should evolve towards QVT. ATL is textual. UMLX is graphical. Once the abstract syntax is standardised, ATL, GReAT and UMLX should just be examples of concrete QVT syntaxes from which users can choose, and between which transformations can translate.

7 Acknowledgement

The author is grateful to anonymous referees, and to Jim Baddoo, Jörn Bettin, Jean Bézevin, Tim Masterton, Richard Metcalfe, Laurent Rioux and members of the Thales MIRROR project for helpful comments on earlier drafts of this article.

8 Summary

We have outlined UMLX, a graphical transformation language that integrates with UML as a mapping between schema. UMLX is a declarative language, and consequently offers scope for powerful optimizations.

We argue that the declarative nature of UMLX enables it to be regarded as a high level language for XSLT from which it derives many important concepts such as referential transparency.

The diagrams in this paper demonstrate the successful configuration of GME as an editor for UMLX, and we have discussed the ongoing parallel development of diagrammatic and manually coded implementations of a compiler for UMLX written in UMLX.

9 References

[1] Aditya Agrawal, Tihamer Levendovszky, Jon Sprinkler, Feng Shi, Gabor Karsai, "Generative Programming via Graph Transformations in the Model-Driven Architecture", OOPSLA 2002 Workshop on Generative Techniques in the context of Model Driven Architecture, November 2002 http://www.softmetaware.com/oopsla2002/karsaig.pdf

[2] Aditya Agrawal, Gabor Karsai, Feng Shi, "A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations", http://www.isis.vanderbilt.edu/publications/archive/Agrawal_A_0_0_2003_A_UML_base.pdf.

[3] Jean Bézevin, Erwan Breton, Grégoire Dupé, Patricx Valduriez, "The ATL Transformation-based Model Management Framework", submitted for publication.

[4] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe, "Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach", In G. Rozenberg, ed., The Handbook of Graph Grammars, Volume 1, Foundations, World Scientific, 1996.

[5] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wganer and A. Corradini, "Algebraic Approaches to Graph Transformation II: Single Pushout Approach and comparison with Double Pushout Approach", In G. Rozenberg, ed., The Handbook of Graph Grammars, Volume 1, Foundations, World Scientific, 1996.

[6] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel and Andrew Wood, "Transformation: The Missing Link of MDA", http://www.dstc.edu.au/Research/Projects/Pegamento/publications/icgt2002.pdf

[7] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle and Peter Volgyesi, The Generic Modeling Environment, http://www.isis.vanderbilt.edu/Projects/gme/GME2000Overview.pdf

[8] Edward Willink, "The UMLX Language Definition", http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/doc/umlx/umlx.pdf.

[9] Edward Willink, "A concrete UML-based graphical transformation syntax - The UML to RDBMS example in UMLX", http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/doc/umlx/M4M03.pdf

[10] Edward Willink, "UMLX Compiler Models", http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/doc/umlx/UmlxCompiler.pdf.

[11] Compuware Corporation and Sun Microsystems, "XMOF Queries, Views and Transformations om Models using MOF, OCL and Patterns", OMG Document ad/2003-08-07, http://www.omg.org/docs/ad/03-08-03.pdf.

[12] DSTC, IBM, "MOF Query/Views/Transformations, Initial Submission", OMG Document ad/2003-08-03, http://www.dstc.edu.au/pegamento/publications/ad-03-08-03.pdf.

[13] OMG, "Model Driven Architecture (MDA)", OMG Document ormsc/01-07-01, http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01.

[14] OMG, "OMG-XML Metadata Interchange (XMI) Specification, v1.2", OMG Document -- formal/02-01-01 , http://www.omg.org/cgi-bin/doc?formal/2002-01-01

[15] OMG, "Meta Object Facility (MOF), 1.4", OMG Document -- formal/02-04-03, http://www.omg.org/cgi-bin/doc?formal/2002-04-03

[16] OMG, "Request For Proposal: MOF 2.0/QVT", OMG Document, ad/2002-04-10.

[17] OMG, "Unified Modeling Language, v1.5", OMG Document -- formal/03-03-01 http://www.omg.org/cgi-bin/doc?formal/03-03-01

[18] QVT Partners, "Revised submission for MOF 2.0 Query/Views/Transformations RFP", OMG Document ad/2003-08-18, http://www.qvtp.org/downloads/1.1/qvtpartners1.1.pdf.

[19] W3C, "Document Object Model (DOM) Technical Reports", http://www.w3.org/DOM/DOMTR.

[20] W3C, "Extensible Markup Language (XML) 1.0 (Second Edition)" W3C Recommendation 6 October 2000, http://www.w3.org/TR/REC-xml.

[21] W3C, "XSL Transformations (XSLT) Version 2.0", W3C Working Draft 2 May 2003, http://www.w3.org/TR/2003/WD-xslt20-20030502