CTIT Technical Report TR–CTIT–03–27, University of Twente

Model Driven Architecture: Foundations and Applications

http://trese.cs.utwente.nl/mdafa2003

Arend Rensink (Editor)

26–27 June 2003, University of Twente

(This page intentionally left blank)

Preface

This report contains the proceedings of the workshop on Model Driven Architecture: Foundations and Application", held at the University of Twente on 26-27 June 2003.

Model Driven Architecture (MDA) proposed by OMG is an approach based on the separation of the specification of system functionality from the specification of the implementation of that functionality on a specific platform. It is aimed at making the software assets more resilient to changes caused by the emerging technologies and makes the role of modeling and models in the current software development much more important. The MDA initiative covers a wide spectrum of research areas some of them are already well established and some are newly emerged. Further efforts are required to bring them into a coherent approach based on open standards and supported by matured tools and techniques. The goal of this workshop is to understand the foundations, to analyze the state of-the-art, to identify problems and solutions, to outline future research directions and to share experience in applying MDA techniques and tools.

Topics of Interest

In the call for papers we asked for full papers and extended abstracts on the following topics:

- Ontologies and domain-specific models
- MDA development process
- Meta modeling and meta models for PIMs and PSMs
- Model transformations: foundations and heuristics
- Model composition
- Aspect-oriented modeling
- Variability management
- Model validation and model checking
- MDA technologies (UML, OCL, XMI)
- Executability of models
- Code generation based on UML profiles
- MDA tools
- Experience reports

We received a total of 25 submissions, from which after a careful reviewing procedure we put together a programme consisting of 7 full papers and 7 short presentations. We have an invited presentation by Wim Bast (of CumpuWare) who is heavily involved in the current Query-View-Transformation (QVT) Request for Proposals of the OMG. The programme also two discussion sessions. Since several of the submitted papers also concerned the QVT RfP, we hope for an interesting exchange of opinions on this subject.

Invited Speaker

- Wim Bast (CompuWare)

Programme Committee

- Mehmet Aksit, University of Twente, the Netherlands
  (editor of the journal special issue on MDA)
- Uwe Assmann, University of Linkøping, Sweden
- Wim Bast, Compuware, the Netherlands
- Klaas van den Berg, University of Twente, the Netherlands
- Jean Bézivin, University of Nantes, France
- Jan Bosch, University of Groningen, the Netherlands
- Paul Clements, Software Engineering Institute, USA
- Krzysztof Czarnecki, University of Waterloo, Canada
- Gregor Engels, University of Paderborn, Germany
- Andy Evans, University of York, U.K.
- Jean-Marc Jźequel, IRISA, France
- Anneke Kleppe, Klasse Objecten, the Netherlands
- Paul Klint, CWI, the Netherlands
- Tom Mens, Vrije Universiteit Brussel, Belgium
- Arend Rensink, University of Twente, The Netherlands (PC chair)
- Bedir Tekinerdogan, Bilkent University, Turkey

Local organisation

- Klaas van den Berg
- Ivan Kurtev, University of Twente, The Netherlands (local chair)

Contents

A model driven approach to model transformations

Biju Appukuttan[1] biju@dcs.kcl.ac.uk
Tony Clark[2] anclark@dcs.kcl.ac.uk
Sreedhar Reddy[3] sreedharr@pune.tcs.co.in
Laurence Tratt[2] laurie@tratt.net
R. Venkatesh[3] rvenky@pune.tcs.co.in

[1] *Tata Consultancy Services, Pune, India. On deputation to Kings College London.*

[2] *Department of Computer Science, King's College London, Strand, London, WC2R 2LS, United Kingdom.*

[3] *Tata Consultancy Services, Pune, India.*

Abstract

The OMG's Model Driven Architecture (MDA) initiative has been the focus of much attention in both academia and industry, due to its promise of more rapid and consistent software development through the increased use of models. In order for MDA to reach its full potential, the ability to manipulate and transform models – most obviously from the Platform Independent Model (PIM) to the Platform Specific Models (PSM) – is vital. Recognizing this need, the OMG issued a Request For Proposals (RFP) largely concerned with finding a suitable mechanism for transforming models. This paper outlines the relevant background material, summarizes the approach taken by the QVT-Partners (to whom the authors belong), presents a non-trivial example using the QVT-Partners approach, and finally sketches out what the future holds for model transformations.

1 Introduction - Transformations and MDA

The OMG Queries/Views/Transformations (QVT) RFP [1] defines the MDA vision thus:

> MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform, and provides a set of guidelines for structuring specifications expressed as models.
>
> The MDA approach and the standards that support it allow the same model specifying system functionality to be realized on multiple platforms through auxiliary mapping

standards... and allows different applications to be integrated by explicitly relating their models.

In less technical terms, MDA aims to allow developers to create systems entirely with models [1] . Furthermore, MDA envisages systems being comprised of many small, manageable models rather than one gigantic monolithic model. Finally, MDA allows systems to be designed independently of the eventual technologies they will be deployed on; a PIM can then be transformed into a PSM in order to run on a specific platform.

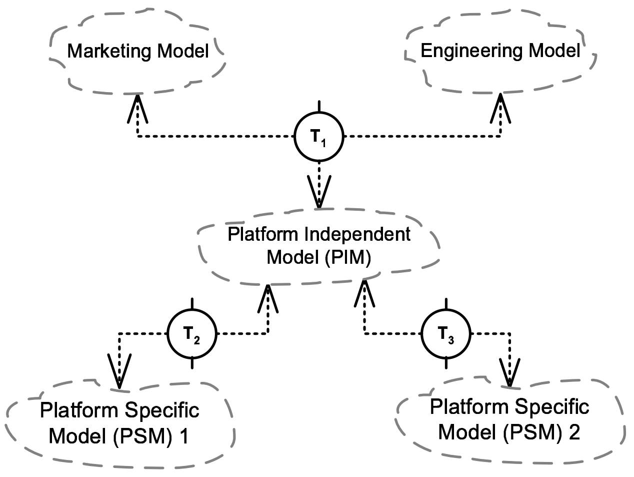

Marketing Model
Engineering Model
T1
Platform Independent Model (PIM)
T2
T3
Platform Specific Model (PSM) 1
Platform Specific Model (PSM) 2

Fig. 1. Transformations and MDA

Figure 1 – based partly on a D'Souza example [2] – shows an overview of a typical usage of MDA. It shows a company horizontally split into multiple departments, each of which has a model of its system. These models can be considered to be views on an overall system PIM. The PIM can be converted into a PSM. In order to realize this vision, there has to be some way to specify the changes that models such as that in figure 1 undergo. The enabling technology is *transformations*. In figure 1 a transformation $\mathtt{T}_1$ integrates the company's horizontal definitions into an overall PIM, and a transformation $\mathtt{T}_2$ converts the overall PIM into PSMs, one for each deployment platform.

The following are some representative MDA related uses where transformations are, or could be, involved:

- Converting a model 'left to right' and/or 'right to left'. This is a very common operation in tools, for example saving a UML model to XML and reading it back in again.
- Abstracting a model. Abstracting away unimportant details, and presenting to the user only the salient points of the model, is a vital part of MDA.
- Reverse engineering. For example, a tool which recovers Java source code from class files.
- Technology migration. This is similar to reverse engineering, but whereas reverse engineering is simply trying to recover lost information, technology migration is effectively trying to convert outdated systems into current systems. For example, a tool which migrates legacy COBOL code to Java.

Transformations are undoubtedly the key technology in the realization of the MDA vision. They are present explicitly – as in the transformation of a PIM to a PSM – and implicitly – the integration of different system views – throughout MDA.

[1] This does not mean that *everything* must be specified fully or even semi-graphically – the definition of model allows one to drill down right to source code level.

2 QVT

In order for MDA to reach its full potential, the ability to manipulate and transform models is vital. Although there has been much discussion [3,4] of the problem area, as well as attempts at filling this gap in the past [5–8], little practical progress has been made. Recognizing the need for a practical solution for transformations, the OMG issued a Request For Proposals (RFP) [1] largely concerned with finding a suitable mechanism for transforming models. This paper is based on the QVT-Partners [2] initial submission [9] to the QVT RFP.

3 Fundamental concepts

It is our view that to provide a complete solution to the problem of a practical definition of transformations, the following complimentary parts are necessary:

(1) The ability to express both specifications and implementations of transformations.
(2) A mechanism for composing transformations.
(3) Standard pattern matching languages which can be used with declarative and imperative transformations.
(4) A complete semantics, which are defined in terms of existing OMG standards.

The solution outlined in this paper can be seen to be chiefly concerned with solving two overarching problems: the need to provide a framework into which different uses of transformations can be accommodated, and the need to provide a standard set of languages for expressing transformations. In solving these needs, the solutions to other fundamental requirements as mentioned earlier in this section follow fairly automatically.

4 A definition of transformations

This section outlines the points of our definition of transformations that are most relevant to this paper. See also section 7.

4.1 Framework

We define an overall framework for transformations that allows one to use a variety of different transformation styles. This framework also transparently allows transformations to change style throughout the lifetime of a system. Such transparency is enabled by identification of two distinct sub-types of transformations: relations and mappings.

Relations are multi-directional transformation specifications i.e. they are declarative. In the general case they are non-executable (in the sense of actually transforming a model), but we have identified useful restricted types of bidirectional relations, which can be automatically refined into mappings. Relations are written in any valid UML constraint language, OCL being an obvious example. Typically relations are used in the specification stages of system development.

Mappings are transformation implementations i.e. they are operational. Unlike relations, mappings are potentially uni-directional. Mappings are expressed in the UML Action Semantics (AS) and thus encompass all programming language implementations. Mappings

[2] `http://qvtp.org/`

can refine any number of relations, in which case the mapping must be consistent with the relations it refines.

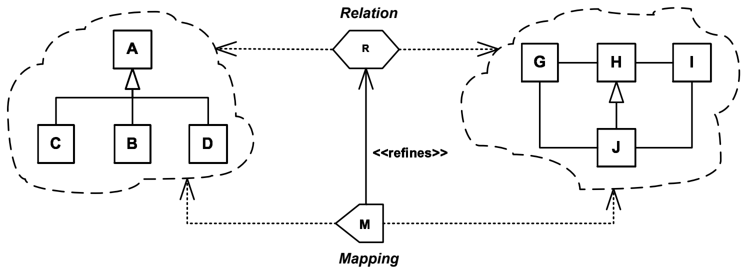


Fig. 2. A high level relation being refined by a directed mapping

Figure 2 shows a relation `R` relating two domains. There is also a mapping `M` which refines relation `R`; since `M` is directed, it transforms model elements from the right hand domain into the left hand domain.

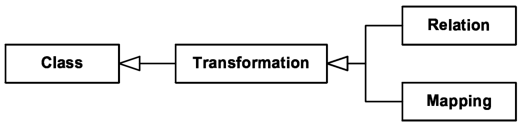


Fig. 3. Transformations, relations and mappings in the MOF hierarchy

Figure 3 shows how transformations, relations and mappings are placed within the MOF [10] hierarchy. As `Transformation` is a super-type of `Relation` and `Mapping`, when we talk about a transformation we effectively mean either a relation or a mapping, we don't mind which one. When we talk about a mapping, we specifically mean a mapping and only a mapping and similarly for relations. The differentiation between specification and implementation is vital. In many complex applications of transformation technology it is often unfeasible to express a transformation in operational terms. For example, during the initial stages of system development, various choices, which will affect an implementation, may not have been made, and thus it may be undesirable to write an implementation at that stage. Another more general reason for the presence of specifications is that transformation implementations often carry around large amounts of baggage, which whilst vital to the transformations execution, obscure the important aspects of a transformation – by using specifications, these important aspects can be easily highlighted. Nevertheless, implementations are vital for the final delivered system. We also propose a standard operational transformation language to prevent the need to drop to low level technologies such as the XML transformation system XSLT (XSL Transformations) [11] – in order for transformations to be a successful and integral part of MDA, it is essential that they be modelled. Our proposal allows transformations to seamlessly and transparently evolve from specifications to implementations at any point during the development life cycle.

4.2 Pattern Languages

Pattern languages are widely used in real world transformation technologies such as Perlesque textual regular expressions and XSL (note that the former is a declarative transformational language, whereas the latter is imperative). Clearly, any solution needs to have pattern languages, as they are a very natural way of expressing many – though not all – transformations. Our solution provides standard pattern matching languages for both relations and mappings; a pattern replacement language is also defined for relations, allowing many specifications utilizing the pattern language to be executable. Furthermore, we also provide graphical syntax to express patterns, as well as the more conventional textual representation.

5 Transformations

Our definition of transformations comes in two distinct layers. Reusing terminology familiar from the UML2 process, we name these layers *infrastructure* and *superstructure*.

5.1 Infrastructure

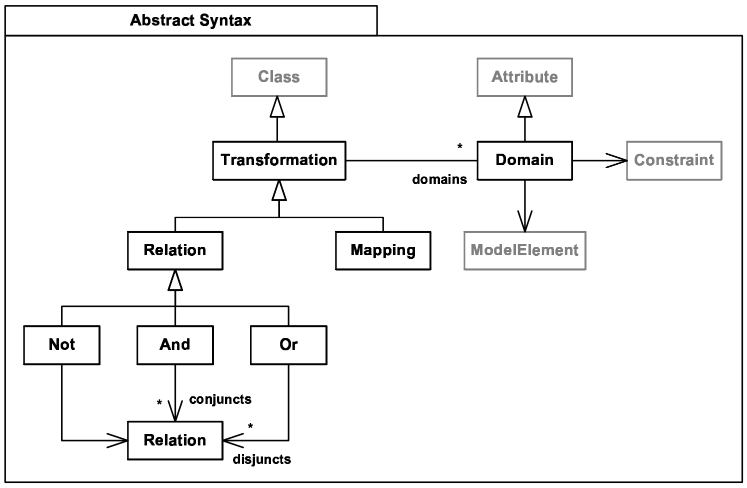


Fig. 4. Infrastructure meta model

Figure 4 shows the infrastructure abstract syntax package. This package can be merged with the standard MOF definition to produce an extended version of MOF. Original MOF elements are shown in grey; our new elements are in black. The infrastructure contains what we consider to be a sensible minimum of machinery necessary to support all types of transformations. The infrastructure is necessarily low-level and not of particular importance to end users of transformations. Its use is a simple semantic core [12].

5.2 Superstructure

Compared to the infrastructure, the superstructure contains a much higher-level set of transformation types and is suitable for end users. Figure 5 shows a transformation meta-model that extends the transformations meta-model given in Infrastructure. The elements Transformation, Relation, Domain, And, Or and Mapping inherit from and extend the corresponding elements in the infrastructure. Elements from MOF core are shown in gray.

The heart of the model is the element Relation. It specifies a relationship that holds between instance models of two or more Domains. Each Domain is a view of the meta-model, and is constituted of Class and association roles. A Role has a corresponding type that the elements bound to it must satisfy. A Domain may also have an associated query to further constrain the model specified by it. The query may be specified as an OCL expression. A Relation also may have an associated OCL specification. This may be used to specify the relationship that holds between the different attribute values of the participating domains. A binary directed-relation is a special case with a source Domain and a target Domain.

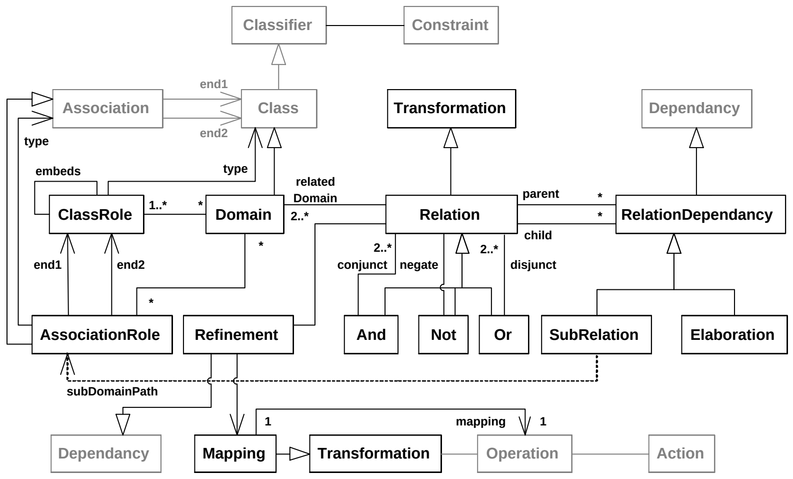


Fig. 5. Superstructure meta model

5.3 Concrete syntax

Our solution defines a graphical concrete syntax for transformations. Figure 6 lists the most important notations.

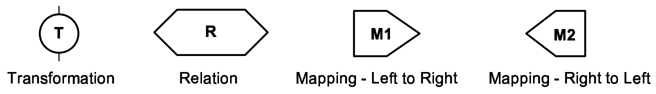


Fig. 6. Concrete Syntax for transformations

6 An example

In order to illustrate the salient features of our approach, in this section we present an example between simplified UML models and XML.

6.1 The example model

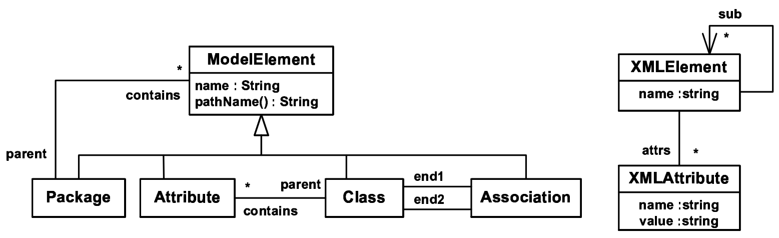


Fig. 7. The example meta-model

Figure 7 shows a simplified model of UML class diagrams. Each `ModelElement` has a name; the `pathName` operation returns a string representing the element's full pathname. The operation is defined thus:

```
context ModelElement::pathName(): String
  if not self.parent then
    self.name
  else
    self.parent.pathName() + "." + self.name
  endif
```

We assume that all elements have a unique pathname. This can be trivially enforced by placing constraints on `Package` and `Class` to ensure that none of their contents share the same name.

Figure 7 (right hand side) shows a simplified model of XML. We prefix both elements in the model by `XML` to avoid having to qualify references via a package name. The model captures the notion of XML elements having a number of attributes, and containing XML elements.

In the rest of this section, we gradually build up a relation from our UML model to XML, from a number of small pieces.

6.2 Building up the transformation

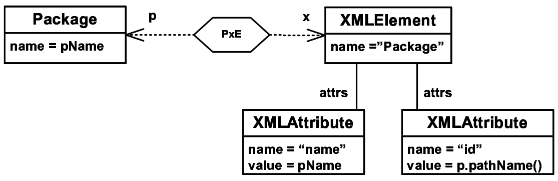


Fig. 8. A UML package to XML relation

Figure 8 shows a relation between the UML `Package` and XML using a pattern language. Although at first glance figure 8 may look like a standard UML class diagram, it should rather be thought of as something in between a class diagram and an object diagram. Notice how some attributes in the transformation have constant values given to them, whilst others have variables – each variable name must have the same value across the diagram.

Thus to examine figure 8 in detail, each `Package` instance is related to an `XMLElement` with the name `Package`. The XML element has two `XMLAttribute`. The first is the name of the package which has a value of `pName`, thus forcing it to be the same literal name as the UML package. To allow us to reference elements (which is necessary for association ends), we also force each XML element to have a unique identifier – the properties of the `pathName` operation mean we can use it to produce unique identifiers. An alternative and more general, though more complex, approach to using `pathName` would be to have a stateful transformation (see section 7.1) which has a counter which is assigned to each element, and maintains a dictionary relating each model element to its unique ID assigned via the counter.

When written in more conventional form, the UML package would be related to the following chunk of XML:

`<Package name=`*pName* `id=`*p.pathName()*`></Package>`

The relations `CxE` and `AxE` for `Class`es and `Attribute`s respectively are much the same as for `PxE` for `Package`.

Figure 9 shows the relation `ASxE` for `Association`. This is more involved than the previous relations as an association is comprised of two association ends which also need to be related to XML. Note that it is not the model elements the association ends reference that

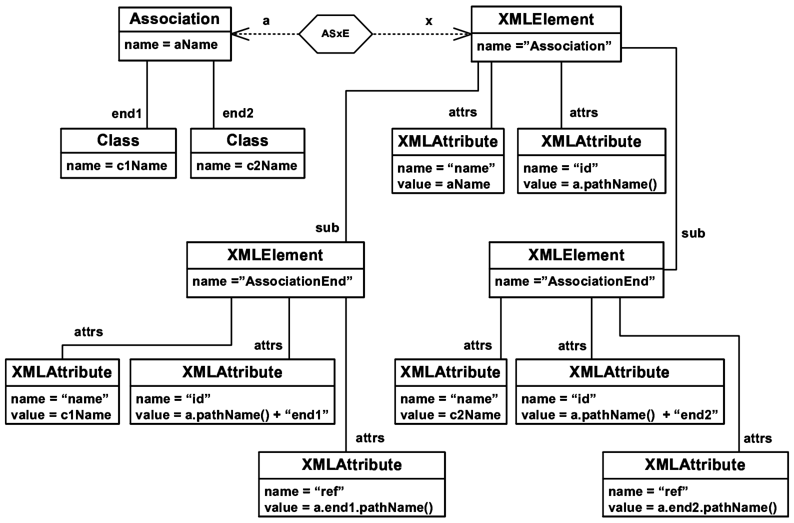


Fig. 9. Transformation of Association

are related, but rather the references themselves. This is where the unique `id` we have forced onto XML elements comes into play. The UML association is thus related to the following chunk of XML:

```
<Association name=aName id=asc.pathName()>
  <AssociationEnd name=c1Name id=asc.pathName()+"end1"
   ref=asc.end1.pathName() />
  <AssociationEnd name=c1Name id=asc.pathName()+"end2"
   ref=asc.end2.pathName() />
</Association>
```

6.3 Putting it all together

In this section, we slot the individual relations in the previous sub-section together to form one overall transformation. This creates several new issues that are not present when the relations exist in isolation.

In general, additional constraints will be needed to ensure a relation is completely modelled. For example, a common issue is the need to ensure that all of the contents of an element (e.g. a UML package) are related to a corresponding element (e.g. an XML element). Figure 10 shows how the individual relations in the previous section slot together. Note the inheritance relationships in this figure. The transformation of the abstract ModelElement is captured by the abstract transformation `MxE`. The information inherited from the abstract ModelElement play a key role in the transformation of the individual elements. Similarly, the individual transformations are derived from the abstract transformation `MxE` defined on the ModelElement.

In order to ensure that all of the contents of an element `Package` are related to a corresponding `XMLElement` the following 'round trip' constraint is needed:

```
context PxE:
  self.p.contains->size() = self.sub->size() and
  self.p.contains->forAll(m |
    self.sub->exists(cxe |
```

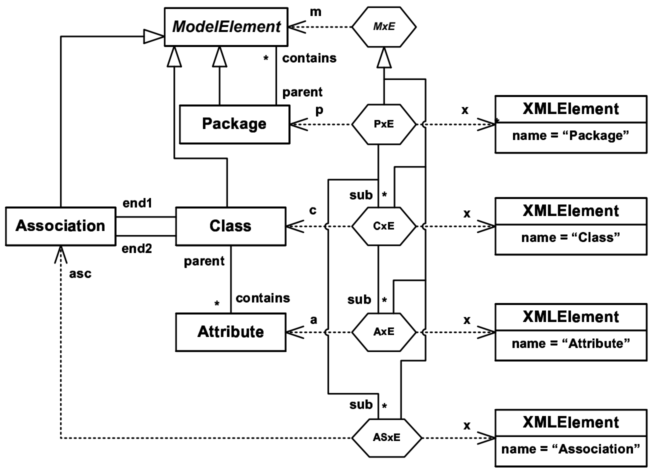


Fig. 10. Transformation composition

```
    cxe.m = m))
```

There are various ways that this constraint can be phrased to achieve the same end result. This particular method makes use of the fact that if the number of contents in `p.contains` is the same as `sub` and every element in `p.contains` has a transformation which is also a member of `sub` then the round trip is enforced. At the moment the user needs to explicitly enforce this constraint via OCL; we anticipate in the future adding a way to allow the user to specify that the round trip needs to be enforced, without forcing them to write out the entire constraint. The relevant constraint could be trivially generated from a boiler-plate – at the time of writing, unfortunately no OCL equivalent to macros or template programming such as found in [13] exists. We expect this shortcoming to be remedied in the relatively near future.

We now use the example object model in figure 11 to illustrate a complete transformation. This model consists of a package `pkg1` which contains two classes `cls1` and `cls2` and an association `assoc1` between these two classes. Furthermore, `cls1` contains an attribute `attr1`.

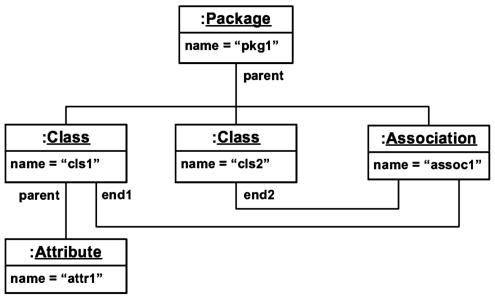


Fig. 11. Object model example to illustrate transformations

Figure 12 shows the complete relations, which combines several of the preceding relations, such as figure 8 and 10, and a few other similar relations which we do not have space for.

The end result of this transformation is the following XML output:

```
<Package name="pkg1" id="pkg1">
  <Class name="cls1" id="pkg1.cls1">
    <Attribute name="attr1" id="pkg1.cls1.attr1" />
  </Class>
  <Class name="cls2" id="pkg1.cls2">
  </Class>
  <Association name="assoc1" id="pkg1.assoc1">
    <AssociationEnd name="cls1" id="pkg1.assoc1.end1" ref="pkg1.cls1" />
    <AssociationEnd name="cls2" id="pkg1.assoc1.end2" ref="pkg1.cls2" />
  </Association>
</Package>
```

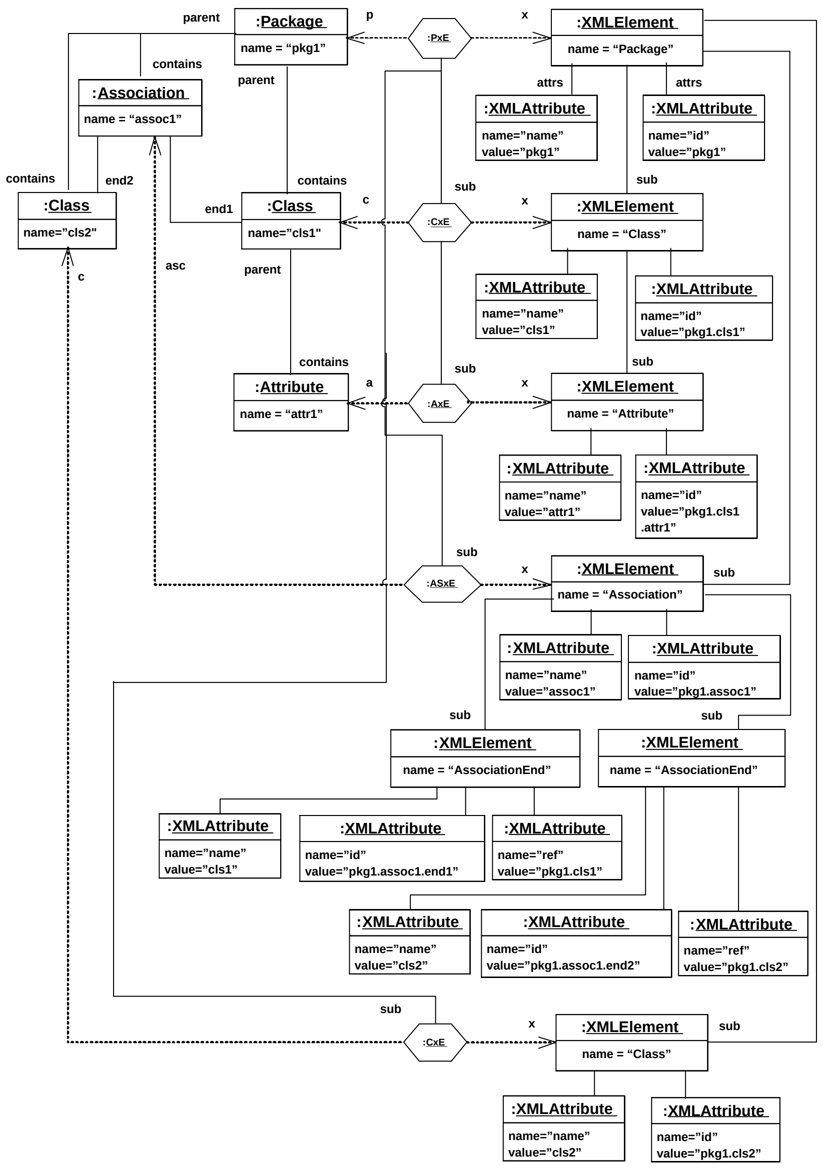


Fig. 12. Complete transformation of the example in figure 11

6.4 Mapping

The example defined thus far is a relation – thus, being declarative, it is not necessarily executable. In our definition mappings, which are operational and potentially directed, transformations can be created which refine relations. Although we do not have sufficient space to write out a complete mapping which refines the relation we have created up until this point, we hope it is fairly trivial to imagine pseudo-code along the following lines which would take in UML and export XML:

```
function uml_to_xml(model:ModelElement):
  if type(model) == Package:
    xml = XMLElement("Package", id=model.pathName())
    for e in model.contains:
      xml.append(uml_to_xml(e))
  ...
```

Of course, this operational definition can be given in any programming language e.g. Java, Python or C++.

7 Other features

In this section we outline some other useful features of our definition of transformations.

7.1 Stateful transformations

In many situations, simple transformations, which perform a one step transformation, are not sufficient. Transformations may need to build up large amounts of information whilst in the process of transforming, particularly, if other transformations are involved in the process and may also need to store information over transformations. A simple example of such a transformation is one, which adds to elements a unique identifier based on an incremented counter. Although one could create a new object in the system to track the counter, it is far more natural and less cumbersome for the transformation itself to maintain the counter. To this end, in our proposal all transformations have state by virtue of the fact that Transformation subclasses Class as shown in figure 3.

7.2 Transformation Reuse

In order for transformations to scale up, it is essential to encompass features for reusing existing transformations and composing further transformations from existing ones. Our proposal caters to this requirement in two different ways – transformations can be reused either through the specialization mechanism or by using a more powerful composition mechanism. A composite transformation is formed of a parent transformation and a number of component transformations which are linked to the parent via logical connectives such as and, etc. The example described in this paper reuses transformations by specializing the `MxE` transformation defined on the ModelElement (figure 10).

8 Conclusions

We originally motivated the need for a practical definition of transformations to allow models to be manipulated; this need is enshrined in the OMG QVT RFP. We then outlined our approach to transformations, and presented a non-trivial example. To summarize, our solution provides: the ability to express transformations as both relations and mappings;

standard pattern languages for both relations and mappings; stateful transformations; powerful mechanisms for reusing transformations and for composing transformations; a succinct definition in two parts utilizing an infrastructure – the simple semantic core, and a superstructure – where the rich end-user constructs exist.

The future for model transformations is hard to precisely predict since it is undoubtedly the case that we are still in the early stages of model transformation technology. We expect approaches such as the one we outline in this paper to be further enhanced and, as real world experience in the area develops, to evolve in different directions. We also expect that in the future specific transformation language variants will be created to handle particular problem domains; nevertheless we feel that most of the fundamental concepts, as outlined in this paper, will hold true no matter the type of transformation involved.

This research was funded by a grant from Tata Consultancy Services.

References

[1] Object Management Group, Request for Proposal: MOF 2.0 Query / Views / Transformations RFP, `ad/2002-04-10` (2002).

[2] D. DSouza, Model-driven architecture and integration - opportunities and challenges, `http://www.kinetium.com/catalysis-org/publications/papers/2001-mda-reqs-desmond-6.pdf` (2001).

[3] J. Bézivin, From object composition to model transformation with the MDA, in: TOOLS 2001, 2001.

[4] M. A. de Miguel, D. Exertier, S. Salicki, Specification of model transformations based on meta templates, in: J. Bezivin, R. France (Eds.), Workshop in Software Model Engineering, 2002.

[5] K. Lano, J. Bicarregui, Semantics and transformations for UML models, in: J. Bézivin, P.-A. Muller (Eds.), The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, 1998, pp. 97–106.

[6] K. Lano, J. Bicarregui, UML refinement and abstraction transformations, in: Second Workshop on Rigorous Object Orientated Methods: ROOM 2, Bradford, May, 1998., 1998.

[7] W. M. Ho, J.-M. Jézéquel, A. L. Guennec, F. Pennaneac'h, UMLAUT: An extendible UML transformation framework (1999).

[8] T. Levendovszky, G. Karsai, M. Maroti, A. Ledeczi, H. Charaf, Model reuse with metamodel-based transformations, in: C. Gacek (Ed.), ICSR, Vol. 2319 of Lecture Notes in Computer Science, Springer, 2002.

[9] QVT-Partners initial submission to qvt-rfp, `ad/03-03-27` (2003).

[10] Object Management Group, Meta Object Facility (MOF) Specification, `formal/00-04-03` (2000).

[11] W3C, XSL Transformations (XSLT), `http://www.w3.org/TR/xslt` (1999).

[12] M. Gogolla, Graph transformations on the UML metamodel, in: J. D. P. Rolim, A. Z. Broder, A. Corradini, R. Gorrieri, R. Heckel, J. Hromkovic, U. Vaccaro, J. B. Wells (Eds.), ICALP Workshop on Graph Transformations and Visual Modeling Techniques, Carleton Scientific, Waterloo, Ontario, Canada, 2000, pp. 359–371.

[13] T. Sheard, S. P. Jones, Template meta-programming for Haskell, in: Proceedings of the Haskell workshop 2002, ACM, 2002.

UMLX : A graphical transformation language for MDA

Edward D. Willink, EdWillink@iee.org
Thales Research and Technology Limited,
Reading, England
19 June 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 [11] for a Query / Views / Transformations (QVT) language to exploit the Meta-Object Facility (MOF) [10], which as from version 2.0 should share common core concepts with the Unified Modeling Language (UML) 2.0 [12]. The initial submissions of 8 consortia[1] have been made, and somewhat surprisingly, only one of them [13] uses a graphical representation of their language.

This paper describes independent work to provide an Open Source tool to support the OMG's Model Driven Architecture (MDA) initiative [8]. 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 [7] [13], 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 [15] is steadily eroding these barriers, and the ease with which XML can be read and written using the Java DOM [14] 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 [16] 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 [9]. 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.

In this section we just introduce the UMLX extensions to UML. The very important area of compiler transformations is deferred until Section 3.

[1] The author is not directly associated with the consortium of which his employer is a part.

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.

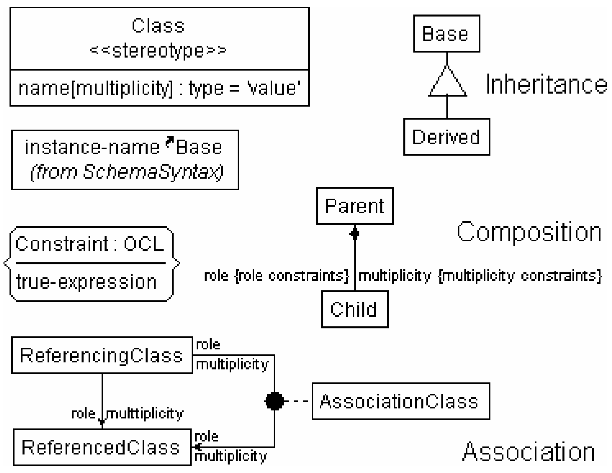


Figure 2.1 Schema Syntax

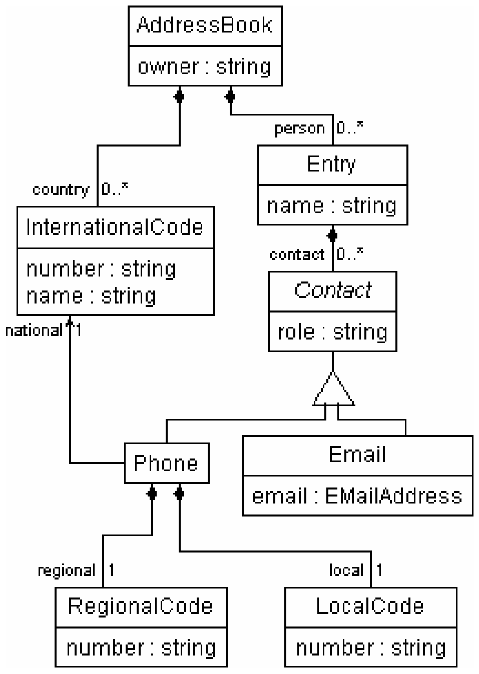


Figure 2.2 Example Schema

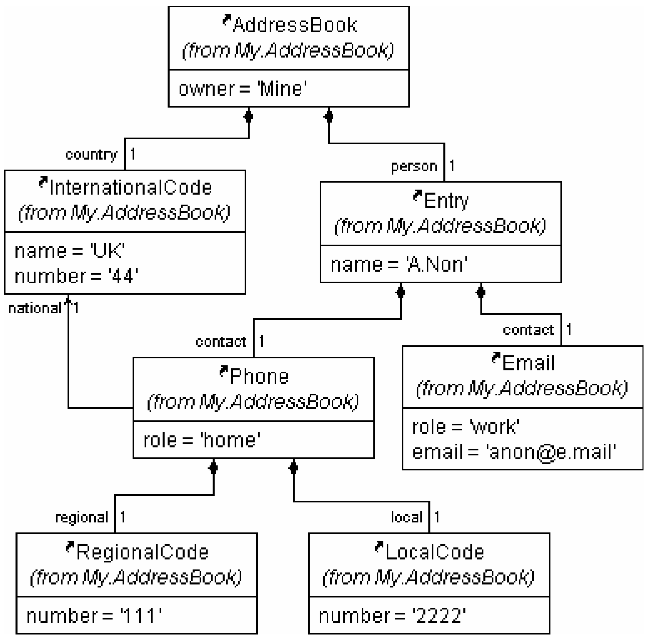


Figure 2.3 Example Instance

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.

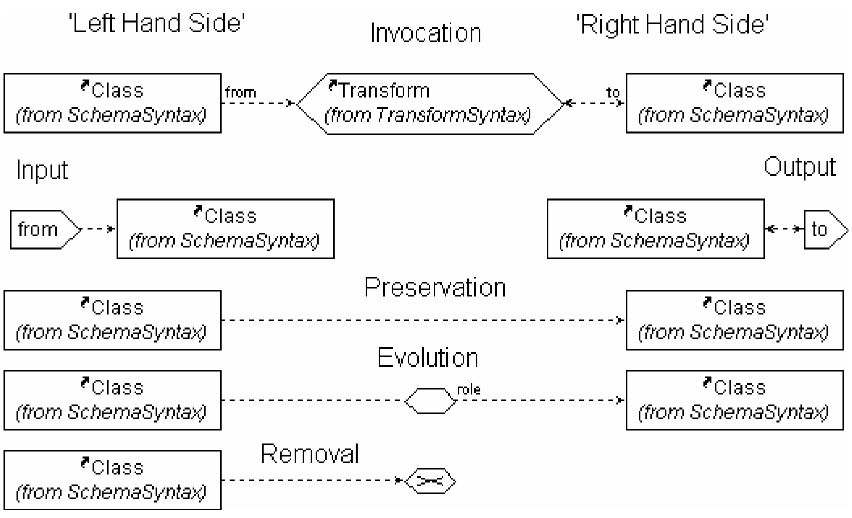

'Left Hand Side'
Invocation
'Right Hand Side'
Class (from SchemaSyntax)
from
Transform (from TransformSyntax)
to
Class (from SchemaSyntax)
Input
Output
from
Class (from SchemaSyntax)
Class (from SchemaSyntax)
to
Preservation
Class (from SchemaSyntax)
Class (from SchemaSyntax)
Evolution
Class (from SchemaSyntax)
role
Class (from SchemaSyntax)
Removal
Class (from SchemaSyntax)

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 [6].

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.

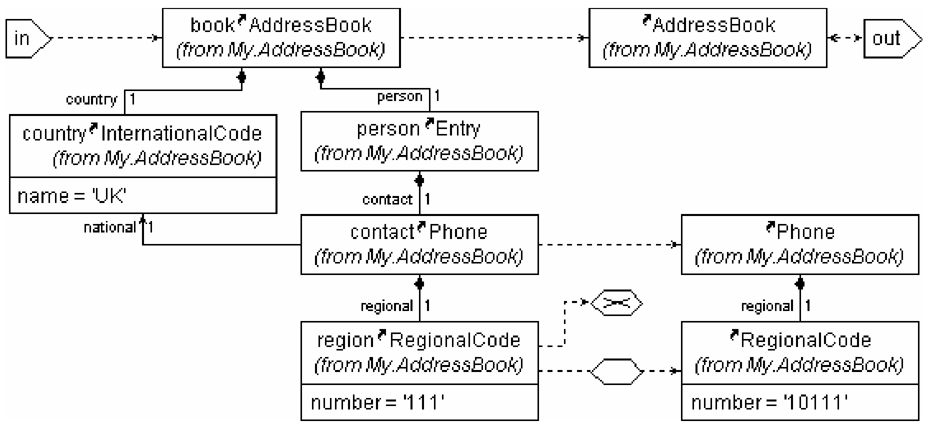

in
book AddressBook (from My.AddressBook)
AddressBook (from My.AddressBook)
out
country 1
person 1
country InternationalCode (from My.AddressBook)
name = 'UK'
person Entry (from My.AddressBook)
contact 1
national 1
contact Phone (from My.AddressBook)
Phone (from My.AddressBook)
regional 1
regional 1
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 result is shared and so may be updated by concurrent 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 `AddressBooks` provides an implicit preservation of its composed contents. Implicit preservation or removal recurses until an explicit transformation 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 transformation 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 [6]. When multiplicities are applied hierarchically, predicates and sub-matches are supported.

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 [6].

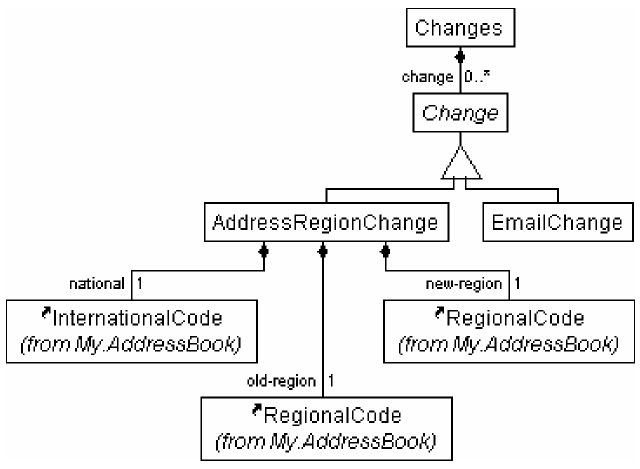
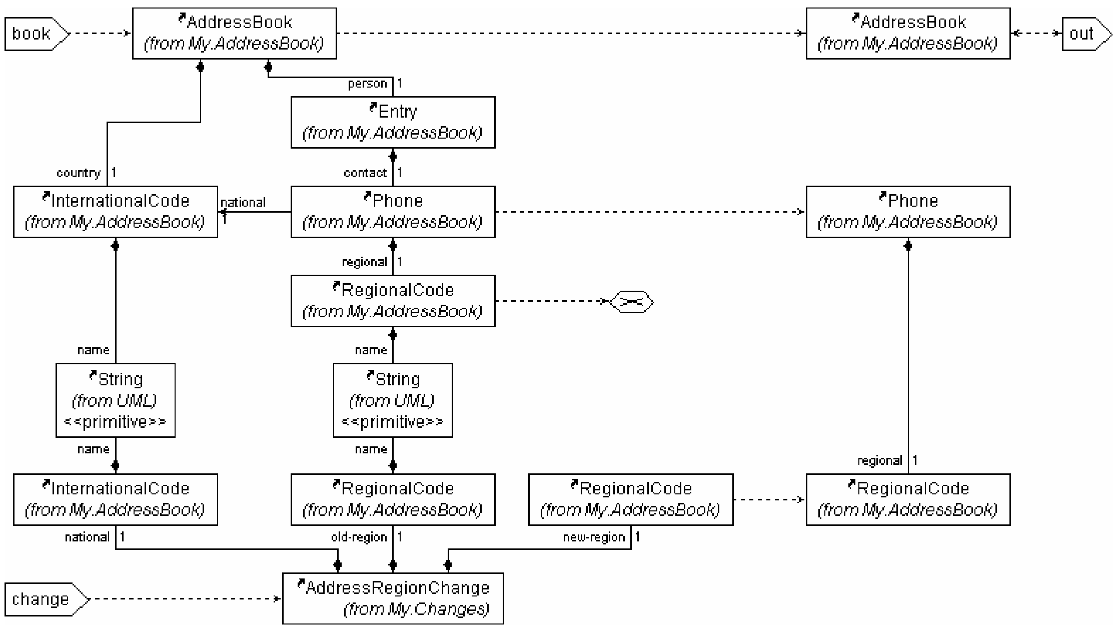

Changes
change 0..*
Change
AddressRegionChange
EmailChange
national 1
new-region 1
InternationalCode (from My.AddressBook)
RegionalCode (from My.AddressBook)
old-region 1
RegionalCode (from My.AddressBook)

Figure 2.6 Contact Changes Schema

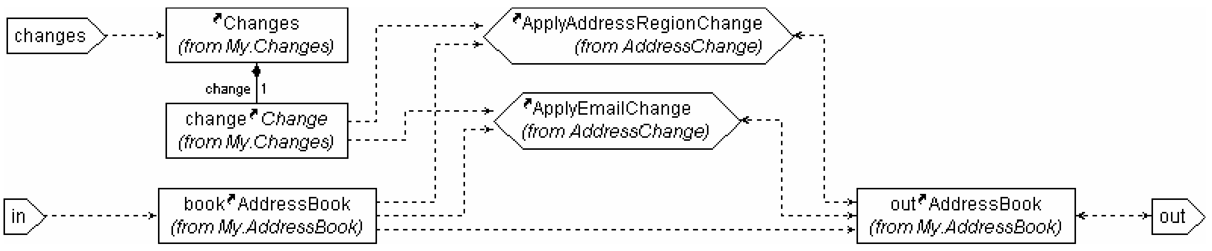
The example application can be made a little more useful by defining the additional schema for changes shown in Figure 2.6. We may now seek to apply a batch of `AddressRegionChange`s and/or `EmailChange`s. An individual `AddressRegionChange` may be realized by the slightly changed transformation, shown 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`.



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 1
Phone (from My.AddressBook)
Phone (from My.AddressBook)
regional 1
RegionalCode (from My.AddressBook)
name
name
String (from UML) <<primitive>>
String (from UML) <<primitive>>
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

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.



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

Figure 2.8 Compound Transformation

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 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 an address book following an upgrade to a tool that uses a new standard AddressBookML.

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 paper, 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.

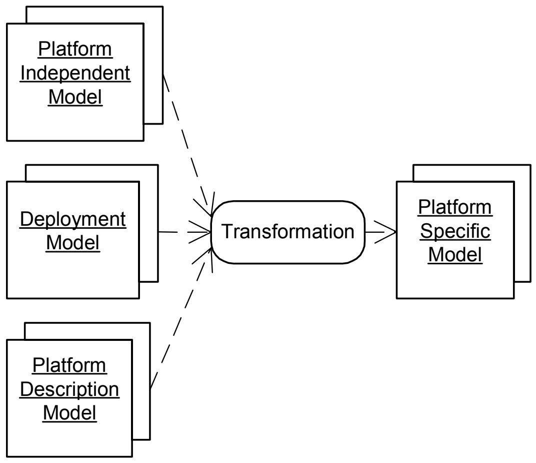


Figure 3.1 MDA Models

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

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.

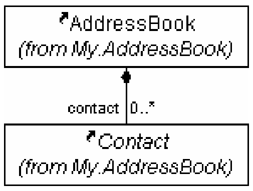


Figure 3.2 UML Composition

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]

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.

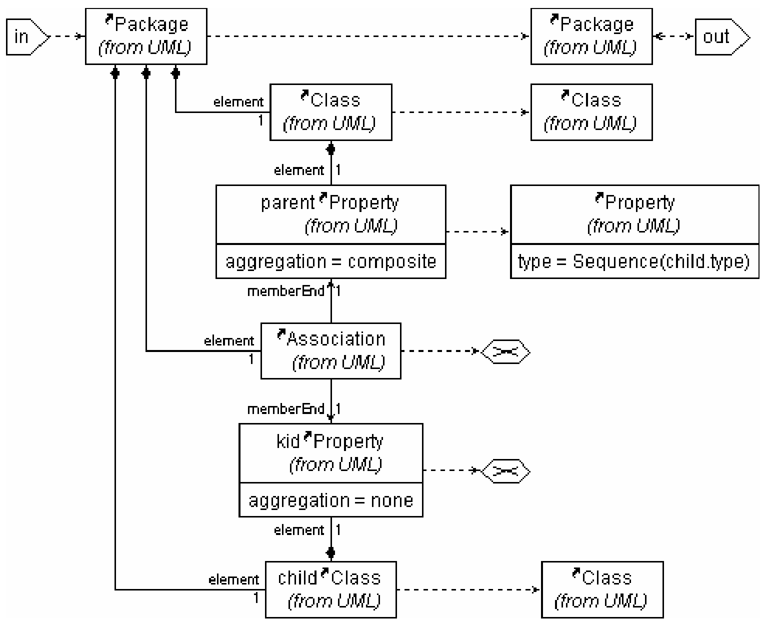


Figure 3.3 Composition Transformation

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.

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.

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

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.

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 [5] 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.

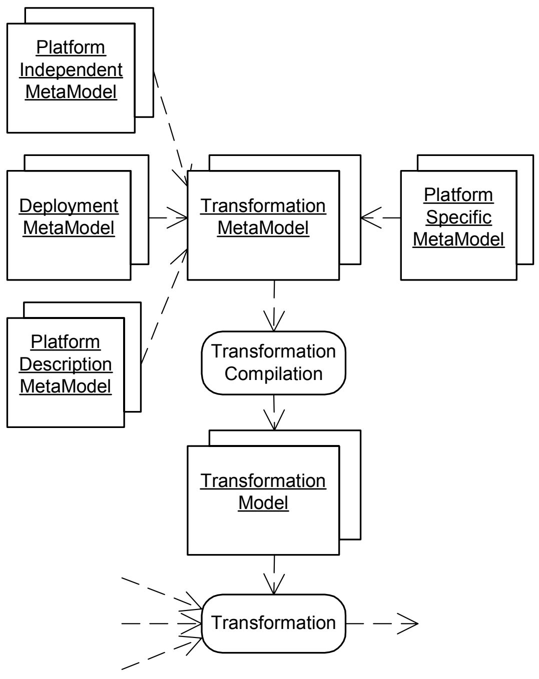

Platform Independent MetaModel
Deployment MetaModel
Platform Description MetaModel
Transformation MetaModel
Platform Specific MetaModel
Transformation Compilation
Transformation Model
Transformation

Figure 4.1 MDA 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 nearly 100 diagrams, so it is clearly inappropriate to present them all here. We will therefore just present the top-levels of the transformation engine and compiler to demonstrate the need to support multiple inputs, multiple schema, workflow and hierarchical transformations, and one detailed transformation to demonstrate the power of the graphical representation.

Although the top-level transformations of the transform engine and the UMLX compiler just define workflow, they are very different from UML activity diagrams, since each hierarchical transformation in the workflow may represent multiple concurrent data-dependent matches.

[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 Engine

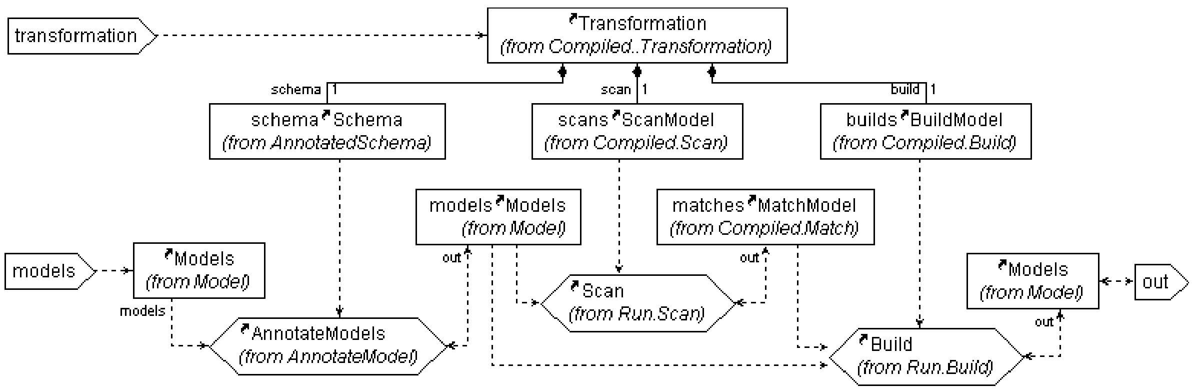


Figure 4.2 Transformation Execution

Execution of an individual transformation, shown in Figure 4.2, involves three activities; `AnnotateModels`, `Scan` and `Build`, and two intermediate pivot models. Each activity is modelled in UMLX as a transformation that makes use of information contained within the `Transformation` model. The `AnnotateModels` activity augments each of the incoming models with instantiation information from their corresponding `Schema`. The `Scan` activity then searches for matches to the structures encoded in the `ScanModel`, and creates a `MatchModel` containing the results. Finally the `Build` activity reacts to the build instructions contained in the `BuildModel`, the matches discovered in the `MatchModel` and the annotated input `Models` to create the output `Models`.

At the compiler-compiler level, the `transformation` and `models` inputs derive from two (meta-)meta-models. At the compiler level, each of the input `models` and each of the `out` models may derive from distinct schema.

4.2 Transformation Compiler

The `Transformation` model and its constituent sub-models is created by the transformation compiler, from the `schema` defining the input and output meta-models and from the meta-models of the `transforms`. Annotation is applied first to the schema and then to the transforms in order to provide the information needed by the `GenerateScan` and `CreateBuild` activities that create the specialised models for the execution engine.

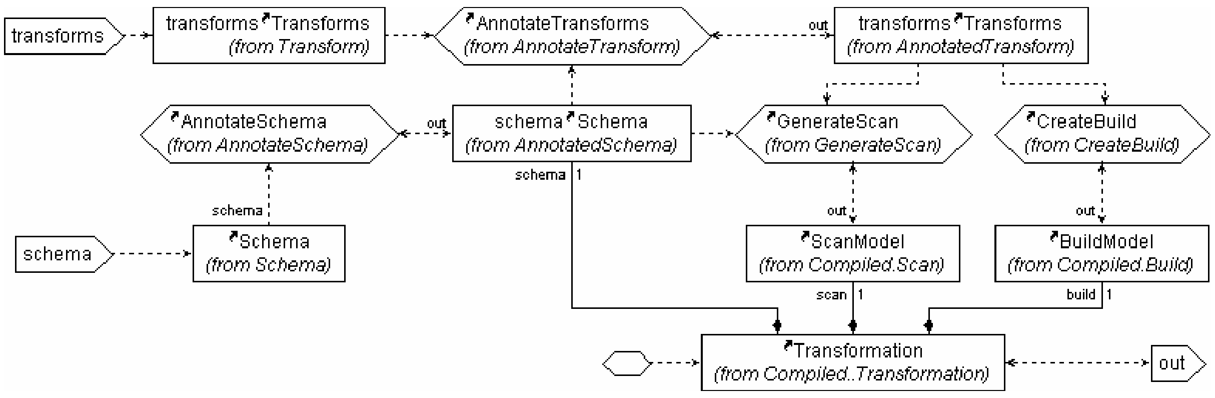


Figure 4.3 Transformation Compilation

4.3 Arc Annotation

An important annotation identifies which composition arc in the schema is instantiated by each composition arc in a transform. This is necessary because although the configuration of GME to support editing of UMLX diagrams ensures that class instances are associated with defining classes, it does not validate the correspondence of relationships.[4]

Figure 4.4 shows a transformation that provides this annotation. It may be read as: for-each `CompositionInstance` in the `Transform`, find the `Composition` in the `Schema` for which the `Classes` at its `Ends` are instantiated by the `ClassInstances` at the same-named `Ends` of the `CompositionInstance`. Wherever a match occurs, the `CompositionInstance` in the output model is updated with a reference to the `Composition` in the schema.

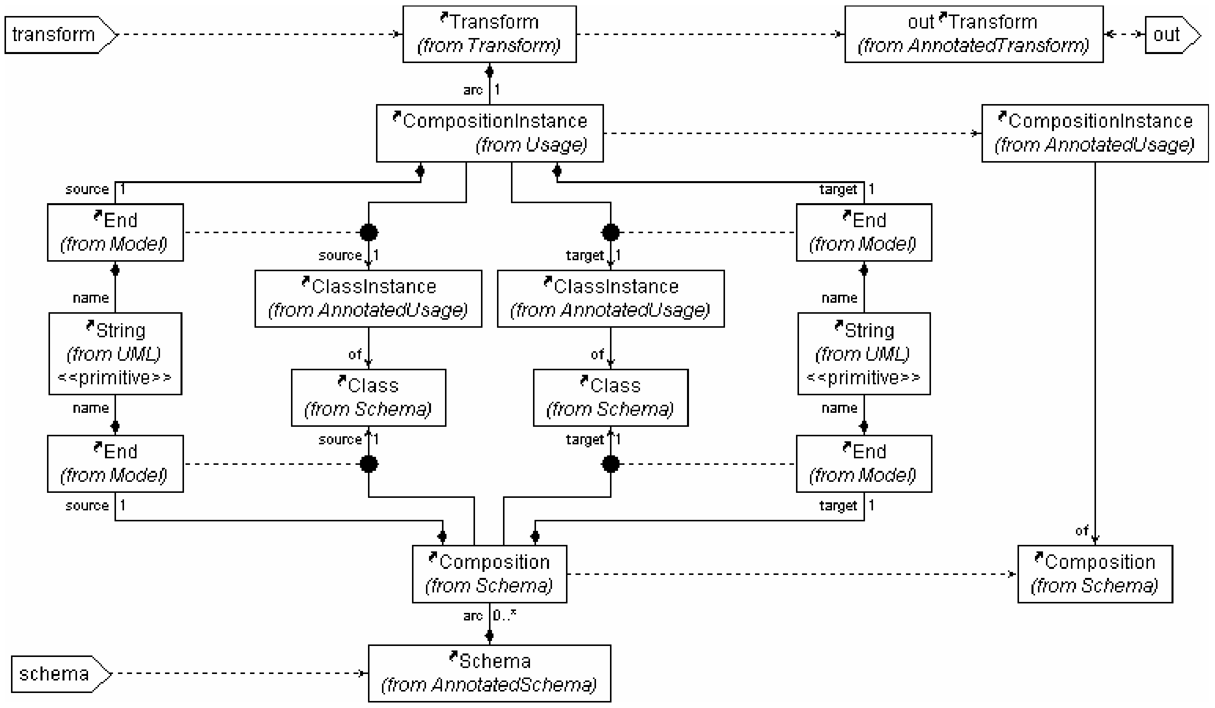

transform
Transform (from Transform)
out Transform (from AnnotatedTransform)
out
arc 1
CompositionInstance (from Usage)
CompositionInstance (from AnnotatedUsage)
source 1
target 1
End (from Model)
source 1
target 1
ClassInstance (from AnnotatedUsage)
ClassInstance (from AnnotatedUsage)
End (from Model)
name
String (from UML) <<primitive>>
of
Class (from Schema)
of
Class (from Schema)
String (from UML) <<primitive>>
name
source 1
target 1
End (from Model)
End (from Model)
source 1
target 1
of
Composition (from Schema)
Composition (from Schema)
arc 0..*
schema
Schema (from AnnotatedSchema)

Figure 4.4 CompositionInstance Annotation

The author regrets that he is unable to program this kind of problem correctly first time in XSLT, and as a result has to engage in a distressing amount of empirical development. The graphical approach enables visual symmetries to be seen, and through validation against a schema, ensures that inter-node references correspond to credible references. The complex and subtle XPath expressions are constructed automatically from something much simpler.

This example has been simplified. The actual transform accounts for inheritance relationships and provides diagnostics for mis-instantiation and multiplicity or type violation. The details of this and all UMLX compiler transforms may be found on the GMT web site at www.eclipse.org/gmt/umlx.

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 transform hierarchy to a simple model.

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

[4] This transformation could usefully form part of an editor validation facility, in order to improve the immediacy and context of diagnostics.

- 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

Gerber et al [4] 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 [7], which we feel is not dissimilar to a textual representation of UMLX. Their concept of tracking before/after instances to correlate multiple transformations is quite similar to establishing 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 [13] draws an interesting distinction between bi-directional mappings and uni-directional transformations, and allows inheritance to be used to specialize transformations. These are areas that remain to be explored in UMLX; it may be that some forms of preservation and evolution are reversible.

The ISIS group at Vanderbilt has pursued the concepts of meta-modelling through the GME tool [5]. 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. 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 distinction between left and right hand sides, there should be greater scope for inter-transform 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 the 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 paper.

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] 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

[5] 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

[6] Edward Willink, "The UMLX Language Definition", http://www.eclipse.org/gmt-home/doc/umlx/umlx.pdf.

[7] DSTC, IBM, "MOF Query/Views/Transformations, Initial Submission", OMG Document ad/2003-02-03, http://www.dstc.edu.au/Research/Projects/Pegamento/publications/ad-03-02-03.pdf.

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

[9] 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

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

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

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

[13] QVT Partners, "Initial submission for MOF 2.0 Query/Views/Transformations RFP", OMG Document ad/2003-03-27, http://www.qvtp.org/downloads/1.0/qvtpartners1.0.pdf.

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

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

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

Model Transformations for the MDA with BOTL

Frank Marschall and Peter Braun

Institut für Informatik
Lehrstuhl Prof. Dr. Manfred Broy,
Software & Systems Engineering
Boltzmannstr. 3
D-85748 Garching

Abstract

In this paper we identify some basic requirements for model transformations for the Model Driven Architecture (MDA) approach and for a model transformation language that is used to specify these transformations. Basically such a language must be precise, allow reasoning about the applicability of specifications, and allow to verify that only valid models are created. Therefore we introduce the Bidirectional Object oriented Transformation Language (BOTL) and show how this language does support the verification of the desired properties. A small running example illustrates our approach.

Key words: Model Driven Architecture, Model Transformation, Metamodel, Refinement, UML

1 Introduction

The MDA (Soley, 2000; OMG-MDA, 2001) introduced by the Object Management Group (OMG) specifies a model based software engineering approach that explicitly separates models at three abstraction layers. The most abstract model, the Computational Independent Model (CIM), deals only with concepts of the application domain. A CIM is refined into a Platform Independent Model (PIM) that contains already computational information about a system but is free from platform specific realization details. These details are part of a Platform Specific Model (PSM), which is a refinement of the PIM.

To derive one model from the other mappings between models must be specified. As already stated in Miller, Mukerji (2003) one must specify mapping rules that specify these transformations. For the specification of mapping rules it is essential to know about the metamodel of the source and target models. This fact is also reflected in the MDA metamodel.

The Unified Modelling Language (UML) (OMG-UML, 2002) is recommended and widely accepted as a specification language for MDA models. The MDA does not specify or prescribe any language for the specification of these model transformations, but currently there is a RFP for a standard to query, create views and transform MOF models for the MDA (OMG-QVT, 2003). Obviously a transformation language for the MDA must be capable to transform object oriented models.

In practice today usually XSL transformations (Clark, 1999) are used to that transform XMI (OMG-XMI, 2002) representations of models. Unfortunately XSL documents lack an intuitive, graphical representation and have some more drawbacks. Since they operate on XMI representations writing MDA transformations is a long winded and fault-prone task. Therefore for example Peltier et al. (2001) proposes a different more abstract approach, which is based upon XSL and helps developing model transformations.

Akehurst (2000) and Gerber et al. (2002) examine different techniques for specifying model transformations. The most promising stem from the area of graph grammars (Rozenberg, 1997; Schürr, 1994). They deliver a theoretical foundation for the transformation of graphs. As a theoretically founded approach they have to be adopted to the concepts of object orientation. Therefore attributed, typed graph grammars extended by constraints may be used for a deep integration with object orientation missing up to now.

Proposals for the QVT, like (DSTC, 2003), provide a specification of transformation semantics. However, none of the existing approaches provides any techniques to ensure or prove that their application won't cause any conflicts and that generated models will be conform to their metamodel. For this purpose we introduce model transformation language BOTL for the transformation of object oriented models. Being mathematically founded BOTL allows reasoning about properties of transformations.

A complete definition of the BOTL is far out of scope of this paper and can be found in Braun, Marschall (2003). Instead we aim to give the reader an idea of the BOTL, its basic concepts, its capabilities, and how the MDA approach can profit from such a language. We show exemplary how this language can be used in the context of the MDA in Section 2. Section 3 gives an overview over techniques that allow one to decide whether the application of a transformation may fail at runtime, Section 4 introduces the concepts that are needed to decide whether generated models will conform to a given target metamodel. Finally, Section 5 summarizes the results.

2 BOTL Transformations for the MDA

As the MDA requires models to be formal, there is a *metamodel* for every kind of MDA model that defines the structure of valid models of that type. A *model transformation specification* defines how a model is derived from an existing model. Thereby the newly created model is denoted as the *target model*, while the existing one is called the *source model*. For each model there must exist a metamodel, called *source* resp. *target metamodel*.

A model transformation specification is called *applicable*, if the transformation it defines can be applied for any arbitrary source model that conforms to the source metamodel. Thereby an applicable model transformation specification must be deterministic, i.e. it must produce always the same target model for a given source model. If it holds that all created target models of a model transformation are conform to the target metamodel, then the according transformation specification is called to be *metamodel conform*. Obviously applicability and metamodel conformity are crucial properties of transformation specifications, especially when the transformations is to be performed by a tool. A prerequisite for reasoning about these properties is a precise *model transformation language* for the specification of transformations.

In the following we provide a very simple example for a model transformation of a part of a platform independent model into a platform specific CORBA model. The example originally stems from Miller, Mukerji (2001) but additional package names, OCL, and natural language constraints have been omitted to keep it more easily understandable.

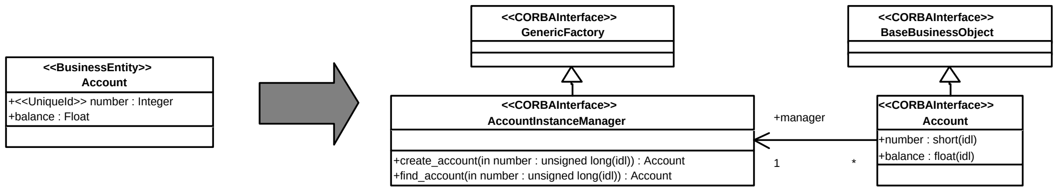


Fig. 1. A graphical representation of a PIM and a PSM that was generated from it.

The left side of Figure 1 shows the sample PIM. A CORBA specific model that was derived from it can be seen at the right side. There exists a metamodel in form of a UML profile for each of the two models. Since the profiles are easy to imagine we don't present them graphically here.

It is easy to understand the idea behind this transformation by inspecting the sample application: For every `BusinessEntity` two CORBA interfaces are created. One that inherits from the interface class `BaseBusinessObject` and an instance manager interface that inherits from the `GenericFactory` interface. Further the factory interface has a `create` and a `find` method that get the attribute stereotyped as `UniqueId` of the `BusinessEntity` as an argument.

Obviously examples are a useful way to illustrate model transformations but they are not sufficient to provide an unambiguous specification for them. Also textual specifications of transformations as they are used today are not unambiguous and allow no reasoning about the applicability of a specified mapping.

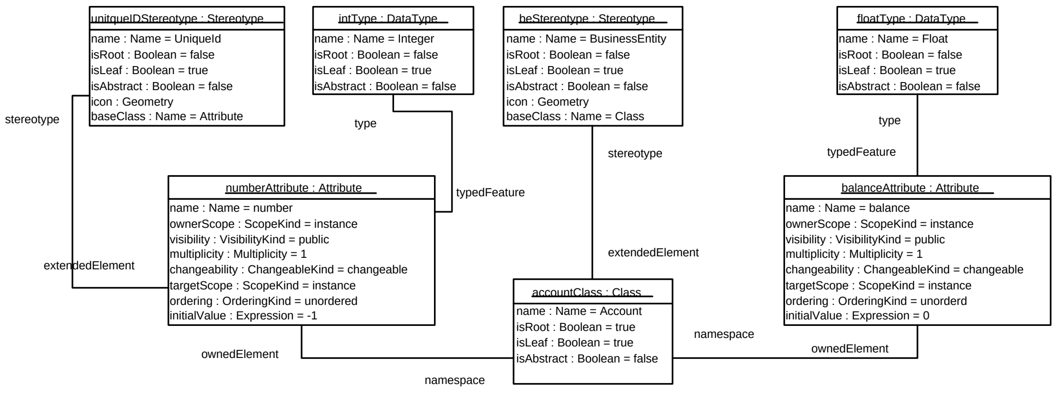

unitqueIDStereotype : Stereotype
name : Name = UniqueId
isRoot : Boolean = false
isLeaf : Boolean = true
isAbstract : Boolean = false
icon : Geometry
baseClass : Name = Attribute
intType : DataType
name : Name = Integer
isRoot : Boolean = false
isLeaf : Boolean = true
isAbstract : Boolean = false
beStereotype : Stereotype
name : Name = BusinessEntity
isRoot : Boolean = false
isLeaf : Boolean = true
isAbstract : Boolean = false
icon : Geometry
baseClass : Name = Class
floatType : DataType
name : Name = Float
isRoot : Boolean = false
isLeaf : Boolean = true
isAbstract : Boolean = false
stereotype
type
type
stereotype
typedFeature
numberAttribute : Attribute
name : Name = number
ownerScope : ScopeKind = instance
visibility : VisibilityKind = public
multiplicity : Multiplicity = 1
changeability : ChangeableKind = changeable
targetScope : ScopeKind = instance
ordering : OrderingKind = unordered
initialValue : Expression = -1
typedFeature
balanceAttribute : Attribute
name : Name = balance
ownerScope : ScopeKind = instance
visibility : VisibilityKind = public
multiplicity : Multiplicity = 1
changeability : ChangeableKind = changeable
targetScope : ScopeKind = instance
ordering : OrderingKind = unorderd
initialValue : Expression = 0
extendedElement
extendedElement
accountClass : Class
name : Name = Account
isRoot : Boolean = true
isLeaf : Boolean = true
isAbstract : Boolean = false
namespace
ownedElement
ownedElement
namespace

Fig. 2. The PIM from Figure 1 in terms of the UML metamodel.

Figure 2 shows the PIM as an instance of UML metamodel, while Figure 1 shows its graphical representation as an UML class diagram. The PIM contains much more elements than its representation as a class diagram but naturally all this information has to be considered by a transformation specification between our given models. Thus to precisely define a transformation between a PIM and a PSM we have to refer to model elements, because the UML metamodel level is not expressive enough therefore. E.g. one cannot differentiate between several instances of the same type in terms of the UML metamodel layer. Thus like Gogolla (2000) and Bottoni et al. (2002) we refer to instances of the UML metamodel to specify transformation rules for UML models.

We propose the rule based model transformation language BOTL for the specification MDA mappings. This language combines the illustrative clearness of a graphical specification with the precision of a formal founded language. BOTL is intended to be supported by a tool that translates fragments of a source model into target model fragments and merges the newly created fragments into a target model. BOTL transforms always object models. The source and target metamodels are consequently class models, which could be mapped to instances UML or MOF meta classes.

Figure 3 depicts a sample BOTL rule. When applied it searches in the source model (the PIM) for occurrences of a model fragment with the same structure as the pattern at its upper side and creates new fragments according to the lower pattern. The objects' identities and attribute values in the model fragment patterns contain terms. Constant values written within quotation marks, a ♢ indicates that the given value is irrelevant. For every match of the source pattern in the source model one new model fragment of the target model is created. The structure of this new fragment is determined by

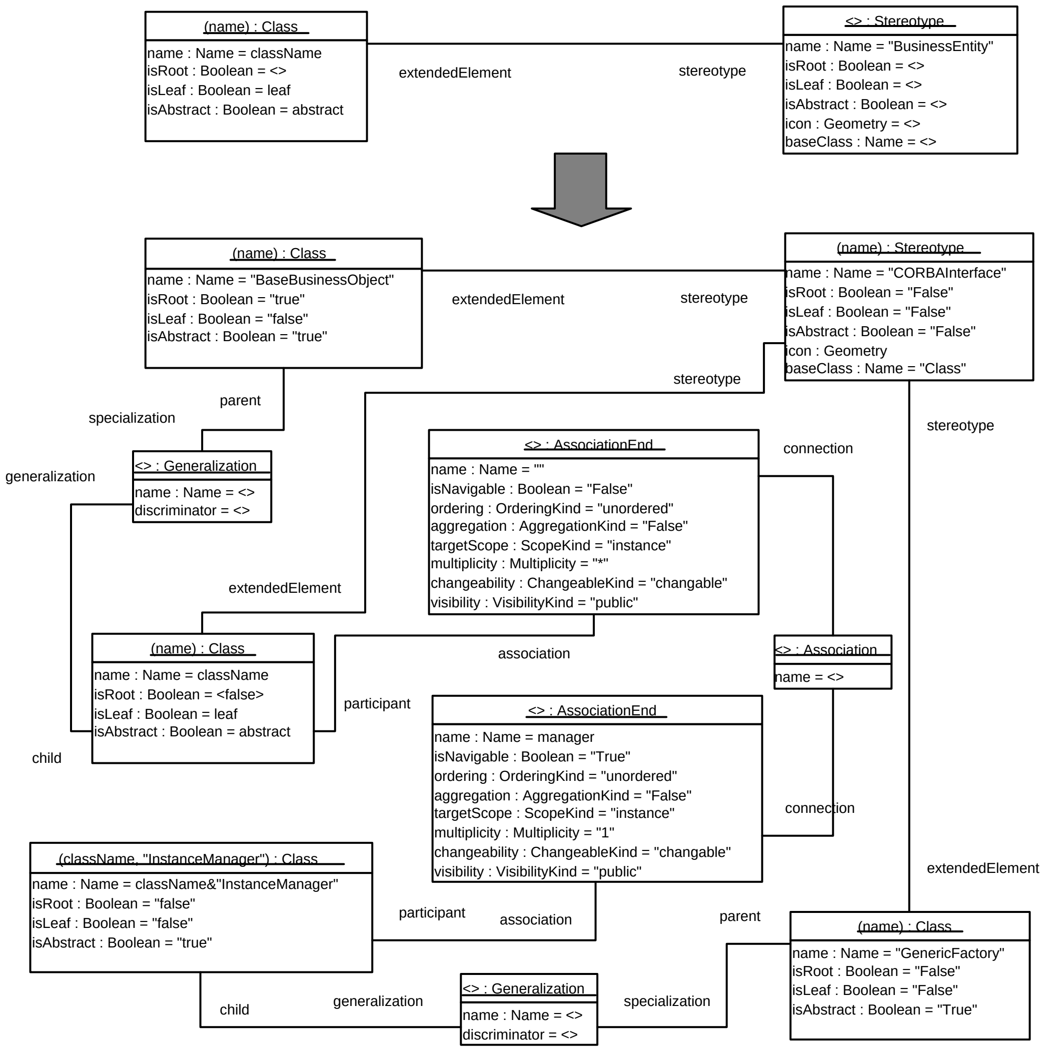

(name) : Class
name : Name = className
isRoot : Boolean = <>
isLeaf : Boolean = leaf
isAbstract : Boolean = abstract
extendedElement
stereotype
<> : Stereotype
name : Name = "BusinessEntity"
isRoot : Boolean = <>
isLeaf : Boolean = <>
isAbstract : Boolean = <>
icon : Geometry = <>
baseClass : Name = <>
(name) : Class
name : Name = "BaseBusinessObject"
isRoot : Boolean = "true"
isLeaf : Boolean = "false"
isAbstract : Boolean = "true"
extendedElement
stereotype
(name) : Stereotype
name : Name = "CORBAInterface"
isRoot : Boolean = "False"
isLeaf : Boolean = "False"
isAbstract : Boolean = "False"
icon : Geometry
baseClass : Name = "Class"
stereotype
parent
specialization
stereotype
generalization
<> : Generalization
name : Name = <>
discriminator = <>
<> : AssociationEnd
name : Name = ""
isNavigable : Boolean = "False"
ordering : OrderingKind = "unordered"
aggregation : AggregationKind = "False"
targetScope : ScopeKind = "instance"
multiplicity : Multiplicity = "*"
changeability : ChangeableKind = "changable"
visibility : VisibilityKind = "public"
connection
extendedElement
(name) : Class
name : Name = className
isRoot : Boolean = <false>
isLeaf : Boolean = leaf
isAbstract : Boolean = abstract
association
<> : Association
name = <>
participant
child
<> : AssociationEnd
name : Name = manager
isNavigable : Boolean = "True"
ordering : OrderingKind = "unordered"
aggregation : AggregationKind = "False"
targetScope : ScopeKind = "instance"
multiplicity : Multiplicity = "1"
changeability : ChangeableKind = "changable"
visibility : VisibilityKind = "public"
connection
extendedElement
(className, "InstanceManager") : Class
name : Name = className&"InstanceManager"
isRoot : Boolean = "false"
isLeaf : Boolean = "false"
isAbstract : Boolean = "true"
participant
association
parent
(name) : Class
name : Name = "GenericFactory"
isRoot : Boolean = "False"
isLeaf : Boolean = "False"
isAbstract : Boolean = "True"
child
generalization
<> : Generalization
name : Name = <>
discriminator = <>
specialization

Fig. 3. A BOTL rule that specifies a part of the desired model transformation.

the target pattern. Since the model patterns serve as placeholders for existing and created model fragments we call them *source* resp. *target model variable*. The objects in a model variable are denoted as *object variables*. Regarding the QVT approach source model variables provide a mechanism to specify queries.

The new attributes' values are computed from the terms in the two model variables' attribute values. ◇ values in the target pattern are replaced by appropriate default values at the end of the transformation. The identity of generated objects is also determined by a term in the target object variable. If two objects with the same identity are created, then these objects are merged into one object, i.e. their attribute values are merged, whereby only ◇ values are overwritten with other values. Merging two associations of the same type between two objects with the same identity yields in one association that has the maximum cardinality of the two associations[1]. Figure 4 depicts the merge

[1] We regard multiple associations of the same type between objects as *one* associ-

operation exemplary. So the fragments that are created by the subsequent application of several rules are be merged to one coherent new model.

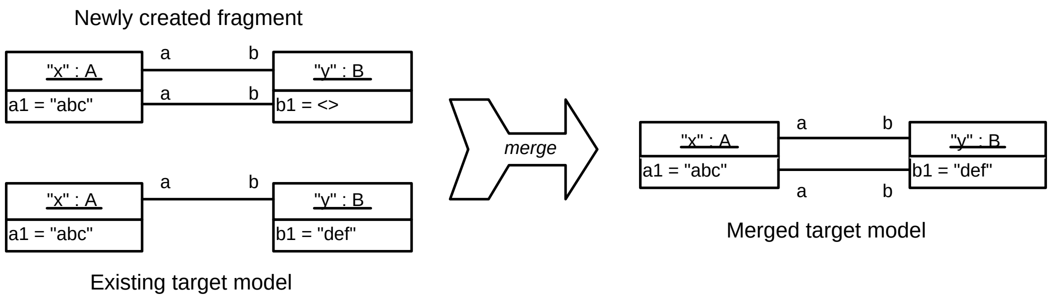


Fig. 4. Merging a newly created model fragment and an existing target model

A BOTL rule set may consist of a number of rules that all create model fragments. Thereby it is interesting to know that the order of pattern matches during one rule application as well as the order of the application of the different rules does not affect the result of a BOTL transformation. Proofs for these statements can be found in Braun, Marschall (2003).

A more detailed explanation of the application of BOTL rules is provided in Braun, Marschall (2002), a formally founded definition can be found in Braun, Marschall (2003). For this work we assume that the semantics of BOTL rules are intuitive enough to be understood by means of the given example.

The sample rule searches the source model for an object of the UML metaclass `Class` associated with a `Stereotype` with a `name` value `"BusinessEntity"`. For every found business entity the rule creates a CORBA interface and an appropriate instance manger interface that inherit from the class `GenericFactory` resp. `BaseBusinessObject` as shown in Figure 1.

The interfaces are of the type `Class`. Their identity is determined by their `name` attributes that serve as primary keys, which is indicated by the `(name)` in the identifier field of these object variables. This ensures that classes with the same name are always mapped to the same objects of the type `Class` in the PSM. The rule creates only the empty CORBA interfaces. Additional rules are needed to copy attributes into the PSM interfaces and to create appropriate `create` and `find` method declarations for the instance manager interface.

Since the instance representation of our example's PSM is even much bigger than the sample PIM we omit it here. We also skip the specification of the complete rule set for the desired transformation and instead discuss how BOTL allows reasoning about applicability and metamodel conformance, which are prerequisites for their usage in the context of the MDA.

ation with a given *cardinality*.

3 Applicability of BOTL specifications

For a tool supported automatic transformation of models, as e.g. the generation of a PIM from a PSM, it must be ensured that the transformation will not fail because of an inconsistent or incomplete specification. Thus a model transformation language must have a notion of applicability and allow (automatic) reasoning about it. We now introduce how BOTL supports this feature.

3.1 Applicability of BOTL rule sets

We call the property that a BOTL rule set produces output for any arbitrary source model that is conform to its source metamodel *applicability* of a rule set. Thereby the transformation of a model might fail for two different reasons.

First it might happen that it isn't possible to calculate a valid value for an attribute. This might be the case e.g. if we should calculate an attribute value from the term $\sqrt{x}$, whereby the variable x is assigned to a negative value. The second critical situation that may arise is when we try to merge two objects with the same identity that contain contradictory values for the same attributes, i.e. both values are different and differ from $\diamondsuit$.

If one of these two cases occurs, then the specified transformation cannot be applied, because there is no unambiguous solution for the resulting target model. Generally we can state that rule set is applicable, if

- each of its rules alone is applicable, and
- there are no two rules that generate two objects of the same type whereby the same attribute gets assigned a value different from $\diamondsuit$, twice.

The second postulate ensures that there are no conflicting attribute values: only one rule of the set can effectively write an attribute's value. The values of primary key attributes don't have to be considered, because two objects with different primary key attribute values won't ever have the same identity.

3.2 Applicability of a BOTL rule

There are still techniques needed to prove that a single rule is applicable. Therefore we identified three criteria that have to hold. For a formal discussion of the presented techniques we refer again to Braun, Marschall (2003):

Computable attribute values: We can determine unambiguous values for the attributes of newly created objects.

No conflicting attributes from one object variable: An object variable in a target model variable does not create different values for the same attribute of one object during the subsequent application of a rule.

No conflicting attributes from different object variables: Different object variables of the same type don't create conflicting attribute values, too.

The following sections give a rough overview on how these properties can be verified. Again we just aim to give the reader an idea of the applied concepts.

3.3 Computable attribute values

The terms in the model variables together with the values of the source and the created target model fragment form an equational system of the kind:

"Found attribute/id value" = "Term in source object variable"
"New object's attribute/id value" = "Term in target object variable"

If we regard a created objects attribute/id values as the unknown variables, then the system must have an unique solution. For the example in Figure 3 the value for the attribute `name` of a newly created instance manger is obtained from the following equational system:

theFoundClass.`name` = `className`
theNewInstanceMgr.`name` = `className & "InstanceManager"`
⇒ theNewInstanceMgr.`name` = theFoundClass.`name&"InstanceManager"`

3.4 No conflicting attributes from one object

Whenever an attribute's *att* value different from $\diamondsuit$ is written into a newly created object with the identity *id* it must hold for every match that:

The set of source objects, that are used to compute the value of *att*, is determined by the set of source objects, that are used to compute *id*.

Therefore BOTL provides some basic algorithms to determine,

- on which source object variables an identity, that is created from a given target object variable, depends,
- on which source object variables the value, that is created from a given attribute of an target object variable depends, and

- whether it holds for two sets A and B of source object variables that: Whenever the elements of A match to the same source objects, then the elements of B match always the same source objects, too.

The last item might hold, if e.g. the objects matched to B have always to-one associations to objects matched by A according to the source metamodel. In the example from Figure 3 this statement is e.g. easy to prove for the object of type `Class` at the lower left corner. Its identifier is determined by the `Class` object of the source model variable and the only attribute value that is not constant (`name`) is also determined by the same source model variable object.

3.5 No conflicting attributes from different objects

The simplest way to ensure this property is to prohibit two or more object variables of the same type in a target model variable. But BOTL also provides another technique to prove this property, if this simple heuristic doesn't work.

The proof technique is based on an equational system that reflects the situation when two different target object variables match to the same target object during two rule applications. In this case the equational system has a restricted solution that reflects the situation when this case may occur. Now consider a set of additional equations, which state that all generated attribute values generated during these two applications are identical. If these new equations do not further restrict the systems solution the property does hold.

The latter technique can be used to prove for the sample rule that there are no conflicts that stem from different target object variables of the class `Class`.

4 Metamodel Conformance of BOTL Specifications

While applicability states that a model transformation specification is realizable, metamodel conformance states that a transformation will yield to valid target models, according to the target metamodel. This section presents the basic concepts that are used to show that a given BOTL rule set generates models that are conform to a target metamodel.

Within BOTL a model is regarded as *metamodel conform*, if

- all objects in the model are of a type that occurs in the metamodel,
- all associations in the model are of a type that occurs in the metamodel and connect objects of the correct types,

- every object in the model has not more outgoing associations of a type than the class association does allow (we denote this property as *upper bounds conformance*), and
- every object has not less outgoing associations of a type than the class association does allow (we denote this property as *lower bounds conformance*).

The first two properties are easy to verify. It has simply to be ensured that the first two postulates do hold for all target model variables. If this is the case, then the application of the rule set can't create any invalid objects or associations. These properties are already guaranteed by the syntax of BOTL rules. Note that BOTL in contrast to the UML does not yet guarantee that no sequences of composite or aggregated objects occur in a target model.

Generally it holds that a BOTL rule set is metamodel conform, if it is *applicable*, *upper bounds conform*, and *lower bounds conform.*

4.1 Upper Bounds Conformance

To verify that a rule set is upper bounds conform it has to be determined for every association in every target model variable how often this association might be maximally created as an outgoing association from the same object. This information can only be gained by reasoning about the identities of the generated objects that associations connect. Generally there are four possibilities for the identity of a new object:

(1) It results from a target object variable with a fixed identifier.
(2) It results from a target object variable with $\diamondsuit$ as its identifier term. I.e. whenever the rule is applied a new object with a unique id is created.
(3) The object's identifier depends on a set of source objects as defined in the rule. I.e. the identifier can be calculated from this set of source objects.
(4) It is not possible to make any statements about the object's identifier.

According to this we can make several estimations about the maximum number of outgoing association of an object that stems from the same model variable association. E.g. if we know that the identifiers of the connected objects variables are both constants or both $\diamondsuit$, then we know that the rule can create only one outgoing association of the given type for every created object.

A case of special interest is when a created target object's identity is one-to-one dependent on the identity of a set of matched source objects. Regarding the source metamodel and the source model variable we can reason about how often this rule can be maximally applied creating the same object. Together with the information about the identifiers at the other hand of the association we can calculate a value that states how often the association might occur as an outgoing association from the same object.

The obtained values are summed up for every end of every association type and compared to the maximal allowed multiplicity of this association type to determine whether this multiplicity constraint might be violated or not.

Since some associations could vanish during the merge process as already indicated in Figure 4, BOTL further comes with a mechanism to determine whether an association in a model variable is redundant. In this case we don't have to consider it for the verification of upper bounds conformance.

4.2 Lower bounds conformance

There is a very simple heuristic to verify that a rule set is lower bounds conform: if every target object variable has the required minimum of outgoing associations according to the target metamodel, then the same holds for all models generated from this rule. This is, because when a new object is created all required outgoing associations are created within the same rule application.

However in some cases this heuristic might not be strong enough to prove lower bounds conformance. For these cases BOTL defines techniques that are very similar to those described in the previous section.

5 Conclusion

In this paper we first highlighted the essential role that appropriate techniques for model transformations play for the MDA approach. A human readable but unambiguous model transformation language is a prerequisite for the successful application of this approach. But such a language must also allow to reason about the applicability of specifications and their ability to guarantee that it creates only valid (i.e. metamodel conform) models.

Therefore we introduced the basic concepts and features of the Bidirectional Object oriented Transformation Language (BOTL). BOTL allows to specify transformations among object oriented models and to verify the desired properties of applicability and metamodel conformance at specification time. Since a detailed discussion of the BOTL verification techniques would have exceeded the scope of the paper, only the most important features and techniques were presented to give the reader an idea about the capabilities of the language.

We propose BOTL as a language for the specification of mappings between the different model layers of the MDA. Further BOTL could be easily extended to specify transformations on a single model. We already use BOTL within the project KOGITO to specify views by defining a mapping of the abstract syntax of a views description technique into a common conceptual model.

Currently we are working on a tool support for BOTL that allows one to specify rules, verify their correctness and generate code for the transformation of various kinds of object oriented models.

References

David H Akehurst: Model Translation: A UML-based specification technique and active implementation approach, Phd. thesis, University of Kent, 2000

Peter Braun and Frank Marschall: Transforming Object Oriented Models with BOTL, In: Graph Transformation (ICGT) 2002, Editor: Paolo Bottoni and Mark Minas, ENCTS series 72.3, Elsevier Science B. V.

Peter Braun and Frank Marschall: BOTL—The Bidirectional Object Oriented Transformation Language, Technical Report, TU München, 2003

J. Clark: XSL Transformations (XSLT) 1.0, WWW Consortium (W3C), 1999

DSTC: Query / View / Transformations, Initial Submission, 2003

A. Gerber, M. Lawley, K. Raymond, J. Steel and A. Wood: Transformation: The Missing Link of MDA, In: Graph Transformation (ICGT), 2002 Editor: Paolo Bottoni and Mark Minas, ENCTS series 72.3, Elsevier Science B. V.

Martin Gogolla: Graph Transformations on the UML Metamodel, ICLAB Workshop on Graph Transformations and Visual Modeling Techniques, 2000, Carleton Scientific

P. Bottoni, F. Parisi-Presicce, G. Taentzter: Coordinated Distributed Diagram Transformation for Software Evolution, Software Evolution through Transformation, ENTCS 74, 2002

Model Driven Architecture (MDA), Editors: J. Miller and J. Mukerji, 2001

MDA Guide Version 1.0, Editors: Joaquin Miller and Jishnu Mukerji, 2001

OMG Model Driven Architecture (MDA), 2001

OMG MOF 2.0 Query / View / Transformations, Request for Proposal, 2003

OMG Unified Modeling Language Specification (Action Semantics), 2002

OMG XML Metadata Interchange (XMI) Specification, Januar 2002

Mikaël Peltier, Jean Bézivin and Gabriel Guillaume: MTRANS: A general framework, based on XSLT, for model transformations Workshop on Transformations in UML (WTUML), Genova, Italy, 2001

Grzegorz Rozenberg: Handbook of Graph Grammars and Computing by Graph Transformation, World Scientific, 1997

Andy Schürr: Specification of Graph Translators with Triple Graph Grammars, WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, 1994, Editor: G. Tinhofer, LNCS 903, Springer

Richard Soley: Model Driven Architecture, OMG Staff Strategy Group, 2000

CODEX - An Automatic Model View Controller Engineering System

Henrik Larsson, Kalle Burbeck

Department of Computer and Information Science
Linköpings Universitet
SE-581 83 Linköping
Sweden

Abstract

There are numerous instances in computer science where a data "core" is present and different ways of looking at that core are wanted. In OMG's Model Driven Architecture (MDA) there are mappings between the platform independent model, the core, and the platform specific models. It would be highly beneficial to have inverses for these mappings so that changes in every model could be propagated to the other models.

Automatic model [1] view controller engineering (MVARE) provides a systematic method of keeping the core, or *model*, consistent with *views* that show certain aspects of the model. It does this by demanding that the transformations between the model and the views have inverses that can be automatically calculated. This paper introduces CODEX, which is an MVARE system. In CODEX, transformations from the model to each view are specified manually, but the inverse transformations, from the views to the model, are generated automatically. The transformations are specified using graph rewriting based on modified double pushout.

---

1 Introduction

Transformations are abundant in computer science. There are countless areas where data of some kind is transferred from one domain to another, UML to source code, high level languages to low level languages, graphical representations to textual

[1] Note that model in MVARE is *not* the same as model in MDA. In sections 2 and 3 the word model is used exclusively in the MVARE sense. In the other sections it will be explicitly stated which meaning is meant if it is not clear by context.

Email addresses: `henla@ida.liu.se` (Henrik Larsson), `kalbu@ida.liu.se` (Kalle Burbeck).

representations and so on. In all of these cases there exist working implementations and not seldom good methods to use.

Also in OMG's model driven architecture (MDA) [1] such transformations are present. One case is the mappings between the platform independent model (PIM) and the platform specific models (PSM). Here the PIM abstracts away implementation details of a system, while the PSMs include information needed for a certain platform.

However, the above mentioned examples all have a common trait; that often transformations in the opposite direction are wanted, that is round-trip engineering systems [2–6]. Here is where the existing methods break down. There are practically no methods to use to construct the inverse of a transformation. It would be highly beneficial if the original transformation could be used to gain the transformation in the opposite direction.

The goal of automatic round-trip engineering (ARE), and automatic model view controller engineering (MVARE), is to introduce a systematic method of specifying transformations in a way such that the inverse transformation can be inferred automatically.

The possibility to calculate inverses of transformations means in terms of PIM and PSM that given a mapping from a PIM to a PSM, the mapping from the PSM to the PIM would be known, see figure 1. Assume that there is a PIM for a system and there are several PSMs of the system for different platforms. Assume further that there is a manually specified transformation from the PIM to each PSM. Now, if each transformation is specified according to the rules of MVARE there will also be transformations from each of the PSMs to the PIM. This means that when changes to the system are wanted, editing can be done in any model (PIM or PSM) and it is still possible to keep the models consistent. Consider, for example, that editing is done in one of the PSMs. Then to bring the other models up to date, the edited PSM is transformed to the PIM, using the inverse transformation, and then the PIM is transformed to the other PSMs.

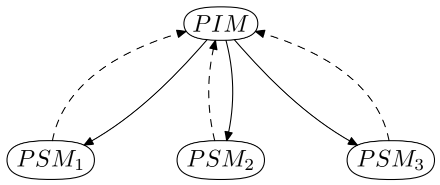


Fig. 1. Transformations between PIM and PSMs. The transformations drawn with solid lines are specified manually and the ones with dashed lines are generated automatically.

This paper introduces an MVARE system named CODEX. It is based on double pushout, a graph rewriting technique [7]. CODEX is the first method that can realize the scenario described in the last paragraph.

2 Background

2.1 *Automatic round-trip engineering*

Automatic round-trip engineering (ARE) [8] takes round-trip engineering [2] one step further. In round-trip engineering a transformation from one domain to another is implemented. Then a transformation in the reverse direction also has to be implemented. The main idea behind ARE is that it should be possible to automatically infer the inverse transformation.

Definition 2.1 (Automatic round-trip engineering) *Let A and B be two domains, and $f : A \rightarrow B$ a transformation function from function space F. If there is a function $i : F \rightarrow F$ which calculates for f its inverse $f^{-1} \in F$ then $R = (A, B, f, i)$ is an* automatic round-trip engineering system (ARE).

2.2 *Automatic model view controller engineering*

An automatic model view controller engineering system (MVARE) [8] is a system where two or more invertible transformations with a common source domain exist. The transformations project the common source, the model, to different targets, the views, that are better suited for dealing with a specific aspect of the system.

Definition 2.2 (View) *A view is a triple $v = \langle O, A, R \rangle$ where O is a set of objects, $A = \{A_i | A_i \subseteq O\, i \in \mathbb{N}\}$ is a set of subsets of O which specifies the domains of the relations in R and R is a set of relations $r : A_0 \times \cdots \times A_i$ for any natural number i.*

The above definition of a view says that everything that is a set of objects and relations is a view. However, a way to identify views that have semantic meaning is needed. This is the purpose of an identification function.

Definition 2.3 (Identification of a view) *Let V be the set of all views, $f : V \rightarrow \{0, 1\}$ be a function such that $f(v) = 1$ iff $v \in V$ and v is a valid view, we say that v is a valid view according to f and f is the identification function for v.*

The identification function can now be used to specify domains of views; views which in some way belong together.

Definition 2.4 (View domains) *Let $D_f = \{d | f(d) = 1, d \in V\}$ where V is the set of all views and f is an identification function, then D_f is the domain of all views valid according to f.*

With views and view domains defined it is now possible to define an MVARE system.

Definition 2.5 (Automatic model view controller engineering) *Let* $A, B_1, \ldots, B_n$ *be* $n+1$ *domains, and* $f_j : A \rightarrow B_j$ *a transformation function from function space* F, $j = 1, \ldots, n+1$. *If there is a function* $i : F \rightarrow F$ *which calculates for every* f_j *its inverse* $f_j^{-1} \in F$ *then* $R = (A, B, f_1, \ldots, f_n, i)$ *is an* automatic model view controller engineering *system (MVARE). A is called the* model domain, *while* $B_1, \ldots, B_n$ *are called the* view domains. *The* f_j *are called the* projections, *the* f_j^{-1} *are called the* integrations.

2.3 Double pushout

Double pushout (DPO) is a graph rewriting technique based on category theory. In DPO graphs and graph morphisms form a category. A very brief description of DPO is given here with a focus on what is necessary to know for this paper. For a more theoretical view, see [7].

A *rule* in DPO is given by three graphs, L, K and R, and two injective morphisms, l and r. See figure 3 for an example of a rule. The leftmost graph is the pattern to match against a graph. The middle graph is the subgraph that is the intersection of L and R and is called the *interface* of the rule. The nodes and edges that are part of L but not K will be removed from the graph when the rule is applied. The rightmost graph states the graph objects that will be added to the graph when applying the rule. The morphisms, l and r, are injective and maps K to L and R respectively.

A DPO rule can only be applied if the match fulfills the *gluing condition*. The gluing condition consists of two parts, the *dangling edges condition* and the *identification condition*.

The dangling edges condition specifies that if a node is deleted by a rule, then all edges to that node have to be explicitly deleted by the rule. The identification condition requires that every object to be deleted has only one pre-image in L.

3 CODEX

CODEX (COmplete reDEXes) is an MVARE system. The domains can be anything that can be represented as graphs. The projections are specified by using a modified double pushout (DPO) technique described below.

Definition 3.1 (CODEX system) *A* CODEX *system consists of* $n+1$ *domains* $M, V_1, \ldots, V_n$, *where* M *is the model domain and* $V_1, \ldots, V_n$ *are the view domains,* *and the* CODEX *transformations* $t_1,\ldots,t_n$*, where* t_i *is applied to the model to get the view* $v_i \in V_i$*,* $i = 1,\ldots,n$*. A* CODEX *transformation* t_i *consists of one or more graph rewrite rules,* $r_{i,j}$*,* $j = 1,2,\ldots$*.*

The construction of the inverse transformation is very simple. The left hand side and the right hand side of all rules "switch sides", that is $L(r_{i,j}^{-1}) = R(r_{i,j})$ and $R(r_{i,j}^{-1}) = L(r_{i,j})$. The inverse transformation t_i^{-1} is now the set of rules $r_{i,j}^{-1}, j = 1,2,\ldots$.

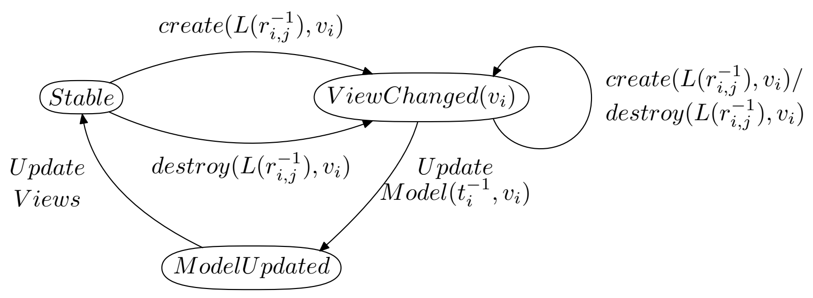
3.1 States and events

This section describes the states the system can be in and which editing events can occur. Figure 2 shows how the states and events interact. In this system only one view may be modified at a time. The possible states of a CODEX system are the following:

- $Stable$ — When the system is in the stable state, every view is consistent with the model. No changes have been made in any view. Any view may be modified, thereby bringing the system to the $ViewChanged(v_i)$ state and preventing changes in other views until the system is in the $Stable$ state again.
- $ViewChanged(v_i)$ — In this state the view v_i has been modified, either by a create or a destroy event. In this state only changes to v_i are allowed.
- $ModelUpdated$ — In this state the model has been updated to be consistent with the view that was changed in the $ViewChanged(v_i)$ state. All views except v_i now need to be updated to bring the system to the $stable$ state.

The four events in a CODEX system are:

- $Create(L(r_{i,j}^{-1}), v_i)$ — This event occurs when a redex of the pattern $L(r_{i,j}^{-1})$ is created in the view v_i. This is only allowed when the system is in the $Stable$ or $ViewChanged(v_i)$ states.
- $Destroy(L(r_{i,j}^{-1}), v_i)$ — This event occurs when the user destroys a redex or node in view v_i. The event can happen in the $Stable$ and $ViewChanged(v_i)$ states.
- $UpdateModel(t_i^{-1}, v_i)$ — The event of the model being updated by applying the transformation t_i^{-1} on the view v_i. This event can occur in the $ViewChanged(v_i)$ state.
- $UpdateViews$ — In the $ModelUpdated$ state this event is triggered to update every inconsistent view according to the model by applying every t_i to the model.



$create(L(r_{i,j}^{-1}), v_i)$
$Stable$
$ViewChanged(v_i)$
$create(L(r_{i,j}^{-1}), v_i)/$
$destroy(L(r_{i,j}^{-1}), v_i)$
$Update$
$Views$
$destroy(L(r_{i,j}^{-1}), v_i)$
$Update$
$Model(t_i^{-1}, v_i)$
$ModelUpdated$

Fig. 2. State chart of a CODEX system.

3.2 CODEX transformations

CODEX transformations are similar to double pushout transformations. The main difference is that in CODEX applications of rules are "remembered" by using tags. The first time a transformation is applied, it is nondeterministic. However, a transformation is always remembered afterwards. This means that once a rule has used a graph object in a rule application in a transformation, its inverse will also do so. Further, any succeeding applications of the transformation on the tagged graph will result in the exact same view.

A number of modifications to double pushout are needed to accomplish this and they are described below. Two views, v_1 and v_2, with one rule each, $r_{1,1}$ and $r_{2,1}$ will be used throughout this section to illustrate examples. The rules $r_{1,1}$ and $r_{2,1}$ are identical and looks as shown in figure 3.

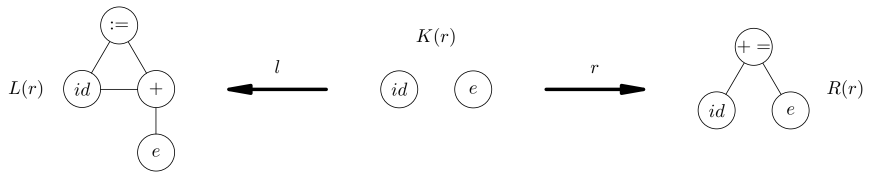


Fig. 3. Two CODEX rules, $r_{1,1}$ and $r_{2,1}$, which will be used throughout this section.

3.2.1 Object roles

Depending on how a graph object is used in a rule it is assigned an *object role*. The possible object roles of a rule are as follows:

- *A-objects* are the objects that are added to the graph when applying a rule.
- *D-objects* are the objects that are deleted from the graph when applying a rule.
- *K-objects* are the invariant objects that are matched in the graph but left unchanged.
- *U-objects* are the objects that are not matched by any rule in the transformation, they are *unaffected*.

- *G-objects or ghosts*. If an object to be deleted by a rule application in v_i has been previously used by another view, $v_j, j \neq i$, then it can't be permanently removed. It has to be saved if it will be possibly to restore v_j. This is solved by *ghosts*, or *G*-objects, which are not shown in the current view, v_i. Figures 5 and 7 illustrate the use of ghosts.

Definition 3.2 (Object role) *Let $v_i \in V_i$ be a view in a* CODEX *system, O be the objects in v_i and $or : O \rightarrow \{A, D, K, G\}$ be the* object role function, *which maps every object to an object role.*

3.2.2 Tags

To make it possible to invert transformations, tags are used to identify redexes of rule applications. A unique rule application tag is assigned to a graph object every time it is matched in a rule application.

Definition 3.3 (Object tag) *The tag is a tuple $\langle vid, rid, aid, or \rangle$, where vid is a view identifier, rid is a rule identifier, aid is a rule application identifier, and or is the object role* as seen from the view V_{vid}.

Objects with the same *vid*, *rid* and *aid* are part of the same redex. The combination of these three identifiers is unique in the CODEX system. With the help of the tags it is possible to identify the objects that are part of a certain application of a rule within a view and therefore also to restore it to the way it was previously presented in the view.

Note that the object role in the tag is seen from the view's perspective. This means, for example, that there will never be any *D*-objects in the model and also that no *A*-objects will exist in a view. When referring to an object role it is always from the perspective of a certain view.

3.2.3 Colors

Unaffected graph objects may or may not be visible in a view. This is up to the implementor or user of the system. Colors are used to express if an object is visible in a view or not. If an object should be shown in a view it will be colored red and if not it will be colored black.

Definition 3.4 (Object color) *Let $v_i \in V_i$ be a view in a* CODEX *system, O be the objects in v_i, and $c : O \rightarrow \{red, black\}$ be the* color function *such that $c(o) = red, o \in O$ iff o is visible in view v_i.*

3.2.4 A CODEX view

To relate back to the definition of views in definition 2.2, it is now possible to state what a CODEX view consists of. A view in codex is a graph, $G = \langle N, E \rangle$, with tags, $TAGS$, and colors, $C = \{red, black\}$. So $O = \{A_0, A_1, A_2, A_3\} = \{N, E, TAGS, C\}$. The relations, R, of a view are

- $r_0 : N \times N$, specifies edges between nodes.
- $r_1 : N \times TAGS \times \cdots \times TAGS$, relates nodes to any number of tags.
- $r_2 : E \times TAGS \times \cdots \times TAGS$, relates edges to any number of tags.
- $r_3 : N \times C$, relates nodes to a color.
- $r_4 : E \times C$, relates edges to a color.

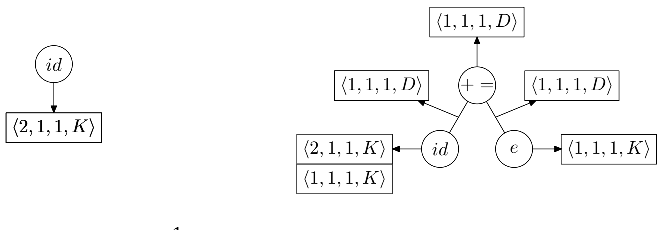
3.3 Events revisited

3.3.1 Creating a redex

Adding objects, $O = \{o_1, \ldots, o_n\}$, to a view, v_i, is only allowed if they form a graph that is identical to $L(r_{i,j})$ for some j. The objects don't have to be new, normally some objects will already be part of the view since the result will be two separate graphs otherwise.

When the objects are created they are assigned a tag to identify the redex they are part of. If the redex was created in view v_i by using rule $r_{i,j}$ and the latest $aid(r_{i,j})$ used was k, then every object of the new redex get the tag $\langle i, j, k+1, or \rangle$. The object role or can be either D or K depending on the rule $r_{i,j}$. All objects in O are colored red. See figure 4 for an example.

If an object in the new redex already exists in the graph then it has to have the object role K. Otherwise it would be possible to delete objects other views have used and expect to exist.



id
$\langle 2,1,1,K \rangle$
$\langle 1,1,1,D \rangle$
$\langle 1,1,1,D \rangle$
$+=$
$\langle 1,1,1,D \rangle$
$\langle 2,1,1,K \rangle$
$\langle 1,1,1,K \rangle$
id
e
$\langle 1,1,1,K \rangle$

Fig. 4. Creating a redex from $r^{-1}_{1,1}$. To the left is the original view. To the right is the resulting view.

3.3.2 Updating the model

As stated above, any number of applications of a transformation, t_i, should always yield the same view, v_i, if the model is unchanged. To accomplish this some restrictions on matching of rules are introduced compared to DPO matching. The reason for the requirement below, and in the next section, that one object has to be red/black is to make sure that identity rules terminate. Other rules than identity rules will not be affected since they have D- or A-objects which cannot have been colored black/red. A match is a morphism, $f : L(r_{i,j}^{-1}) \rightarrow v_i$ such that

- In the redex of the match all objects must have a tag $\langle i, j, k, *\rangle$, where k is a constant integer and $*$ means any value.
- The match must satisfy the DPO gluing condition.
- The label of o and $f(o)$ must be the same.
- The object role of o and $f(o)$ must be the same.
- At least one object in the range of f must be red.

When a valid match is found the rule application is done as in DPO with the addition of adding and modifying tags. When applying a rule, $r_{i,j}^{-1}$, there are two main cases:

(1) if $\exists o, p(p = f(o) \wedge p$ is a ghost$)$ — This case indicates that earlier when matching $r_{i,j}$ on the model, the rule $r_{i,j}$ did not match a redex added by $r_{i,j}^{-1}$ but used combinations of K and A objects from other rules.

The tags of the objects of the redex will be modified in the following way:
- K-objects of the match will be transferred to the model, but the old matching tag $\langle i, j, k, K\rangle$ will be replaced by a new one, $\langle i, j, l, K\rangle$, where l is a new aid.
- D-objects and their tags will be removed from the graph.
- G-objects will have their $\langle i, j, k, G\rangle$-tags replaced by a $\langle i, j, l, A\rangle$-tag, thus reviving them.

Figure 5 shows an example of this case.

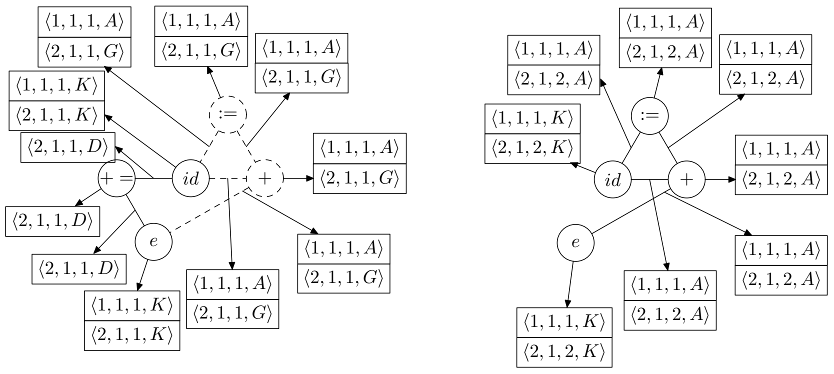


Fig. 5. Updating the model when there are ghosts present. To the left is v_2. To the right is the model, m, after application of $r_{2,1}^{-1}$. Ghosts are drawn with dashed lines.

(2) if $\neg\exists o, p(p = f(o) \wedge p$ is a ghost$)$ — This means that the redex once was created in the view v_i.

The tags of the objects of the redex will be modified in the following way:
- K-objects of the match will be transferred to the model, but the old matching tag $\langle i, j, k, K\rangle$ will be removed and replaced by a new tag $\langle i, j, l, K\rangle$, where l

is a new *aid*.
- D-objects and their tags will be removed from the graph.
- A-objects of $R(r_{i,j}^{-1})$ will be added to the graph, with new tags $\langle i,j,l,A\rangle$.

In both cases all objects $o \in R(r_{i,j}^{-1})$ will be colored black.

3.3.3 Updating the views

A view, v_i, is updated by applying the transformation t_i to the model, m. As in the event of updating the model it is necessary to introduce restrictions on when a rule can be applied. A match is a morphism, $f : L(r_{i,j}) \to m$ such that.

(1) if $\exists p(\forall o(p = f(o) \wedge p$ has a tag $\langle i,j,k,*\rangle))$, where $*$ means any value.
- The match must satisfy the DPO gluing condition.
- The label of o and $f(o)$ must be the same.
- The object role of o and $f(o)$ must be the same.
- At least one object in the range of $f(o)$ must be black.

(2) if $\neg\exists p(\forall o(p = f(o) \wedge p$ has a tag $\langle i,j,k,*\rangle))$, where $*$ means any value.
In this case one additional restriction has to be made compared to case 1 above.
- If the object role of o is A, then there must not be any tag $\langle i,*,*,*\rangle$ associated with $f(o)$.

If this was not the case then a rule could turn any object into a ghost. Even objects that have been used by other rules in the same view, which would yield a different view than expected.

When applying $r_{i,j}$ the following changes will be made to tags in the redex:

- K-objects of the match are transferred to the view v_i, but the old matching tag $\langle i,j,k,K\rangle$ will be replaced by a new tag $\langle i,j,l,K\rangle$, where l is a new *aid*.
- A-objects will be removed if they no other tag than $\langle i,j,k,A\rangle$ associated with them. If an A-object has any other tags, it will be made into a ghost. In case 1 this is done by by replacing the $\langle i,j,k,A\rangle$ tag by a new tag $\langle i,j,l,G\rangle$. In case 2 a new $\langle i,j,l,G\rangle$ tag is added.
- D-objects of $L(r_{i,j}^{-1})$ will be added to the graph with new tags $\langle i,j,l,D\rangle$.

The objects $o \in L(r_{i,j}^{-1})$ will be colored red. If any ghosts were created, they will be colored black. An example of case 1 is shown in figure 6 and an example of case 2 is shown in figure 7.

3.3.4 Destroying a redex

A redex is deleted from a view by removing from the objects of that view all tags of that redex. A redex is identified by the unique combination of *vid*, *rid* and *aid* in

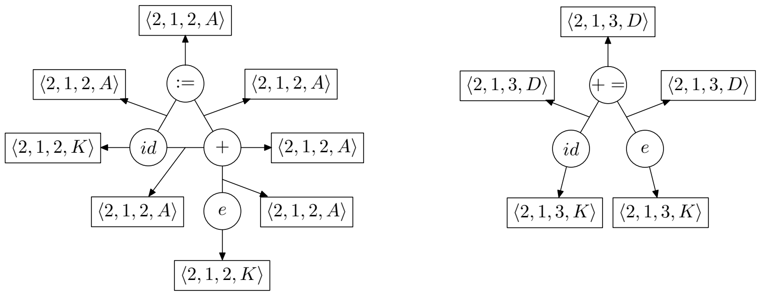


Fig. 6. Updating a view where the match is an earlier rule application. To the left is the model. To the right is v_2 after application of $r_{2,1}$.

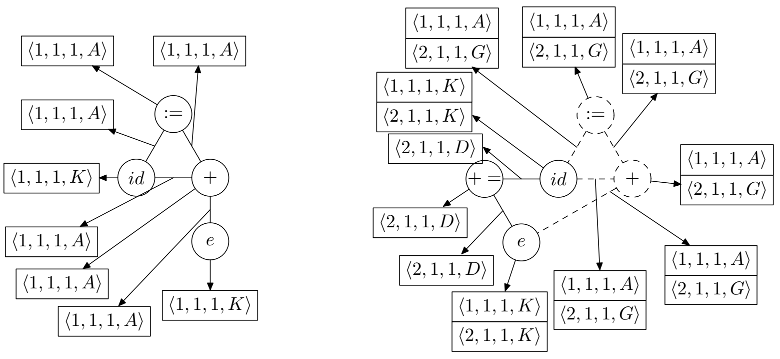


Fig. 7. Updating a view with ghosts. To the left is the model. To the right is v_2 after application of $r_{2,1}$.

a tag. Assume that the redex to be destroyed has $vid = i$, $rid = j$ and $aid = k$. Then find all objects $O = \{o \in v_i | o \text{ has a tag } \langle i, j, k, *\rangle\}$, where $*$ means any object role. Every object of O will have exactly one such tag since every object is only used by the same redex once, and the tags are unique. Remove this tag from the objects. If, when removing a tag from an object, there are no remaining tags associated with that object, then destroy it.

If a single object is to be destroyed then the system has to make sure that there are no incomplete redexes left in the view. So when destroying an object also all redexes it is part of have to be destroyed in the way described above.

4 Discussion

This paper introduced an MVARE system called CODEX. CODEX is based on modified double pushout graph rewriting. The addition of tags and the restrictions on matches make it possible to automatically generate inverse transformations. The property that the inverse can be generated automatically provides a very appealing way to keep the model and the views consistent. It is no longer necessary to construct the inverse by hand in an ad-hoc fashion.

This paper addresses the problem with keeping the platform independent and platform specific models of MDA consistent. If the mappings between the PIM and the PSMs are defined as described in this paper then it is possible to edit any of the different models and still keep them consistent with each other.

If modeling a system where memory usage is critical, the cost of keeping all the tags like CODEX does may be too large. In such cases there has to be a point where the tags are discarded and the transformations will be irreversible.

The approach used here builds on double pushout graph rewriting and because of the nature of DPO there are limitations on what CODEX can do. It can not handle one-to-many or many-to-many mappings in a convenient way because of the identification condition. Where such mappings are wanted it may still be possible to remake the models or to use CODEX to model important one-to-one mappings. Further, the dangling edges condition makes it hard to delete arbitrary constructs of a model because the context it appears in has to be known. Despite these drawbacks we believe MVARE has a lot to offer to MDA in terms of consistency between PIMs and PSMs by making it possible to not only transform models in the forward direction but also allowing edited, transformed models to be transformed back to the original domain without any extra efforts.

References

[1] Architecture Board ORMSC, Model driven architecture (MDA), Tech. Rep. ormsc/2001-07-01, ORMSC, ed. Joaquin Miller, Jishnu Mukerji (July 2001).

[2] A. Henriksson, H. Larsson, A definition of round-trip engineering, technical report, http://www.ida.liu.se/∼henla/papers/roundtrip-engineering.pdf (January 2003).

[3] The Togethersoft tool-suite, http://www.togethersoft.com.

[4] U. A. Nickel, J. P. W. Jörg Niere, A. Zündorf, Roundtrip engineering with FUJABA, extended abstract, http://citeseer.nj.nec.com/383005.html.

[5] D. Bojič, D. Velaševič, URCA approach to scenario-based round-trip engineering, OOPSLA 2000 Workshop on scenario-based round-trip engineering .

[6] H. Knublauch, T. Rose, Round-trip engineering of ontologies for knowledge-based systems, in: Proc. of the Twelth International Conference on Software Engineering and Knowledge Engineering (SEKE), Elsevier Science B.V., 2000, http://citeseer.nj.nec.com/knublauch00roundtrip.html.

[7] H. Ehrig, M. Korff, M. Löwe, Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts, in: Lecture Notes in Computer Science vol. 532, Springer Verlag, 1991, pp. 24–37.

[8] U. Assmann, Automatic roundtrip engineering, Electronic Notes in Theoretical Computer Science Vol. 82, Issue 5 .

A Generic Deep Copy Algorithm for MOF-Based Models

Ivan Porres, Marcus Alanen *

TUCS Turku Centre for Computer Science
Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14A, FIN-20520 Turku, Finland

Abstract

This paper discusses the problem of duplicating a subset of a model based on the Meta Object Facility (MOF), such as a UML model. We propose a new algorithm that works at the metamodel level and deals with the specific semantics of UML models. This algorithm can be used as a building block to implement tools that transform models.

Key words: MOF, Metamodeling, Model Transformation, UML, Deep Copy

1 Introduction

Software models have become a primary artifact in software development thanks to the advent of the Unified Modeling Language (UML) [1] and approaches such as the Model Driven Architecture (MDA) [2] promote the usage of models during the whole development life cycle. In this context, models are created, reviewed and updated constantly. However, in order to be practical, MDA requires the construction of specialized tools to manipulate the models.

In this paper we discuss the problem of the efficient duplication of a subset of a model. This operation is needed while performing many model transformations and refactorings. Its implementation may seem trivial: most object-oriented programming languages have a deep copy operator that can copy arbitrary data structures, including data structures that contain circular references.

* Corresponding author.
Email addresses: `ivan.porres@abo.fi` (Ivan Porres), `marcus.alanen@abo.fi` (Marcus Alanen).

However, a generic deep copy operator will ignore the special semantics of a MOF-based model. In many situations, the standard deep copy operator will copy too much, while the simple or shallow copy operator will copy too little. In the article, we will define a new operator called modelcopy that duplicates a subset of a model and yields an intuitively expected result.

We proceed as follows: Section 2 describes the purpose and requirements of a copy operator for MOF-based models. Section 3 shows the modelcopy algorithm in Python pseudocode, and explains the inner workings of the algorithm. We also show how we meet the requirements stated in Section 2, and discuss the implementation details. We describe an application of the algorithm in Section 4, particularly referring to our System Modeling Workbench (SMW) [3] toolkit. Finally, we conclude in Section 5 by stating the importance of useful and correct tools in the manipulation of models in UML (and other languages). In this context, we believe the modelcopy operator is a basic algorithm that can be used in many tools for model transformation.

1.1 The UML Metamodel

In order to understand an algorithm that transforms a UML model we must understand the internal representation of UML, i.e., the UML metamodel. We will use the model in Figure 1 as an example. This figure shows a simple UML class diagram. It contains two classes, Company and Person, and an association between the classes named work. The association has two ends, named employee and employer. An instance of Company can be associated with many instances of Person while a Person may be associated with just one Company. The class Person also contains two attributes, Name and Address of type String. The diagram does not contain the class String, so we assume that it is defined somewhere else.

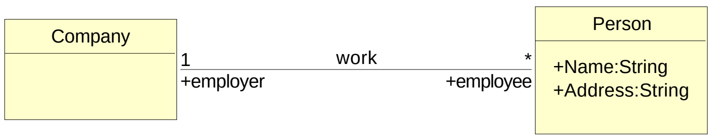


Figure 1. Example Model

All the elements that can appear in a UML diagram, such as Class, Association and Attribute, are defined in the UML metamodel, which is defined using UML itself. Figure 2 shows a simplification of a part of the UML metamodel that describes UML class diagrams. It describes the abstract syntax of a UML class diagram, e.g., what is a UML Class or a UML Association and how they are related. The metamodel contains metaclasses, such as ModelElement and

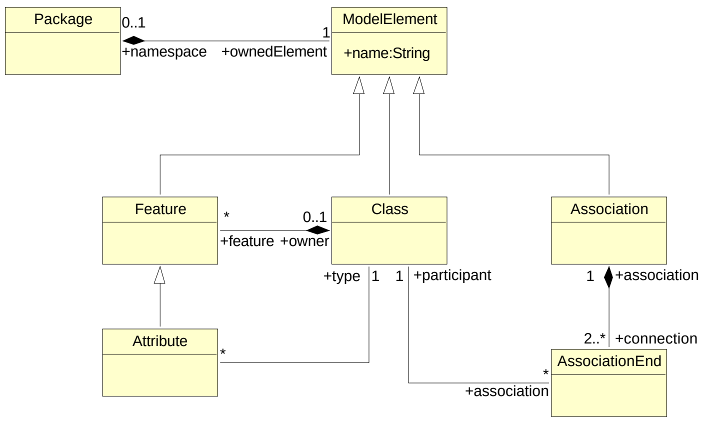


Figure 2. Example Metamodel

generalizations between metaclasses, e.g., Feature is a subclass of ModelElement. Metaclasses may contain attributes, such as name in ModelElement and can be connected using associations, such as the line between Package and ModelElement.

Associations describe how we can connect and compose different model elements. The metamodel shown in Figure 2 shows that a UML Package may own many other model elements, for example classes and associations. Associations in the metamodel are also bidirectional. If a Package owns a ModelElement, the Package is the namespace of the ModelElement. The association between Package and ModelElement is a composition. This is represented by the black diamond next to Package. A composition represents a whole / part relationship. A ModelElement is a part of a Package and a Package owns the ModelElement.

Figure 3 shows how the model described in Fig. 1 fits into the UML metamodel described in Fig. 2. This figure is an object diagram. Each rectangle in the diagram is an instance of a metaclass and each link between two objects represents an instance of a metamodel association. This is also how a UML model is stored into an XMI [4] document or a repository according to the UML standard. This diagram also makes explicit some information from our example: the Name and Address features of Person are linked explicitly with String.

2 The Problem

Once we know how we can represent UML models as a collection of objects and links between objects, we can proceed to explain the copy operation in more detail. As an example, let us assume that we want to copy the model

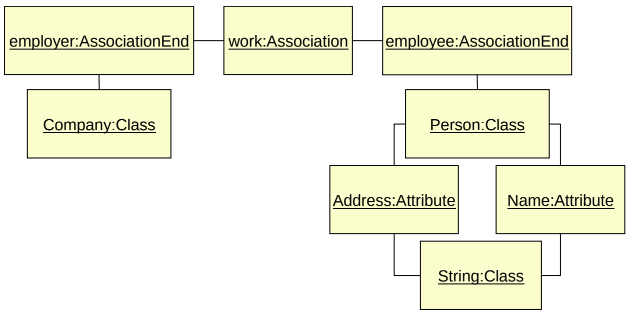


Figure 3. Metamodel Representation of the Example Model

element that represents the class Person.

Most modern programming languages implement several copy operators. Shallow copy usually copies a single object while deep copy copies a whole data structure recursively, e.g., the object passed as a parameter and all other objects that are transitively connected to the object. However, we cannot apply the standard shallow or deepcopy operators as such to copy a UML class in a model. If we use the shallow copy operator, we will obtain a copy of Person, but the copy will not have any features. The Name and Address attributes will not be copied since, by definition, shallow copy only copies one object, and features can belong to at most one class. Also, the original Name and Address attributes cannot and should not be shared between the original and the copy of Person. This is probably not the result expected intuitively. More rigorously, we can argue that we have violated the semantics of the composition association between a class and its features, since we have split a composite from its parts. The solution may seem to use deep copy to copy recursively the class Person and all its features. However, all elements in a model are connected together since all associations are bidirectional. Deep copy will create a copy of Person, but also of the rest of the model.

Figure 4 shows the expected result for copying class Person in our example model. Modelcopy should duplicate the class and its attributes. The new class, named Copy of Person in the figure, is not associated to Company since that would also require a duplication of the association. Therefore not all meta-associations between the model elements are preserved. However, the attributes of the class Copy of Person should have a type. As we can see in the figure, the meta-association between Name and String should be preserved in the copy.

A second example is when we want to copy Person and work simultaneously. Modelcopy should create a copy of both elements. However, Copy of work should be connected to the Copy of Person in one side but keep its original connection to Company in the other side. This example is shown in Fig. 5.

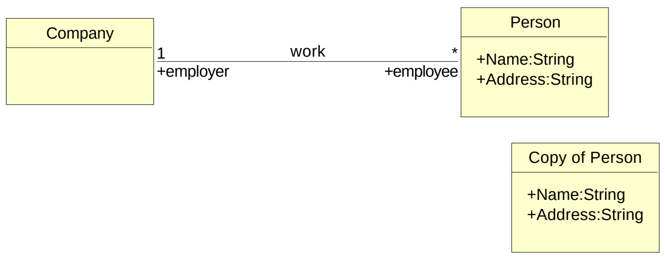


Figure 4. Copy of Person

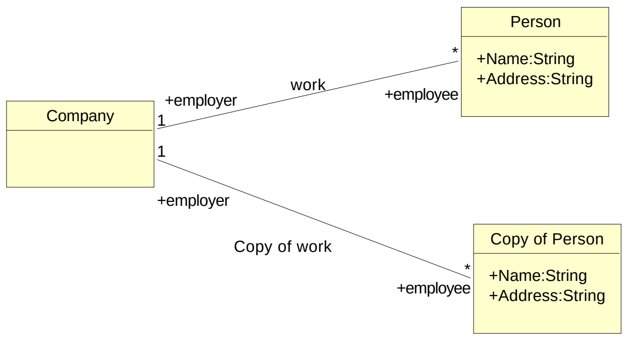


Figure 5. Copy of work and Person

Figure 6 shows our third example where we want to copy Person, Company and work. In this case we should get two new copies of the classes and an association connecting the copies. An alternative outcome of this operation is shown in Fig. 7. We consider this result to be incorrect since it is not consistent with the second example.

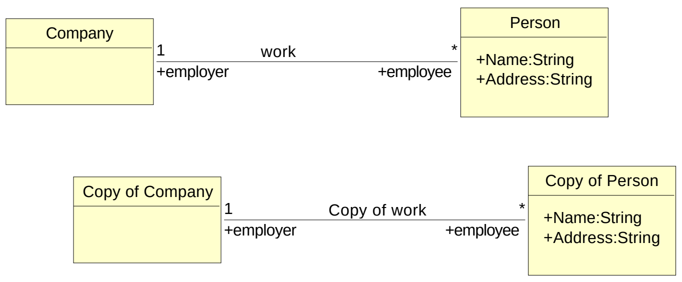


Figure 6. Copy of Company, Person and work

We should note that copying an element in a model is a different action than adding a graphical representation of an element into several diagrams. For example, the class Person could appear again in a different class diagram in the same model, e.g., describing the payroll system in a company. However, in this case, both diagrams represent the same class Person. This is supported explicitly in the UML metamodel (Section 3.5.1 of [1]). Any model element may have many (graphical) representations and a graphical representation may describe

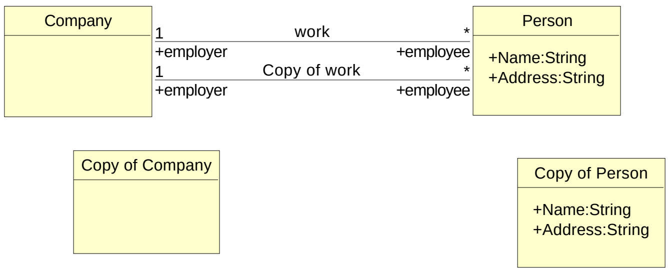


Figure 7. An Alternative Result for the Copy of Company, Person and work

many graphical elements. Adding the class Person into another class diagram does not involve duplicating the class, but creating a new presentation for it.

Due to the observations from the previous examples we are now ready to define the modelcopy operator more rigorously: given an ordered collection of elements C=[c_1,..,c_n] that belong to model M, the modelcopy operator returns a new ordered collection C'=[c_1',..,c_n'] where c_i' belongs to M and is a well-formed and connected copy of c_i.

(1) The elements c_1,..,c_n to be copied should belong to the same model. We say that two elements belong to the same model if they are transitively connected.
(2) All copied elements should be well-formed. This implies:
    (a) If element e is copied all parts of e (elements participating transitively in a composition meta-association of e) should also be copied.
    (b) Copies should preserve the ordering and multiplicity constraints of the original meta-associations.
(3) All copied elements should be connected: Each association between a copied element and an element in the original model should be preserved except when:
    (a) This would break a multiplicity constraint of the association.
    (b) The element in the original model is also being copied. In this case the association should occur between the copied elements.

We can see an example of the application of rules 2.a and rules 2.b in the Copy of Person in Fig. 4. The copy also contains a copy of the attributes Name and Address, and in the same order as the original. It does not have a connection to the work association, which is an application of rule 3.a. We can observe the application of rule 3.b in Fig. 6. If we do not consider this rule we obtain the result shown in Fig. 7.

We should note that a modelcopy algorithm as described above works at the MOF level. It does not guarantee that the copied models will be well-formed at the modeling language level. For example, there is a well-formed rule in UML that says that a region in a Statechart may have only one initial state.

If we use the modelcopy algorithm to duplicate an initial state the model will not be well-formed.

In the next section we will give a detailed algorithm for the modelcopy operator.

3 The Modelcopy Algorithm

In order to describe an algorithm that performs a modelcopy operation in a generic way, we need a mechanism to query the structure of the modeling language, i.e., to query the metamodel used in a given model. We assume that there is a reflection interface that, given a model element, returns its metaclass and the attributes and associations of the metaclass. Examples of such an interface can be found in [3] and [5].

In this article, we assume that the function classOf(e) returns the metaclass of model element e. featuresOf(c) returns a list of all features of the metaclass c. A feature can be an attribute, or an association end. Each feature has an attribute with its name, an attribute named kind that contains one of these values: Attribute, Association or Composition. If the feature is an Association or a Composition, there will be an attribute named multiplicity that contains the multiplicity constraint of the association. Valid values are either one, for exactly zero or one child element, or unlimited, for any amount of child elements. The attributes otherName and otherMultiplicity will describe the name and multiplicity of the other association end.

The modelcopy algorithm in Fig. 8 assumes an important property of the metamodel. If we consider each metaclass as a node in a graph and each composition association a directed arc from the composite to the part, the resulting graph should be a tree. This assumption is implicit in the semantics of MOF, holds for all versions of UML and it is also an implicit requirement in the XMI specification.

In the UML metamodel there are some associations that are marked as ordered. The order of the elements of these associations should be preserved throughout model transformations. We have designed the modelcopy algorithm so that it preserves the ordering of the association ends if the methods that manipulate the association ends also preserve the order.

The following clarifications of the pseudocode might be in order. If c is a collection, the command for o in c binds o successively to each element in c. Also, c.append(o) adds a new element to the collection. If the collection is ordered, the element is placed at the end of the collection. If d is a dictionary, d[k]=o adds a new object o with the key k to the dictionary, deleting any previous value of d[k]. The expression k in d returns true if the key k is present in the dictionary. The expression d[k] returns the current mapping of k, and d.getValue(k,default) returns the value d[k] if it is defined, otherwise it returns default. The statement get(o, f) returns a collection with the values of field f of object o, if that field does not have a multiplicity constraint, and otherwise it returns the single value (object) of field f in object o, or None, meaning there is no object in field f. The function setValueNoUpdate(o, f, v) is used to set the value v of field f of an object o, without updating the other end, i.e., without updating the object v. The function insertValueNoUpdate(o, f, v) is used to append the value v of field f of an object o, also without updating v.

The algorithm works in two passes, modelcopy1 and modelcopy2. In the first pass, it creates a new instance of each element where needed, and in the second pass, it sets each new element's associations correctly.

The function modelcopy1 creates a mapping between an element in the original model to a newly-created element in the copied1 dictionary. The created elements have default values without any connections to each other. We assume the ordinary UML containment policy, that compositions and attributes are owned by exactly one element, and that all elements except the root of the tree are owned by an element. Thus, modelcopy1 is guaranteed to create all objects in the subtrees, and only those objects.

The mapping is created by visiting each element in C transitively under the composition or attribute association (lines 4–8), and creating a new instance of each element encountered (line 3). This satisfies our requirement that if an element is copied, all parts of it are copied. Termination is guaranteed by the copied1 dictionary and the test on line 2 in modelcopy1.

The function modelcopy2 sets the connections between elements. This is accomplished by going trough the dictionary of newly created objects and setting their connections to point to the same elements as the original element's connections point to, or to new elements in the mapping copied1 if they exist.

The most important part of this pass is to not update the other end of a bidirectional association automatically. The reason for this is rather subtle. The appending of new associations must be done in the same order as they were defined, separately at both ends. Otherwise, the other automatically updated end would have its elements in the wrong order. As an example, take elements $e_1, \ldots, e_n$ and their corresponding ordered features $f_1, \ldots, f_n$. Say $f_1 = [e_2, e_3]$ and $f_2 = [e_3, e_1]$. Copying the elements creates new, initially empty, features f'_i; appending e'_2 to f'_1 yields $f'_1 = [e'_2]$ correctly, but f'_2 should not be modified, otherwise it would become $f'_2 = [e'_1]$. Instead, f'_1 is built separately by appending elements to only f'_1. Then, f'_2 can be built by first

```
1 def modelcopy(Collection c): Collection
2     copied1, copied2 = new Dictionary, new Dictionary
3     result = new Collection
4     for e in c:
5          result.append(modelcopy1(e, copied1))
6     for e in copied1:
7          modelcopy2(e, copied1, copied2)
8     return result

1 def modelcopy1(Element e, Dictionary copied1): Element
2     if e not in copied1:
3          copied1[e] = new getClassOf(e)
4          for f in getFeaturesOf(classOf(e)):
5              if f.kind != Association:
6                  for value in get(e, f.name):
7                      modelcopy1(value, copied1)
8     return copied1[e]

1 def modelcopy2(Element old, Dictionary copied1, Dictionary copied2): Element
2     if old not in copied2:
3          copied2[old] = true
4          newE = copied1[old]
5          for f in getFeaturesOf(class(old)):
6              if f.kind == Association:
7                  if f.multiplicity == ZeroOrOne:
8                      v = get(old, f.name)
9                      if f.otherMultiplicity == ZeroOrOne and not v in copied1:
10                         setValueNoUpdate(newE, f.name, None)
11                     else:
12                         setValueNoUpdate(newE, f.name,
                                            copied1.getValue(v, v)
13                         if v and not v in copied1:
14                             insertValueNoUpdate(v, f.otherName, newE)
15                 else:
16                     for v in get(old, f.name):
17                         if f.otherMultiplicity == ZeroOrOne and not v in copied1:
18                             skip
19                         else:
20                             insertValueNoUpdate(newE, f.name,
                                                   copied1.getValue(v, v)
21                             if v and not v in copied1:
22                                 insertValueNoUpdate(v, f.otherName, newE)
23             else:
24                 # It is an attribute or a composition
25                 if f.multiplicity == ZeroOrOne:
26                     value = get(old, f.name)
27                     setValueNoUpdate(newE, f.name,
                                        modelcopy2(value, copied1, copied2))
28                 else:
29                     for value in get(old, f.name):
30                         insertValueNoUpdate(newObj, f.name,
                                              modelcopy2(value, copied1, copied2))
31    return copied1[old]
```

Figure 8. Modelcopy Algorithm Pseudocode

appending e'_3 and afterwards e'_1.

Thus, we update only one end of an association, at lines 12, 20, 27 and 30, for associations and compositions alike. This is no problem for copied elements that reference other copied elements since we are guaranteed to visit both ends of an association anyway, by lines 24–30.

An association with a multiplicity constraint of exactly one cannot connect to the original and the copied element. In this case all we can do is skip this connection for the copied element. This is done at lines 9–10 and 17–18. How big a problem this is depends highly on the metamodel. When associations between a non-copied and copied element are valid, copied elements must immediately update the associations of non-copied elements, since we will never visit the non-copied elements in modelcopy2. This is done at lines 13–14 and 21–22. An additional small caveat is that non-copied elements have their new associations in a non-deterministic order. This can be fixed by yet a third pass, which is not included in this paper.

As such, the operation performed is a copy of elements reachable from the set of elements under the attribute and composition relations, with all associations between the copied elements set, and associations between copied and non-copied elements set, multiplicity constraints permitting. In the worst case, the algorithm terminates after copying all elements in the model. This results in an exact copy of the original model, in which the original and copy do not share any associations between each other.

3.1 Implementation

The modelcopy algorithm has been implemented in the SMW toolkit. It is a collection of tools implemented in Python for manipulating software models. The SMW toolkit is divided into several components: the kernel, a networked repository, the model transformation framework and the graphical modeler. The SMW kernel implements the reflection interface needed to implement the modelcopy algorithm as described above. It supports UML 1.1 to 1.4 and user-defined modeling languages. The modelcopy algorithm has been implemented in the SMW kernel and therefore is available in all SMW applications and modeling languages.

4 Example: Template Instantiation

An application of the modelcopy operator is template instantiation (Chapter 2 of [6], [7]). A template is a (part of a) model that we want to reuse in other models. Template instantiation is the introduction of a template in a model. Each element in the template should be copied into the current diagram. Optionally, it is possible to rename some of the elements in the template based on one or more template parameters. Figure 9 shows a simple algorithm for template instantiation. This algorithm accepts a parameter that is used to rename the copied elements. This step is performed in lines 5–6 of the algorithm.

We show an example of this mechanism in Figure 10. The leftmost side of the figure shows a template for the Subject-Observer pattern [8], while the right side shows an instance of the template where the <<name>> parameter has been replaced with the string "Temperature".

```
def instantiate(template,newName,diagram):
    new=modelcopy(template)
    for e in new:
        diagram.addPresentation(e)
        if e.name:
           e.name=string.replace(e.name,"<<name>>",newName)
```

Figure 9. Template Instantiation Algorithm

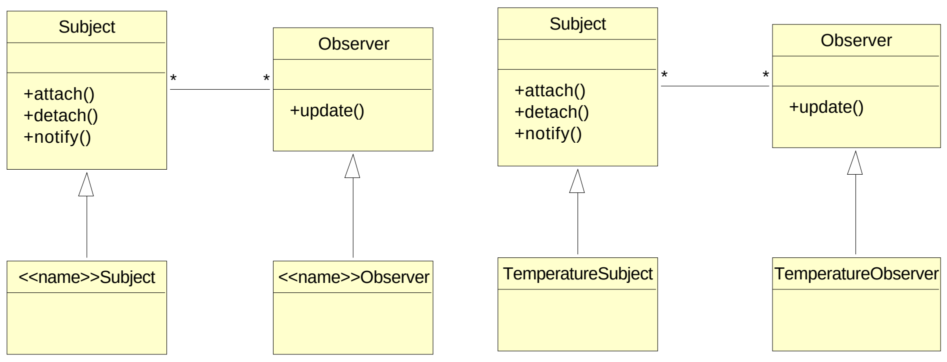


Figure 10. Template Instantiation Example

Another obvious application of the modelcopy algorithm is to implement the cut, copy and paste commands in a UML model editor, of which an example has been seen in Fig. 5. Due to space constraints, a more thorough description is given in [9].

5 Conclusions

This article presents an algorithm that duplicates subsets of models described using UML or any other MOF-based modeling language. Although the problem may seem trivial, we have shown that its solution is rather complex and it requires an extensive knowledge of the MOF standard. The algorithm requires two phases and two temporary data structures while standard deep copy algorithms need only one phase and one temporary data structure.

We like to think that this algorithm is a useful building block to construct advanced tools that can manipulate any UML and MOF-based model. The MDA approach can only be realized if we are able to construct sophisticated tools that can manipulate models in a generic way. MDA tools should support many different profiles or extensions to the original modeling languages. They should also be able to transform models from one profile to another.

We consider that a generic modelcopy operator is necessary in many of these transformations.

Many researches have studied the properties of the UML language, but the construction of tools to manipulate UML models is seldom discussed. The UML standard itself does not discuss the construction or design of UML tools. Other implementation standards and libraries, such as the Java Metadata Interface [5], are targeted to tool builders but they do not discuss the need of a copy operator. This fact is surprising since we consider this a common operation in model manipulation and refactoring.

The SMW Toolkit, including the implementation of the modelcopy operator and its application in a UML model editor is available under an open source license at `http://www.abo.fi/~iporres/`.

References

[1] OMG, OMG Unified Modeling Language Specification, version 1.4, September 2001, available at `www.omg.org`.

[2] OMG, OMG Model Driven Architecture, OMG Document ormsc/2001-07-01. Available at `www.omg.org` (July 2001).

[3] I. Porres, A Toolkit for Manipulating UML Models, Tech. Rep. 441, Turku Centre for Computer Science, available at `www.tucs.fi` (January 2002).

[4] OMG, OMG XML metadata interchange (XMI) specification, oMG Document formal/00-11-02. Available at `www.omg.org`.

[5] S. Iyengar, et al., Java metadata interface (JMI) specification, available at `java.sun.com`.

[6] Adaptive, D. Process, P. Technology, Softlab, Siemens, Unambiguous UML (2U) 3rd Revised Submission to UML 2 Infrastructure RFP, OMG document ad/2003-01-08 (January 2003).

[7] T. Clark, A. Evans, S. Kent, A metamodel for package extension with renaming, in: UML 2002 - The Unified Modeling Language. Model Engineering, Concepts, and Tools. Fifth International Conference, Dresden, Germany, Vol. 2460 of LNCS, Springer, 2002.

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison Wesley, 1995.

[9] I. Porres, M. Alanen, A Generic Deep Copy Algorithm for MOF-Based Models, Tech. Rep. 486, Turku Centre for Computer Science (2002).

Supporting Model-to-Model Transformations: The VMT Approach[1]

Shane Sendall[†], Gilles Perrouin[❖], Nicolas Guelfi[❖], Olivier Biberstein[+]

[†]Swiss Federal Institute of Technology in Lausanne
Software Engineering Laboratory
CH-1015 Lausanne EPFL, Switzerland[2]
sendall@acm.org

[❖]Luxembourg University of Applied Sciences
SE2C-Software Engineering Competence Center
6, Rue Coudenhove-Kalergi
L-1359 Luxembourg-Kirchberg, Luxembourg
{gilles.perrouin, nicolas.guelfi}@ist.lu

[+]Berner Fachhochschule - University of Applied Sciences
Biel School of Engineering and Architecture
Quellgasse 21, CH-2501 Biel, Switzerland
Olivier.Biberstein@hta-bi.bfh.ch

Abstract

The model-driven architecture approach (MDA) promotes software development as driven by a thorough modeling phase where application code is automatically generated from a platform specific UML model (PSM). The idea is that the PSM is itself derived from a platform independent UML model (PIM). Such code generation and model derivation are examples of model-to-model transformations.

This paper presents the Visual Model Transformation (VMT) approach, which includes a transformation language and a tool to support UML model transformations. The transformation language is a visual declarative language that supports the specification, composition and reuse of model transformation rules. These rules make use of the OCL language and a visual notation to indicate the selection, creation, modification and removal of model elements. An abstract denotational semantics based on graph transformation is sketched for the VMT language. We also present the MEDAL tool, which is a prototype build on top of IBM/Rational XDE development environment, and is a first step towards tool support for the VMT approach.

1 Introduction

More so than ever, software is hard to construct and maintain. One of the main factors that make it difficult is the size and complexity of the problem to be addressed. Abstraction and Separation of Concerns offer some of the best tools to combat problem size and complexity in software design. In particular, the use of software models has become a popular way to harness the principles of abstraction and separation of concerns.

In this context, the Unified Modeling Language (UML) [Omg03a], which became an Object Management Group (OMG) standard in 1997, is used by the majority of software modeling techniques and approaches. UML can support many different kinds of abstractions and separation of concerns. UML is a rich language that can be used to develop a set of inter-related models. The number and complexity of such models can vary depending on the abstraction level and kind of view taken, where the overall model is defined as the composition of these models.

There are a number of model engineering techniques that can be applied during software development, which include: refinement and derivation of models toward a software realization,

[1] This work is supported by the Luxembourg Ministry of Higher Education and Research under the Title I project n°MEN/IST/01/001

[2] From August 2003, Shane's address will be: Software Modeling and Verification Lab., University of Geneva, CH-1211 Geneva 4, Switzerland.

reverse engineering models to a higher level of abstraction, generation of models that act as views of existing models, and the synchronization of models.

These techniques are examples of *Model Transformations*. A model transformation involves taking one or more models as input and producing one or more models as output according to a set of rules specific to the purpose in hand. Performing these model transformations by hand can be quite a cumbersome and error-prone task. Ideally, such tasks should be automated in order to improve developer productivity and reduce human error.

Our work on model transformation is performed in the context of the FIDJI project [FIDJI] of the Luxembourg University of Applied Sciences in collaboration with the Swiss Federal Institute of Technology in Lausanne and the University of Applied Sciences in Biel, Switzerland.

The FIDJI team has experimented with concrete model transformations while developing architectural frameworks [GS02, GR03]. Theoretical issues addressed by the FIDJI project [BG00, DiM99, Gue01] and more practical needs of our architectural frameworks led the team to define a systematic approach to perform generic model transformations. Tool support for this approach was prototyped in a tool called MEDAL (uMl gEneric moDel trAnsformer tooL), which is an add-in to IBM/Rational's XDE UML modeling tool [XDE].

Owing to our experiences with MEDAL and our wider vision for model-based software development, we are working on an approach that is capable of transforming any set of UML models to any set of UML models. Our goal is to provide an approach that offers a means to specify and execute any UML-to-UML model transformation that could be useful during the model engineering activities of software development.

Our proposal for achieving this objective is called the Visual Model Transformation (VMT) approach. It offers a visual and declarative language for specifying UML model transformations.

This paper is organized the following way: Section 2 describes the first prototype developed and highlights the lessons learned; it sets the stage for the description of the VMT approach, which is given in Section 3. A formalization of the VMT approach using Graph Transformation Theory is proposed in Section 4. Section 5 presents related work in the field of model transformations. Finally, Section 6 concludes and presents future work.

2 MEDAL – A First Cut at Tool Support for Model Transformation

2.1 Context

Our interest in model transformation began with our work developing architectural frameworks using IBM/Rational's XDE UML modeling tool. The framework, which we call JAFAR [GR03], makes use of transformations to refine high-level models into J2EE artifacts. These transformations are defined in XDE and executed by its pattern engine. In XDE, a transformation is treated as a UML collaboration (also called "Model Template"), which has a set of template parameters. Using a scriptlet language, it is possible to perform actions on these parameters. It is also possible to define pre and post conditions to model template applications. For more sophisticated transformations, one can attach a Java program that realizes the actual transformation (referred to as a "callout").

XDE 1.5, which will be released officially in July 2003, will add support for OCL and UML profiles. These two features are particularly relevant to model transformations. OCL is tailored to operate with UML and one of its purposes (together with the specification of model constraints) is the navigation through UML models and selection of subsets of them. UML profiles [Omg03b] are extension mechanisms that allow the specialization of the UML metamodel to fit particular business domains or platforms. Using an UML profile, one can provide to the modeler a set of predefined model elements via stereotypes and tagged-values. Together with a set of rules, UML profiles can be employed to define mappings into particular platforms (J2EE, CORBA…). So far, many profiles have been defined [Omg03b], which each target a specific platform or domain.

Kozaczynski and Thario showed in [KT02] how to make use of OCL and UML profiles in XDE to automatically transform RUP user-experience models into artifacts for the Struts framework [Struts03]. Starting from that experiment, we investigated how XDE 1.5 could help one to specify and perform more generic model transformations. The result of this initial work was a first prototype, which we named MEDAL [GS03] (uMl gEneric moDel trAnsformer tooL).

The objectives of this prototyping phase were to define a Domain Specific Visual Language (DSVL) dedicated to UML Model Transformations, which provides a means to define and compose model transformations in UML.

2.2 MEDAL Features

We designed the first release of the tool to act as a visual “wrapper” to the existing XDE’s transformation mechanism. One can distinguish two kinds of tasks for specifying a transformation process: the definition of individual transformations and the definition of the sequencing between transformations.

To define a single transformation, we use a class diagram in which a model template is seen as a package and parameters are shown as stereotyped OCL notes. To enhance the expressiveness of notes we have defined a small extension to OCL called MxOCL (MEDAL’s extended OCL) allowing user aliases to be entered inside OCL expressions. The same UML notes mechanism is employed to specify the source context of the transformation and the target location.

To compose transformations, we use UML activity diagrams. For this purpose, an activity in an activity diagram corresponds to a transformation and a transition between activities corresponds to a sequential control flow between transformations. In particular, we used a subset of the control flow constructs offered by UML activity diagrams (i.e., decisions, transitions, guard conditions) as a means to compose transformations. The combination of these diagrams forms the MEDAL transformation language. A sample overview of the different views of this language can be found in Figure 1. This figure shows a simple transformation that adds accessor and mutator methods to all classes that are contained in a given package. These methods are added for all the attributes on the classes, and the attributes are made private, if they are not already.

The meta-model of the MEDAL language as well as its well-formedness rules (expressed as OCL constraints) are described in an UML profile. This profile is evaluated by the tool to check diagrams and any errors are reported by the tool to the user. This language is out of the scope of this paper. Interested readers are referred to [GS03, SRP03].

2.3 Lessons Learned — Moving Towards VMT

Activity diagrams (“Transformation Sequencing Rules” see Figure 1) have revealed to be a good representation for composing transformations. However, notations for loops and other more complex control structures have to be studied further. Concerning the use of “Model Template Application Definition” diagrams (see Figure 1), the main issue is to cope with a huge number of parameters or accesses to elements. As the number of MxOCL notes and parameters can be high, using notes to specify a great number of parameters can become cumbersome. We must also consider the point that in case of obfuscated access to some model elements, the MxOCL expression included in a note can be very hard to read.

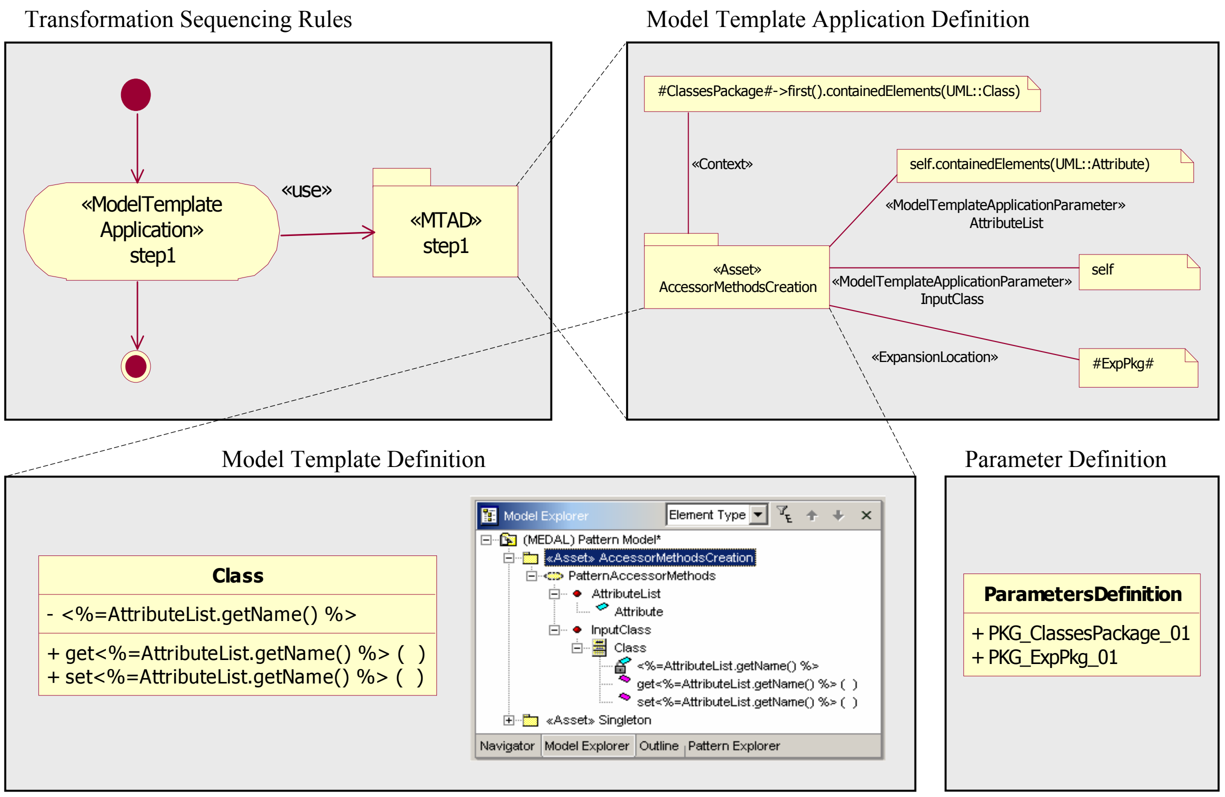

Transformation Sequencing Rules
«ModelTemplate Application» step1
«use»
«MTAD» step1
Model Template Application Definition
#ClassesPackage#->first().containedElements(UML::Class)
«Context»
self.containedElements(UML::Attribute)
«ModelTemplateApplicationParameter» AttributeList
«Asset» AccessorMethodsCreation
«ModelTemplateApplicationParameter» InputClass
self
«ExpansionLocation»
#ExpPkg#
Model Template Definition
Class
- <%=AttributeList.getName() %>
+ get<%=AttributeList.getName() %> ( )
+ set<%=AttributeList.getName() %> ( )
Model Explorer
Element Type
(MEDAL) Pattern Model*
«Asset» AccessorMethodsCreation
PatternAccessorMethods
AttributeList
Attribute
InputClass
Class
<%=AttributeList.getName() %>
get<%=AttributeList.getName() %> ( )
set<%=AttributeList.getName() %> ( )
«Asset» Singleton
Navigator
Model Explorer
Outline
Pattern Explorer
Parameter Definition
ParametersDefinition
+ PKG_ClassesPackage_01
+ PKG_ExpPkg_01

Figure 1- MEDAL Language Overview

3 The VMT Approach

In this section, we present the VMT approach, which includes our proposal for a UML model transformation language. We give an overview of VMT and describe the various components of the transformation language.

3.1 Approach Overview and Background

The VMT approach proposes a principally visual language for describing transformations between models specified with UML. In the language, a transformation is defined in terms of a set of *transformation rules*. Each transformation rule defines the way that one or more target UML diagram elements are created, changed, and/or deleted as a function of zero or more source diagram elements. Intuitively, this process can be seen as a mapping from source to target diagram elements, where target elements may be created in the process. A transformation rule is described by a *rule specification*. A rule specification consists of two parts: a *matching schema* and a *result schema*. The matching and result schemas are inspired from the work on graph transformation approaches (see Section 4).

The matching schema of a rule specification defines the condition under which the rule has permission to fire, the input arguments for the transformation, and those input arguments that will be deleted with the execution of the rule. A matching schema is represented as a graph, which has two roles: 1) it is used to define the condition that must be fulfilled for the rule to be permitted to fire, and 2) it is used to define the binding relation between source model elements and input arguments of the transformation rule. Intuitively, nodes in the matching graph can be seen as placeholders for elements in the source model. It is also possible to have a node that is a placeholder for a set of elements. Equally, it is possible to define a prohibited element, which is represented also as a node. This concept is used to strengthen the firing condition for the rule. In particular, if a match is found for a prohibited node, then the rule no longer has permission to fire.

A result schema defines the target diagram(s) as a function of the elements bound by the matching schema. A result schema is also represented as a graph. Intuitively, a node in the result graph represents an element of a target diagram. In particular, a node can represent a newly created element, a modified element from the source model, or an unmodified element from the source model. Like for the matching schema, it is also possible to have a node that represents a set of target model elements.

To illustrate the role of the matching and result schemas in a rule specification, suppose that we would like to transform an association class with certain multiplicities, and between two classes, into a class with binary associations (with the corresponding multiplicities). Figure 2 illustrates the transformation. It represents the transformation in terms of the before (left) and after (right) state, in the same vein that we would describe the matching schema and result schema, respectively, of the rule specification for this transformation.

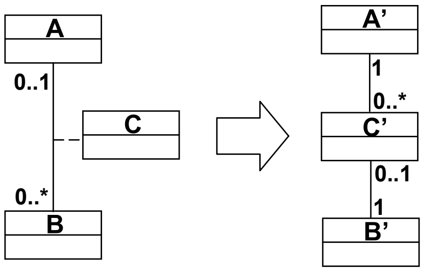


Figure 2 – Graphical Depiction of an Example Transformation

A model transformation can modify existing models or it can create new models (or a combination of both); we refer to these two categories of transformation as modification and creation transformations, respectively. A modification transformation involves changing an already existing model, or set of models, through the addition, modification and removal of elements. For example, a modification transformation could define the application of a design idiom to a UML class diagram. In this case, the target diagram, i.e., the output class diagram, is a slightly modified version of the source diagram. The example cited in Figure 2 is also an example of a modification transformation. A creation transformation involves creating only new elements as a result of the transformation. For example, a creation transformation could define the creation of a (randomly generated) end-to-end scenario, represented by a UML sequence diagram, from a UML state diagram. In this case, the target diagram, i.e., the resulting sequence diagram, only includes the new elements, which can be traced back to elements in the state diagram.

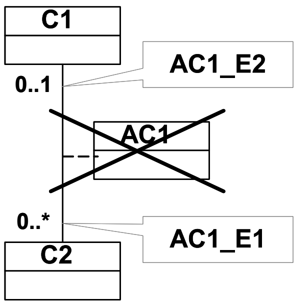
When defining a transformation, it can sometimes be easier to formulate the description in terms of a set of simpler transformations instead of a single more complex transformation. In our approach, this technique is made possible by defining each of the simpler transformations as a distinct rule and defining the required ordering of rule application using a separate description, which we call a *rule ordering schema*. We find it useful to have these two levels of description as we have observed that it promotes scalability and reuse, and it promotes a simpler rule specification language, due to the separation of concerns. In addition to rule sequencing, a rule ordering schema allows one to express rule iteration and conditional branching of rules. Furthermore, it allows one to map outputs of a rule to inputs of another rule. A rule ordering schema may also refer to other transformation definitions. Thus it is possible to have a composition hierarchy. This feature promotes transformation reuse and scalability.

Summarizing the execution process of a transformation description: upon the firing of a transformation, one or more transformation rules are applied to the one or more source models, according to the control flow defined by the rule ordering schema.

Even though the graphical language that our approach proposes offers a widely applicable and useful set of abstractions for specifying transformations, we believe that some of the complex algorithms required for model transformations in general are easier defined in a procedural language, such as, Java, C#, etc. As such, we propose a means to integrate our language with general-purpose programming languages. In particular, we chose the Java object-oriented programming language [SGB00] as the target language. The idea is that our graphical language would be used to describe the transformation rules— matching and result schemas—and the orderings of the transformation rules—rule ordering schema, and the Java programming language would be used to define those parts of the transformation that are easier expressed with it, e.g., complex transformation algorithms. We propose integration at the programming language level by generating code for the chosen programming language from the language of rule specifications and rule ordering schemas. In this way, we would allow users the full expressive power of a general-purpose programming language, yet a set of abstractions that are fine-tuned to transformations. We believe that this compromise offers quite some potential because it leaves the door open for the user to work in a way that is most comfortable to him/her. Due to space considerations, we do not go into any further details of this aspect of the approach.

3.2 Example

To illustrate a transformation described using the VMT approach, we revisit the example introduced in Section 3.1. This transformation involves taking an association class and replacing it by a class with two associations to the previously connected classes. Since this is a simple example, we will define the transformation using only a single rule specification, and we will omit the rule ordering schema (since there is only a single rule).



id: C1
type: Class
name: *any*
association: Set{AC1_E2}
constraint: *any*
feature: *any*
generalization: *any*
isActive: false (default)
ownedElement: *any*
specialization: *any*

additional selection constraints:
-

id: C2
type: Class
name: *any*
association: Set{AC1_E1}
constraint: *any*
feature: *any*
generalization: *any*
isActive: false (default)
ownedElement: *any*
specialization: *any*

additional selection constraints:
-

id: AC1_E2
type: AssociationEnd
name: *any*
aggregation: none (default)
association: AC1
changeability: *any*
isNavigable: *any*
multiplicity: 0..1
ordering: unordered (default)
participant: C1
qualifier: empty (default)
targetScope: instance (default)
visibility: public (default)

additional selection constraints:
-

id: AC1 -> REMOVED
type: AssociationClass
name: *any*
association: *any*
connection: Set{AC1_E1, AC1_E2}
constraint: *any*
feature: *any*
generalization: *any*
isActive: false (default)
ownedElement: *any*
specialization: *any*

additional selection constraints:
-

id: AC1_E1
type: AssociationEnd
name: *any*
aggregation: none (default)
association: AC1
changeability: *any*
isNavigable: *any*
multiplicity: 0..*
ordering: unordered (default)
participant: C2
qualifier: empty (default)
targetScope: instance (default)
visibility: public (default)

additional selection constraints:
-

Figure 3 – Graphical Depiction of the Matching Schema for the Association Class Transformation

Figure 3 shows the matching schema for the transformation. The UML-like class diagram in Figure 3 depicts the matching graph in model-level view mode. We also offer another visual form for the graph (not shown here): metamodel-level view mode, in which case the graph is shown in a notation neutral manner—all the different kinds of UML model elements are shown as objects in a UML-like object diagram. Each of the 5 boxes shown in Figure 3, which we refer to as a properties-constraint box, provides a set of conditions that must be observed in the selection of a source model element. The correspondence between the boxes and the diagram is made by the id property value of the box and the label of the element in the diagram. Note that we envisage that the tool would not show all boxes at the same time. Instead, the tool would provide a window pane for the diagram and one for a single properties-constraint box. The user could then select the appropriate properties-constraint box by selecting the corresponding element in the diagram.

Figure 3 shows two classes, labeled C1 and C2, and an association class, labeled AC1, which connects these two classes. Two association ends, labeled AC1_E1 and AC1_E2 are labeled using "callout" boxes. This notation is used because association ends do not have a corresponding graphical representation in UML. Also, the association class is graphically crossed out, which means that the element will be removed with the execution of the transformation.

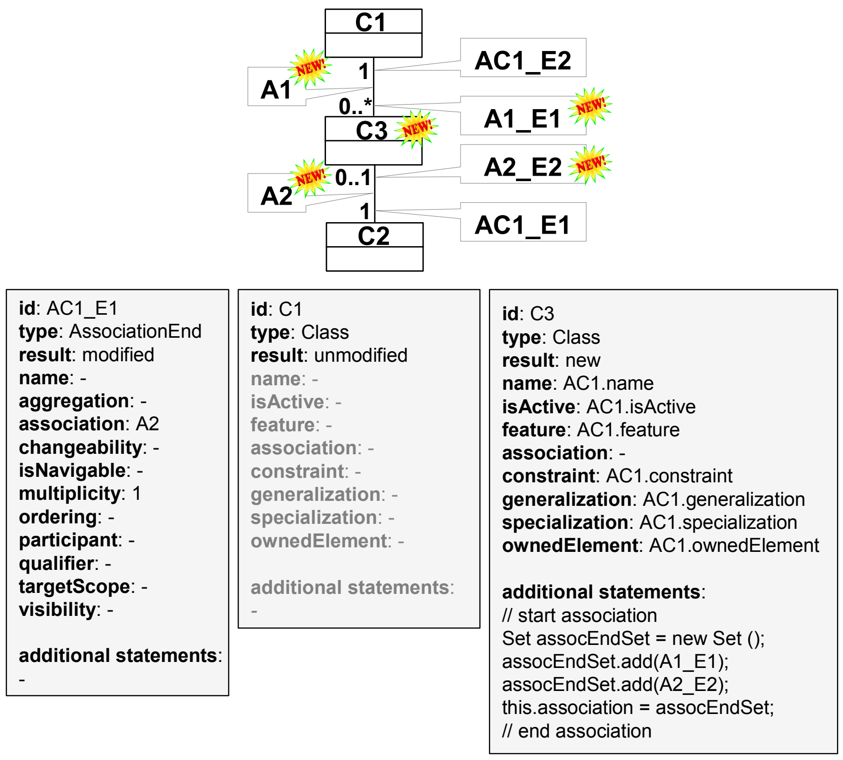
Looking closer at each properties-constraint box, the id property provides a name that can be used to refer to the bound element. The other properties shown below id are properties of the metaclass of the element; as such, this list is specific for each different kind of metaclass of the element. The any condition simply indicates that the corresponding property is unrestricted, i.e., a possible candidate element may have any value for this property. There is also a space for additional selection constraints. This clause provides an additional constraint, written in OCL, that must be observed for a valid match.

The meaning of the matching schema depicted in Figure 3 is the following: The transformation will try to bind all elements of the matching graph to elements in the source model. This will only occur if the source model has two classes with an association class connecting them, which has the multiplicities 0..1 and 0..* at either end. Each properties-constraint box makes some additional restrictions on binding, e.g., elements that bind to either C1 or C2 must have the isActive property equal to false, elements that bind to either AC1_E1 or AC1_E2 must be unordered and not an aggregation, etc. If all constraints of the diagram and properties-constraint boxes are satisfied then a binding is made and the transformation rule will fire.

Figure 4 shows the result schema for the transformation. The three boxes that surround the UML-like class diagram depict the properties-definition boxes for three of the nine element definitions implied by the diagram. Note that the other boxes have been omitted for space considerations. Figure 4 shows three classes, labeled C1, C2 and C3, two associations, labeled A1 and A2, and four association ends, labeled AC1_E1, AC1_E2, A1_E1 and A2_E2. In addition, five "new" labels are shown. These labels signify that the corresponding element definition will result in a new element (as opposed to a modified or unmodified existing one).

Looking closer at each properties-definition box, the element definition with id C1 states that the element bound to C1 (a class) by the matching schema will be left unchanged. The element definition with id AC1_E1 states that the bound association end will be modified in two ways: its multiplicity will be set to 1 and its related association property will be set to the association denoted by A2. Finally, the element definition with id C3 states that a new class will be created that has mostly the same properties of the association class that was bound to id AC1, with one exception: it will have the association ends of A1_E1 and A2_E2.

The meaning of the result schema depicted in Figure 4 is the following: if the rule has permission to fire (according to matching schema), it will create one class (C3), two associations (A1 and A2) and two association ends (A1_E1 and A2_E2), and modify two association ends (AC1_E1 and AC1_E2), according to the statements given in their corresponding properties-definition boxes. And, the transformation will leave untouched two classes (C1 and C2).



id: AC1_E1
type: AssociationEnd
result: modified
name: -
aggregation: -
association: A2
changeability: -
isNavigable: -
multiplicity: 1
ordering: -
participant: -
qualifier: -
targetScope: -
visibility: -

additional statements:
-

id: C1
type: Class
result: unmodified
name: -
isActive: -
feature: -
association: -
constraint: -
generalization: -
specialization: -
ownedElement: -

additional statements:
-

id: C3
type: Class
result: new
name: AC1.name
isActive: AC1.isActive
feature: AC1.feature
association: -
constraint: AC1.constraint
generalization: AC1.generalization
specialization: AC1.specialization
ownedElement: AC1.ownedElement

additional statements:
```
// start association
Set assocEndSet = new Set ();
assocEndSet.add(A1_E1);
assocEndSet.add(A2_E2);
this.association = assocEndSet;
// end association
```

Figure 4– Graphical Depiction of the Result Schema for the Association Class Transformation

4 Formal Aspects of Model Transformation

As mentioned in the above section, our approach considers a model, roughly speaking, as a graph. Thus, the theoretical foundation of model transformation in VMT can be easily expressed in terms of graph transformations that have been studied in [Roz97]. This section presents briefly such aspects adapted to the VMT approach. First of all, we give some basic definitions related to the notion of graph transformation as the application of elementary transformation rules and then we discuss how such transformation rules are used in VMT. Finally, we present the correspondence between the notion of model considered in VMT and the notion of graph presented below.

4.1 Graph Transformation

A ***labeled graph*** *G=(N,E,l)* consists of a finite set of nodes *N*, a finite set of edges *E* such that the elements of *E* are 2-elements subsets of *V* denoted by $E \subseteq [V]^2$, and a mapping *l* assigning a *labeling* symbol to each edge. A graph *K* is a ***subgraph*** of *G*, denoted by $K \subseteq G$, if the node and edge sets of *K* are subset of the respective sets of *G*, and the label mappings of *K* coincide with the one of *G* restricted to *K*. We say that *K* has an ***occurrence*** in *G*, if there is a mapping *o*, which maps the nodes and edges of *K* to the nodes and edges of *G*, respectively, and preserves labelings.

A *graph transformation* consists of applying *transformation rules* to a graph iteratively. Each rule application transforms a graph into another graph by specifying a graph pattern and the elements that are removed and added (a modified element can be removed and added with new values). A ***transformation rule*** *r=(M,R,A,c)* consists of a graph *M=(N,E,l)* called the ***matching schema***, $R=(N_R,E_R,l_R)$ corresponds to the nodes, the edges, and the labelings that are removed ($N_R \subseteq N$, $E_R \subseteq E$), $A=(N_A,E_A,l_A)$ groups the components that are added, and *c* is a condition for the application of the rule.

An ***application*** of a rule *r=(M,R,A,c)* to a given graph *G* yields a resulting graph *H*, provided that *H* can be obtained from *G* in the following four steps:

1. CHOOSE an occurrence of the matching schema *M* in *G*.
2. CHECK the condition *c*.
3. REMOVE the components of *R* from *G,* in addition all the edges incident to a removed node are removed.

4. ADD the components of *A* in *G*, in addition all the nodes related to an added edge are also added.

The condition *c* is in general the fact that the matching schema is isomorphic to the chosen occurrence, but other conditions can be added such that some global conditions on the graph to be transformed. The application of a rule *r* to a graph *G* yielding a graph *H* is called a ***direct derivation*** from *G* to *H* through *r,* denoted by $G \Rightarrow_r H$ or simply by $G \Rightarrow H$. Given a set of rules *P,* the successive direct derivations $G_0 \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \ldots \Rightarrow G_n$ is a ***derivation*** from G_0 to G_n by rules of *P*. Since there are possibly several occurrences of a given matching schema (that can even overlap) and because there are many transformation rules and the direct derivation order is relevant, all the derivations form a set of graphs, which can be infinite. A set of terminals symbols *T* together with a set of rules *P* and an initial graph *S* form a ***graph grammar.*** The set of all graphs labeled with terminal symbols of a set *T* that can be derived from an initial graph *S* and a set of rules *P*, is the ***language*** generated by the graph grammar made of *S*, *T*, and *P*. This language is denoted by *L(S,T,P).*

4.2 Graph Transformation and the VMT Approach

We observe that at the meta-model level the notion of attributes is essential. Thus, we introduce the notion of ***attributed labeled graph*** [Roz97] that consists of a graph with attributed nodes and edges, i.e. an underlying graph structure equipped with (named) attributes for the edges and the nodes. Formally speaking, attributes are represented as two mappings that map, respectively, the edges and the nodes into the global domains of possible values for attributed edges and nodes. This notion of attributed labeled graph is a central notion of the abstract syntax of VMT. The notions of occurrence, transformation rule, and derivation remain very close to the definitions given in the previous section, the greatest change concerns the definition of the occurrence that, now, has to preserve not only the labelings, but also the mappings that associate attributes to the nodes and the edges.

The VMT approach provides a **rule ordering schema** that allows the user to formulate a graph transformation as a chain of simpler transformations. A rule ordering schema is nothing else than an attributed labeled graph in which the edges are directed. Naturally, we introduce directed graph by means of two additional mappings that associate a start and an end node to each edge in order to obtain an ***attributed directed labeled graph***. Thus, a rule ordering schema is formally represented as an attributed directed labeled graph in which each node corresponds to a single transformation rule. Since no choice is allowed for the application of a single transformation rule, the application of such a rule must terminate and the successive direct derivations of a single rule must be confluent, i.e. give a unique result. The chain of derivations generated by the rule ordering schema produces a unique graph that is the result of the graph transformation that belongs to the language generated by the set of transformation rules.

5 Related Work

In this section, we survey some of the approaches and technologies that are related to our approach and/or influenced its development in one way or another.

One approach to UML model transformation is to encode them directly in a procedural language using an API to the model repository offered by a UML tool. An advantage of this approach is that developers do not need any extra training to become operational. However, a disadvantage is that encoding transformations in a procedural language can be time-consuming and difficult to understand and maintain due to a lack of high-level abstractions for transformation specification. Also, tool model repositories usually restrict the kind of transformations that can be performed, simply because the API may not let one access or manipulate the required model information.

One proposal that promises to raise the level of abstraction of operations on UML models is UML's action language [Omg03a]. The language has been proposed as a way to procedurally define UML transformations [MB02, SPH+01] and is a special-purpose language for manipulating UML models. However, due to its "general-purpose" context, the UML action language still suffers, albeit less chronically, from a lack of high-level abstractions for dealing with model transformations.

Another technique is to treat UML models as graphs. Much work has been performed on graph grammars and graph transformation systems. Graph transformations are realized by the application of transformation rules, which are rewriting rules for graphs. A transformation rule consists of a graph to match, commonly referred to as LHS, and a replacement graph, commonly referred to as RHS. If a match is found for the LHS graph, then the rule is fired, which results in the matched sub-graph of the graph under transformation being replaced by the RHS graph. The PROgrammed GRaph REplacement System (PROGRES) approach [SWZ97] offers a mean to not only specify transformation rules but to also define the sequencing of these rules, described using imperative constructs. There are several other tools that are based on graph transformation theory; such as AGG, GenGed [BEW02], and one dedicated to UML and Java, Fujaba [FUJABA].

The use of logic programming languages have also used in the context of model transformations. In [Whi02], a framework dedicated to UML model transformations implemented in MAUDE programming language [CDE+01] is presented.

Milicev proposes a graphical language for specifying model transformations [Mil02]. The approach proposes an extended UML object diagram as notation for developing the mapping specification. The diagrams are extended with the concepts of conditional, repetitive, parameterized, and polymorphic model creation, using UML's stereotype extensibility mechanism. Execution support for the proposed language is offered by a mapping to C++ code. Also, the approach offers the ability to reuse fragments of mapping specification through the use of parameterized model creation. An important limitation of the approach is its underlying assumption that the selection of source model elements for the transformation can be easily expressed in a general-purpose programming language, i.e., C++. If one were faced with complex selection criteria, it would be very likely that these selection conditions would become complex and hard to maintain. In fact, it would be at least useful to offer a language that is tailored for such a purpose, such as, UML's Object Constraint Language (OCL) [Omg03a].

UML's Object Constraint Language [WK98] has also been proposed as a way to declaratively describe UML model transformations, e.g., [PVJ02, SPL+01]. Kleppe et al. [KWB03] define a transformation language that uses OCL to specify the conditions for the firing of a transformation and the elements that are input to and output of the transformation. One nice feature of their approach is the way that it facilitates the definition of bi-directional transformations, so that one can perform a transformation in either direction. The approach has many similarities to our approach. However, we chose a principally graphical notation, compared to their fully textual notation, which we believe has advantages in terms of usability and conciseness of description.

6 Conclusion

For model-driven software development approaches to become a reality in mainstream software development practice, software development tools need to be able to better automate the creation, evolution and maintenance of the models used throughout the software lifecycle. In this direction, one of the keys areas is the easy description and execution of UML model transformations.

In this paper, we presented the Visual Model Transformation (VMT) approach. The VMT approach offers a visual and declarative language to specify UML model transformations. Transformations can be defined in the proposed language by specifying transformations rules and defining the order in which these ones are to be executed.

Further work on the VMT approach will consist in continuing work on the design and implementation of the second version of MEDAL tool to support the proposed VMT approach, and developing the formal foundations of the language in order to be able to prove transformation properties based on the source and target model semantics.

7 Acknowledgements

We would like to thank Wojtek Kozaczynski and Jim Thario who gave us a lot of inputs on Model transformations and Rational XDE. We also would like to thank the other SE2C team members for their valuable comments.

8 References

[BEW02] R. Bardohl, C. Ermel and I. Weinhold, “AGG and GenGED: Graph Transformation-Based Specification and Analysis Techniques for Visual Languages”, Electronic Notes in Theoretical Computer Science, volume 72 issue 2, Elsevier Science B.V, 2002

[BG00] D. Buchs and N. Guelfi; “A formal specification framework for object-oriented distributed systems”. IEEE Transactions on Software Engineering, special issue on Formal Methods for Open Object Based Distributed Systems, July 2000, vol. 26, n°7, pp 635-652.

[CDE+01] M. Clavel, F. Durän, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. Quesda; “Maude: Specification and Programming in Rewriting Logic”. Theoretical Computer Science, 2001.

[DiM99] G. Di Marzo Serugendo; “Stepwise Refinement of Formal Specifications Based on Logical Formulae: from COOPN/2 Specifications to Java Programs”. Ph.D. Thesis, no 1931, Ecole Polytechnique Fédérale de Lausanne, Département d'informatique, CH-1015, Lausanne, Suisse, 1999.

[Eclipse03] Eclipse IDE website: http://www.eclipse.org

[FIDJI] N. Guelfi, G. Perrouin, B. Ries and P. Sterges; “FIDJI Project Annual Activities Report”. Applied Computer Science Department technical report n° TR-DIA-03-01, Luxembourg University of Applied Sciences, Luxembourg-Kirchberg, Luxembourg, 2002.

[FUJABA] Fujaba project website : http://www.fujaba.de

[GP02] N. Guelfi and G. Perrouin; “Rigourous Engineering of Software Architectures: Integrating ADLs, UML and Development Methodologies”. In Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA’02), November 4-6 2002 Cambridge, USA.

[GR03] N. Guelfi, B.Ries,” JAFAR2:An Extensible J2EE Architectural Framework for Web Applications”, submitted to PFE-5, Fifth International Workshop on Product Family Engineering, November 4-6, 2003, Sienna, Italy

[GS02] N. Guelfi and P. Sterges; “JAFAR: Detailed Design of a Pattern-based J2EE Framework”. In Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA’02), November 4-6 2002 Cambridge, USA.

[GS03] N. Guelfi and P. Sterges, “MEDAL: A CASE Tool Extension for Model-driven Software Engineering”, submitted to SwSTE’03, IEEE International Conference on Software - Science, Technology & Engineering, November 4-5 2003, Herzelia, Israel

[Gue01] N. Guelfi; “Flexible Consistency In Software Development Using Contracts and Refinements”. 2nd International Workshop on Living With Inconsistency, part of the International Conference on Software Engineering - ICSE'01, May 2001, Toronto, Canada.

[KT02] W. Kozaczynski and J. Thario; “Transforming User Experience Models To Presentation Layer Implementations”. OOPSLA 2002, Second Workshop on Domain-Specific Visual Languages available on http://www.cis.uab.edu/info/OOPSLA-DSVL2/Papers/

[Mil02] D. Milicev; “Domain Mapping Using Extended UML Object Diagrams”. IEEE Software, pp. 90-97, March/April 2002.

[Omg01a] OMG CWM Partners; “Common Warehouse Metamodel (CWM) Specification”, Feb. 2001. http://www.cwmforum.org/spec.htm

[Omg01b] OMG Architecture Board ORMSC; "Model Driven Architecture (MDA)". July 9, 2001 (draft). http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01

[Omg02a] OMG TC; "MOF 2.0 Query/Views/Transformations RFP", 2002. http://cgi.omg.org/cgi-bin/doc?ad/02-04-10

[Omg03a] OMG Unified Modeling Language Revision Task Force; "OMG Unified Modeling Language Specification". Version 1.5, March 2003. http://www.omg.org/technology/documents/formal/uml.htm

[Omg03b] OMG UML profiles, 2003. http://www.omg.org/mda/specs.htm#Profiles

[PVJ02] D. Pollet, D. Vojtisek, J-M. Jézéquel; "OCL as a Core UML Transformation Language". WITUML: Workshop on Integration and Transformation of UML models (held at ECOOP 2002), Malaga, Spain, June 2002.

[Roz97] G. Rozenberg (ed.); "Handbook of graph grammars and computing by graph transformation: Volume I Foundations". World Scientific Publishing, 1997.

[SGB00] J. Gosling, B. Joy, G. Steele, and G. Bracha; "The Java Language Specification". Second Edition, Addison Wesley, June 2000

[SPH+01] G. Sunyé, F. Pennaneach, W-M. Ho, A. Le Guennec and J-M. Jézéquel; "Using UML Action Semantics for Executable Modeling and Beyond". Proceedings of 13th International Conference, CAiSE 2001, Switzerland. LNCS (Lecture Notes in Computer Science), no. 2068, pp. 433-447, Springer Verlag, 2001.

[SPL+01] G. Sunyé, D. Pollet, Y. Le Traon and J-M. Jézéquel; "Refactoring UML Models". UML 2001 — The Unified Modeling Language: Modeling Languages, Concepts, and Tools, 4th International Conference, Canada; Gogolla and Kobryn (Ed.), LNCS (Lecture Notes in Computer Science), no. 2185, pp. 134-148, Springer Verlag, 2001.

[SRP03] P. Sterges, B. Ries, G. Perrouin "UML-to-UML model generic Transformations: the MEDAL tool", Applied Computer Science Department technical report n° TR-DIA-03-02, Luxembourg University of Applied Sciences, Luxembourg-Kirchberg, Luxembourg, 2002.

[Struts03] Apache Struts framework website: http://jakarta.apache.org/struts/index.html

[SWZ97] A. Schürr, A. Winter and A.Zündorf; "The Progress Approach: Language and environment". In Chapter13 of G. Rozenberg (eds), *Handbook of graph grammars and computing by graph transformation: Volume II Applications, Languages and Tools*, World Scientific Publishing, 1997.

[Whi02] J. Whittle; "Transformations and Software Modeling Language: Automating Transformations in UML". Jézéquel, Hußmann & Cook (Eds.): Proceedings of UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany. Lecture Notes in Computer Science no. 2460, pp. 227-242, Springer, 2002.

[WK98] J. Warmer and A. Kleppe; "The Object Constraint Language: Precise Modeling With UML". Addison-Wesley 1998.

[XDE] Rational XDE Web Site; Rational Software Corporation, 2003. http://www.rational.com/products/xde/index.jsp

Attribute grammars as tools for model-to-model transformations

May Dehayni*, Louis Féraud
{dehayni, feraud}@irit.fr

Institut de Recherche en Informatique de Toulouse, Université Paul Sabatier
118, route de Narbonne
31062 Toulouse cedex 4 France

Abstract. In this paper, we propose a system for model transformation. First, we define the problem, followed by the need for model transformation. Next, other approaches of model transformations, such as the XSLT and MTrans languages, are presented. This brief survey gives the model transformation characteristics and situates our approach, which is based on the formal semantics specified by attribute grammars. The semantics offered by this formalism brings the rigor of formal semantics while providing good performance at the implementation by a specific software called an evaluator. Thus the system architecture consists of essentially defining a textual abstract syntax of the source meta-model and in building the attribute grammar that expresses the transformation. The input of the automatically generated evaluator is some text that describes the source model where syntax is defined by an abstract grammar. The output of the evaluator is another text that corresponds to the target model in accordance with its grammar. This paper will present the transformation from EDOC/ECA to CCM model, which is integrated in the MDA approach *i*.e., the mapping from PIM to PSM.

Keywords: meta-modeling, model transformation, XSLT, attribute grammars, evaluator.

1. Introduction

About twenty years ago, the transition from procedural to object technology occurred. The revolution that object technology caused in software engineering has continued with the emergence of model engineering. Currently, models are treated like first-class entities in software development [4], [5] particularly in the MDA (Model Driven Architecture) approach [26], [33] and the problem of model processing appears crucial and thus motivates our contribution.

As a first class citizen, a model is an entity that can be handled with precise properties. It can be named, extended, instantiated, transformed, and it is formal in the sense that it has a syntax and precise semantics defined according to a model called "meta-model". A meta-model belongs to the fourth-level architecture standardized by the OMG as the MOF (Meta Object Facility) [13]. A meta-model may also be viewed as the language that generates the models. Practically, in model engineering the majority of manipulations are apprehended as being model

* Lebanese National Council For Scientific Research scholar.

transformations. The model transformation problem based on meta-modeling has become a modeling issue for which several approaches have been proposed. Among these approaches, we can quote, the UMLAUT framework [9], a solution dedicated to the UML meta-model, the general language XSLT [15], for the transformation of models in the XMI format, and some particular solutions based on semantic networks [12] or using specific model transformation language like MTrans [3], [20].

This paper consists of a new proposal for model transformation. This problem will be studied through the following criteria: expressivity, implementation facility, and effectiveness. First, the context of the study will be defined as well as the problem and the need for model transformation. Then some transformation approaches will be presented. A description of our approach based on the formal semantics of attribute grammars will then be depicted. The EDOC/ECA to CCM meta-model transformation will be given. In the last section, will be discussed the characteristics and the benefits of our approach.

2. An overview of model transformation

Model transformation is of primordial importance in meta-model evolution, to facilitate the cooperation between developers, and to amalgamate models.

The four layers reflective framework MOF proposed by OMG, together with the meta-modeling techniques allow the description of meta-models. The problem of model transformation based on the MOF can be then stated in the following way: "*Given a source model 'm_1' described by a meta-model 'MM_1' we would define an automatic process making it possible to obtain a model 'm_2' conforming to a meta-model 'MM_2'; 'MM_1' and 'MM_2' being MOF compliant* ". In fact, this process is similar to the one applied in compiler construction, where a source text, which obeys some syntax and some semantics, is transformed into a target text that conforms to another syntax and another semantics; our approach is primarily based on this similarity.

3. Some approaches of model transformation

In this section, we discuss briefly some of the model transformation approaches.

3.1 Graph grammars

A meta-model can be viewed as a language: according a syntax and semantics to generate models as words. Generally, (meta-) models are represented in a graphic formalism, very often using UML class diagrams. As a result, models can be viewed as graphs. It is then natural to consider the use of graph grammars for expressing model transformation [6], [30].

The transformation process is based on syntactic graph rules; it consists in finding a Left Hand Side (LHS) graph and replaces it by a Right Hand Side (RHS). Some authors [21] take into account the non-determinism in rule applications on the sub-graphs to restrain the interests of this approach.

Thus, in the remainder of this paper, we focus on transformation methods using a textual model representation. This format is the one used by the standardized XMI format; such a form is also used in the tool UML Rose. Currently, the model transformation expression can be carried out according to various approaches; in this section some of them will be presented in order to study their characteristics and to introduce our transformation framework.

3.2 Transformation based on the use of an API

Many modeling tools build model repositories in accordance with the MOF specification and generate Application Programming Interfaces for each meta-model supported, such as for “dmof” [31] and “univers@lis” [32]. These APIs can also be used to describe the model transformation process by means of programs written in an imperative language: Java, C++, Eiffel, Python, *etc*. This solution provides the user with a set of interfaces used to describe the transformation process as a series of instructions that allow to generate the corresponding target model from a source model. The use of APIs to describe a transformation process is a powerful solution because, in general, programming languages have good performance at runtime. However, it is relatively difficult to give a code independent specification because the programmer is responsible of organizing and describing all stages explicitly in terms of imperative programs.

3.3 The XSLT language

The XMI standard (XML Model Interchange) [14] offers a special standard format to exchange meta-models. This format is generated by a DTD (Document Type Definition) in XML, which conventionally defines the representation of all (meta-) models. Relations between a DTD and an XML document, between a meta-model and the instantiated models are similar: both of them involve a word (XML document / model) of a language generated by a grammar (DTD / meta-model). As models are XML texts it appears that the XSLT language is a convenient solution for model transformation. XSLT [15] is a transformation language devoted to define XML document stylesheets. A program in XSLT is a sequence of rules. An XSLT rule consists of two parts: the first one identifies a pattern in the source document, the second one builds a part of the target document corresponding to the generated entities. XSLT can be considered roughly as a declarative language but certain aspects are close to imperative programming languages. Its principle rests on tree transformations. An XSLT processor applies the transformation rules described in a stylesheet to the tree representing the source XML document, namely the syntactic tree generated by the grammar specified in the DTD. Then XSLT is therefore suitable for tree transformation based models that are generally structured as graphs. Hence this language is not quite suitable for model transformations.

Although, XSLT is an appropriate standard for XML document transformation, it suffers from limitations when dealing with model transformations, because the class of transformations offered by this language is based on a top-down parsing on sub-trees and a bottom-up parsing on the ancestors of a selected node on the syntactic tree of an XML document. Moreover XSLT has limited calculation information on a tree [19]. Thus, realization of several traversals on the source model tree, essential in certain model transformation situations, imposes the creation of intermediate trees of intermediate computations, which can make the transformation process less powerful, difficult to read, and redundant. For example, let us consider in an UML diagram class models where an entity refers to other entities, or a diagram where a subclass references the inherited information from its ancestors. In such situations, it is essential to seek the element (syntactic node) referred in the source model tree, which is relatively difficult to implement in XSLT.

3.4 Some specific languages for model transformation

Among the specific approaches to model transformation, we can quote the following languages and tools:

The UMLAUT tool [9] defines a transformation framework for UML models. Its architecture is based on the use of preset transformation functions in an extensible library. The user can add his own functions in the Eiffel language and use the API generated by the tool specifically for the UML meta-model.

Model transformation can be obtained using a declarative language. In [28] such a language is proposed to transform UML class diagrams. This language uses the declarative style based on unification. It is a part of a transformation environment, which offers a way to also transform constraints expressed in OCL (Object Constraint Language) [34]; some of these constraints may be automatically inferred. A model or a meta-model is defined by a concrete syntax, an abstract syntax, and semantics domain. The authors of [2] propose an approach to define transformations within each above level. The key concept of this approach is the concept of a relation where the relationship between concepts is defined by patterns. The MIA (Model In Action) [12] language based on sNets (semantic Networks), is a declarative rule-based language. An inference engine activates the transformation rules. The MTrans (Model Transformation) [3], [20] is a declarative and imperative rule-based language. Each rule describes a stage of the transformation process. The sequences of the various stages is explicitly controlled by the user thanks to operators that allow to carry out explicit calls of transformation rules. MTrans reuses OCL to express navigation between model elements. The initial MTrans implementation produced an object program in XSLT, while the latest implementation generates a Python program on top of MOF-based interfaces using the univers@lis meta-modeling tool [32].

3.5 Model transformations by attribute grammars

The compilation techniques carry out program transformations from source programming language to a target language. For model transformation, it seems that the use of some of these techniques could be successful, because a meta-model is also a tool that defines a language. The syntax of a programming language is represented by a Chomsky grammar or by an abstract grammar [1] while its semantics is defined by a semantic formalism. Semantics can be specified by several formalisms; let us quote the operational [25], denotational [24], and axiomatic [27] semantics. In spite of the theoretical power of these semantics, they appear difficult to implement and they do not generally offer good performance in terms of automatic generation tools. In contrast, semantics offered by attribute grammars [7], [8], [17] bring formal rigor while providing good performance at the implementation level.

Attribute grammars (abbreviated AGs) were first introduced by Knuth [11] to describe syntactic-based translations. This approach is a purely declarative programming paradigm directed by syntax. It consists in computing information called "attributes" on an abstract syntax tree for each symbol. The set of attributes of a symbol is divided in two disjoined subsets respectively called inherited and synthesized attributes. An attribute can be of any data type; its value is computed by semantic rules associated with each grammar production. Attribute computation consists in solving a system of semantic equations [18]. The order of attributes computation is provided by an attribute dependence graph. This order is automatically determined by software called an evaluator, which accepts as input an AG specification. The output is an executable code, which computes the semantics (*i.e.*, attributes values) of a source text and generates the semantics described by the AG.

To increase the performances of evaluators, it appeared necessary to define subclasses of AG. The definition of subclasses is based on the dependence relations between attributes. Among the most significant classes we can, non-exhaustively, distinguish the purely synthesized grammars (attributes can be evaluated during the bottom up syntactic analysis), the Left Attribute Grammars (abbreviated LAG, where it is possible to make in parallel syntactic analysis and attribute computation), the Ordered Attribute Grammars (abbreviated OAG, the order is total over attributes) [10], [23], [29] and the strongly non-circular attribute grammars [18] (order fixed dynamically on each syntactic tree).

There exist several AG evaluators each of which has a language to express the semantic rules [7]. For our work, we have chosen the Cornell Synthesizer Generator, which can process OAG grammars. This choice appears as a good compromise between the expressive power of processed attribute grammars and runtime performance. In the following section, our approach for AG based model transformation will be presented with the representation of its architecture and giving an explanation of its various stages. Then, a short example of the EDOC/ECA to CCM transformation is presented. Next, our approach will be discussed.

4. Transformation framework based on Attribute Grammars

The architecture of our model transformation system based on meta-modeling by an AG is depicted below:

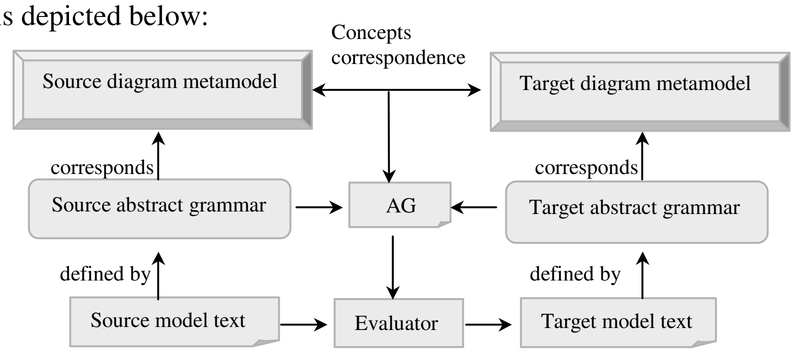


Fig. 1. Architecture of model transformation system by AG

The meta-models that occur in our transformation framework are defined by an abstract grammar using the concepts of phyla and operators [22]. The first step to build this system consists in choosing a textual format representation of the (meta-) models, then to define its abstract syntax, and to give an AG describing the transformation. The transformation framework generates a textual representation of the target model. Its input is the text that describes the source model, the syntax of which is defined by an abstract grammar of source meta-model. The output is another text corresponding to the specification of the target model in accordance with its grammar.

4.1 Definition of abstract syntax

The XMI is the most used textual format for model representation. However, it is a very verbose language. Thus, recently the OMG has standardized the definition of a textual language called HUTN (Human Usable Textual Notation), a more readable textual representation for MOF compliant models. HUTN closely resembles XMI: thus it is easy to convert from HUTN format into XMI and back.

The mapping between the concrete syntax and the following abstract syntax is obtained by canceling the terminal symbols and the purely linear productions [22]. A phylum is the analogue of a non-terminal, an operator is similar to the right part of a production. The abstract productions look like:

$phy_0 : Operator(phy_1\ phy_2 \ldots\ phy_n)$ where $phy_0, \ldots, phy_n$ are phyla and " : " is analogous to "$\rightarrow$" in concrete syntax.

To avoid confusing notations, attributes of object-oriented concepts are called "oo-attributes", we keep the word attribute to denote attributes occurring in AG.

A meta-model defines a set of concepts, their properties and their relations. It is generally represented by an UML class diagram. UML notation is defined by an abstract syntax and its semantics are described in UML. The syntax of meta-models is given by a minimal set of UML concepts that are the key concepts of MOF, which primarily consists of:

- class, which describes entities by their properties which are represented by oo-attributes and references;
- oo-attribute, which defines the properties attached to each class; and
- relation, which represents connections between classes.

Each concept is defined by a phylum *i.e.* a node of the abstract syntax tree. Then the representation of a meta-model in an abstract-grammar consists of the phyla associated to their concept and the operators defined for each phylum; they allow gathering the phyla corresponding to the components of the concept (oo-attributes, references, and their composite concepts). In the next section, an example of model transformation based on attribute grammars is given.

4.2 The transformation of ECA to CCM model by attribute grammars

This example is borrowed from [3], where the transformation has been implemented using MTrans language [20]. Transformation of model entities consists in computing their properties (here oo-attributes and references). In order to specify their target entities, we associate to each phylum representing the source concept a set of attributes essentially defining its components. These components are its: name, oo-attributes, references, sub-entities and a synthesized attribute "target", which specify target entities in accordance with the correspondence relations between source and target meta-models entities. Oo-attributes and references of target entities can be computed from the properties values of source entities or can be initialized by default values.

To clarify and explain our approach, we consider the transformation example of the meta-model EDOC/ECA (Entreprise Distributed Object Component / Entreprise Component Architecture) [35] into CCM (CORBA Component Model) [16]. The former is a platform independent model (PIM), which specify abstract component model independently of any platform. The latter is a Platform Specific Model (PSM) that defines components implementation in CORBA middleware.

The UML profile for EDOC is an OMG RFP that defines a specification of distributed applications by independent platform components using UML. The ECA part of this specification defines a model of components in which each component is described by its properties, its ports, its connections with other components and its sub-components. A port is simple (flowPort) or complex (protocolPort). The former defines a simple interaction between components while the latter defines a complex interaction, which is realized by its sub ports. A port is defined by its name, its direction (initiator, responder), its size (one or more) and its mode (synchronous, asynchronous).

The CCM specification defines a structural meta-model, which specifies a component by its name, its events (emits, consumes, publishes) and interfaces (facets, receptacles).

Each ECA component will be transformed into a CCM component type and a CCM home. The ports of an ECA component will be translated, depending on the communication nature (external, internal), the port: mode, direction, size into either an (emits, consumes, publishes) events or (facets, receptacles) interfaces of the corresponding CCM target entity.

The principal difference between ECA and CCM is that the first meta-model permits recursive composition of components while the second does not. In our approach of transformation based on AG, this problem can be solved by the fact that each (processComponent) phylum representing a component is, among others, derived in its ports and the identities of its sub components. This enables us to compute by inherited or synthesized attributes of the component phylum all the needed information to transform it into a CCM entity, and consequently to avoid making a deep traversal on the component structures of the source model several times to seek necessary information to transform it. In the follow, an abstract grammar of the ECA source meta-model is given. After, an attribute grammar that defines transformation of an ECA model into CCM meta-model is given.

4.2.1 An abstract grammar for EDOC/ECA source meta-model

The abstract grammar is defined in SSL (Synthesizer Specification Language) of the Cornell Synthesizer Generator. A phylum is associated to each concept of the source meta-model. Operators are also defined for each phylum; they allow gathering the phyla corresponding to the components of the concept. Let us consider the " flowPort " ECA concept that is defined by an identifier, a direction, a size and a mode then we define the phylum " flowPort " derived in " FlowPortDef (id direction size mode). In the follow some abstract syntax productions of the ECA abstract grammar are given:

```
metamodelEDOC:MetamodelEDOC(id communityProcess);
communityProcess:CommunityProcessDef(id processComponentList connectionList);
processComponentList:ProcessComponentListNil()|ProcessComponentListDef(processComponent
processComponentList);
```

```
processComponent:ProcessComponentDef(id protocolPortList flowPortList subComponents);
protocolPortList:ProtocolPortListNil()|ProtocolPortListDef(protocolPort protocolPortList);
protocolPort:ProtocolPortDef(id flowPortList);
flowPortList:FlowPortListNil()|FlowPortListDef(flowPort flowPortList);
flowPort:FlowPortDef(id direction size  mode);
subComponents:SubComponentsNil()|SubComponentsDef(id subComponents);
id:IdentifierNull()|Identifier(IDENTIFIER);
size:One()|Many();
mode:Async()|Sync();
direction:Initiates()|Responds();
connectionList:ConnectionListNil()|ConnectionListDef(connection connectionList);
connection:ConnectionDef(id id id id);
```

4.2.2 An attribute grammar for the transformation of an EDOC/ECA to CCM

An AG that transforms ECA components and their ports into CCM entities consists in defining attributes of phyla and their associated semantic rules. Now, the following table gives the attributes that occur in the AG. Here, " s " and " i " denote respectively synthesized and inherited attributes.

Phylum	Attribute	Mode
FlowPort	name	s
	d	s
	s	s
	m	s
	connections	i
	PCsubComp	i
	PCname	i
	internalPC	i
	target	s
FlowPortList	connections	i
	PCsubcomp	i
	PCname	i
	internalPC	i
	target	s
protocolPort	name	s
	connections	i

Phylum	Attribute	Mode
protocolPort	PCsubComp	i
	PCname	i
	internalPC	i
	target	s
protocolPortList	connections	i
	PCsubComp	i
	PCname	i
	internalPC	i
	target	s
subComponents	l	s
ProcessComponent	name	s
	connections	i
	idSubComp	i
	myIdSubComp	s
	target	s
ProcessComponent List	connections	i
	idSubComp	i
	idSubComp	s
CommunityProcess	name	s
	idSubComp	s

Table 1. Attributes definition

Semantic rules express models transformation by establishing correspondence between concepts of source and target meta-models. The definition of these rules consists in expressing any concept of source meta-model by using one or more concepts of the target meta-model. These rules are defined in the SSL language: "$$" symbol denotes left hand side phylum of an abstract production. Some semantic rules, describing the computation of a "flowPort" (resp. process component) properties: name, size, mode, direction and the generation of the specification of their target entities, are given in the following table. The computation of the "flowPort.target" attribute is done by the function "PortTarget", which generates target CCM events or interfaces specification, using string concatenation operators, according to the target model structure and depending on the "flowPort" phylum attributes values.

Production	Semantic rules
flowPort:FlowPortDef(id direction size mode);	$$.name=id.v; $$.s=size.s; $$.m=mode.m; $$.d=direction.d; $$.target=PortTarget($$.PCname, $$.internalPC,$$.name,$$.d,$$.s,$$.connections,$$.PCsubComponents);

Table 2. Semantic rules from ECA " flowPort " to CCM " events "

Thanks to the AG, it is possible to automatically transform an ECA model into CCM model. The AG consists of 25 abstract syntax productions, of 67 semantic rules involving 44 attributes. It is to be noted that the expression of the transformation is purely declarative, *i.e.,* there is no mix between rules and algorithms devoted to semantics as in the API based approach. In contrast with similar approaches based on graph grammars [2], which necessitate the assistance of a programmer, the above transformation process is completely automatic since the abstract syntax and the semantic rules are defined. Parse tree and semantic equations imply an attribute dependence graph. In the follow, the decorated parse tree of a part of the ECA meta-model and the dependence graph of the "connections" attribute of the "flowPort" phylum are depicted below:

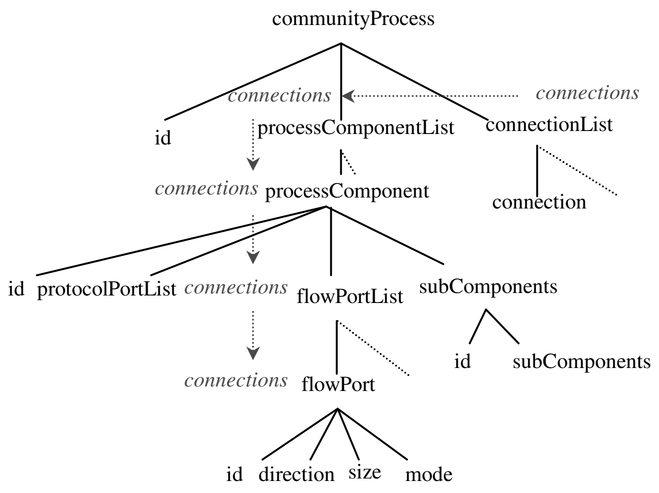


Fig. 2. Decorated parse tree of a part of the ECA meta-model

In the dependence graph, attributes computation order must follow the edges, which change to the maximum of a level on the derivation tree. The edge "→" denotes that the left attribute value must be computed before the right one. In the example, the AG contains inherited attributes (connections, PCname, PCsubComponents, internalPC) for the "processComponent" phylum of which the computation depends on its super node "processComponentList". Then the specified AG belongs to the LAG class and attributes can be computed in parallel with the syntactic tree construction using a recursive evaluator. But, if some phyla like “connectionList” and “processComponentList” are scattered in a way where the connections appears before the process component then the dependence between “processComponentList.connections” and “connectionList.connections” attributes is a right dependence. In this case, the AG is not yet LAG but OAG and requires an evaluator generator of the same processing power as Cornell Synthesizer Generator.

As it was previously stated, as many (meta-) models are in the XMI format, this fact leads to simply express transformations using XSLT. Processing the above example in XSLT would be a bit difficult because of the existence of inherited attributes. More generally, an OAG grammar allows several traversals when computing attributes; these traversals are automatically performed by the evaluator. Nevertheless for the same computation XSLT requires to explicitly program the sequence of tree visits. This requirement may lead to possible programming errors and to performance corruption of the transformation process.

Considering now the implementation of this example in MTrans it appears that the transformation necessitates to perform a deep traversal on the tree structure of the source model, as described in [3], in order to find the entities descriptions that are essential to generate target model entities specification.

5. Conclusion

We have presented the AG formalism as a tool for model-to-model transformations. This transformation tool is declarative, in the sense that the user must only specify the rules to be used to evaluate the attributes values and never the order of computation. This order is implicitly and automatically given in accordance with the dependence relations between attributes. The model transformation framework based on AG rests on abstract syntax. The main advantage of this kind of syntax offers to decrease the size of the syntactic trees necessary to perform attribute evaluations.

The use of AG for model transformation offers the traditional benefits of this formalism: As being theoretically based, an AG guarantees precision. Semantic rules being expressed in a declarative mode on the syntactic structure, transformations are expressed with concision and clearness without being overwhelmed in code. The modularity of a specification, written in AG, is ensured by its block structure derived from grammar productions [18]. Moreover, this formalism constitutes an executable method of specification, since it describes only a computation in terms of an AG and then automatically produces a program, which carries it out [22]. As the evaluator code is automatically obtained, it results that an effective implementation is very simple.

The implementation of our model transformation approach is done in the Cornell Synthesizer Generator that generates incremental evaluators. This property offers the possibility of replacing a sub-tree of a syntactic tree representing a model by another sub-tree: the propagation of new attribute values on the whole tree is then automatically processed. Thus, using such an incremental evaluator allows the introduction of reusability in the model transformation. In certain cases, defining a new model transformer can be obtained by a mere attributed sub-trees substitution.

If we try to qualitatively compare the AG based approach to some other ones, it appears that the main benefit is the complete automatization of the transformer framework. In fact, the use of XSLT requires to explicitly program the necessary tree traversals in order to obtain a target model. Moreover, XSLT is processed by an interpreter, the MTrans generates a Python program that is an interpreted object oriented language while the Cornell Synthesizer generates a compiled evaluator. The programmer of a model transformation in MTrans must explicitly control the various stages of the process; this is by invoking suitable rules and by implementing functions that permits to make deep traversals of the source model structure in order to seek required information. These qualitative comparative elements between the XSLT, MTrans and the AG transformation approaches are summarized in the following table:

Criteria / approach	Semantic/syntax coupling	Syntax tree parsing	Parsing management	Execution
XSLT	Filter expressions and semantics are mixed	Downward and ascending parsing	Explicit	Interpreted
MTrans	Filter expressions and semantics are mixed	Depth parsing	Explicit	Interpreted
Attribute Grammars	Syntax semantic separation	Many ascent descent parsing	Implicit	Compiled

A new transformation language standard called QVT (Query Views Transformation) has recently appeared, and we are currently working on a comparison of our approach with the QVT based one.

6. Acknowledgement

We are very grateful to Mariano Belaunde, from France Telecom R&D, for his interesting discussions, comments, and remarks. This work is supported by the project RNTL-TRAMs from the French Ministry of Industry.

7. References

1. Aho, A.V. & Sethi, R. & Ullman: COMPILERS: Principles, Techniques and Tools; Addison-Wesley (1986)
2. D. Akehust, S. Kent: A Relational Approach to Defining Transformation in a Meta-model. In *Proc. UML 2002*, Dresden, LNCS 2460. Springer-Verlag (2002)
3. M. Belaunde, M. Peltier: From EDOC components to CCM components: a precise mapping specification. *In Proc. ETAPS 2002*, Grenoble, France. LNCS 2306 Springer (2002)
4. J. Bezivin: From Object Composition to Model Transformation with the MDA. In *Proc. TOOLS'USA*, Santa Barbara (2001)
5. S. Crawley *et al.*: Meta-meta is better. *Workshop on Distributed Applications and Interoperable Systems* (1997)
6. J.Cuny *et al.*: Graph-grammars and their application in computer science. 5th *internat. Workshop* Williamsburg VA USA, LNCS 1073 (1994)
7. J. Deransart, M. Jourdan: Attribute Grammars and their Applications. In *Proc. WAGA 90*, LNCS 461. Springer-Verlag (1990)
8. M. Jazayeri *et al.*: The intrinsically exponential complexity of the circularity problem for attribute grammars. *ACM* 18 (1975)

9. W. Ho *et al.*: UMLAUT an extensible UML transformation framework. *ASE'99*. IEEE (1999)
10. U. Kastens: Ordered Attributed Grammars. *Acta Informatica* 13(3), (1980)
11. D. Knuth: Semantics of context free languages. *Mathematical Systems theory* (1968)
12. R. Lemesle : Technique de Modélisation et de Méta-modélisation. PhD thesis (2000)
13. OMG, Meta Object Facility Specification v1.3, ad/99-09-05 (1999)
14. OMG, eXtensible Metadata Interchange XMI specification version 1.1, ad/00-11-02, (2000)
15. OMG, eXtensible Stylesheet Language Transformation version 1.0 (1999)
16. OMG, CORBA Components – Volume II, MOF-based Meta-models. http://tao.doc.wustl.edu/Components/CCM/99-07-02(CCM-V2).pdf
17. J. Paakki: Attribute Grammar paradigms -- A high-level methodology. *ACM Computer surveys*, 27(2), (1995)
18. D. Parigot : Transformations, Évaluation incrémentale et optimisations des grammaires attribuées : le système FNC-2, PhD thesis (1988)
19. D. Parigot *et al.*: L'apport des technologies XML et objets pour un générateur d'environnements: SmartTools, *revue Objet* (October 2002)
20. M. Peltier: Transformation entre un profil UML et un méta-modèle MOF: application du langage MTrans. In *Proc. LMO* (2002)
21. G. Perrouin: Models transformations: methods and tools. CEA, SE2C Architecture seminar (April 2003)
22. W. Reps, T.Teitelbaum: The Synthesizer Generator: A system for constructing language-based editors, Spinger-Verlag (1989)
23. Reps, T. Teitelbaum: The Synthesizer Generator. In *Proc. ACM SIGSOFT/SIGPLAN*, (Pittsburgh, PA, April 1984), *ACM/SIGPLAN* Notices 19 (May 1984)
24. D. A. Schmidth: Denotational Semantics, A methodology for language Development Wm C.Brown Publishers Dubuque, Iowa (1988)
25. Slonneger & Kurts: Formal Syntax and Semantics of Programming Languages. Addition Wesley (1995)
26. R. Soley and the OMG Staff Strategy Group Object Management Group. Model Driven architecture White Paper Draft 3.2 – (November 2000)
27. D.A. Watt: Programming Language Syntax and Semantics. Prentice-Hall (1991)
28. J. Whittle: Transformation and software modeling language: automatic transformation in UML. In *Proc. UML 2002*, Dresden, LNCS 2460. Springer-Verlag (2002)
29. D. Yeh: On incremental evaluation of ordered attributed grammars. BIT 23 (1983)
30. A. Zündorf: Graph pattern matching with PROGRES. In J.Cuny, H.Ehrig, G. Engels, G.Rozenberg eds. Graph-grammars and their application in computer science. 5th internat. Workshop Williamsburg VA USA, LNCS 1073 (1994)
31. www.dstc.com/Products/CORBA/MOF
32. http://universalis.elibel.tm.fr/site/
33. http://www.omg.org/mda
34. http://www.omg.org/docs/formal/01-09-67.pdf
35. http://www.omg.org/edoc

MDA and Integration of Legacy Systems: An Industrial Case Study

Parastoo Mohagheghi[1], Jan Pettersen Nytun[2], Selo[2], Warsun Najib[2]
[1]Ericson Norway-Grimstad, Postuttak, N-4898, Grimstad, Norway
[1]Department of Computer and Information Science, NTNU, N-7491 Trondheim, Norway &
[1]Simula Research Laboratory, P.O.BOX 134, N-1325 Lysaker, Norway
[2]Agder University College, N-4876 Grimstad, Norway
parastoo.mohagheghi@eto.ericsson.se, jan.p.nytun@hia.no

Abstract

The Object Management Group's (OMG) Model Driven Architecture (MDA) addresses the complete life cycle of designing, implementing, integrating, and managing applications. There is a need to integrate existing legacy systems with new systems and technologies in the context of MDA. This paper presents a case study at Ericsson in Grimstad on the relationship between the existing models and MDA concepts, and the possibility of model transformations to develop models that are platform and technology independent. A tool is also developed that uses the code developed in Erlang, and CORBA IDL files to produce a structurally complete design model in UML.

1. Introduction

The success of MDA highly depends on integration of legacy systems in the MDA context, where a legacy system is any system that is already developed and is operational. Legacy systems have been developed by using a variety of software development processes, platforms and programming languages. Ericsson has developed two large-scale telecommunication systems based on reusing the same platforms and development environment. We started a research process (as part of the INCO project [3]) to understand the development process in the context of MDA, and to study the possibility to transform from a PSM to a PSM at a higher level of abstraction, or to a PIM. Part of the study is done during a MSc thesis written in the Agder University College in spring 2003 [8]. We studied what a platform is in our context, which software artifacts are platform independent or dependent, and developed a tool for model transformation, which may be part of an environment for round-trip engineering.

The remainder of the paper is structured as follows: Section 2 describes some state of the art. Section 3 presents the Ericsson context, and section 4 describes platforms in this context and transformations. Section 5 describes a tool for transformation, and the paper is concluded in section 6.

2. Model-Driven Architecture

The Model-Driven Architecture (MDA) starts with the well-known and long established idea of separating the specification of the operation of a system from the details of the way that the system uses the capabilities of its platform [5]. The requirements for the system are modeled in a *Computation Independent Model* (CIM) describing the situation in which the system will be used. It is also common to have an information model (similar to the ODP information viewpoint [4]) that is computation independent. The other two core model concepts in MDA are the *Platform Independent Model* (PIM) and the *Platform Specific Model* (PSM). A PIM describes the system but does not show details of how its platform is being used. A PIM may be transformed into one or more PSMs. In an MDA specification of a system, CIM requirements should be traceable to the PIM and PSM constructs that implement them, and vice versa [5]. Models are defined in the Unified Modeling Language (UML) as the OMG's standard modeling language. UML meta-models and models may be exchanged between tools by using another OMG standard, the XML Metadata Interchange (XMI).

Model transformation is the process of converting one model to another model of the same system [5]. An MDA *mapping* provides specifications for transformation of a PIM into a PSM for a particular platform. Mapping may be between a PIM to another PIM (model refinement for example to build a bridge between analysis and design), PIM to PSM (when the platform is selected), PSM to PSM (model refinement during realization and deployment), or PSM to PIM (reverse engineering and extracting core abstractions).

Like most qualities, platform independence is a matter of degree [5]. When a model abstracts some technical details on realization of functionality, it is a PIM. However it may be committed to a platform and hence be a PSM.

3. The Ericsson Context

GPRS (General Packet Radio Service) provides a solution for end-to-end Internet Protocol (IP) communication between a mobile entity and an Internet Service Provider (ISP). Ericsson has developed two products to deliver GPRS to the GSM (Global System for Mobile communication) and WCDMA (Wideband Code Division Multiple Access) networks [1].

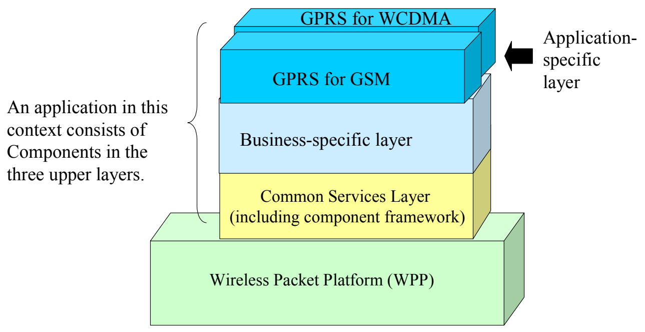

GPRS for WCDMA
GPRS for GSM
Application-specific layer
An application in this context consists of Components in the three upper layers.
Business-specific layer
Common Services Layer (including component framework)
Wireless Packet Platform (WPP)

Fig. 1. The GPRS Nodes software architecture

Figure 1 is one view of the software architecture, where the hierarchical structure is based on what is common and what is application specific. Other views of the architecture reveal that all components in the application-specific and business-specific layers use a component framework in the common services layer, and all components in the three upper layers use the services offered by WPP [6]. Size of each application is over 600 NKLOC (Non-commented Kilo Lines Of Code measured in equivalent C code). Software components are mostly developed internally, but COTS components are also used. Software modules are written in C, Erlang (a functional language for programming concurrent, real-time, and distributed systems [2]), and Java (only for user interfaces). The software development process is an adaptation of the Rational Unified Process (RUP) [7]. UML modeling is done by using the Rational Rose tool.

4. Platforms and Transformations

Figure 2 shows the software process from requirements to the executables, several models representing the system, and the relationships between these models and the MDA concepts.

The use case model, domain object model, use case specifications and supplementary specifications (textual documents) are developed in the Requirement workflow. Requirements of the system are then *transformed* to classes and behavior (as described in sequence diagrams) in the Analysis workflow. Design is a *refinement* of analysis, adding new classes, interfaces and subsystems, and assigning them to components. Elements in the design model are subsystems, blocks (each subsystem consists of a number of blocks), units (each block consists of a number of units) and software modules (each unit is realized in one or several modules). IDL files are either generated from the component model, or written by hand. From these IDL files, skeletons and stubs are generated, and finally realization is done manually.

Some subsystems in the design model make a component framework for real-time distributed systems that uses CORBA and its Interface Definition Language (IDL), and Erlang/OTP for its realization (OTP stands for Open Telecommunication Platform, which offers services for programmers in Erlang [2]). In the design phase, it may be seen as a technology-neutral virtual machine as described by MDA (a virtual machine is defined as a set of parts and services, which are defined independently of any specific platform and which are realized in platform-specific ways on different platforms. A virtual machine is a platform, and such a model is specific to that platform [5]).

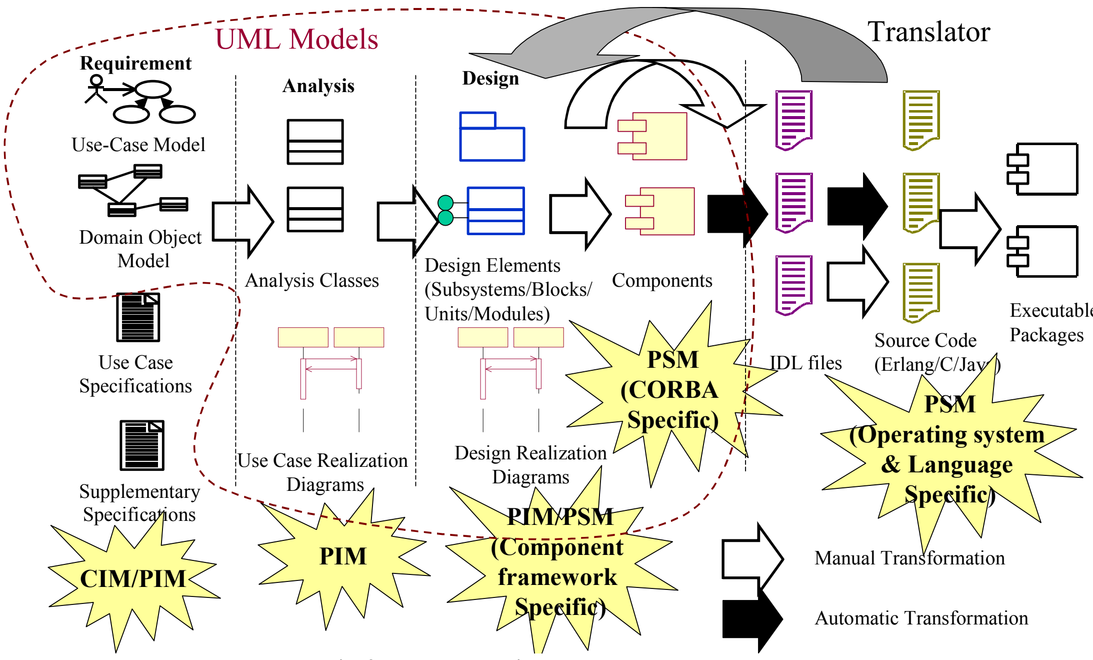

UML Models
Translator
Requirement
Use-Case Model
Domain Object Model
Use Case Specifications
Supplementary Specifications
Analysis
Analysis Classes
Use Case Realization Diagrams
Design
Design Elements (Subsystems/Blocks/ Units/Modules)
Design Realization Diagrams
Components
IDL files
Source Code (Erlang/C/Java)
Executable Packages
CIM/PIM
PIM
PIM/PSM (Component framework Specific)
PSM (CORBA Specific)
PSM (Operating system & Language Specific)
Manual Transformation
Automatic Transformation

Fig.2. From requirements to executables

RUP calls moving from one model to another one for translation, transformation or refinement. Hence software development in the adapted RUP process may also be seen as a series of transformations. However a transformation in RUP is different from a transformation in MDA, since a transformation in MDA starts from a complete model and have a record of transformation. UML models and other artifacts developed in the requirement workflow describe the system in the *problem domain* (as required by the GPRS specifications), and not in the *solution domain*. These are part of a PIM that is not computationally complete. Models in the analysis workflow describe the system in the solution domain and are also part of a PIM. It is first in the design workflow that we could have a computationally complete PIM (that contains all the information necessary for generating code), but it is dependent on the component framework with its realization in CORBA and OTP. On the other hand, each PSM at a higher level of abstraction is a PIM relative to the PSM at the lower level (less technology dependent). The curved gray arrow in Figure 2 shows a tool called Translator, which is described in section 5.

We notice that most transformations are done manually and therefore:

- There is a risk for inconsistencies between textual requirements and the UML models, between different UML models, and between UML models and the code. Inspections and testing are performed to discover such inconsistencies, which are costly.
- Developers may update the code, IDL files, or the design model without updating other models.
- Not all models are developed completely. The analysis model (consisting of analysis classes and sequence diagrams describing the behavior) is only developed for a fraction of use cases. The reason is simply the cost. Another example is the design model where not all the units are completely modeled. If the platform changes, there is not a complete PIM for generation of a PSM in another platform.

5. The Translator

We studied the possibility of reverse engineering the code in order to develop a complete PIM or PSM. We restricted our study to the Erlang environment in the first phase. Our method is based on:

- Filtering out parts of the code that is platform specific, where a platform in this context is the Erlang/OTP platform and CORBA. Among these aspects were operations for starting and restarting the applications and processes, consistency check, transaction handling (a set of signaling messages interchanged between software modules aiming at completion of a common task), and communication mechanisms during message passing.

- Combing the code with IDL files: Erlang is a dynamically typed language, and the programmer does not declare data types. Therefore we had to use the IDL files to extract data types.
- Using XMI for model exchange.

We studied several commercial tools but ended with making our own tool, the *Erlang to XMI Translator*. The reason was that none of the tools supported reverse engineering from Erlang code or from the sequence diagrams in the design model (although these diagrams are neither complete nor always synchronized with changes in the code).

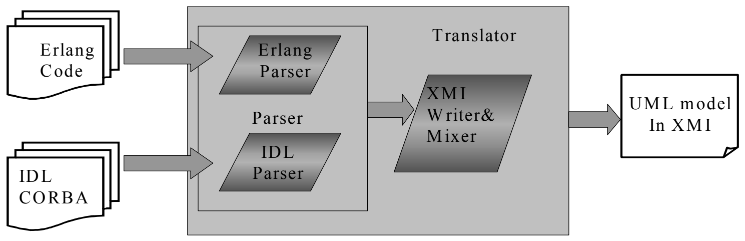


Fig. 3. The Erlang to XMI Translator

The resulting UML model is in XMI, which may be opened by other tools such as Rational Rose (the Rose plug-in for XMI must be installed). As we recognized the need to be able to separately parse single subsystems (parsing the total system takes too long time and a subsystem may be updated at any time), we have developed an XMI mixer that combines separate XMI files (from the translator or other tools that export UML models in XMI) and generates a complete model. The tool is developed in Java. The resulting model has the following characteristics:

- It is still dependent on the internally developed component framework and uses its services. However, it is independent of CORBA, the Erlang language and OTP.
- It is a *structurally complete model*, and shows the complete structure of the design model. However it does not have information on the behavior. We have not extracted the behavior of the system that is described in the code. To do so, we would need an action semantics language.
- It is using XMI version 1.0 and UML version 1.4.

Some characteristics of Erlang make the transformation more complex than for other programming languages. In Erlang, data types are not specified, and therefore we used the IDL files for identifying data types. Another problem was that Erlang allows defining methods with the same name, but different number of parameters in a single software module. Although internal coding guidelines recommends using different method names, sometimes programmers have kept these methods to keep the code backward compatible. In these cases we chose the method with higher number of parameters, and recognize that the code should be manually updated.

As mentioned in section 4, the component framework may be seen as a virtual machine, realized in CORBA and Erlang/OTP. It also includes design rules for application developers that describe how to use its services, and templates for programmers that include operations for using these services in Erlang (and C as well). We mapped each Erlang file to a UML class, and the exported methods in an Erlang file were mapped to public operations in the UML class. However we removed methods that depend on the OTP platform. This removal makes the model platform independent, but the virtual machine looses some of the services that were not described in a technology-neutral way; e.g. services for starting the system and transaction handling.

We recognized the following advantages of raising the level of abstraction by transforming a PSM to another PSM:

- The model is synchronized with the code. Any changes in the code can be automatically mirrored in the model by using the developed tool.
- The UML model may be used to develop the system on other platforms than CORBA or other languages than Erlang. It may also be integrated with other models or be used for future development of applications.

- The model is exchangeable to by using XMI.
- The new UML model may be used during inspections or for developing test cases.

6. Discussion and Conclusions

Ericsson uses Erlang for its good performance and characteristics suitable for concurrent, distributed applications. But Erlang is not in the list of languages supported by commercial MDA tools. However our study confirmed the possibility and low cost of developing a tool that helps to keep the UML models synchronized with the code.

Reverse engineering is a complex task. We described some challenges we met during transforming a PSM to another PSM. Some of them are specific to the Erlang programming language, while an interesting issue was the difficulty to distinguish between aspects of the component framework that are platform-independent (and hence may be realized in other platforms without further changes) and those that are platform dependent, where a platform in this context is OTP. The Translator gives a PSM that is structurally complete, but transformation to a structurally complete PIM should be done manually by developing a model for the component framework that is platform independent.

Another important issue is the difficulty to extract behavior and constraints automatically from the code. We could draw sequence diagrams manually by using the code, but they can't be used by Rose (or any other tool) to generate code in other programming languages. Therefore we can't develop a computationally complete PIM or PSM.

The next steps in the study may be:

1) Study the possibility to develop a platform independent model for the component framework, and a Platform Description Model (PDM) that describes the framework realization.
2) Study the possibility to extract objects from the developed PIM (in the design model) to have a complete object-oriented class diagram. Neither Erlang nor C is object-oriented languages, while future development may be object-oriented.
3) Develop a similar translator for the C language.

Developing legacy wrappers is another approach when integrating legacy systems, which is not evaluated in this case and may be subject of future studies.

The study helped us to better understand the MDA approach to software development and to identify the problems and opportunities with the approach. Although organizations may find it difficult to use the MDA approach for their legacy systems, some aspects of the approach may already be integrated into their current practice.

Acknowledgement

We thank Ericsson in Grimstad for the opportunity to perform the case study.

References

[1] L. Ekeroth, P.M. Hedstrom, "GPRS Support Nodes", Ericsson Review, No. 3, 2000, 156-169.

[2] For more details on Erlang and OTP, see www.erlang.se

[3] The INCO (INcremental and Component-based development) project is a Norwegian R&D project in 2001-2004: http://www.ifi.uio.no/~isu/INCO/

[4] ISO, RM-ODP [X.900] http://www.community-ML.org/RM-ODP/

[5] MDA Guide V1.0: http://www.omg.org/docs/omg/03-05-01.pdf

[6] P. Mohagheghi, R. Conradi, "Experiences with Certification of Reusable Components in the GSN Project in Ericsson, Norway", Proc. 4th ICSE Workshop on Component-Based Software Engineering: Component certification and System Prediction, ICSE'2001, Toronto, Canada, May 14-15, 2001, 27-31.

[7] Rational Unified Process: www.rational.com

[8] Selo, Warsun Najib, "MDA and Integration of Legacy Systems", MSc thesis, Agder University College, Norway, 2003.

Handling QoS in MDA: a discussion on availability and dynamic reconfiguration[1]

João Paulo Almeida[a], Marten van Sinderen[a], Luís Ferreira Pires[a] and Maarten Wegdam[a, b]

[a]*Centre for Telematics and Information Technology, University of Twente*
PO Box 217, 7500 AE, Enschede, The Netherlands
[b]*Lucent Technologies, Bell Labs Advanced Technologies EMEA Twente*
Capitool 5, 7521 PL, Enschede, The Netherlands

Abstract. In this paper, we discuss how Quality-of-Service (QoS) can be handled in the Model-Driven Architecture (MDA) approach. In order to illustrate our discussion, we consider the introduction of availability and dynamic reconfiguration QoS concepts at platform-independent level. We discuss the consequences of the introduction of these concepts in terms of the realizations of platform-independent models in different platforms. The platforms considered provide varying level of support for the QoS concepts introduced at the platform-independent level.

1 Introduction

There is a general agreement that distributed applications and services only achieve their desired impact if they properly cope with Quality-of-Service (QoS) issues such as performance and availability. In order to enable that, QoS issues should be addressed throughout a service's development life cycle. In this paper, we discuss how QoS can be handled in the Model-Driven Architecture (MDA) approach [9].

The concept of platform-independence plays a central role in MDA development. Platform-independence is a quality of a model that indicates the extent to which the model relies on characteristics of a particular platform. A consequence of the use of platform-independent models (PIMs) to specify a design is the ability to refine or implement a design on a number of target platforms. Platform-specific designs are specified through platform-specific models (PSMs).

For the purpose of this paper, we assume that services are ultimately realized in some specific object- or component-middleware technology, such as CORBA/CCM [10], .NET, and Web Services. Ideally one could strive for PIMs that are absolutely neutral with respect to all different classes of middleware technologies. However, at a certain point in the development trajectory, different sets of platform-independent modelling concepts may be used, each of which is needed only with respect to specific classes of target middleware platforms.

In this paper, we motivate the introduction of QoS concepts at the platform-independent level; we present *availability* as a QoS characteristic to be considered at the platform-independent level, as well as *dynamic reconfiguration* as a means to satisfy availability QoS constraints. We present some consequences of the introduction of these concepts in terms of the realizations of platform-independent models in different middleware platforms. For that, we consider platforms that provide varying levels of support for dynamic reconfiguration.

2 The Need for QoS Concepts for Platform-independent Design

Awareness of the qualitative aspects of a service starts in the initial phases of its design, when the service designer states the qualitative properties required from the service, e.g., that the service should support a certain level of availability and should perform according to certain temporal constraints. Since services should be specified at a level of abstraction at which the supporting infrastructure is not considered, service specifications are middleware-platform-independent by definition.

In order to illustrate our discussion, let us consider a groupware service that facilitates the interaction of users residing in different hosts. Initially, the service designer states QoS properties that are to be satisfied by the service. At subsequent stages of the design trajectory, the designer is confronted with design decisions. In the design of the groupware service, we consider the following

[1] This work is partly supported by the European Commission in context of the MODA-TEL IST project (http://www.modatel.org) and the Telematica Instituut in the context of the ArCo project (http://arco.ctit.utwente.nl/).

alternatives: (i) a centralized (server-based) design, and (ii) a distributed (peer-to-peer) design. Figure 1 depicts these two solutions. In solution (i), a server facilitates the interaction between users. In solution (ii), symmetric components facilitate the interaction without a centralized application-level component.

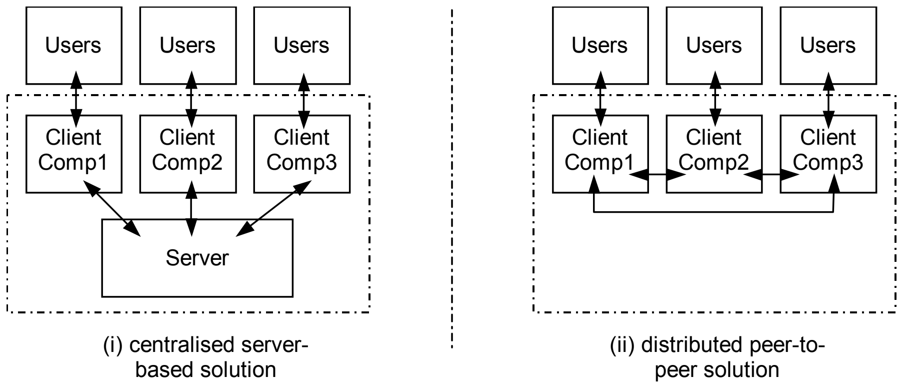


Figure 1 Alternative designs for the groupware service

Ideally, it should be possible to capture stable aspects of a system's architecture in a platform-independent manner. Therefore, it would be desirable to select between alternative designs (i) and (ii) during platform-independent design. Nevertheless, platform-specific aspects such as the supported distribution transparencies (as defined in the Reference Model for Open Distributed Processing (RM-ODP) [6]) play an important role in the selection of an adequate architecture. For example, in case the platform provides support for replication transparency, solution (i) would not introduce a single point of failure in the architecture, and therefore would be acceptable as an alternative for the implementation of a highly available service.

Apparently, this places the designer in a dilemma, since platform selection would affect platform-independent design for the qualitative aspects. In order to solve this dilemma, QoS-aware MDA should allow the designer to express, at platform-independent level, (QoS) requirements on platform-specific realizations. These requirements should guide and justify design decisions at a platform-independent-level and provide input for platform selection.

3 Selection of Concepts for Platform-independent Design

QoS constraints may be satisfied by QoS mechanisms that may be implemented in the application and in target middleware platforms. Ideally, application developers should profit from the provision of distribution transparencies as a means to satisfy QoS constraints, shifting complexity from the application design to the platform.

This applies both at the platform-specific and platform-independent levels. At platform-independent level, these transparencies apply to what we call an *abstract platform*. The choice of abstract platform defines which (platform-independent) properties or aspects are actually considered and which (platform-specific) properties or aspects are abstracted from in a platform-independent design.

A platform-independent design relies on the (platform-independent) concepts provided by an abstract platform in an analogous way as a platform-specific design relies on platform concepts. In order to expose this relative notion of platform, we prefer the term abstract platform rather than the more general terms "meta-model" or "concept space", as adopted in [4]. In order to define an abstract platform, one must carefully observe:

1. *Portability requirements for the platform-independent design*. The abstract platform should be generic enough to allow a mapping to different target platforms.
2. *The needs of application designers*. The abstract platform should provide concepts and facilities that ease platform-independent design, e.g., by providing required or desirable distribution transparencies at platform-independent level.
3. *The extent to which abstract platform and concrete target platforms differ. The* gap between abstract platform and concrete platforms has direct consequences for the mappings between platform-independent and platform-specific design.

Figure 2 illustrates the factors that influence the choice of abstract platform.

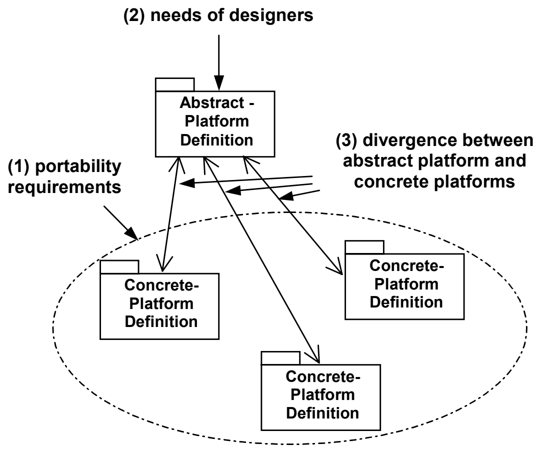

(2) needs of designers
Abstract - Platform Definition
(1) portability requirements
(3) divergence between abstract platform and concrete platforms
Concrete-Platform Definition
Concrete-Platform Definition
Concrete-Platform Definition

Figure 2 Forces in the choice of abstract platform

The forces exercised by factors (2) and (3) are often contradictory:

- Raising the provided support to observe the needs of designers may increase the gap between the abstract platform and concrete platforms. This is the case, for example, for the support of replication transparency [6] in the abstract platform, when a target platform has no support for the replication of components. By introducing replication transparency at platform-independent level, dealing with replication is deferred to platform-specific realization.
- Reducing the gap between support provided by the abstract platform and concrete platforms may lead to an abstract platform that handicaps the designer. This is the case, for example, for a “minimal” abstract platform that supports a common denominator of a broad class of middleware platforms such as point-to-point one-way message exchange. Patterns such as request/response and multicast message exchange, when necessary, are expected to be addressed by application developers in the platform-independent design of the application.

Differences in the architectural concepts used to build platform-independent designs and those concepts supported by the target platform may result in the use of intricate combinations of constructs in the platform-specific design. This may have an impact on the complexity of the mapping between platform-independent and platform-specific design and on some quality attributes of platform-specific design. It is questionable whether in case of really disparate abstract and concrete platforms, mappings are even feasible or can provide platform-specific designs with appropriate quality properties. For example, these mappings may sacrifice traceability from corresponding platform-independent designs, as well as intuitiveness for developers that are accustomed to a particular concrete platform.

Narrowing the gap between an abstract platform and concrete platforms is a challenging activity. Introducing new concrete platforms because of (unpredicted) changes in portability requirements may mean that the gap between the abstract platform and the newly introduced concrete platform is large. Besides that, narrowing the gap between an abstract platform and a particular concrete platform may enlarge the gap between the abstract platform and other concrete platforms.

4 Availability and Dynamic Reconfiguration

In the following, we consider availability as an example QoS characteristic, defined as the percentage of time that the system under consideration functions without disruptions (due to, e.g., faults or planned upgrades), the mean time between disruptions and the mean time to repair [14]. We also consider *dynamic configuration* as a means to satisfy availability QoS constraints. .

The aim of dynamic reconfiguration [3, 7, 8, 14] is to allow a system to evolve incrementally from one configuration to another at run-time. Dynamic reconfiguration exploits parallelism to improve a system’s overall availability. While certain activities of a system are affected during reconfiguration, other activities are left unaffected. Developing systems that can be dynamically reconfigured is a complex task, since a developer must ensure that dynamic reconfiguration results in a correct and useful system.

Reconfiguration is specified in terms of entities and operations on these entities. In this paper, we focus our attention on component replacement and migration. *Component replacement* allows one version of a component to be replaced by another version, while preserving component identity. We use the term version of a component to denote a set of implementation constructs that realizes the component. The new version of a component may have functional and QoS properties that differ from the old version. Nevertheless, the new version of the component should satisfy both the functional and QoS requirements of the environment in which the component is inserted. *Component migration* means that a component is moved from its current node to a destination node. Migrations of components can, e.g., be necessary when a certain node has to be taken offline.

A system evolves incrementally from its current configuration to a resulting configuration in a *reconfiguration step*. A reconfiguration step is perceived as an atomic action from the perspective of the application. We distinguish between simple and composite reconfiguration steps. A *simple reconfiguration step* consists of the execution of a reconfiguration operation that involves a single component. A *composite reconfiguration step* consists of the execution of reconfiguration operations involving several components. Composite steps are often required for reconfiguration of sets of related components. In a set of related components, a change to a component *A* may require changes to other components that depend on *A*'s behavior or other characteristics.

Support for dynamic reconfiguration can be clearly related to availability requirements when we consider disruptions (downtimes) due to upgrades of a system. A system without support for dynamic reconfiguration would typically be taken off-line a number of times during its lifetime for upgrades, causing downtimes. These downtimes could be avoided with dynamic reconfiguration.

4.1 Reconfiguration Transparency

Dynamic reconfiguration interferes with system activities and, therefore, requires special attention from the perspective of run-time reconfiguration management [7]. A system can become useless in case the preservation of consistency is ignored. The system under reconfiguration must be left in a "correct" state after reconfiguration. In order to support the notion of correctness of a distributed system, three aspects of correctness requirements are identified [8]. This notion of correctness is addressed in several dynamic reconfiguration approaches described in the literature (e.g., [7, 8, 14]).

From the perspective of application developers, platforms should ideally provide *reconfiguration transparency* [14]. The objective of reconfiguration transparency is to mask, from an application component and from its environment, the ability of a system to execute reconfiguration steps involving the component. Reconfiguration transparency hides from application developers the details and differences in mechanisms used to overcome the difficulties introduced by reconfiguration.

4.2 Dynamic Reconfiguration Concepts for Platform-independent Design

We introduce dynamic reconfiguration concepts in a platform-independent design by specializing the notion of a component to include the distinction between *reconfigurable* and *non-reconfigurable* components. Reconfigurable components can be *migrateable*, *replaceable* or both *migrateable* and *replaceable*. This allows a designer to establish these distinctions at a platform-independent level, specifying which components may be manipulated by reconfiguration operations in reconfiguration steps.

A (composite) reconfiguration step is specified by a set of simple reconfiguration steps. The definition of a replacement reconfiguration step identifies a component to be replaced and establishes its new version. The definition of a migration reconfiguration step identifies a component to be migrated and establishes its new location. Reconfiguration steps are committed to and handled by a reconfiguration manager component entailed by the abstract platform.

5 Platform-specific Realization

Platform-specific realization may be straightforward when the selected concrete platform corresponds (directly) to the abstract platform. When this is not the case, more effort has to be invested in platform-specific realization. In general, we distinguish two contrasting extreme approaches to proceeding with platform-specific realization:

1. *Adjust the concrete platform*, so that it corresponds directly to the abstract platform. This may imply the introduction of platform-specific (QoS) mechanisms, possibly defined in terms of

internal components of the concrete platform. Since modifying a concrete platform is typically not feasible (e.g., re-implementing the CORBA ORB to match the abstract platform would be too expensive), extension of this platform in a non-intrusive manner is often the preferred way to adjust the concrete platform. In this approach, the boundary between abstract platform and platform-independent design is preserved in platform-specific design.

2. *Adjust the platform-specific design of the application,* to preserve requirements specified at platform-independent level. This may imply the introduction of (QoS) mechanisms in the platform-specific design of the application. This approach may be suitable in case it is impossible to adjust the concrete platform, e.g., due to the lack of extension mechanisms and/or the cost implications of these adjustments.

Approaches to realization that adjust both concrete platform and the platform-specific design of the application are positioned somewhere between these two extreme approaches.

Different middleware platforms provide varying levels of support for dynamic reconfiguration with varying levels of transparency. In order to discuss the consequences of the introduction of reconfiguration transparency at the platform-independent level for platform-specific realization, we consider the following platforms: CORBA enhanced by portable extensions with the Dynamic Reconfiguration Service (DRS) [1, 14]; and, CORBA with compliance to the Online Upgrades (Draft Adopted) Specification [11].

5.1 CORBA + DRS

The Dynamic Reconfiguration Service we have proposed in [1, 14] provides reconfiguration transparency for CORBA application objects, supporting both simple and composite reconfiguration steps. The DRS has been implemented by extending CORBA implementations through the use of portable interceptors, which are standardized extension mechanisms for CORBA ORB implementations [10]. In this realization, there is a direct correspondence between the DRS-enabled CORBA platform and the abstract platform, illustrating approach 1 to realization.

5.2 CORBA with compliance to the Online Upgrades

The Draft Adopted Specification for Online Upgrades [11] provides interfaces to manage the upgrading of the implementation of a single CORBA object instance. The specification allows these interfaces to be used to upgrade multiple CORBA object instances, but provides minimal mechanisms to coordinate the upgrading of multiple CORBA object instances.

This means that there is a compromise to be made with respect to support for composite reconfiguration steps. If composite reconfiguration steps are supported by the abstract platform, then realization on top of CORBA with online-upgrades is impaired. Nevertheless, a subset of the platform-independent designs can be mapped directly, namely the ones that do not use composite reconfiguration steps. Therefore, we could have considered an alternative abstract platform that provides support for simple reconfiguration steps only. This abstract platform would be defined as a “subset” of the platform that supports composite reconfiguration steps. This latter realization illustrates again approach 1.

The differences between the platforms could be reconciled by following approach 2, where transformations would introduce mechanisms at the application-level to coordinate the upgrading of multiple CORBA object instances. This would, however, sacrifice reconfiguration transparency at platform-specific level.

6 Conclusions

Since platform selection would affect platform-independent design for the qualitative aspects, QoS-aware MDA should allow designers to express, at platform-independent level, QoS requirements on platform-specific realizations. This is done through the definition of an abstract platform, which determines which (QoS) properties or aspects are actually considered and which are abstracted from in a platform-independent design.

Differently from the “traditional” discussion that couples levels of transparency to the support provided by middleware platforms, in MDA, platform-independent models are “decoupled” from their corresponding platform-specific counterparts by mappings. This adds a new dimension to the discussion on the level of support provided by a platform or located in the application. There is some degree of freedom between the selection of transparencies for the abstract platform and the provision of transparency for the concrete platform. In this paper, we identify three factors that should be observed when defining an abstract platform. We also discuss, in a general sense, the implications of choosing the level of transparency of the abstract platform for the mapping on the concrete target platforms.

As an example, we have considered availability as QoS characteristic, and dynamic reconfiguration as a means to satisfy availability QoS constraints. Dynamic reconfiguration concepts have been introduced in a platform-independent design and two contrasting approaches to platform-specific realization have been considered. These approaches are further illustrated using two target platforms (CORBA + DRS, CORBA + Online Upgrades) with different levels of reconfiguration transparency.

The work presented in this paper is related to the OMG-promoted work on MDA core technologies, such as UML and extensions, MOF, etc. These technologies include, more recently, QoS meta-modelling techniques and UML language extensions, such as in [5], QML and responses to OMG RFP on Modeling QoS and FT Characteristics and Mechanisms [12]. In this paper we have addressed the conceptual aspects, abstracting from language aspects. Platform-independent designs must be specified in suitable modelling languages, therefore these efforts are complementary to the conceptual discussion presented in this paper.

References

[1] J. P. A. Almeida, M. Wegdam, M. van Sinderen, L. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA, *Proc. 3rd Intl. Symposium on Distributed Objects & Applications (DOA 2001)*, Rome, Italy, Sept. 2001.

[2] J. P. Almeida, M. van Sinderen, L. Ferreira Pires, D. Quartel, “A systematic approach to platform-independent design based on the service concept”, *Proc. 7th Intl. Conf. on Enterprise Distributed Object Computing (EDOC 2003)*, Brisbane, Australia, Sept. 2003, to appear.

[3] C. Bidan, V. Issarny, T. Saridakis, A. Zarras. A dynamic reconfiguration service for CORBA, *Proc. IEEE Intl. Conf. on Configurable Distributed Systems*, May 1998.

[4] D. Exertier, O. Kath and B. Langois. PIMs Definition and Description to Model a Domain. MASTER IST-project D2.1, Dec. 2002.

[5] A.T. van Halteren, *Towards an adaptable QoS aware middleware for distributed objects*, Ph.D. Thesis, University of Twente, Enschede, The Netherlands, 2003.

[6] ITU-T / ISO, *Open Distributed Processing - Reference Model - Part 3: Architecture*, ITU-T X.903 | ISO/IEC 10746-3, Nov. 1995.

[7] J. Kramer, J. Magee. Dynamic configuration for distributed systems. *IEEE Trans. on Software Engineering* 11(4), April 1985, pp. 424-436.

[8] K. Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems*. Ph.D. thesis, Imperial College, London, UK, March 1999.

[9] Object Management Group, *Model driven architecture (MDA)*, ormsc/01-07-01, July 2001.

[10] Object Management Group, *Common Object Request Broker Architecture: Core Specification*, Version 3.0, formal/02-12-06, Dec. 2002.

[11] Object Management Group, *Online Upgrades Draft Adopted Specification*, ptc/02-07-01, Jul. 2002.

[12] Object Management Group. *UML Profile for Modeling QoS and FT Characteristics and Mechanisms RFP*, ad/02-01-07, February 2002.

[13] L. A. Tewksbury, L. E. Moser, P. M. Melliar-Smith, Coordinating the Simultaneous Upgrade of Multiple CORBA Objects, *Proc. 3rd Intl. Symp. on Distributed Objects and Applications (DOA 2001)*, Rome, Italy, Sept., 2001.

[14] M. Wegdam, *Dynamic Reconfiguration and Load Distribution in Component Middleware*, Ph.D. Thesis, University of Twente, The Netherlands, 2003.

An MDA–Based Approach to Managing Database Evolution (position paper) [⋆]

Eladio Domínguez [a] Jorge Lloret [a] Ángel L. Rubio [b]
María A. Zapata [a]

[a] *Dpto. de Informática e Ingeniería de Sistemas. Facultad de Ciencias. Edificio de Matemáticas. Universidad de Zaragoza. 50009 Zaragoza. Spain.*

[b] *Dpto. de Matemáticas y Computación. Edificio Vives. Universidad de La Rioja. 26004 Logroño. Spain*

1 Introduction

In this paper we are going to present the status of an ongoing research about the relationships between a particular architecture [5] that tackles with a concrete problem related to database technology –the problem of database evolution- and the Model-Driven Architecture (MDA [14]).

The requirements of a database do not remain constant during its life time and therefore the database has to evolve in order to fulfil the new requirements. Since database evolution activities consume a large amount of resources [13] they are considered of great practical importance and, as a consequence, much research has been focused on analyzing ways of facilitating this task [2,16]. In particular, among the several problems that are related to evolution activities (see [11]), one of the most important is that of 'forward database maintenance problem' (or 'redesign problem', according to [16]). This problem faces how to reflect in the logical and extensional schemata the changes that have occurred in the conceptual schema of a database. As a contribution towards achieving a satisfactory solution to this problem (that has not been found yet, despite a lot of efforts by different researchers [16,13]), some of the authors of the present paper have presented in [5] an architecture for managing database evolution.

On the other hand, the MDA is an initiative led by the Object Management Group (OMG) that embodies "the expanded vision necessary to support interoperability with specifications that address integration through the entire systems life cycle: from business modeling to system design [...] and evolution"[14]. Therefore, although MDA deals mainly

[⋆] This work has been partially supported by DGES, projects TIC2000-1368-C03-01, TIC2002-01626, and by Ibercaja-University of Zaragoza, project IB 2002-TEC-03

Email addresses: noesis@posta.unizar.es (Eladio Domínguez), jlloret@posta.unizar.es (Jorge Lloret), arubio@dmc.unirioja.es (Ángel L. Rubio), mazapata@posta.unizar.es (María A. Zapata).

with software development and less with data modeling issues, it is implied explicitly in evolution tasks. We have found out that several key features of MDA are shared by our above-mentioned architecture for managing database evolution. Firstly, the notion of mapping, used in MDA in order to transform Platform Independent Models (PIMs) into Platform Specific Models (PSMs) –and backwards-, is also a central point when dealing with database modeling and evolution. Secondly, both approaches made an intensive use of metamodels in order to represent 'modeling knowledge'.

In this paper we make a proposal of how to get both worlds (database evolution and MDA) closer. The main aim of the work is to investigate how to take advantage of the advances in the MDA field (in particular the development of MDA-based tools) in order to use them in a databases evolution context. The remainder of the paper is organized as follows. Section 2 explains our view of the relationships between database engineering and MDA, presenting in Section 3 the specific relationships between MDA and our architecture for managing database evolution. Finally, some conclusions are outlined in Section 4.

2 Database Engineering and MDA

Several recent papers have already addressed some similarities between database technology and MDA concepts (see [15,10]). However it is far from being clear how should database engineering concepts be reinterpreted in terms of MDA.

Traditionally, in the databases field, the term 'conceptual schema' refers to a model that captures the user's information requirements, for example by means of an Entity/ Relationship (ER) Model. Following the ANSI-SPARC Architecture [3], a conceptual schema is independent of any physical implementation (and of any Database Management System, DBMS). Moreover, a conceptual schema is independent of any computational aspects. Because of that, we think that the 'conceptual schema' idea corresponds with the MDA 'Business Model' concept. Unfortunately, the MDA Specification Document leaves out Business Models quickly, and focuses its attention on PIMs and PSMs, so much so that Business Models are omitted from the MDA Metamodel Description (page 12 of [14]). We advocate for making use of the footnote on page 7 of this Document, where it is said that "while [Business Model] need not be explicitly present in a particular usage of the MDA Scheme, MDA accommodates it consistently in the same overall architecture".

The next step in a database development process is to obtain, starting from the conceptual schema, a 'logical schema' that is described using the (Object-) Relational Model. The logical schema is closer to computational aspects, but it is not dependent on any particular DBMS. Therefore, a logical schema can be seen as a Platform Independent Model from the point of view of MDA. This idea is strengthened by analyzing one of the core specifications of the MDA, the Common Warehouse Metamodel (CWM [4]). This specification includes a metamodel for Relational data resources, that is based in the SQL Standard [12]. However, as it has been proven in the literature [18,17], each particular DBMS implements its specific version of SQL, in such a way that a (object-) relational schema that it is assumed to be in accordance with the standard, can be not valid for a particular DBMS. This is the main reason because we think that a description of a 'logical schema' based on the (object-) relational approach and/or the standard SQL should be considered as a 'Platform Independent Model', and that a description of that schema using the particular version of SQL of a particular DBMS should be considered as a 'Platform Specific Model'. These ideas are strengthened by the functionality of different software tools that are used to automatize the data modeling process, such as DB-MAIN [11]. A user of this tool can draw a conceptual (ER) schema, that is automatically translated to a logical (relational) schema. Then, the tool allows the user to guide the translation process of this logical schema towards SQL code adapted to a particular platform (DBMS), such as Oracle.

In order to illustrate graphically these ideas, in Figure 1 we show a modification of the MDA Metamodel Description from the one presented in [14]. There are three basic differences between the original description and our proposal. Firstly, Business Models are explicitly included as a new (meta)class, which leads to include two associations between 'Business Model' and 'PIM' classes. Secondly, an association class 'BM-PIM Mapping Techniques' is included, in order to represent the translation modeling knowledge from Business Models (conceptual schemas in the database context) to PIMs (logical schemas). It seems surprising to us that in the original MDA Metamodel Description there are explicit association classes to represent 'PSM Mapping Techniques' (from PSM to PSM) and 'PIM Mapping Techniques' (from PIM to PIM), but there not exists analogous association classes to represent 'PIM to PSM Mapping Techniques' or 'PSM to PIM Refactoring Techniques'. Our proposed inclusion of the class 'BM-PIM Mapping Techniques' (that in our opinion it is essential, at least in the database context), makes us think about the necessity of other classes (like, for instance, 'PIM-BM Refactoring Techniques', useful for database reverse engineering). Lastly, our metamodel description includes a 'BM Mapping Techniques' association class, which is a key point in a database evolution setting, since, as we have said before, in the forward database maintenance problem the changes in the conceptual schema (Business Model) are taken into account to determine changes in the logical and extensional schemata.

We recognize that this is not the only possible interpretation. There exists some controversy in the literature about the use of ER schemata as Business Models [9]. As another example, in [10] a different approach is considered, since ER schemata are identified as PIMs, and relational schemata are identified as 'prototypical' PSMs. Moreover, relational schemata are used somehow as Business Models in [1]. However we think that all these approaches are not mutually exclusive, since it is a matter of 'level of detail'. Different kinds of ER schemata can be used at different levels of abstraction and with different levels of detail. In fact, this is likely quite near of the MDA vision, since it defines concepts such as 'abstraction', 'refinement' and 'viewpoint'.

3 Database evolution and MDA

In this section we are going to outline the basic characteristics of our proposed architecture for managing database evolution (for details, see [5]). Afterwards, we are going to show which are the relationships between this architecture and our interpretation (that has been described in the previous section) of the database concepts in terms of MDA.

Our proposed architecture for managing database evolution makes use of a structural artifact that consists of three components: an information schema, an information base and an information processor. The information schema defines all the knowledge relevant to the system (and therefore it plays a 'metamodel' role), the information base describes the

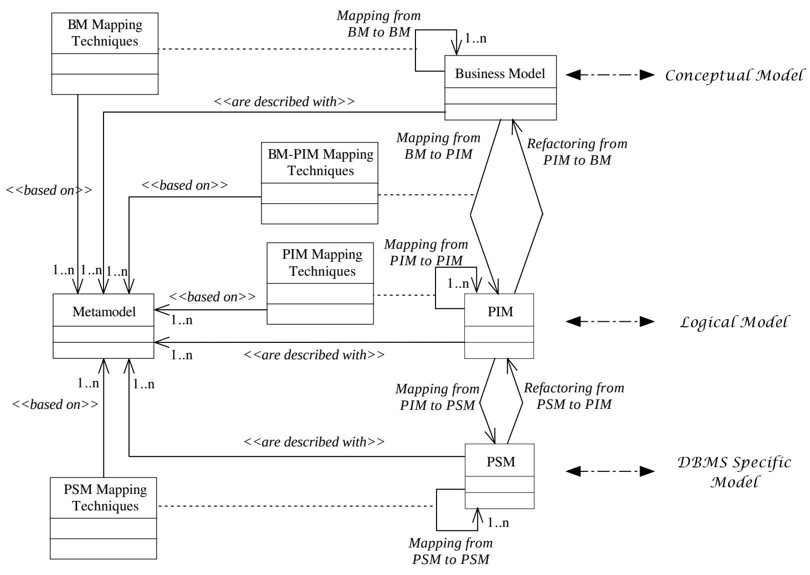


Fig. 1. MDA Metamodel Description (modified from [14]) and its relation with database concepts

specific objects perceived in the Universe of Discourse (a 'model' role), and the information processor receives messages reporting the occurrence of events in the environment. In order to respond to the events received, the information processor can send structural events towards the information base and/or towards the information schema and can generate internal events that inform other processors of the changes performed in it. This structural artifact is used within our architecture giving rise to four structures which are used to store, respectively, the conceptual modeling knowledge, the translation process, the logical modeling knowledge and the extension. The corresponding components of each one of these structures as well as the way in which they are related appear in Figure 2. The name of each one of these components has been modified in an attempt to capture the type of knowledge that they store.

There are two main characteristics of this architecture that make it different of other proposals. On the one hand it includes an explicit translation component that stores information about the way in which a concrete conceptual database schema is translated into a logical schema. This component plays an important role in enabling the automatic propagation of evolution from the conceptual to the extensional schemata. On the other hand, a meta–modeling approach [6] has been followed for the definition of the architecture. Within this architecture, three meta–models are considered which capture, respectively, the conceptual, logical and translation modeling knowledge.

There are several relationships between this architecture and MDA concepts. First of all, both make a explicit use of metamodels. For example, metamodels are used within the architecture in order to check the validity of events issued from the environment to carry out evolution tasks. Second, the translation information is stored explicitly in our architecture, so that it embeds a certain notion of 'mapping'. This component is essential in our database evolution architecture, because when a modification of the conceptual schema is carried out, a new set of elementary translations is determined without it being necessary to apply once again the translation algorithm from scratch. Last, the consideration of MDA concepts have made us to think about the relationship between the logical and extensional information systems of our architecture. In [5] we said that "the logical database schema can be seen as the information base of the logical information system or as the information

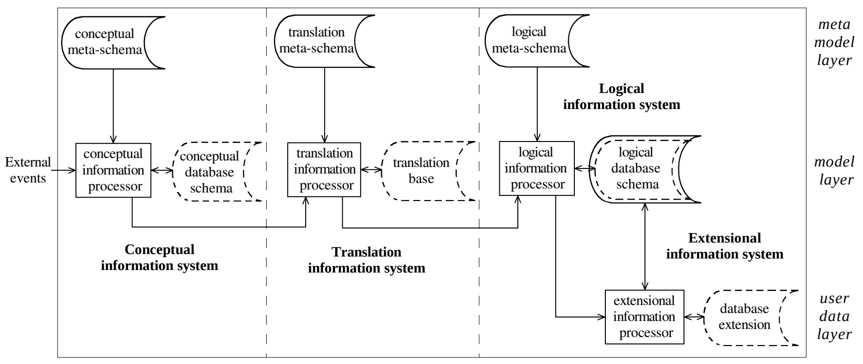

conceptual meta-schema
translation meta-schema
logical meta-schema
meta model layer
Logical information system
External events
conceptual information processor
conceptual database schema
translation information processor
translation base
logical information processor
logical database schema
model layer
Extensional information system
Conceptual information system
Translation information system
extensional information processor
database extension
user data layer

Fig. 2. Architecture for Database Evolution

schema of the extensional one. For this reason two different components of our architecture store the same information". However, this situation led us to define in our architecture "some rules, called *correspondence rules* (in the same sense as in [13]). These rules govern the correspondence between the elements of each one of the two components". The MDA notions suggest that these 'correspondence rules' must play a more relevant role, since they are acting as 'PIM to PSM Mapping'. We think that this point needs of further research.

Finally, it is necessary to point out that we have developed an implementation of the architecture that has allowed us to test its functionality. This implementation is based on the RDBMS Oracle 8i and the Programming Language PL/SQL. In this implementation we use the DBMS as a kind of 'MOF Repository', in a very similar way as described in Chapter 9 of [8]. In particular, Frankel says in that Chapter that "to a MOF Repository, transformations rules are just another kind of metadata that it manages according to the general pattern", and that "one strategy for producing the code to execute the transformations is to create a generator that reads a set of M1 [Model Layer] transformation rules and generates the transformation code that executes the rules on M0 [User-Data Layer] Data". This description corresponds quite accurately with the behavior of our current implementation.

4 Conclusions

In this paper we have shown some ideas of an ongoing research about relating MDA and a particular architecture for managing database evolution. We think that the database evolution context can be an interesting and valid application area for MDA, and we have seen several points where both approaches possibly can benefit from each other:

- The experience accumulated along the years by database researchers and practitioners in the fields of schemata translation and evolution can be useful to make advances in the MDA field. For example, the interpretation of our database evolution architecture in MDA terms has led us to propose an enhancement of the MDA metamodel.
- The other way round, taking the MDA ideas into account in the database context can give place to introduce new viewpoints on current database approaches. In our case, the use of the MDA approach has suggested us the need of further research about the notion of 'correspondence rules' within our evolution architecture.
- The database evolution context can take advantage of the advances in the MDA field. In particular, we see the foreseeable development of MDA-based tools as an opportunity for database researchers and practitioners.

References

[1] Serge Abiteboul, Victor Vianu, Brad Fordham, Yelena Yesha, Relational Transducers for Electronic Commerce, *Journal of Computer and System Sciences*, Vol. 61, N. 2, 2000, 236–269.

[2] L. Al-Jadir, M. Léonard, Multiobjects to Ease Schema Evolution in an OODBMS, in T. W. Ling, S. Ram, M. L. Lee (eds.), *Conceptual modeling, ER-98*, LNCS 1507, Springer, 1998, 316–333.

[3] ANSI/SPARC Report, *ACM SIGMOD Newsletter*, Vol. 7, N. 2, 1975.

[4] Common Warehouse Metamodel Specification Version 1.1, OMG Document formal/03-03-02.

[5] E. Domínguez, J. Lloret, M. A. Zapata, An architecture for Managing Database Evolution, *Proceedings of ER 2002 Workshop on Evolution and Change in Data Management*, To appear in LNCS, 2002, 64–75.

[6] E. Domínguez, M. A. Zapata, J. J. Rubio, A Conceptual Approach to Meta–Modelling, in A. Olivé, J. A. Pastor (Eds.), *Advanced Information Systems Engineering, CAISE'97*, LNCS 1250, Springer, 1997, 319–332.

[7] R. A. Elmasri, S. B. Navathe, *Fundamentals of Database Systems (3rd ed.)*, Addison-Wesley, 2000.

[8] David Frankel, *Model Driven Architecture – Aplying MDA to Enterprise Computing*, Wiley Publishing, 2003.

[9] Michalis Glykas, George Valiris, Formal methods in object oriented business modeling, *Journal of Systems and Software*, Vol. 48, N. 1, 1999, 27–41.

[10] M. Gogolla, A. Lindow, M. Richters, P. Ziemann, Metamodel Transformation of Data Models, *Workshop in Software Model Engineering, Dresden, Germany, http://www.metamodel.com/wisme-2002/*, 2002.

[11] J. L. Hainaut, V. Englebert, J. Henrard, J. M. Hick, D. Roland, Database Evolution: the DB-MAIN approach, in P. Loucopoulos (ed.), *Entity-Relationship approach- ER'94*, Springer Verlag, LNCS 881, 1994, 112–131.

[12] ISO/IEC 9075-2: 1999, *Information Technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation)*, 1999.

[13] J. R. López, A. Olivé, A Framework for the Evolution of Temporal Conceptual Schemas of Information Systems, in B. Wangler, L. Bergman (eds.), *Advanced Information Systems Eng., CAiSE 2000*, Springer, LNCS 1789, 2000, 369–386.

[14] J. Miller, J. Mukerji (eds.), Model Driven Architecture (MDA), Object Management Group, Document number ormsc/2001-07-01, July 9, 2001.

[15] Bernard Morand, Models transformations: from mapping to mediation, *Workshop in Software Model Engineering, Dresden, Germany, http://www.metamodel.com/wisme-2002/*, 2002.

[16] A. S. da Silva, A. H. F. Laender, M. A. Casanova, An Approach to Maintaining Optimized Relational Representations of Entity-Relationship Schemas, in B. Thalheim (ed.), *Conceptual Modeling- ER'96*, Springer Verlag, LNCS 1157, 1996, 292–308.

[17] C. Turker, Schema Evolution in SQL-99 and Commercial (Object-) Relational DBMS. In: H. Balsters, B. De Brock, S. Conrad (eds.), *Database Schema Evolution and Meta-Modeling, Post-Proceedings of the 9th Int. Workshop on Foundations of Models and Languages for Data and Objects*, LNCS 2065, 2001, 1–32.

[18] C. Turker, M. Gertz, Semantic Integrity Support in SQL:1999 and Commercial (Object-) Relational Database Management Systems, *The VLDB Journal* Vol. 10, No. 4, 2001, 241–269.

A Model Driven Architecture for REA based systems

Signe Ellegaard Borch, Jacob Winther Jespersen,
Jesper Linvald, Kasper Østerbye*
IT University of Copenhagen, Denmark
* Corresponding Author (kasper@it-c.dk)

Abstract. An important aspect in the OMG model driven architecture is the separation of the platform independent model (PIM) and the platform specific model (PSM) of a system. This paper presents a system generator for a specific kind of accounting systems where the PIM is specified in a domain specific XML specification, and where the PSM is automatically generated by using platform dependent templates. One contribution of this paper is to describe how we have been working with the concepts of MDA in a domain specific context. The accounting systems we generate are based on the REA model of accounting proposed by William McCarthy. Another contribution of the paper is to show that one can generate parts of an accounting system based on REA specifications.

1 Introduction

The REA accounting model was first presented by William McCarthy [1], and later elaborated in numerous papers, foremost in the work of McCarthy and Geerts [2], and it has been a main inspiration for the ebXML standardization effort [5]. An important aspect of the REA model is its attempt to capture three important aspects of business modeling in one framework, namely accounting, inventory, and supply chain management. In this paper, however, our main emphasis is using REA as a meta-model. Our models based on the REA meta-model are expressed in XML, and plays the role of platform independent models. These models are translated into EJB code for a specific J2EE server using code-templates.

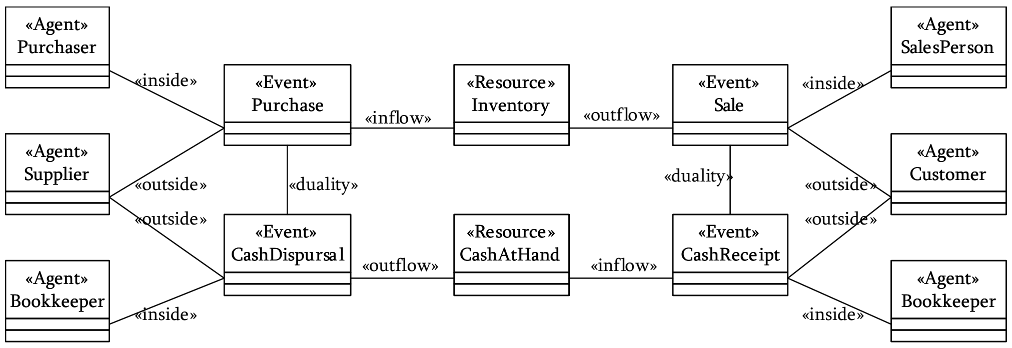
The choice of XML technology for specifying the platform independent model, rather than UML is deliberate. XML is widely supported by a range of transformation and verification tools, which unlike most UML transformation technology are freely available for the Java platform, or through open source projects.

The paper is structured as follows: First, we will briefly present the REA meta-model, and give a concrete example. Second, we will present the architecture of our implementation of the model driven architecture. Finally, we will conclude with some experiences and perspectives.

2 The REA meta-model

The REA model of accounting is a meta-model for building accounting systems, which is centered on registration of the basic economic events that occur within an organization. In its first incarnation, presented by William McCarthy in [1], the model consists of three basic meta-types, Resources, Events, and Agents. Each Event (such as a sale, payment, purchase, etc.), is related to a Resource (Inventory, Cash…) and two Agents, an internal Agent responsible for the Event, and an external Agent with whom the Event takes place. In [2], this basic model is extended to handle contractual issues as well.

A simple cycle shop that purchases bicycles and sells them again can be modeled as:



Besides the three REA entity meta-types, there is a number of association meta-types. Events are associated using a duality relation. This captures a simple exchange situation: When the customer receives a bicycle through a sales event, he must also be part in a cash receipt event (as the shop does not give away bicycles). For each event that is associated with a resource through an outflow association (meaning that the resource is decreased), its dual event must be attached to a resource through an inflow association.

In the above model, no cardinalities, attributes, or methods are described. Due to the fact that we are working in a domain specific context these aspects can be deduced from the model on the basis of the underlying ontology: In the simple model, each entity type has an attribute ID, Resources have an additional description, Agents have a name, and Events have a date, a quantity, and a value.

Resources have a static method to find out how much is in stock, which is the total quantity over its inflow events, minus the total quantity over outflow events. For a given instance of a Resource, e.g. men’s bike, black, Centurion, 7 gear basic, one can find the number in stock by looking only at inflow and outflow events that relate to that specific inventory item.

Similarly, one can find out how much money a given customer owes by calculating the value of all outflow events minus the value of all inflow events associated with a given customer.

Cardinalities are as follows: One Agent can participate in many Events. One Event has one internal and one external Agent. A Resource can participate in many Events; a given Event is only about one Resource[1]. The duality association is many to many, unless otherwise specified. Notice that these methods, attributes and cardinalities are given by the REA meta-model, and need not be specified in any concrete model.

3 Design and Implementation

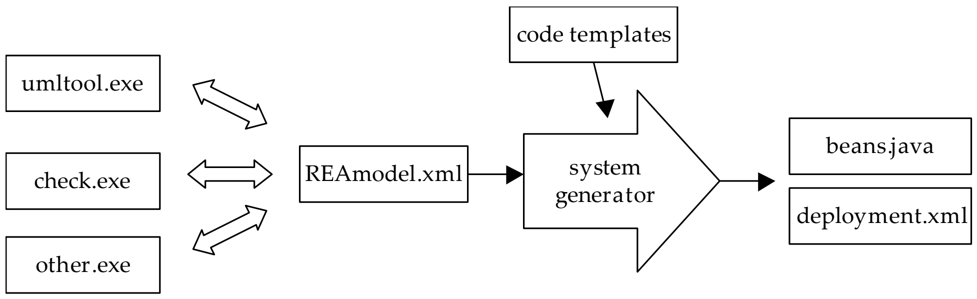
In the following we will describe an implementation of MDA on the basis of the REA meta-model.

3.1 Overall design

The overall design is illustrated in the figure below. The main component is the system generator, which based on the REA model and code templates generates the files needed to build the system.

[1] Obviously, a given sale can consist of several things, e.g. two bikes, one helmet, 4 lamps, and a bell. This multiplicity is specified at the contractual level, not at the operational level we present here.

The REAmodel.xml file describes the platform independent model in a simple XML format with tags representing the meta-model entities Resource, Event and Agent.



The system generator assumes the REA-model to be well formed, an assumption that is honored by the checking tool to the left. Leaving the checking tool out of the system generator has the advantage that the system generator becomes simpler, furthermore, the consistency of the REA meta-model has not yet been fully specified, and it is therefore a good idea to leave it as a separate tool. Furthermore, it is possible for a different team to work on the checker independently of the team working on the system generator.

We have chosen to use a simple customized XML Schema rather than building upon say XMI [9], which is clearly strong enough to express the models. The primary reason is that we wanted to be able to start designing without worrying about the inner workings of XMI. However, we are building an extension to RationalRose that will produce our REA-model from a UML class diagram, where each of the meta-types of REA is specified as UML stereotypes. We see UML as *one* important way to specify REA models, though we envision others. Thus, it is useful for us to keep a XML representation as the input to the system generator.

The system generator is the heart of the tool. It reads the REA-model, builds an internal intermediate model, and uses this intermediate model together with the templates to generate the java and XML files necessary for the system to run on the J2EE platform.

3.2 Model transformation

This section describes the process of transforming a REA model into a J2EE application through multiple automated transformations in more details.

The first transformation is carried out on the basis of the XML REA specification, which is parsed and mapped to a Java model. This model corresponds one-to-one with the XML REA specification. The purpose of this step is to provide a Java based specification of the PIM to give as input to the Velocity template engine [3].

The reason why we have chosen Velocity as our template engine instead of using other technologies such as XSLT is that it is Java-centric and therefore fits better into the Java environment we are working in. From the templates, we generate Bean classes with XDoclet [2] tags. This is the point where the mapping from the platform independent model to the platform specific model takes place.

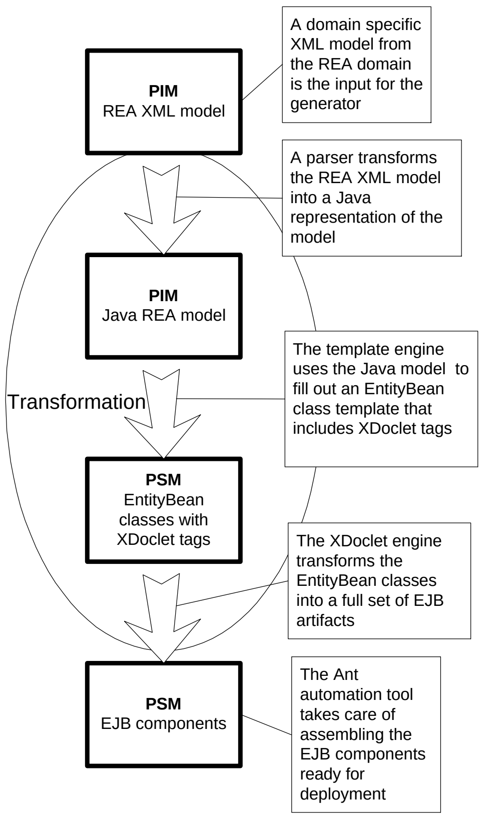
XDoclet generates the necessary interfaces and deployment descriptors to make up complete EJBs on the basis of the tagging done in the previous step. The reason for using XDoclet as an intermediary transformational step is to simplify code generation. As an alternative, we could have used multiple templates each corresponding to an EJB specific artifact (i.e. one template for the deployment descriptor, one for the LocalHome interface etc). The XDoclet approach has the advantage that we only have to maintain a single template for the components, which makes the template easier to debug.

The generated source files are finally packaged for deployment on a specific application server using the automation tool Ant [10].

Interpreting our approach to the terminology of the MDA guide [7], we first notice that neither the PIM nor the PSM are MOF based. The PSM has a reified meta-model in the form of an XML Schema, whereas there is no reified meta-model for the XDoclet output. We use a *direct transformation* approach c.f. [7, 3.7], which, despite of the non-reified PSM meta-model, is an example of a *meta-model transformation* [7, 3.10.2].

The fact that we are working with a domain specific PIM with rich semantics gives us the possibility of building in knowledge about the domain into the system generator.

This means that we do not have to annotate the PIM with additional information concerning business rules – for example on how to compute inventory stock or accounts receivable. Instead, we can rely on the fact that our transformation tool will recognize declarative statements in the PIM about which business rules to apply and translate them into the PSM. The MDA guide mentions *automatic translation* in [7, 4.1.4], which is what we do. However, our semantics are not defined using an action language, but is part of the REA meta-model itself.


PIM
REA XML model
A domain specific XML model from the REA domain is the input for the generator
A parser transforms the REA XML model into a Java representation of the model
PIM
Java REA model
Transformation
The template engine uses the Java model to fill out an EntityBean class template that includes XDoclet tags
PSM
EntityBean classes with XDoclet tags
The XDoclet engine transforms the EntityBean classes into a full set of EJB artifacts
PSM
EJB components
The Ant automation tool takes care of assembling the EJB components ready for deployment

4 Conclusions

This paper has described how we utilize a Model Driven Architecture in a domain specific environment. From a strictly declarative REA application specification (the PIM) our tool is able to create a Java Enterprise Application (the PSM) ready for deployment to a J2EE server. The user of the tool is not required to do manual coding.

As is apparent, we have focused on transformation of meta-models, i.e. we have devised a particular PIM-to-PSM mapping based on a typological view of the REA entities. This approach seems to work well in our case due to the common characteristics of our PIM elements; a commonality that is a result of the ontological foundation (REA) they share. The rich semantics in this foundation is what allows us to turn seemingly sparse specifications into meaningful applications in a fully automated transformation process.

The latter observation is in line with OMG's proclamation that semantically rich input models is an important condition to make full automation in the transformation a feasible approach [8], page 15. Another important condition mentioned is the independence of legacy, which also holds in our case.

The capabilities of the applications that our tool generates are currently limited to basic handling of the application data objects (the Resources, Events and Agents) and to carrying out simple business processing, e.g. the stock and accounts receivable calculations. Notably, our REA specifications do not currently include declarations concerning business rules, workflow, or information visualization (e.g. report generation), nor can they be marked up to control platform specific issues, e.g. to fulfill QoS requirements.

However, going forward it seems promising to extend the capabilities in these areas by applying the MDA pattern repeatedly to the transformation process. For example, when workflows are modeled generically in REA-based systems, such models can be merged with a PIM (constituting a PIM-to-PIM mapping) before going to the PSM.

In [6] the notion of Platform Description Model is presented. The MDA guide [7], only defines the term Platform Model. Our interpretation of Platform Model is that it is not a formal model, but is expressed at a conceptual level, whereas Bezivin and Ploquin's notion of PDM is a technical artifact. In our system templates play the role of PDM, and we believe that the system generator should be able to generate code for various platforms if equipped with a different set of templates, though that remains to be verified.

5 References

[1] William E. McCarthy. *The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment,* The Accounting Review (July 1982) pp. 554-78.

[2] William E. McCarthy, and Guido L. Geerts. *An Accounting Object Structure For Knowledge-Based Enterprise Model*, IEEE Intelligent Systems, July/August 1999 (pp. 89-94)

[3] Velocity Template Engine. http://jakarta.apache.org/velocity/

[4] XDoclet - http://xdoclet.sourceforge.net/

[5] ebXML (Electronic Business using eXtensible Markup Language). http://www.ebxml.org/

[6] Jean Bézivin and Nicolas Ploquin, Combining the Power of Meta-Programming and Meta-Modeling in the OMG/MDA Framework, OMG's 2nd Workshop on UML?for Enterprise Applications, San Francisco, USA, December 2001

[7] MDA Guide Version 1.0. Edited by Joaquin Miller and Jishnu Mukerji. http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf

[8] Model Driven Architecture (MDA). http://www.omg.org/docs/ormsc/01-07-01.pdf

[9] XML Metadata Interchange (XMI). http://www.omg.org/cgi-bin/doc?formal/2003-05-01

[10] The Apache Ant Project. http://ant.apache.org/

Model Driven Architecture as Approach to Manage Variability in Software Product Families

Sybren Deelstra, Marco Sinnema, Jilles van Gurp, Jan Bosch
Department of Mathematics and Computer Science, University of Groningen,
PO Box 800, 9700AV Groningen, The Netherlands,
{s.deelstra|m.sinnema|jilles|j.bosch}@cs.rug.nl, http://segroup.cs.rug.nl

Abstract

In this paper we portrait Model Driven Architecture (MDA) as an approach to derive products in a particular class of software product families, i.e. a configurable product family. The main contribution of this paper is that we relate MDA to a configurable software product family and discuss the mutual benefits of this relation. With respect to variability management, we identify two main benefits of applying MDA to product family engineering, i.e. postponing binding time and mechanism selection to application engineering, and the independent evolution of domain concepts, product family assets and the transformation technique. In addition, we identify variability management as a solution to the problem of round-trip transformation in MDA.

Keyword(s): Model Driven Architecture, Software Product Families, Variability Management

1. Introduction

The notion of software product families has received substantial attention during the 1990s, and has proven itself in a large number of organizations. Software product families are organized around a product family architecture that specifies the common and variable characteristics of the individual product family members, as well as a set of reusable components that implement those characteristics. Products within a product family are typically developed in a two stage process, i.e. a domain engineering stage and a concurrently running application engineering stage.

Domain engineering involves, amongst others, identifying commonalities and differences between product family members and implementing a set of shared software artifacts (e.g. components or classes) in such a way that the commonalities can be exploited economically, while at the same time the ability to vary the products is preserved. During application engineering individual products are derived from the product family, viz. constructed using a subset of the shared software artifacts. If necessary, additional or replacement product-specific assets may be created.

The ability to derive various products from the product family is referred to as variability. Managing the ability to handle the differences between products at various stages of development is regarded as a key success factor in software product families. Variability is realized through variation points, i.e. places in the design or implementation that are necessary to make functionality variable. Two important aspects related to variation points are binding time and realization mechanism. The term 'binding time' refers to the point in a product's lifecycle at which a particular alternative for a variation point is bound to the system, e.g. pre- or post-deployment. The term 'mechanism' refers to the technique that is used to realize the variation point (from an implementation point of view). Several of these realization techniques have been identified in the recent years, such as aggregation, inheritance, parameterization, conditional compilation (see e.g. [5] [1]).

Model Driven Architecture (MDA), on the other hand, was introduced by the Object Management Group only a few years ago. MDA is organized around a so-called Platform Independent Model (PIM). The PIM is a specification of a system in terms of domain concepts. These domain concepts exhibit a specified degree of independence of different platforms of similar type, (e.g. CORBA, .NET, and J2EE) [11]. The system can then be compiled towards any of those platforms by transforming the PIM to a platform specific model (PSM). The PSM specifies how the system uses a particular type of platform [11].

The main contribution of this paper is that we relate MDA to a particular class of software product families, i.e. a configurable software product family, and that we discuss the mutual benefits of this relation. With respect to variability management, we identify two main benefits of applying MDA to product family engineering, i.e. postponing binding time and mechanism selection to application engineering, and the independent evolution of domain concepts, product family assets and the transformation technique. In addition, we identify variability management as a solution to the problem of round-trip transformation in MDA.

The remainder of this paper is organized as follows. In the next section, we discuss a classification of different types of software product families. In section 3 we relate MDA to on of these classes, i.e. a configurable product family. In addition, we provide our view on variability management in this context, as well as how variability management and MDA benefit from such an approach. Related work is presented in section 4 and the paper is concluded in section 5.

2. Product Family Classification and Model Driven Architectures

The basic philosophy of software product families is intra-organizational reuse through the explicitly planned exploitation of commonalities between related products. In our experience with industry we have identified that organizations employ widely different approaches to product family based development. Based on [3], we organize these approaches according to the s*cope of reuse*, i.e. the extent to which the commonalities between related products are exploited.

- **Standardized infrastructure:** Starting from independent development of each product, the first step to exploit commonalities between products is to reuse the way products are built. Reuse of development methodologies is achieved by standardizing the infrastructure with which the individual applications are built. The infrastructure consists of typical aspects such as the operating system, components such as database management and graphical user interface, as well as other aspects of the development environment, such as the use of specific development tools.
- **Platform:** With a standardized infrastructure in place, the next increase in scope of reuse is when the organization maintains a platform on top of which the products are built. A platform consists of the infrastructure discussed above, as well as artifacts that capture the domain specific functionality that is common to all products. These artifacts are usually constructed during domain engineering. Any other functionality is implemented in product specific artifacts during application engineering. Typically, a platform is treated as if it was an externally bought infrastructure.
- **Software Product Line:** The next scope of reuse is when not only the functionality common to all products is reusable, but also the functionality that is shared by a sufficiently large subset of product family members. As a consequence, individual products may sacrifice aspects such as resource efficiency or development effort in order to benefit from being part of the product line, or in order to provide benefits to others. Functionality specific to one or a few products is still developed in product specific artifacts. All other functionality is designed and implemented in such a way that it may be used in more than one product. Variation points are added to accommodate the different needs of the various products.
- **Configurable Product Family:** Finally, the configurable product family is the situation where the organization possesses a collection of shared artifacts that captures almost all common and different characteristics of the product family members, i.e. a configurable asset base. In general,

new products are constructed from a subset of those artifacts and require no product specific deviations. Therefore, product derivation is typically automated once this level is reached (i.e. application engineers specify a configuration of the shared assets, which is subsequently transformed into an application).

In the next section, we discuss how MDA relates to these product family classes and briefly discuss how products are derived in this context. In addition, we discuss the mutual benefits of combining MDA and variability management.

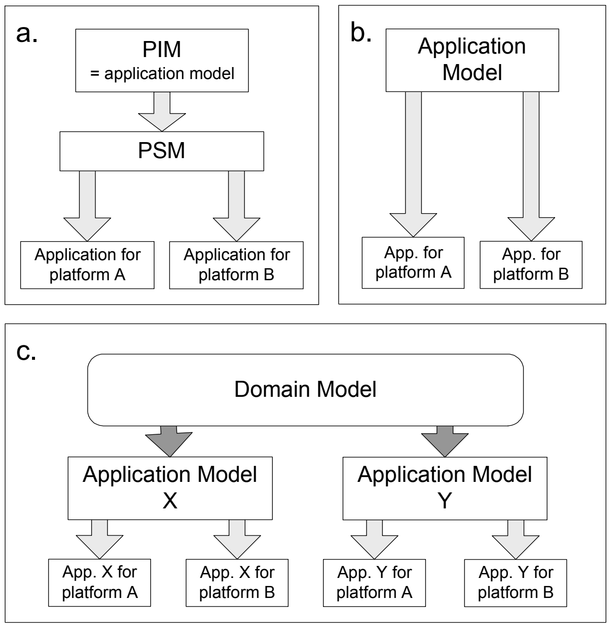


Figure 1 – How the MDA approach relates to software product family engineering. *Figure a. shows the traditional MDA approach. Figure b. shows the platform as a variation point that is resolved during the transformation step. Figure c. shows how MDA can be extended to enable model driven product family engineering.*

3. Variability Management in an MDA approach to Software Product Families

Conceptually, a software system that is specified according to the MDA approach specifies an application model for a family consisting of products that implement the same functionality on different platforms (see Figure 1a). The choice for the alternative platforms is a variation point in such a family. This variation point is separated from the application model in the sense that it is not visible in the specification anymore. Instead, it is provided in the infrastructure (consisting of the different platforms), and managed in the transformation definition (see Figure 1b).

The main benefit of MDA compared to traditional SPF development, is that the management of the platform variation point is handled automatically by the transformation step and is not a concern for the product developer anymore. The underlying platform, however, is not the only variation point that needs to be managed in a product family. The various product family members differ on both a conceptual level, i.e. their functional characteristics, as well as on the infrastructure level, i.e. which shared software assets are used to implement the alternative concepts.

MDA can easily be extended to accommodate this need, by adding a domain model that specifies places where alternative concepts can be selected. Selection of different concepts from this domain model then results in different application models, which, provided that the right transformation definitions are implemented, can already be automatically transformed within to the existing MDA approach (see Figure 1c). This presumes that the MDA transformations are powerfull enough to deal with both platform and other technical variability, and that all variations in the domain models can be implemented or provided by the infrastructure (consisting of the asset base). Relating the latter aspect to the classification of product families described in section 2, MDA thus only presents a real benefit for a configurable product family.

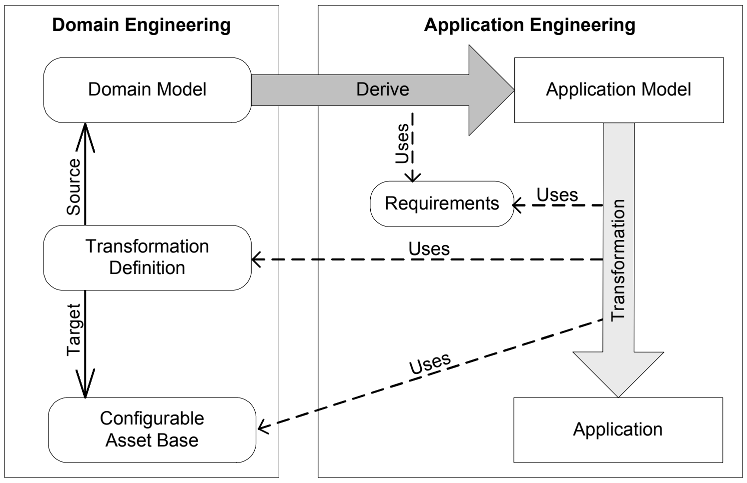


Figure 2 – Product family engineering in an MDA world. *Domain concepts specify the functionality of the product family independent from the implementation details of the asset base. The transformation definition then defines how the domain concepts are mapped on the asset base. Based on the product requirements, an application model is derived by selecting alternative concepts specified in the domain model. The application model is then compiled to a working application using the transformation definition, the asset base and the product requirements.*

3.1 Product derivation in an MDA world

Assuming that there is a configurable software product family, the two main processes can now be described as follows (see also Figure 2):

Domain engineering: Domain engineering involves the following three main activities. The first activity involves the creation and maintenance of the asset base, which consists of reusable and/or variable assets. The second activity involves the specification of the domain model, which consists of the domain concepts. The third activity involves the definition of the transformation definition, which specifies how instances of the domain model are converted into a working application using the asset base. This transformation definition could be a compiler for a domain specific language or an elaborate environment such as KOALA [10], where applications are specified in an ADL and some glue scripts.

Application engineering: Application engineering then boils down to creating an application model by selecting alternative domain concepts from the domain model. In this process alternative concepts are selected based on customer requirements. The application model is then compiled to a working application using the transformation definition, the asset base and the customer requirements. Essentially, the transformation process binds infrastructure level variability that is not visible in the domain model anymore, as well as variation points that are selected on the conceptual level, to the realization of those variation points in the infrastructure.

3.2 Benefits

The main advantages for variability management in this approach to product family engineering are as follows.

- **Postponing binding time and mechanism selection to application engineering.** Rather than polluting the application model with implementation details of the variation points on the conceptual level, such as binding time and mechanism, the mapping to the realization in the infrastructure is handled in the transformation. The main advantage of this separation in comparison to traditional product family development is that the choice for a particular binding time and realization mechanism can now be postponed to the stage at which the application model is compiled. In other words, provided that the appropriate transformations are defined, it is possible to select a binding time and mechanism for a variation point that is most suited to the application requirements during application engineering.
- **Independent evolution of domain concepts, transformations and infrastructure.** The separation of specification and implementation furthermore allows for the independent evolution of the variation points and variants specified in the domain concepts, and the variability realized in the asset base. Managing the evolution of variability in both domain concepts and infrastructure is effectively moved to the transformation definition, as the transformation definition is responsible for the mapping between domain concepts and the infrastructure. A main advantage in comparison with traditional product family development is that applications can benefit from infrastructure or transformation evolution with a simple re-compilation, rather then requiring adaptations to the application model.

In addition to the advantages that MDA presents with respect to variability management, variability management in turn presents a possible solution to the problem of round-trip transformation in MDA. We elaborate on this below.

- **Using variability management to eliminate the need for round-trip transformation.** In cases where the transformations turn out to be not powerful enough, systems that are specified in platform independent models require adaptations at the level of the platform specific model. The problem with such a situation is that once changes have been made at the platform specific level, the PIM cannot be changed and re-compiled or all changes to the PSM will be lost. The MDA community therefore identified the need for round-trip transformation, i.e. a reversible transformation between the PIM and the PSM. This need can be removed however, by specifying the platform specific variations as variation points in the PIM. These variation points then explicitly specify where those variations are allowed, and during the transformation these platform specific variants are inserted in the PSM. This approach is similar to programming assembly within a higher level language such as C.

4. Related Work

In the past few years, a number of books on software product families emerged, such as, [14] [6] [4] and our co-author's book [2]. In a paper by the same co-authour [3] a maturity classification of product families is presented that consists of the following levels: standardized infrastructure, platform, software product line, configurable product base, programme of product lines, and product populations. In this paper we extracted four of those levels according to the scope of reuse.

A number of common techniques for realizing variability are discussed in [5] and [1]. Other techniques are focused around some form of infrastructure-centered architecture where variability is realized in the component platform, e.g. CORBA, COM/DCOM or Java-Beans [13], or around code fragment superimposition, e.g. Aspect-, Feature- and Subject-oriented Programming, [8], [12] and [7], respectively.

A paper related to deriving different application models is [9], which proposes extending UML with architectural constraints to enable automatic derivation of product models from the product family architecture.

5. Conclusions

While product families have received wide adoption in industry during the 1990s, MDA was proposed by the OMG only a few years ago. In this paper, we portrayed MDA as an approach for a particular class of product families, i.e. a configurable product family. We discussed how products are derived in such an approach and how MDA and variability management benefit from each other.

The main contribution of this paper is that we related MDA to a configurable software product family and discussed the mutual benefits of this relation. With respect to variability management, we identified two main benefits of applying MDA to product family engineering, i.e. postponing binding time and mechanism selection to application engineering, and the independent evolution of domain concepts, product family assets and the transformation technique. These aspects are not tackled by traditional product family development. In addition, we identified variability management as a solution to the problem of round-trip transformation in MDA.

Acknowledgements

This research has been partially sponsored by the EU under contract no. IST-2001-34438.

6. References

[1] M. Anastasopoulos, C. Gacek, *Implementing Product Line Variabilities*, Symposium on Software Reusability (SSR'01), Toronto, Canada, Software Engineering Notes, Vol. 26, No. 3, pp. 109-117, May 2001.

[2] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*, Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7, 2000.

[3] J. Bosch., *Maturity and Evolution in Software Product Lines; Approaches, Artifacts and Organization*, in Proceedings of the SPLC2, August 2002.

[4] P. Clements, L. Northrop, *Software Product Lines – Practices and Patterns*, Addison-Wesley, 2002.

[5] I. Jacobson, M. Griss, P. Jonsson, *Software Reuse – Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997.

[6] M. Jazayeri, A. Ran, F. Van Der Linden, *Software Architecture for Product Families: Principles and Practice*, Addison-Wesley, 2000.

[7] M. Kaplan, H. Ossher, W. Harrisson, V. Kruskal, *Subject-Oriented Design and the Watson Subject Compiler*, position paper for OOPSLA'96 Subjectivity Workshop, 1996.

[8] G. Kiczalez, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, *Aspect Oriented Programming*, in Proceedings of 11th European Conference on Object Oriented Programming, pp. 220-242, Springer Verlag, 1997.

[9] L. Monestel, T. Ziadi, J. Jézéquel, *Product line engineering: Product derivation*, Workshop on Model Driven Architecture and Product Line Engineering, associated to the SPLC2 conference, San Diego, August 2002.

[10] R. van Ommering, F. van der Linden, J. Kramer, J. Magee, *The Koala Component Model for Consumer Electronics Software*, IEEE Computer, vol. 33-3, pp 78-85, March 2000.

[11] MDA Guide v1.0, OMG, 2003.

[12] C. Prehofer, *Feature Oriented Programming: A fresh look on objects*, in Proceedings of ECOOP'97, Lecture Notes in Computer Science 1241, Springer Verlag, 1997.

[13] C. Szyperski, *Component Software – Beyond Object-Oriented Programming*, Pearson Education Limited, 1997.

[14] D. M. Weiss, C.T.R. Lai, *Software Product-Line Engineering: A Family Based Software Development Process*, Addison-Wesley, 1999.

Towards a UML Profile for Service-Oriented Architectures [1]

Reiko Heckel, Marc Lohmann, and Sebastian Thöne

Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn, Germany

1 Introduction

Application development naturally starts with functional requirements given by the business that shall be supported. The model capturing these requirements is then refined taking more and more aspects of the technology and target platform into account. To support and automate in particular the later steps of this process is the objective behind the OMGs model driven architecture (MDA), summarized in Fig. 1.

The development of distributed applications is supported by platforms like Web services [2] or Jini [3]. The aspects of the technology shared by these platforms, like the roles of *service providers, requesters*, and *registries* as well as their *publish, find*, and *bind* operations, are conceptualized in the service-oriented architectural style (SOA). To support the model-driven development of SOA applications at the platform-independent Level 2 of Fig. 1, a notation has to be defined to support the representation of SOA concepts. Moreover, these concepts have to be given a semantics which allows to interpret the behavior of applications modeled in that style.

In this paper, we fulfill the demand for a suitable syntax for this domain by sketching a UML profile for SOA by means of an example. Once the profile is properly defined, its semantics can be given in terms of a graph transformation

Email addresses: `reiko@upb.de` (Reiko Heckel), `mlohmann@upb.de` (Marc Lohmann), `seb@upb.de` (Sebastian Thöne).

[1] Research partially supported through the European Research Training Network *SegraVis* (on *Syntactic and Semantic Integration of Visual Modeling Techniques*) and the International Graduate School *Dynamic Intelligent Systems* at the University of Paderborn

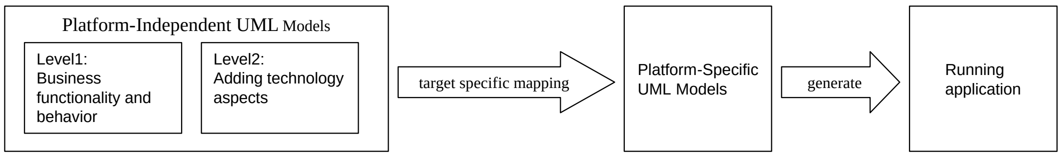

Platform-Independent UML Models
Level1: Business functionality and behavior
Level2: Adding technology aspects
target specific mapping
Platform-Specific UML Models
generate
Running application

Fig. 1. OMG's outline of MDA

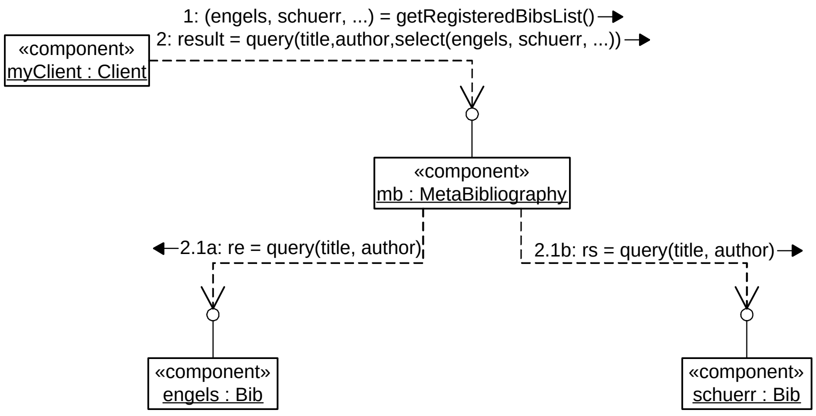

1: (engels, schuerr, ...) = getRegisteredBibsList()
2: result = query(title,author,select(engels, schuerr, ...))
«component» myClient : Client
«component» mb : MetaBibliography
2.1a: re = query(title, author)
2.1b: rs = query(title, author)
«component» engels : Bib
«component» schuerr : Bib

Fig. 2. Collaboration diagram modeling execution of a query

model for the SoA style introduced in [1], expressed here for conciseness using collaboration diagram notation.

The paper is organized as follows: Below, we introduce the platform-independent business architecture of our sample application (cf. Level 1 of Fig. 1), a meta-bibliography service integrating an open number of local bibliographies. Then, in Section 3, we sketch the SoA profile and its semantics given by the behavior of the SoA operations. Following the MDA outline, Section 4 provides the SOA model of the publication service based on the profile (cf. Level 2 of Fig. 1), and Section 5 concludes the paper.

2 Business Architecture

In this section, we introduce the running example of the paper. The objective behind the meta-bibliography is to provide an integrated bibliography service for the *SegraVis* network (see `www.segravis.org`) by loose coupling of individual solutions that are already present at the 12 member sites. Each site has to wrap its local bibliography data source (a relational data base, BibTeX file, etc.) inside a Web service interface. The meta-bibliography is accessed over the web by a browser and an incoming request is forwarded to all known bibliography services. The application is also used as a running example for exercises in a lecture on *Model-Based Development of Web Applications* at the University of Paderborn (see `www.upb.de/cs/ag-engels/`).

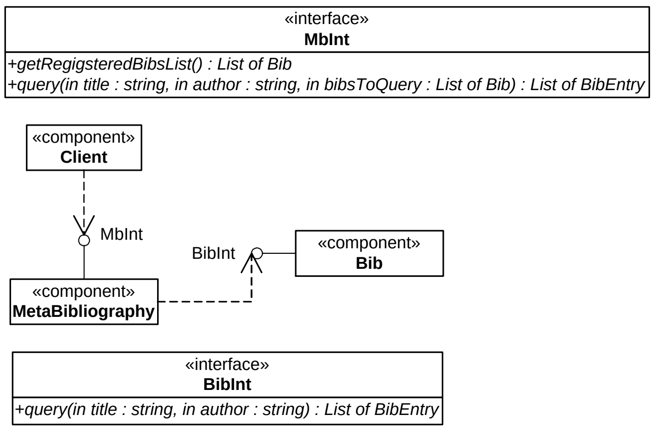

«interface»
MbInt
+getRegigsteredBibsList() : List of Bib
+query(in title : string, in author : string, in bibsToQuery : List of Bib) : List of BibEntry
«component»
Client
MbInt
BibInt
«component»
Bib
«component»
MetaBibliography
«interface»
BibInt
+query(in title : string, in author : string) : List of BibEntry

Fig. 3. Business architecture derived from collaboration diagram

Figure 2 shows a collaboration diagram modeling a sample scenario for executing a query on the meta bibliography. Before clients can send a query, they have to select those local bibliographies where the query should be executed. Therefore, clients retrieve a list of registered bibliographies, i.e., all local bibliographies that are registered with the meta bibliography. Then, the client selects the bibliographies of the research groups of Gregor Engels in Paderborn and Andy Schürr in Darmstadt. The search string that is entered afterward is forwarded concurrently to all selected bibliographies. From the scenario, we can derive the UML component diagram in Fig. 3 to identify the (types of) components and interfaces used in the application.

3 Service-Oriented Architectures in UML

The roles and artifacts of the service-oriented style are represented as follows (see upper left of Fig. 4). Services are components with stereotype ≪service≫. Provider and requester are interpreted as roles of components implementing and using an interface, respectively. (We ignore here the business perspective of, e.g., the provider as organization owning the service.) A service registry is a service implementing a special interface (marked by "?" as shorthand for a stereotype ≪registry≫) where service descriptions can be published by providers and queried by requesters.

A service description is represented as a UML package marked by a stereotype ≪desc≫, see upper right of Fig. 4. The models contained in this package specify the service and its interface(s) as indicated by the ≪specifies≫ dependencies. Requesters store their requirements for a service in packages marked by a stereotype ≪req≫. We assume a conceptual relation ≪satisfies≫ to represent the fact that all required properties are guaranteed by a description. Further, we have to represent the fact that a component (requester or registry)

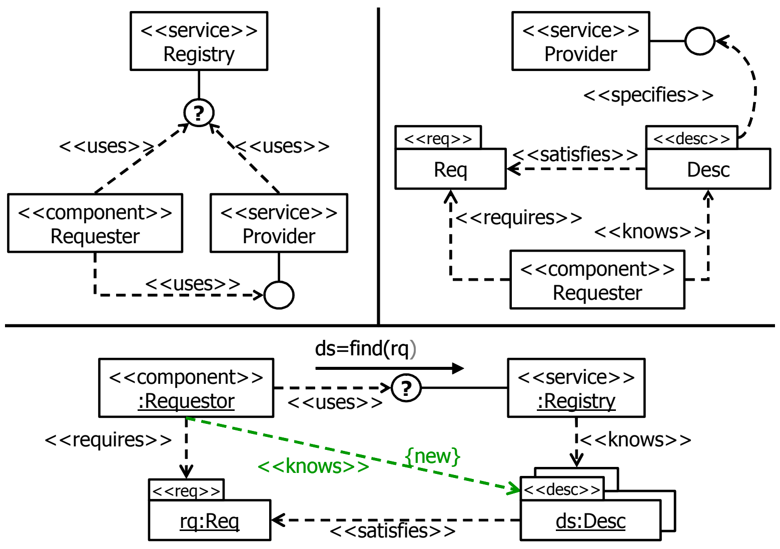


Fig. 4. Syntax of UML for SOA and semantics of the *find* operation

has access to a certain description by a ≪knows≫ relation.

The SoA style provides three distinguished operations, to *publish* and *find* service descriptions, and to *bind* to a service. In the bottom of Fig. 4 a collaboration diagram is provided to specify the semantics of the find operation, by which a requester obtains all service description known to a registry that satisfy the requester's requirements.

Similarly, there are collaboration diagrams defining the operational semantics of the other two operations. These collaboration diagrams can be further refined into graph transformation rules, as expressed in [1]. The operational semantics covers only the communication and architectural reconfiguration of components and services. Other aspects, like the contents of service descriptions, including operation signatures, data models, etc. are left open to further refinement in the platform-specific model.

4 SOA Model

Based on the notation given in the previous section, we can specialize our business model to the SoA style. This requires to add to the initial configuration a registry which holds the list of all local bibliographies participating in the network together with appropriate service descriptions as well as requirements. Moreover, the operations of the original application scenario have to be interleaved with find and bind operations. One possible such specialization is shown in Fig. 5.

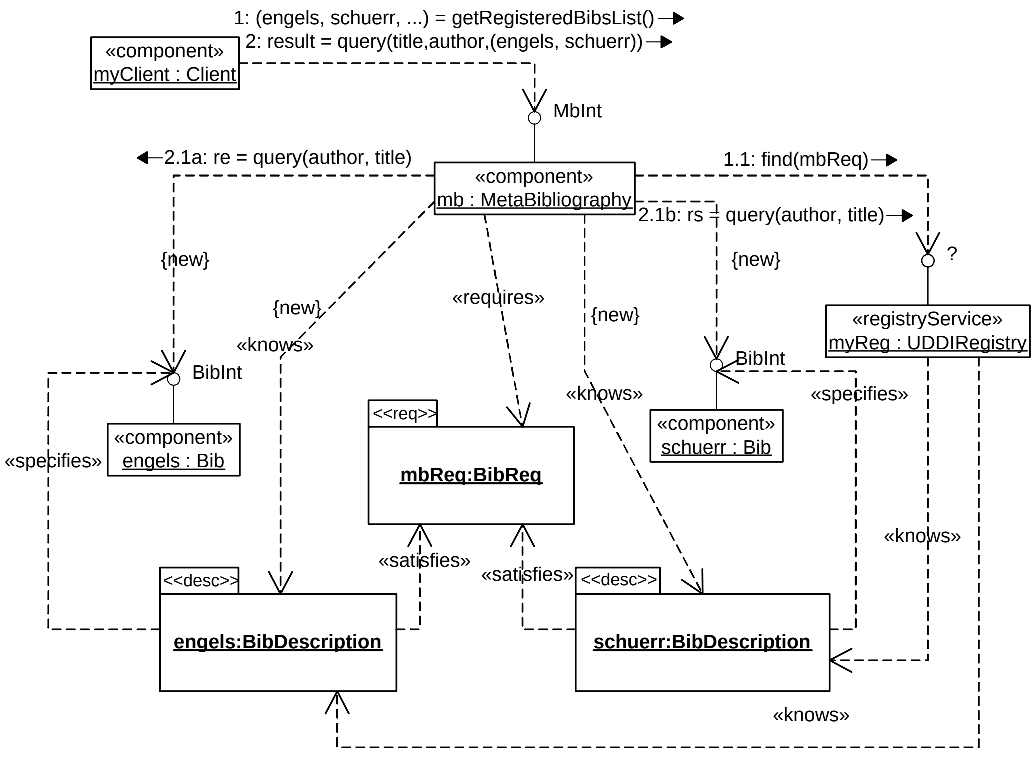


Fig. 5. SoA-specific collaboration diagram

5 Conclusions

The UML profile for service-oriented architectures sketched in this paper requires further refinement, in particular, if platform-specific details, e.g., of XML-based Web services shall be added. This aspect is, however, separated from the purely architectural view of the notation so far, as it is concerned with the contents of service descriptions and queries.

Still at the architectural level, other types of diagrams are involved, like class diagrams with interfaces to define signatures and data types of operations and sequence diagrams as alternative presentation of interactions. They are all left out here because of space limitations.

Based on the formal semantics of the architectural style, which is expressed in [1] by means of a graph transformation system, we are also planning support for validation of SoA models by simulation using a graph transformation tool like Fujaba (see `www.fujaba.de`) or by model checking using one of the emerging approaches to model checking graph grammars [5,4].

References

[1] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and validation of service-oriented architectures: Application vs. style. In *Proc. ESEC/FSE 2003*, Helsinki, Finland, September 2003. To appear.

[2] M. Champion, C. Ferris, E. Newcomer, and D. Orchard. *Web Service Architecture, W3C Working Draft*, 2002. `http://www.w3.org/TR/2002/WD-ws-arch-20021114/`.

[3] Sun microsystems. *Jini Architectural Overview - Technical White Paper*, 1999.

[4] A. Rensink. Model checking graph grammars. In *Proc. Workshop on Automated Verification of Critical Systems (AVoCS 2003), Southampton (UK)*, April 2003.

[5] D. Varró. Towards symbolic analysis of visual modelling languages. In Paolo Bottoni and Mark Minas, editors, *Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques*, volume 72 of *ENTCS*, pages 57–70, Barcelona, Spain, October 11-12 2002. Elsevier.

A Synthesis-Based Approach to Transformations in an MDA Software Development Process

Ivan Kurtev, Klaas van den Berg

Software Engineering Group, Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE, Enschede, the Netherlands
Email: {kurtev, vdberg}@cs.utwente.nl

Abstract
In an MDA software development process, models are repeatedly transformed to other models to finally achieve a set of models with enough detail to implement the system. Generally, there are multiple ways to transform one model into another model. Alternative target models differ in quality properties and the selection of a particular model is determined on the base of specific requirements. Current transformation languages only provide means to specify transformations but do not help to identify and select among alternative transformations. In this paper we propose a synthesis-based software development process with a set of techniques for constructing a transformation space for a given transformation problem. The process takes a source model, its meta-model and the meta-model of the target, and quality requirements as input and generates a transformation space.

Key words: MDA, Software Development Process, Synthesis, Model Transformations

1. Introduction

The Model Driven Architecture (MDA) approach proposed by OMG [4] aims to promote interoperability and portability. Current software development processes such as the Unified Process [1] have to be adapted to this approach. A software development process can be described using the Software Process Engineering Meta-model (SPEM) [7]. A software development process is defined as collaboration between entities called process roles that perform operations called activities on concrete, tangible entities called work products. Activities consist of a number of steps. A discipline partitions activities within a process according to a common theme. In the Unified Process nine disciplines are described, e.g. Business Modeling, Requirement Management, Analysis and Design, and Implementation.

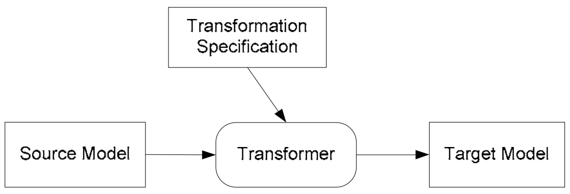


Figure 1. The generalized MDA Pattern [3]

MDA software development involves a specification of the system functionality in a set of Computation Independent Models (CIMs), Platform Independent Models (PIMs), and Platform Specific Models (PSMs). A key characteristic of the MDA is the notion of model transformation. Model transformations appears at any level: CIMs to CIMs, CIMs to PIMs, PIMs to PIMs, PIMs to PSMs and PSMs to PSMs. In the MDA Guide [3] this is captured in the MDA pattern, which can be generalized as depicted in Figure 1. The transformation specification can consists of - for example - mapping rules based on (meta-) models. A model transformation is a set of transformation rules and techniques used to operate on a source model to produce another model, the target model. In general, transformation rules relate constructs in the source model to the constructs in the target model (see Figure 2).
The source and target models are at level M1 according to the Meta Object Facility (MOF) [5] terminology and their source and target meta-models are at level M2. The source model M_A is an instance of the source meta-model MM_A and it has to be transformed to a new model that is an instance of the target meta-model MM_B. The two meta-models determine the possible transformations rules. The resulting target models may differ from each other in the qual-

ity properties they possess, such as performance or adaptability. Software engineers have to compare and choose among the alternatives based on quality requirements.

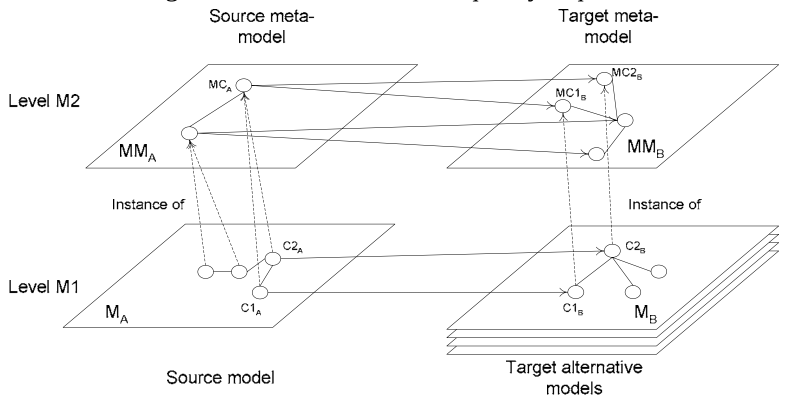


Figure 2 Multiple alternative transformations for a given source model

The diversity of quality requirements prevents the usage of a fixed set of transformation rules. For a concrete problem the software engineer must be able to identify the transformations that lead to a model with the desired quality properties. Unfortunately, current transformation languages and techniques do not provide means to identify alternative transformations and to compare them based on specific quality characteristics of the resulting models. This paper uses a synthesis-based approach to an MDA software development process. It applies a technique to explicitly model a set of alternative transformations for the source model. This technique also allows specification of quality properties of the models. The quality properties are used as selection criteria among the alternatives.

In section 2 we explain the problem addressed in the paper. Section 3 explains the synthesis-based approach to software development. Section 4 describes techniques for constructing transformation spaces and selecting alternatives from them. Section 5 discusses the applicability of these techniques in the context of the model transformations in MDA.

2. Problem Statement

An MDA software development process involves transformations from models to models as depicted in the MDA pattern. We show the presence of alternative model transformations with an example of transforming UML class diagrams to XML schemas. At level M1 we have a UML model (source) and alternative XML schemas (target). At level M2 we have the UML meta-model and an XML Schema meta-model that can be derived from the XML Schema specification [8] (There is no standard XML Schema meta-model expressed according to MOF but some proposals already exist [6]). The UML class model can be considered as the PIM and the XML-Schema as the PSM. For example, the *Class* construct defined in the UML meta-model may be mapped to Element declaration or Complex type definition construct in the XML Schema meta-model. Then at level M1 there are two possibilities for mapping each class in the source model: either to an element declaration or to a complex type. These possibilities can be combined in alternative mappings that produce alternative XML schemas. We can identify some transformations that produce alternative schemas. Even for simple source models there are more than one correct target models. The first problem is to select among the alternatives. It is rarely the case that every alternative is a good solution. Usually various requirements must be fulfilled. Software engineers are often faced with this situation but usually the process of comparing alternatives is more implicit than explicit. The second problem is that there is no support for identification of alternative transformations. The transformation to the required schema may not be always trivial and obvious and then a systematic approach is required.

3. Synthesis-based Software Development

In the Introduction we described activities and work products in SPEM. MDA work products are models, transformation specifications and transformers. An MDA software development process can be seen as a sequence of applying MDA patterns, starting with the user requirements specified in a CIM and after the final transformation resulting in an operational application. The target model in one step serves as source model in the following step. These activities can be grouped into disciplines around the MDA levels.

There is no easy fit of the MDA work products into activities as defined in the Unified Process. In this paper we demonstrate how the synthesis-based approach to software development can be applied in an MDA context. The major activities in synthesis-based software development are the following: technical problem analysis, solution domain analysis, and alternative space analysis. Each activity is refined into several steps [9]. The three activities could be carried out for each transformation.

As transformations are at the core of MDA we now focus on the third activity in which we implicitly use results of the technical problem analysis and the solution domain analysis (e.g. the analysis of the XML Schema, the Schema meta-model, the UML domain class model and the UML meta-model). A complete analysis of this case study is given in [2].

The important concepts in the MDA transformation problem are depicted in Figure 3.

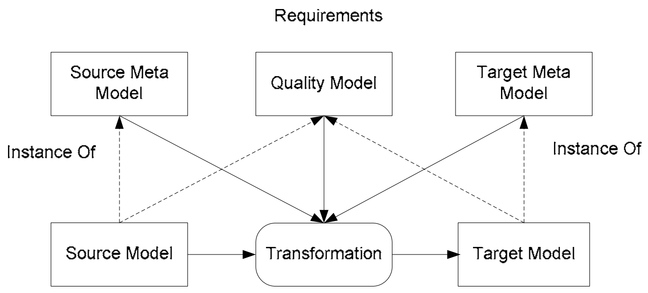


Figure 3 The transformational problem in the MDA pattern

The source model specifies the system functionality at certain level of abstraction. The source model is an instance of the source meta-model. We aim at identifying transformations that produce target model(s), which preserve the functionality in the source model and realizes that functionality on the target platform. The target meta-model (or target platform) is considered as a constraint and in some cases the required functionality cannot be implemented with the features provided by the constructs in the target meta-model. Quality requirements for the models such as adaptability, extensibility, performance, etc. are specified in the quality model. Those requirements are imposed on the source model and they must be preserved in the resulting target model where these quality properties are interpreted in terms of the target meta-model. In Figure 3 we denote the fulfillment of the quality requirements by the source and the target models as conformance to the quality model.

In summary, the transformational problem in MDA is the problem of specifying a transformer that takes the source model, its meta-model, the quality model and the target meta-model as input and generates target model(s) that conforms to the target meta-model and the quality model. The alternative space analysis is directed at the identification and specification of feasible transformations between the source model and the target model.

4. Alternative Space Analysis

In this section we describe the activity Alternative Space Analysis of the synthesis-based software development process in the context of the MDA transformation problem. The workflow for this activity is shown in Figure 4.

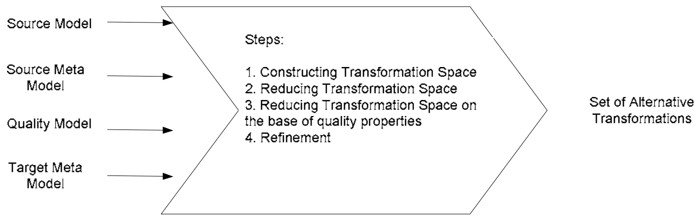

Source Model
Source Meta Model
Quality Model
Target Meta Model
Steps:
1. Constructing Transformation Space
2. Reducing Transformation Space
3. Reducing Transformation Space on the base of quality properties
4. Refinement
Set of Alternative Transformations

Figure 4 Workflow of the activity Alternative Space Analysis

This activity takes a source model, its meta-model, the quality model and the meta-model of target model as input and generates a *transformation space* for the source model, i.e. a set of alternative transformations for a given source model. A *transformation space* is a multidimensional space spanned over a number of independent *dimensions***.** Each dimension is associated with a number of *coordinates* that form a *coordinate set*. Every element in the space represents a transformation that produces a model that is an instance of the target meta-model. Since a transformation space may be too large some operations are defined to reduce the space. The required quality characteristics of the target model are represented in a quality model. The concepts from the quality model are woven with the source model elements and they indicate characteristics that the target model must possess. The result of the activity is a set of alternative transformations that can be used to produce the transformer required for the model transformation. We describe the process of constructing transformation space for the simple example model in Figure 5. It contains two classes related with a generalization relation. The quality requirement in that example is extensibility of the model regarding expected specializations of *Class2*.

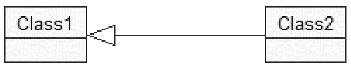


Figure 5 Example source model: UML class diagram with two classes with generalization relation

This model will be transformed to an XML schema. The example shows that even in this simple case there are multiple valid schemas that can be generated. Two alternative XML schemas for the source model are given in Figure 6.

```
<complexType name='class1'>
   ………………..
</complexType>

<complexType name='class2'>
  <complexContent>
     <extension base='class1'>
      …….
     </extension>
  </complexContent>
</complexType>
```

(a)

```
<element name='c1' type='class1'/>

<complexType name='class1'>
   ………………..
</complexType>

<element name='c2' substitutionGroup='c1'>
  <complexType>
    <complexContent>
      <extension base='class1'>
       …….
      </extension>
    </complexContent>
  </complexType>
</element>
```

(b)

Figure 6 Two alternative XML schemas derived from the source model

The activity of alternative space analysis has the following 4 steps.

1. Constructing Transformation Space. The dimensions of a transformation space are determined from the constructs in the source model. A subset of the constructs in the source model is selected and one dimension is defined for each construct in that subset. A construct from the source model used to define a given dimension is always an instance of a construct from the source meta-model. The coordinate set for that dimension is determined on the base of the construct from the source meta-model. For the source meta-model construct the set of possible target constructs from the target meta-model is determined. Then this set is used to form the coordinate set for the dimension. A point in a transformation space represents an alternative transformation of the source model. For every source construct the target construct is determined as the coordinate of the point over the dimension corresponding to that source construct.

The transformation space for the example in Figure 5 contains three dimensions: `Class1`, `Class2` and `Generalization`. The first two correspond to the classes and the third corresponds to the relation between the classes. Coordinate sets for these dimensions are defined below:

```
coordinateSet(Class1)={CT, E, MG, AG}
coordinateSet(Class2)={CT, E, MG, AG}
coordinateSet(Generalization)={Der, Subst, Cont, Ref, E}
```

Coordinate sets are derived from the constructs available in the XML Schema meta-model. Abbreviations stands for Complex Type Definition (`CT`), Element Declaration (`E`), Attribute Declaration (`A`), Attribute Group (`AG`) and Model Group Definition (`MG`). For the `Generalization` dimension we choose among the XML Schema relations. They are Derivation (`Der`), Substitution (`Subst`), Containment (`Cont`) and Reference (`Ref`). Figure 7 shows a graphical representation of the transformation space for our example. It has 4 * 4 * 5 = 80 alternatives.

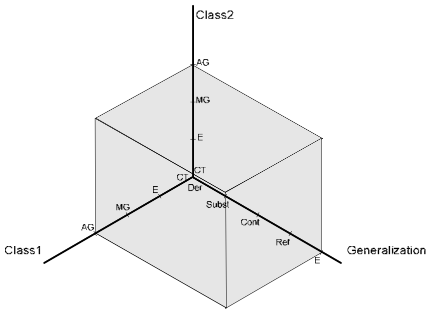


Figure 7 Transformation Space with three dimensions corresponding to two classes and generalization

2. Reducing Transformation Spaces. It is possible to generate all the alternatives in a transformation space and to compare them. The number of alternatives is usually large even for simple source models. However, it is unnecessary to generate the whole space of alternatives. Instead, the software engineer may reduce the space either by selecting or by excluding alternatives from the transformation space.

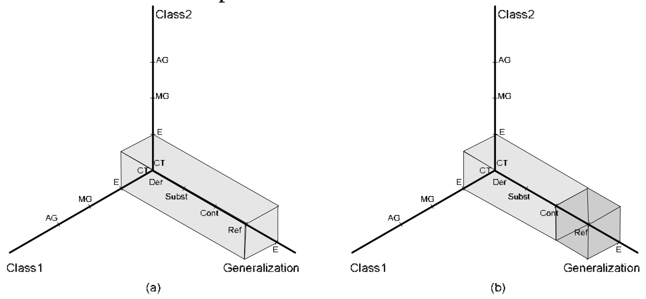


Figure 8 Transformation Spaces after selection (a) and exclusion (b) operations

For example the software engineer may decide to *select* only Element Declaration (`E`) and Complex Type (`CT`) as alternatives for UML Class. He specifies criteria for selection from the transformation space, e.g. based on best practices or heuristic rules. The space after this selection is shown in Figure 8a. Furthermore, the software engineer may decide to *exclude* Reference (`Ref`) and Element Declaration as alternatives for the generalization. The points that satisfy the criteria are excluded from the transformation space. The space after the application of the exclusion operation is shown in Figure 8b. The dark shaded area shows the part that is excluded.

3. Reducing Transformation Space on the base of Quality Properties. Transformation space may be further reduced by considering the quality requirements that the target model

must fulfill. The quality requirements are used to create the quality model. Once the software engineer has the quality model constructs explicitly represented he can identify instances of those constructs in the source model. Then, he can use selection criteria based on them.
For instance, in our example we may require extensible XML schemas that allow adding specializations of *Class2* without changing the generated schema. The quality model in that case distinguishes two types of components: extensible and inextensible. We consider *Class2* as extensible. The interpretation of the extensibility in terms of the target XML schema is that the content model of the extensible component is reused by the components created from the expected specializations of *Class2*. According to that interpretation the schema in Figure 6b is not extensible because the complex type is defined as anonymous and therefore its content cannot be reused through the schema derivation mechanism. The schema in Figure 6a allows derivations and can be considered as extensible. The process of assigning quality properties to the source model and the operations that support it are described in more details in [2].
4. Refinement. This is the last step of the activity for alternative space analysis. Once the space is sufficiently reduced the software engineer may generate the alternatives explicitly. However, these alternatives are not complete transformations yet. Some additional details are required before the transformation is specified and executed. For instance, the XML element declaration components always require a type to be defined.

5. Conclusion

We presented a synthesis-based approach for an MDA software development process with focus on the activity of alternative space analysis. In current software development processes such as the Unified Process, no explicit attention is given to exploration and selection of alternatives. For MDA there are inherently many possible alternative transformations from source to target models. In the synthesis-based approach to the software development process alternative space analysis is treated as a separate activity with specific techniques for generating and reducing the transformation space. Although transformation spaces tend to be large even for simple models, they are purely conceptual. The software engineer does not need to generate all the alternative transformations from a transformation space. A number of reduction steps are applied and the size of the space is reduced. The structure of a transformation space specified by dimensions and coordinate sets provides a framework to reason about the alternatives in general instead of per alternative individually.
Since software engineers generally have to fulfill both functional and quality requirements, they should be able to identify and compare the quality properties of the functionally equivalent alternative target models for the same source model. The proposed techniques capture the quality requirements in a quality model that can be used in criteria for reducing the transformation space.

References

1. Kruchten, P. (2000). *The Rational Unified Process*, Addison-Wesley-Longman.
2. Kurtev, I, Berg, K.G. van den, Aksit, M. (2003). *UML to XML-Schema Transformation: a Case Study in Managing Alternative Model Transformations in MDA.* To appear in FDL'03 .
3. MDA Guide (2003). Version 1.0, Object Management Group, omg/2003-05-01.
4. Miller, J., Mukerji, J. (2001). *Model Driven Architecture (MDA).* OMG Document available at http://www.omg.org
5. OMG/MOF. (2000). *Meta Object Facility (MOF) Specification.* OMG Document available at http://www.omg.org
6. OMG/XMI. (2001). *XML Metadata Interchange (XMI) Version 2.* OMG Document available at http://www.omg.org
7. Software Process Engineering Metamodel Specification (2002). Version 1.0, Object Management Group formal/02-11-14.
8. Thompson, H., Beech, D., Maloney, M., Mendelsohn, N. (2001). *XML Schema Part 1: Structures,* W3C Recommendation. http://www.w3.org/TR/xmlschema-1
9. Tekinerdogan, B., Aksit, M. (2002). *Synthesis-Based Software Architecture Design*. In Aksit, M. (ed.) 'Software Architectures and Component Technologies', Kluwer Academic Publishers