Metamodelling for MDA

First International Workshop
York, UK, November 2003
Proceedings

Edited by

Andy Evans
Paul Sammut
James S. Willans

Table of Contents

Mappings

Tools

Preface

The OMG Model Driven Architecture (MDA) promises to be a revolutionary step in software engineering by making models primary artefacts. This raises the level of abstraction in the software development process, and thus helps manage the complexity and change inherent in today's systems. Whilst a lot of focus has been given to the transformation of platform-independent models to platform-specific models, the scope of MDA potentially covers the modelling of all aspects of a system throughout its lifecycle. Metamodelling provides the foundation for this vision, by enabling meaningful metamodels of languages to be defined precisely and unified in a consistent framework. To this end, the OMG's Meta-Object Facility (MOF) is stipulated as the language in which all languages for MDA are expressed. As another iteration of the UML revision process nears completion however, many issues have been raised concerning the means by which MOF is applied, such that languages constructed to support MDA are reusable, flexible and meaningful. This workshop then aims to investigate the fundamental principles, tools and techniques that are needed for the metamodelling of languages, in order to provide a foundation for MDA and the long-term future for the role of modelling in software development.

1 Models and Mappings

Fundamentally MDA concerns a wide variety of models and mappings between those models, allowing integration and transformation of those models. Whilst there are many kinds of mapping, they can be fit broadly into three main categories:

- refinement or vertical mappings, which relate system models at different levels of abstraction, such as PIM to PSM mappings and reverse engineering mappings;
- horizontal mappings, which relate or integrate models covering different aspects or domains within a system, but at the same level of abstraction;
- variant mappings, which relate models of different members of a product family, or different stages of the evolution of a system.

These categories of mapping can be thought of as the dimensions in a three dimensional modeling space, as depicted in Figure 1.

2 Languages and Metamodelling

Models are defined using modelling languages, and if those languages are to be used for anything more sophisticated than drawing pictures, their syntax and semantics must both be precisely defined. Whilst formal languages such

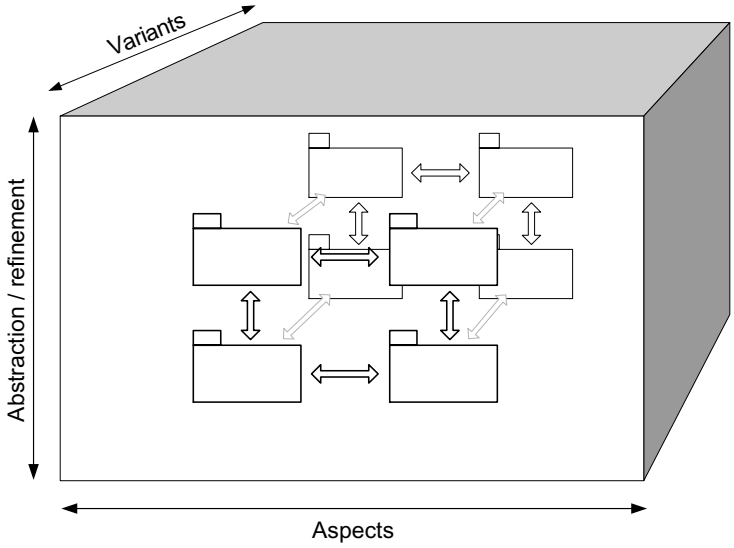


Fig. 1. Dimensions in The Modeling Space.

as Z can be used to realise such language definitions, their syntax are often obtuse and inaccessible, making them unpopular with industry. An alternative approach is metamodelling, where language definitions are themselves modelled, constructed in a language that resembles the languages used to construct system models. In this way, metamodelling not only enables language definitions easier to understand (since they are written in a familiar notation), but potentially they can be manipulated and integrated in the same way as system models (since they are all models). The latter of course depends entirely on the modelling tools that are available, but the potential is nevertheless there.

In MDA, models can be constructed in whatever modelling language the user chooses, but in order for models from one language to be integrated with or transformed into models from another language, those languages must themselves be integrated. This is achieved by:

1. defining the modelling languages using the same metamodelling language;
2. providing a mapping that semantically integrates the two languages.

In MDA, all modelling languages are ultimately defined in the OMG standard metamodelling language, called the Meta-Object Facility (MOF). Another OMG standard in the pipeline is the Query/View/Transformations (QVT) language, which will be the language designated for describing all language mappings. MOF and QVT provide the unifying foundation for MDA, since they facilitate the key metamodelling activities of metamodel construction and metamodel integration.

Providing standardised metamodelling and mapping languages is only part of the challenge however. Metamodels of real languages are complex artefacts just like models of today's enterprise systems, and must be similarly well thought out and architected. Thus there is also the need for strong metamodelling principles,

processes and methodologies. As another iteration of the UML revision process nears completion, it is clear that this is an area which is currently poorly understood, as many issues have been raised concerning the means by which MOF is applied, such that languages constructed to support MDA are reusable, flexible and meaningful.

3 Standards and Tools

Standards such as MOF and QVT described above, are crucial for MDA to work, since they distil common understanding and best practice, and provide conformance points for tools, that in turn enable tool interoperability. Other important MDA related standards are the Unified Modelling Language (UML), and the XML Metadata Interchange (XMI) format.

As important as, if not more so than, standards are tools. Without the availability of tools to carry out the appropriate applications, MDA simply cannot work. There is a vast range of modelling tools on the market, that have been particularly popular since the advent of UML. However most of these tools are tied to one or a few languages that are hard coded in the tool. MDA will require more flexible 'metatools', that can be adapted to support a much wider range of languages. Metamodelling makes this feasible, since a metatool can import language definitions defined in MOF and integrate them using QVT mappings.

It is unlikely that there will be one tool that will carry out every envisagable MDA application. Rather there will probably be tools for constructing and integrating metamodels, tools for system modelling, tools for model and metamodel analysis, model compilers and interpreters, model transformation tools, tools for running model simulations, and so on. XMI is the XML based serialisation format that will allow these tools to work seamlessly together in a unified MDA environment.

4 Aims and Scope of the Workshop

The workshop seeks to explore the theories, principles and practice by which language metamodels are constructed, transformed, related and used to support MDA. It also aims to highlight lessons that have been learned from putting these theories and principles into practice, and investigate the next generation of tools that are needed to fully realise the MDA vision. The papers and discussions at this workshop reflect the current challenges facing MDA in the area of metamodelling, and touch on many of the issues described above.

A key aspect of this workshop is that it aims to bring together the theories, views and experiences of both academia and industry. Solutions coming from industry tend to be focused and practical, yet often lack the generality needed to apply them in a wider arena, due to inevitable time constraints and business pressures. Solutions from academia on the other hand, will often be precise, powerful and generic, yet may fall short on practical aspects such as usability, implementability and applicability to real problems. Workshops such as these

will hopefully provide the bridge between industry and academia, so that powerful, practical, implementable, generic, technologically sound solutions to the challenges of MDA can be found.

November 2003

Andy Evans, Program Chair
Paul Sammut, Organising Chair
James S. Willans, Organising Chair
Metamodelling for MDA Workshop 2003

Workshop Sponsors

Organisation

Chairs

Paul Sammut, University of York, UK
James S. Willans, University of York, UK

Program Chair

Andy Evans, University of York, UK

Programme Committee

Colin Atkinson, University of Mannheim, Germany
Wim Bast, Compuware, The Netherlands
Jean Bezivin, University of Nantes, France
Steven Crook-Dawkins, University of York, UK
Tony Clark, Xactium Ltd., UK
Andy Evans, University of York, UK
Robert France, Colorado State University, USA
Anneke Kleppe, Klasse Objecten, Netherlands
Stephen J. Mellor, Project Technology, USA
Alan Moore, Artisan Software, UK
Richard Paige, University of York, UK
Pete Rivett, Adaptive, UK
Paul Sammut, University of York, UK
Jos Warmer, Klasse Objecten, The Netherlands
James S. Willans, University of York, UK

Invited Speakers

Tony Clark, Xactium Ltd., UK
Stephen J. Mellor, Project Technology, USA
Alan Moore, Artisan Software, UK

Hosted by

Software and Systems Modelling Team,
Department of Computer Science,
University of York, UK

Calling a Spade a Spade in the MDA Infrastructure

Colin Atkinson
University of Mannheim
68161 Mannheim, Germany
atkinson@informatik.uni-mannheim.de

Thomas Kühne
Darmstadt University of Technology
64283 Darmstadt, Germany
kuehne@informatik.tu-darmstadt.de

Introduction

Metamodeling has a pivotal role to play in the realization of the MDA. It is therefore essential that the MDA community establish a clear and sound view of what metamodels are, what purposes they serve, and what form they consequently should take. A lot of progress has been made in this direction in recent years, but there are still some fundamental issues that need to be sorted out.

In our view, one of the most fundamental problems is that the role of metamodeling in the MDA approach is generally looked at from one angle only. The "accepted wisdom" at the moment is that the role of metamodeling is to support language definition and/or extension. This "metamodeling is language definition" view is found in the preambles to many of the UML/MOF related OMG documents, and is even implicit in this workshop's Call for Papers.

We do not challenge the fact that one of the most important functions of metamodeling in MDA is to support the definition of languages. However, we believe that characterizing language definition as the *only* role of metamodeling is overly simplistic and ultimately detrimental to the evolution of MDA technology. To do justice to metamodeling and properly characterize its role we need to go back to the foundations of "modeling" and "meta-ness" and analyze what their integration means.

Calling a Spade Something Else

At the outset of this discussion we need to point out, of course, that the meaning of a term is determined by a common understanding within the community using it. Thus, if the MDA community (and in particular the OMG) chooses to define "metamodeling" as language definition, and to characterize other modeling activities involving meta-ness as "not metamodeling", then it is perfectly free to do so. And that would be the end of the debate. But before we go down this route we should be sure that

a) such a definition makes sense, and serves to promote rather than hinder the development of MDA technology.

b) such a definition is deliberately selected in preference to other possible definitions with a full awareness of the pros and cons.

At present we do not believe that either of these requirements holds. In particular, we do not think that characterizing metamodeling as just language definition does justice to it, and we do not think that this characterization is becoming popular because it is the best, but because the full spectrum of metamodeling is not being recognized.

In the spirit of calling a spade a spade, in the remainder of this position paper we try to present a complete characterization of metamodeling for MDA.

Form versus Content

Whenever human beings wish to communicate (whether in written or spoken form) they make statements in some kind of language. Usually communication concerns things and facts from reality but since language is part of that reality it is possible to make statements about (the use of) language as well. Consider the element "spade" in following sentence, for example.

"You should call a spade a spade"

Taking a grammatical or linguistic viewpoint, one would argue that "spade" is a "noun" and that the first occurrence of "spade" plays the role of a "subject". Hence, "noun" and "subject" are *linguistic* classifiers for "spade". Taking a semantic or ontological viewpoint, however, the term "spade" can be understood to refer to a certain type of tool or to an iconic symbol found on playing cards. Hence, "tool type" and "card identification mark" are *ontological* classifiers for "spade".

This already gives us four classifiers for "spade", and it is easy to come with a whole host of others. However, all the classifiers come in one of two fundamental flavors. The first kind of classification deals with the *form* of the statement element (what it is when we *mention* it), while the second deals with the *content* of the statement element (what it is when we *use* it). These two fundamental dimensions of classification [1] exists whatever kind of statement we are dealing with. In the case of a visual model (which is just a statement expressed in a graphical rather than a textual style), the linguistic (meta-) dimension deals with the classification of model elements according to their form (e.g. Class, Association, Attribute) and the ontological (meta-) dimension deals with the classification of model elements according to their content

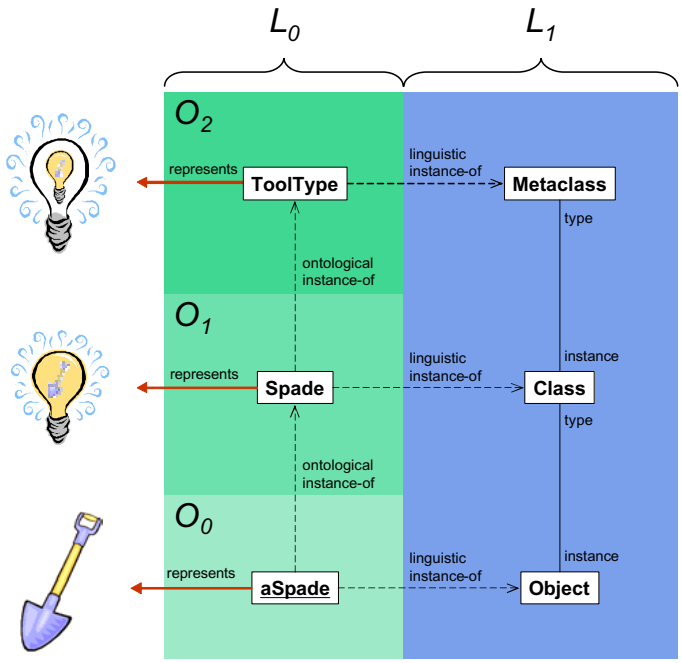


Fig. 1 Two metamodeling dimensions

(e.g. ToolType). In Fig. 1 the real world is shown in the leftmost column, a model of it in the middle column (L_0), and a language definition for the model element vocabulary in the rightmost column (L_1). Thus, the linguistic meta-boundary runs vertically between L_0 and L_1, whereas the ontological meta-boundaries run horizontally (within L_0).

Perspective and Perception

Cleary the ontological classification of types has just as much "metaness" about it as the linguistic classification of types, yet the term "metamodeling" is typically reserved for "linguistic metamodeling" only. One could argue that the O-level hierarchy is just another flavor of linguistic metamodeling by viewing the O_n level as defining a language to be used in O_{n-1}. But this would imply that adding class "Spade" to level O_1 amounts to "extending the language for level O_0". The central question therefore becomes:

> "When an object-oriented programmer defines a new class, is he/she extending the language?"

The usual answer is "no." When programmers write classes they usually don't think of themselves as extending the programming language, but rather as *using* the language. In fact, one of the key ideas that (especially Smalltalk) programmers must learn is that not all fundamental concepts are captured in the core language definition, but are provided as part of a standard library. Likewise any addition to one of the O-levels in Fig. 1 is not a language extension, but a use of the language defined in L_1.

Still, even though adding new types to level O_1 does not correspond to language extension, it is clear that metamodeling is taking place. Just as "Class" corresponds to the set of all classes (e.g., "Spade", "Person", etc.), "ToolType" corresponds to the set of all tool kinds (e.g., "Spade", "Hammer", etc.). It is, thus, unjustified to characterize any extension of L_1 as "metamodeling" and any extension of O_2 as "not metamodeling".

It appears that perspective plays an important role in characterizing model extensions as "meta-modeling" or not. Obviously, tool builders and members of standard consortiums take it for granted that L_1 extensions constitute "meta-modeling" exclusively. Yet from the perspective of a modeling language's user, the type hierarchy formed by the O-levels is much more relevant. In other words, ontological metamodeling is "user (content) metamodeling" and linguistic metamodeling is "standard (form) metamodeling".

Conclusion

Ontological metamodeling does not depend on the existence of an explicit O_2 level, but actually is alive and well in the UML today. However, rather than being supported by a natural extension of the existing O_0 —O_1 levels, it is implicitly supported via stereotypes and profiles. The effect is the same, but the metamodeling character of

profile creation is either suppressed or misleadingly cast as a “language extension activity”. In effect, metamodeling other than for language definition goes on all the time, but the predominant “standard definition” perspective reserves the term “metamodeling” exclusively for its own purposes.

However, at the end of the day “standard definition” is a means to an end and not an end to itself, so it is the user’s perspective which should be predominant. Therefore, user metamodeling should be recognized as such and be cleanly supported instead of forcing all the baggage associated with stereotypes, tagged values and the rest of profile paraphernalia on users. Important MDA techniques such as type level transformations in both “framework based” and “marking mechanism” versions [2], actually call for ontological metamodeling support.

We believe that if the current unbalanced view of metamodeling continues, the evolution of MDA technology will be stifled and the full potential of metamodeling will not be fulfilled.

REFERENCES

1. Colin Atkinson and Thomas Kühne, Rearchitecting the UML Infrastructure, ACM journal "Transactions on Modeling and Computer Simulation", Vol. 12, No. 4, 2002
2. David S. Frankel, Model Driven Architecture: Applying MDA to Enterprise Computing, OMG Press, 2003

Do MDA Transformations Preserve Meaning?

An investigation into preserving semantics

Anneke Kleppe[1], Jos Warmer[2]

[1]Klasse Objecten, Netherlands
A.Kleppe@klasse.nl

[2]De Nederlandsche Bank, Netherlands
J.B.Warmer@dnb.nl

Abstract. The definition of model transformation given by the OMG states that for all model transformations, the (sub)system defined by the input model must be the same as the (sub)system defined by the output model. If this definition holds, model transformations can be considered to be preserve the meaning of their input models. In this paper we explore the correctness of the given definition. We compare the ideal situation in which model transformations do preserve meaning, with many current day situations in which model transformations may be applied. We identify a number of situations in which the given definition is dubious. Based on the observation that this definition only holds in an ideal world, we argue that model transformations do not preserve semantics. At best, they define the semantics of the language in which the input model is written in terms of the semantics of the language in which the output model is written.

Keywords

metamodeling, OCL, semantic domain, UML, MDA, Model Driven Architecture, model transformations

1 Introduction

The Model Driven Architecture (MDA) ([1] and [2]) is a way to develop software by transforming an input model into an output model. The output model is usually closer to the source code of the system to be developed, than the input model. Using a series of transformations, also called model transformations, the software system is developed from a Platform Independent Model (PIM) to source code. In this process a series of model transformations will be linked together. In this paper we wish to zoom in on a single model transformation. Therefore, we will not use the terms PIM and PSM (Platform Specific Model). Instead we will use the terms *source model* and *target model*. The source model is the input to the transformation, the target model is the output of the transformation.

In the MDA Guide Version 1.0 ([1]) the concept of model transformation is defined as follows:

Definition *"Model transformation is the process of converting one model to another model of the same system."*

Given a source model, this model describes a system. In other words, the meaning of a model is the system it describes. A model transformation from a source model to a target model is defined such that the target model must describe the same system. Therefore, the target model must have the same meaning. The definition thus implies that model transformations preserve the meaning of the model(s) being transformed.

In this paper we explore the given definition. Going from the ideal situation, in which model transformations do preserve meaning, to the current day situations in which model transformations may be applied, we identify a number of situations in which this definition is at least dubious. Based on the observation that this definition can only hold in an ideal world, we argue that model transformations in practical situations do *not* preserve semantics. At best they define the semantics of the language in which the source model is written in terms of the semantics of the language in which the target model is written.

2 The Ideal Transformation

In figure 1, a model transformation that preserves meaning, is depicted. The figure shows that both source and target model are written in some language, and that there must be some *transformation specification* (or *mapping*) that defines how elements from one language are to be transformed into another. The transformation specification is defined at the language level, while the transformation itself works at the model level. This ensures that the same transformation can be executed time and again. The figure also shows the system that is described by both the source and target model.

Whether the source model plays the role of PIM, PSM, or CIM (Computational Independent Model) is unimportant with regard to reasoning about transformation semantics, and the same holds for the role of the target model. Likewise, the source and target languages may either be different, or the same.

Now suppose we have a transformation T, specified by transformation specification TS. T transforms a model M1, written in language L1, to a model M2, written in language L2. More important, T is a transformation that preserves meaning, therefore the system S1, specified by model M1, must be the same as the system S2, specified by model M2. Because the systems S1 and S2 are the same, the figure shows only one system.

With this we can formulate a definition of the preservation of semantics of transformations:

Definition *A Model Transformation preserves semantics if and only if the systems described by its source model and its target model are identical*[1]

1. This definition can be more subtle. A transformation can be defined as preserving semantics if te target model has at least the semqantics implied in the source model. It might potentially add semantics, as long as it doesn't invalidate the originalk semantics of te source model. This can be compared to Liskovs Substitution Principle. The systen described by the target model must behave as the system described by the source model. We have chosen the simpler definition to make the discussion points more clear.

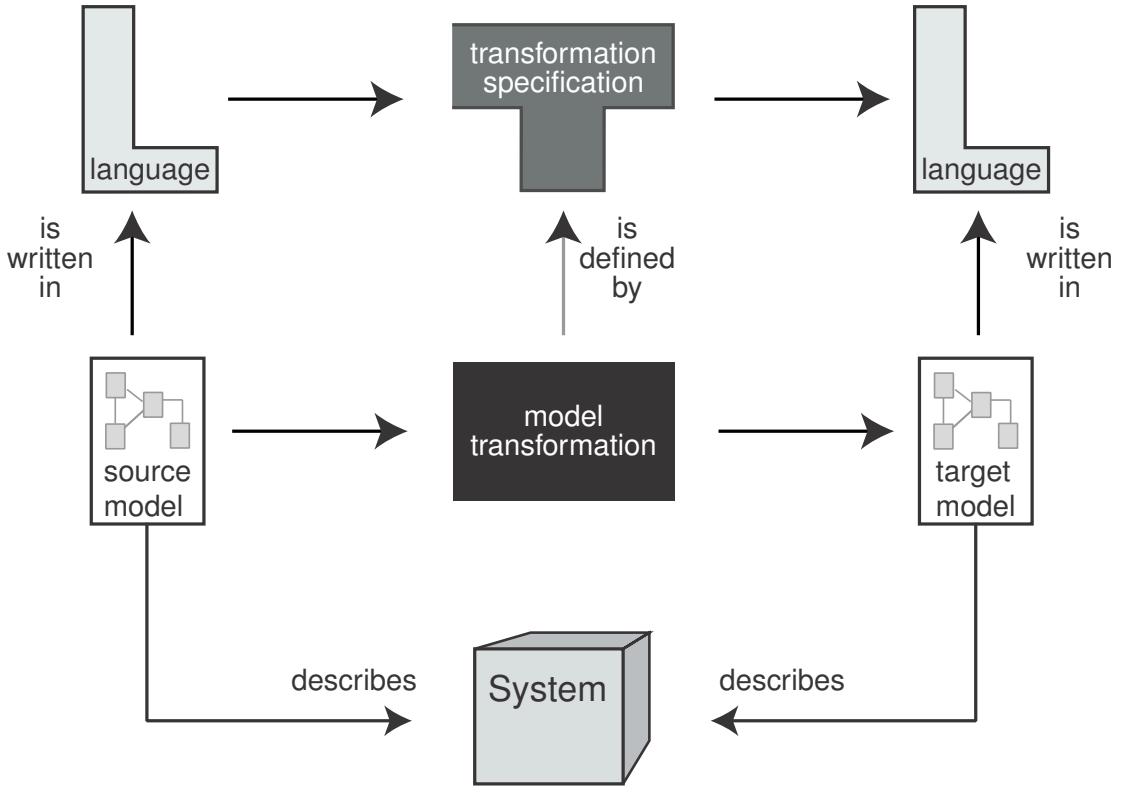


Figure 1 A Model Transformation preserving meaning

The system described by a model written in a certain language, is always a subset of the semantic domain of that language. Thus, when two models written in different languages describe the same system, the semantic domains of both languages should have something in common. Figure 2 shows the relationship between the semantic domain of source and target languages, in case the transformation is meaning-preserving. The system described by both the source and target model is part of the intersection of both semantic domains.

Now, we can formulate the following definition of preservation of semantics by a transformation specification. Note both semantic domains reside on level M0 in the OMG architecture.

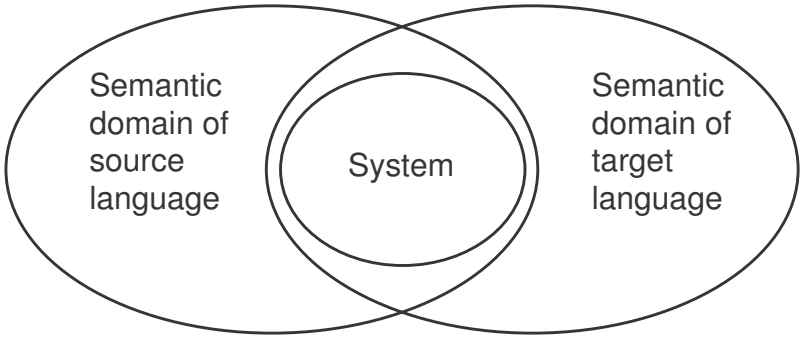


Figure 2 Relationship between semantic domains of source and target languages

Definition *A Transformation Specification can preserve semantics if and only if the intersection of the semantic domains of its source language and and its target language is not empty.*

When looking at the day-to-day practise, a number of problems can be identified that undermine the given definitions. In the next section these are explained.

3 Day-to-day Occurences of Transformations

In this section we analyze several occurrences of transformations that bcan be found in the current world.

3.1 Problem 1: Semantics of Either Source or Target Language is Not Defined

When MDA gets foot in the industry, the most common transformation will be one that takes a UML model as source, and produces program code as target. A problem with UML is that it has no definition of its semantics. The semantics described in the UML standard consists mostly of remarks and comments in plain English. There is no defined mapping from a UML model to a system at the M0 level. In fact, the UML language does not even define a semantic domain over which its meaning could be defined. Proof of this lack of semantics is the ever growing list of conference papers that attemp to give semantics to a part of the UML ([3], [4], [5], and [6]). Within the OMG languages are defined by using the Meta Object Facility (MOF). The MOF does not provide facilities for defining semantics, it only allows one to define the abstract syntax of a language. Therefore all OMG languages have this problem. This situation is depicted in figure 3.

It is not just UML that lacks a semantics definition, many languages used in software development share the same problem, e.g. XML, CWM. These languages define syntax only. Therefore, lack of a semantics definition of either source or target language is a situation that needs to be taken into account for model transformations. If a language has no semantics definition, a model in this language has (by definition!) no meaning. Transformations from such a language cannot possibly preserve meaning, as there is no meaning to be preserved.

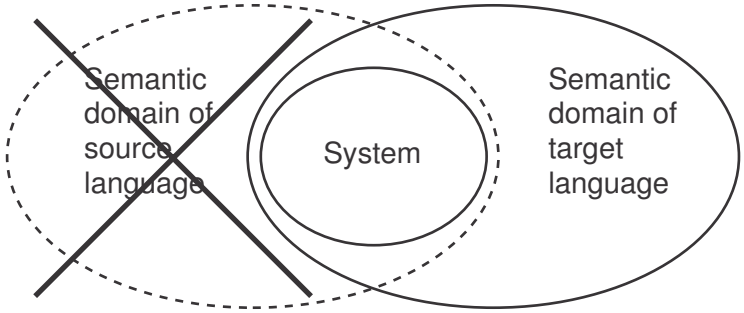


Figure 3 Problem 1: Source language has no semantics

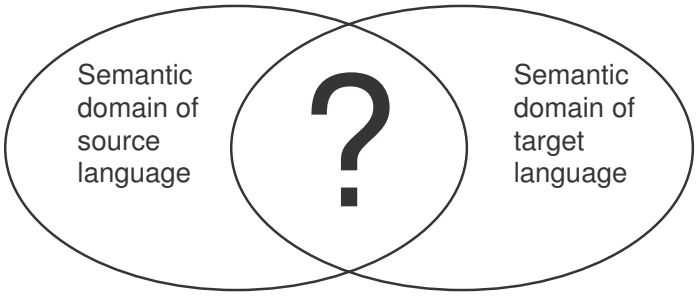


Figure 4 Problem 2: Semantics can not be compared

3.2 Problem 2: Semantics of Both Languages is Defined in Different Formalism

Another situation that will often arise, is where the formalisms used to define the semantics of source and target language are different. Suppose we want to transform a model consisting of UML class diagram and OCL invariants into C#. The OCL specification ([8], and [7]) defines a semantics for such a model. In fact it defines two semantics for this model: one using a mathematical formalism and one using the UML/OCL language itself. The semantics for C# are defined by its compilers; as most programming languages C# has a operational semantics. This situation is depicted in figure 4.

The question is, how can we determine whether the system defined by the UML/OCL model is the same as the system compiled from the C# code? To be able to determine that these two models describe the same system, we need to be able to compare the two semantic definitions that are written using different formalisms. In the example described, this would mean that we must compare one of the two semantics for UML/OCL with the operational semantics of C#. Comparing either the mathematical semantics, or the UML-based semantics with an operational semantics has not been done before, nor do we see that this can be achieved easily. To enable this proof we need a mapping between semantic formalisms. In theory, this might be done. In practice, this is considered impossible.

Unfortunately, this is a common situation. UML statecharts are very well understood and a large number of semantics definitions have been published (for instance [3]). Still, it is unclear whether the source code of systems built using these statecharts indeed conforms to their specification.

In general, comparing semantics definitions is a difficult and unsolved problem. In fact, the same problem was encountered during the writing of the OCL standard: how do we proof that the denotational (mathematical) semantics is equal to the UML-based semantics?

3.3 Problem 3: Both Languages Have Different Expressive Power

A third situation occurs when source and target languages have different expressive power. For instance, in SQL or COBOL you cannot express events or exceptions, whereas in Java you can express both. In SQL you cannot express complex behavioral

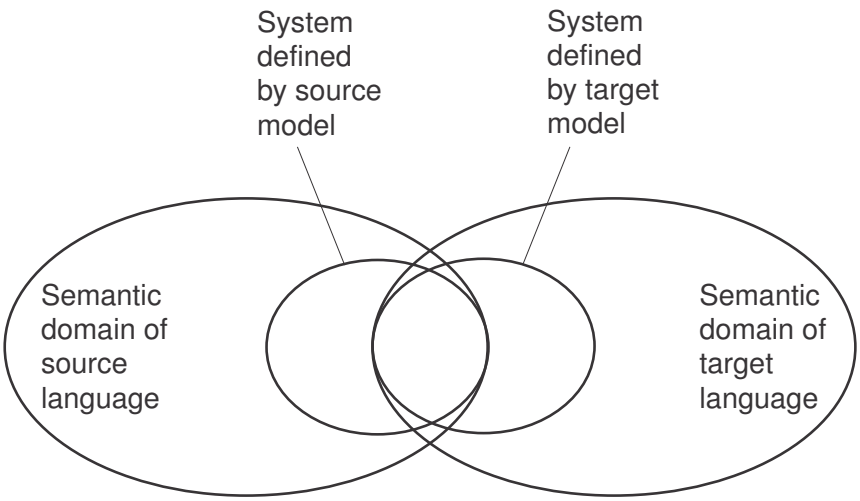


Figure 5 Problem 3: Languages have different expressive power

algoritms, in programming languages like C#, or in UML models you can. This situation is depicted in figure 5.

Whenever the expressive power of two languages differs, a system modeled with one language will usually not be exactly the same as a system modeled using the other language. Thus, definition Definition will not hold. Only if the languages have a common subset, we may be able to transform models within this subset. This is rather unpractical, because we lose the power of both the languages we use. Nobody is happy with using the greatest common denominator. At the very best, the overlap between the two languages, i.e. that which they have in common, will be large enough to model a substantial part of the source and target systems.[1] This leads us to the next problem.

3.4 Problem 4: When are Two Systems the Same?

Another common situation in which transformations will be applied, is the refactoring of models. In this case both source and target language are the same, so the same semantics definition applies for both source and target models. As an example we take a transformation between two Java source files that transforms all public attributes into private attributes with getter and setter methods. Table 1 shows two very simple classes: TestByteCode1 and TestByteCode2. TestByteCode1 is the output to this transformation. TestByteCode2 is the input.

1. A more complicated example is when one PIM is transformed into several PSM's. In this case the combination of all semantics of the PSM languages together might have the same expressive power as the PIM language. This case is very complex to handle, because we need to be able to define the exact relationship between the semantics of the various PSM languages.

As C#, Java has operational semantics defined by its compilers. When the two classes in Table 1 are compiled, their corresponding .class files differ. Thus, in theory, the op-

Table 1: Classes with and without public attribute

Class with public attribute	Class with private attribute

`public class TestByteCode1`
`{`
`  private int attribute;`
 	
`  public int getAttribute() {`
`  return attribute;`
`  }`
 	
`  public void setAttribute(`
`  int attribute) {`
`  this.attribute = attribute;`
`  }`
`}`
	`public class TestByteCode2`
	`{`
	`  public int attribute;`
	`}`

erational semantics for TestByteCode1 should be consider different from the semantics of TestByteCode2. Still, in practise the systems are considered to be the same. Furthermore, their behavior is different. Both have a different interface,which will lead to differences in the classes that reference either of the TestByteCode classes. In its turn, this will lead to differences in the .class files for an arbitrary number of classes in the system.

So, what exactly is meant by the phrase “the same system”? If we cannot take the compiler output as indicator, what else can we do? We could try testing. In theory, it is impossible to use testing as a method to establish whether two system are the same, because we can never test all possible inputs against all possible outputs. In practise, testing two systems for equally, simply to proof that a transformation preserves meaning, will not be done. One does not build two systems, when only one is needed. Another possibility would be to define explicitly which language structures are equivalent tp eachother. A language might contain many equivalence classes of structures that we consider “the same”. This is a hard and timeconsuming task, which we don’t see anybody doing right now or in the near future. The question when two systems are the same remains unanswered. We do not see an easy way out of this dilemma. This situation is depicted in figure 6.

4 Transformation Specifications as Means to Define Semantics

In the previous section we showed that there are a number of problems with the definition of model transformation given by the OMG. To claim that given semantics are always preserved by any model transformation is unrealistic. Because we consider MDA

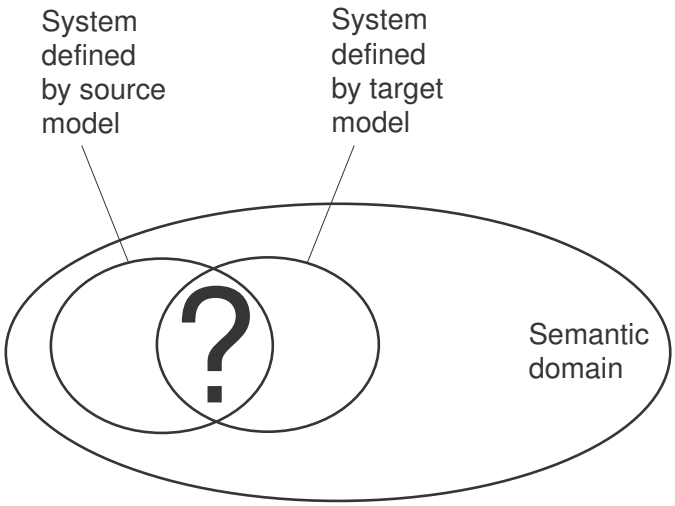


Figure 6 Problem 4: Uncomparable systems

to have an enormous potential for the software industry, we choose not to discharge MDA and model transformations, simply because transformations do not always preserve meaning. We would like to look at transformation specifications from a different viewpoint, taking into account the intention of model transformations.

The intention of model transformations is to relate models in such a way that these models describe the same system. So, let us suppose the source and target model **do** describe the same system. In that case we can claim that the transformation specification relates the semantics of source and target language. Table 2 lists the possible combinations. What is interesting is that if the target language has a defined semantics, the transformation specification defines either the semantics of the source language, or a mapping between the semantics of source and target language. In other words, model transformations are a means to specify the semantics of the source language. In fact this is a common way of defining new languages in the field of mathematics. We understand a new mathematical language because we understand the mapping to an existing mathematical language.

Table 2: The relationship between semantcs of source and target languages

Source Language	Target Language	Relationship defined by Transformation
has no semantics	has no semantics	no preservation of semantic possible => no relation between semantics of source and target language
has semantics	has no semantics	complete loss of semantics => no relation between semantics of source and target language

Table 2: The relationship between semantcs of source and target languages

Source Language	Target Language	Relationship defined by Transformation
has no semantics	has semantics	transformation specification defines the semantics of the source language in terms of the semantics of the target language.
has semantics	has semantics	transformation specification defines a mapping between semantics of source and target language.

When we have the common situation where the source model is in UML (it has no semantics) and the target model is in a programming language (having operational semantics), a transformation specification actually defines the semantics of UML based on the semantics of the programming language.

5 Families of Languages

Finally, there is a situation that will become more and more apparent. In this situation, a number of transformation specifications exist for one source language, and the source language does not have its own semantics. Each transformation specification defines a different semantics for the source language. Thus, the source language should be considered to be a language that has a number of different semantics. Indeed, such a language should be considered to be not a single language, but instead a family of languages, as the UML has been called before (references).

For instance, using the UML profiling mechanism a number of related languages have come into being. As there are (this is not an exhausting list):

- UML Profile for Enterprise Application Integration (EAI) ([9])
- UML Profile for Enterprise Distributed Object Computing (EDOC) ([11])
- UML Testing Profile ([12])
- UML/EJB Mapping Specification ([10])

The common notion is that all of these languages use the same notation (UML), however, their semantics are determined by the mapping between the notation and concepts that are elsewhere defined. It can even be argued that the success of UML is based on the fact that people can use it in many different ways.

6 Summary

We have shown that it is impossible to hold the position that every MDA model transformation must always preserve the meaning of its source model. A number of problems with this viewpoint have been identified. The problems range from languages that have no semantics at all, to the (un)ability to determine whether two systems are the same.

Rather than discarding the notion of model transformations, we take the viewpoint that model transformations must be regarded as a means to provide semantics for the language of its source model, or a means to specify the relationship between the semantics of the languages used for source and target model.

References

[1] *MDA Guide Version 1.0.1*, OMG, document number: omg/2003-06-01, 12th June 2003

[2] Anneke Kleppe, Jos Warmer, Wim Bast, *MDA Explained, The Model Driven Architecture: Practise and Promise*, 2003, Addison-Wesley

[3] D. Latella, I. Majzik, and M. Massink, *Towards a formal operational semantics of UML statecharts diagrams*, in The 3rd international conference on Formal Methods for Open Object-Distributed Systems, 1999, Kluwer Academic Publishers

[4] K. Lano and J. Bicarregui, *Semantics and Transformations for UML models*, in The Unified Modeling Language '98, Beyond the notation, 1998, Springer-Verlag

[5] Stephen J. Mellor, Stephen R. Tockey, Rodolphe Arthaud, Philippe Leblanc, *An Action Language for UML: Proposal for a Precise Execution Semantics*, in The Unified Modeling Language '98, Beyond the notation, 1998, Springer-Verlag

[6] Gunnar Övergaard and Karin Palmkvist, *A Formal Approach to Use Cases and Their Relationships*, in The Unified Modeling Language '98, Beyond the notation, 1998, Springer-Verlag

[7] Jos Warmer and Anneke Kleppe, *The Object Constraint Language, Getting your Models Ready for MDA*, 2003, Addison-Wesley

[8] OMG Document ad/2003-01-07, *Response to UML 2.0 OCL RfP ad/2000-09-03, Revised Submission, Version 1.6,* January 6, 2003, available from www.klasse.nl/ocl and www.omg.org

[9] *UML™ Profile and Interchange Models for Enterprise Application Integration (EAI) Specification*, OMG, Draft Adopted Specification, January 2002

[10] Java Community, *UML/EJB Mapping Specification*, Process document JSR26, 2001.

[11] *A UML Profile for Enterprise Distributed Object Computing*, Version 1.0, Revised 22 August 2001, OMG Document Number: ad/2001-08-20

[12] *UML Testing Profile (final submission)*, March 2003, available from www.omg.org

MDA components:
Challenges and Opportunities

[1]Jean Bézivin, [2]Sébastien Gérard, [3]Pierre-Alain Muller, [4]Laurent Rioux

[1] ATLAS Group (INRIA & IRIN)
University of Nantes
44322 Nantes cedex 3, France
jean.bezivin@sciences.univ-nantes.fr
[2] CEA - Centre d'Etudes de Saclay
CEA-List/DTSI/SLA/L-LSP
F-91191 Gif sur Yvette
France
sebastien.gerard@cea.fr
[3] ESSAIM - University of Mulhouse
68093 Mulhouse, France
pa.muller@uha.fr
[4] Thales TRT
Domaine de Corbeville
91404 Orsay
France
laurent.rioux@thalesgroup.com

Abstract. The MDA™ approach aims at providing a precise and efficient framework for software and system production and maintenance. Artifacts used or developed within the corresponding processes should be considered as assets for the organizations where the MDA is being deployed. These processes may themselves become part of the accumulated explicit knowledge of enterprises. We are going therefore to talk about "MDA components" to designate the elements of this information capital. However this concept of MDA component is still loosely specified. Our objective in this paper is to contribute to a more precise definition of this central notion. We also propose a set of criteria that should help to understand more clearly what a MDA component candidate with good properties is. The paper is more intended to set up a basis for constructive discussion than to offer definitive answers and closed solutions. Among the proposals, we suggest here to consider precisely the notion of "tool signature", i.e. the set of precise services any tool is offering. We also suggest making clear the distinction between pure MDA tools and import/export tools, this distinction being based on the notion of technological space. Such proposals and others are not only made for descriptive reasons, but are intended to support a strong operational view for an interoperability framework for model management. The ultimate goal is to reach a high level or reusability and adaptability of these MDA components in modern software production and maintenance schemas.

1 Introduction

The MDA™ [10, 15] approach aims at providing a precise and efficient framework for software and system production. Artifacts used or developed within the corresponding processes should be considered as assets for the organization where the MDA is being deployed. Therefore we are going to talk about "MDA components". However this notion is still loosely specified. Our objective is to contribute to a more precise definition of this central notion. We also propose a set of criteria that should help to understand more clearly what a MDA component candidate with good properties is.

The paper is more intended to set up a basis for constructive discussion than to offer definitive answers and closed solutions to the hot issue of defining the MDA component paradigm. As much as possible we have followed the OMG standards and terminology. However, from place to place, in order to give a coherent and global vision of MDA, and for the sake of abstraction as well as resilience if face of the evolution of standards, we have taken the freedom to introduce some new terms.

Among the proposals made in this paper, we suggest to consider precisely the notion of "tool signature", i.e. the set of precise services any tool is offering to manage models. We also suggest making clear the distinction between pure MDA tools and import/export tools. This distinction is based on the notion of technological space (TS) that has been introduced in [8]. Such proposals and others are not only made for descriptive reasons, but are intended to support a strong operational view for a Model Interoperability Framework (MIF).

If the MDA paradigm is to succeed, then we need to understand clearly what MDA components are because they will be considered as the basic units of industrial assets. To achieve this purpose, at least the following issues have to be solved. How it is possible to characterize an MDA component, i.e. which properties does it have to fullfill (*e.g.: (P1) An MDA component is independent of any given proprietary platform1; (P2) An MDA component is independent of any given specific case TOOL2; (P3) An MDA component can be stored and retrieved in a uniform way…*)? *(P4) An MDA component may be deployed in the context of different software processes or methods*. But also, how MDA components are designed, implemented, used, adapted, named, accessed, etc.

An MDA component is a packaging unit for any artifact used or produced within an MDA-related process, including the process itself. This being said, several questions immediately arise. For example how is an MDA-component related to or different from a general software component? Is there a market for pre-built MDA components, i.e. can we envision the availability of large libraries of such components that can be acquired and extended for specific needs? How will these libraries be organized? Can we define a classification of MDA components? If these components are stored and retrieved in a uniform manner, it is likely that the various tools available in the model engineering scope will apply only to certain categories of such components. How to define with accuracy the spectrum of component types each tool

[1] e.g: C#/DotNet, Java/EJB…
[2] e.g: Poseidon, Rational Rose…

is able to process? These are some of the problems to which we need to bring initial answers in the rest of this paper.

One of the main ideas presented here is that most, if not all, of the considered MDA components contents are models[3]. This is consistent with the overall MDA philosophy: “Models as first class citizens”. From this observation we may infer that the notion of meta-model (i.e. the "type" of a model) should be the basis for the aforementioned classification and definition of the spectrum of possibilities of each individual tool. We consider this to be the central idea for organizing the MDA landscape.

Obviously this is a very abstract starting point that should be much elaborated. For instance how could we agree on a standard global identification of meta-models? Should we use for example identification schemas similar to XML name spaces? This brings in turn more general questions like the definition of a standard MDA meta-model. Could meta-models defined outside the OMG normalization process also be considered? Let’s imagine an organization, like OASIS in the XML landscape compared to W3C, hosting collaborative definitions of specific domain meta-models. What would be the status of these meta-models compared to the normal recommended standards of the OMG? What would be the status of a global model4 identifying all known meta-models? Which naming and accessing schema will be most suitable?

If the MDA community is not yet ready to answer all of these questions, we need however to anticipate providing soon some initial answers. In order to facilitate our thinking in this area, let us postulate the existence of a very general abstract framework intended to host such MDA components and the corresponding processing tools. This infrastructure called below MIF (for Model Interoperability Framework) is only intended to draw the big picture for an efficient discussion.

This paper is organized as follows. First we introduce the notion of technological space. Then we discuss what are MDA models and therefore what are the basic entities populating the MDA TS. We offer, in the following section, a description of a MDA Workbench and the Model Interoperability Framework. The next sections present a particular layer of this organization called Basic Access Model Services and discuss model transformations. Before the conclusion we give some practical and typical illustrations of MDA components.

2 The notion of technological space

All along this paper, the concept of Technological Space (sometimes abbreviated here to TS) is used to illustrate our work on MDA paradigms definitions. This section aims then at giving the reader the minimal information required to understand the concept (more details may be found in [8]). A TS is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It is often associated to a given user community with shared know-how, educational support, common literature and even workshop and conference meetings. It is at the same time

[3] We consider here profiles, meta-models and meta-meta-models as special cases of models.
[4] We call this a "mega-model" later.

a zone of established expertise and ongoing research and a repository for abstract and concrete resources. Although it is difficult to give a precise definition of Technological Space, some spaces can be easily identified (Fig. 1): Programming languages concrete and abstract syntax (Syntax TS), Ontology engineering (Ontology TS), XML-based languages and tools (XML TS), Data Base Management Systems (DBMS TS), Model-Driven Architecture (MDA TS) as defined by the OMG as a replacement of the previous Object Management Architecture (OMA) framework, etc.

Each of these previous TSs has basic properties and features ensuring strong and weak points: (i) the Syntax TS is a very rich and stable technology that has been maturing for more than fifty years with formal tools (e.g. context-free grammars) and mapping these onto executable machines; (ii) the Ontology engineering TS has some very precise definition tools like conceptual graphs and description logics; (iii) the MDA TS has received industrial agreement and backup on its Unified Modeling Language (UML) and Meta Object Facility (MOF) standard recommendations. It has made available many industrial CASE tools like supporting model creation/browsing/…; (iv) the DBMS TS has a long record of dealing with huge volumes of structured data; (v) the XML TS has also wide industrial acceptance in the field of semi-structured data representation. It has also some interesting and widely available tools such as XSLT transformation engines.

Fig. 1 illustrates also that no TS is an island. There are bridges among the spaces and these bridges also have particular properties. In Fig. 1 we do not represent all the bridges between the various TSs and the figure does not suggest any superiority for any one of them.

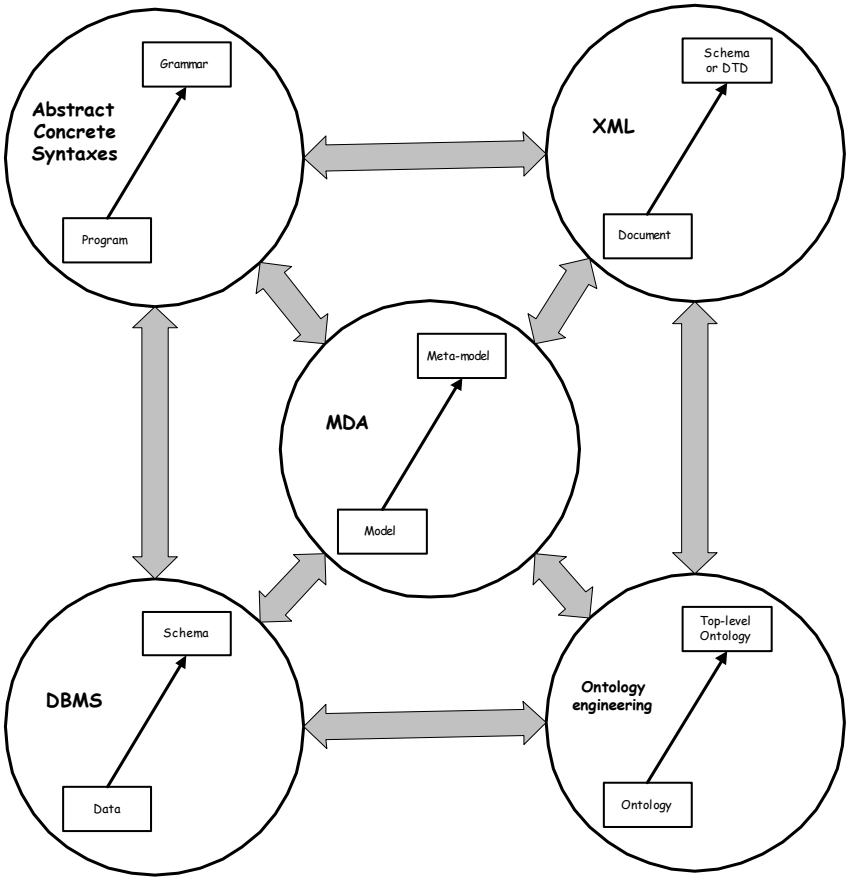


Fig. 1. Samples technological spaces with some inter-relationships

3 MDA models overview

One of the main common concepts of the MDA identified in the various ongoing thinking about MDA is of course the paradigm of model ([4, 6, 14]).

3.1 Models and structural types of models

A model represents a particular aspect of a system under construction, under operation or under maintenance. A model is written in the language of one specific meta-model. A meta-model is an explicit specification of abstraction, based on shared agreement. A meta-model acts as a filter to extract some relevant aspects from a system and to ignore all other details. A meta-meta-model defines a language to write meta-models.

A meta-meta-model is usually self-defined using a reflexive definition, and is based at least on three concepts (entity, association and package) and a set of primitive types. The OMG MDA postulates the use of the MOF as the unique meta-meta-model for all IT-related purposes. The MOF contains all the universal features necessary to build meta-models and to operate on them (those features are not specific to a particular domain language).

The UML meta-model plays different roles in the MDA. First it defines the language used for describing object-oriented software artifacts. Second, its kernel is synchronized with the MOF for practical reasons, as there are many fewer meta-modelers (people building meta-models) than modelers (people building models). As a consequence, it is not optimal to build specific workbenches only for these few meta-modelers. By making the MOF correspond to a subset of UML, it is possible with some care to use UML tools for both usages. As a consequence the MDA is not only populated by first class MOF meta-models, but also with extensions of meta-models defined by profiles for specific purposes. This is mainly done for practicality (for instance widening the market of UML tools vendors), but this duality may lead to some redundancy between profiles and MOF meta-models (it is even possible to find conversion tools).

A meta-model defines a specific modeling language. It may be compared to the formal grammar of a programming language. In the case of UML the need to define variants of the base language was expressed. The UML meta-model was then equipped with extension mechanisms so as to define specializations called UML profiles. These extension mechanisms have later been moved upward to the MOF level, so that they may be shared by all MOF meta-models.

There are many examples of UML profiles (e.g. "UML for Corba", "UML for C++", "UML for Scheduling Performance and Time" (real-time applications), "UML for EJB", "UML testing", "UML for EAI", "UML for QoS and fault tolerance", "UML for EDOC" (Enterprise Distributed Object Computing", etc.). Some are standardized by OMG working groups and other are independently defined by user

groups or even by individuals. The situation is similar to the XML galaxy where independent users may decide to develop specific DTDs or XML schemas. In the case such a development becomes popular or widely available, it may constitute a de facto standard. UML CASE (Computer Aided Software Engineering) tools vendors are another important source for profiles. These organizations may provide libraries of profiles, usually adapted to the specificities of their tools and with limited portability.

A MOF meta-model is composed of three parts: terminological, assertional and pragmatics. The terminological part roughly corresponds to UML class diagrams depicting the key concepts of the meta-models and their static features (attributes and associations) as well as detailed textual descriptions. The assertional part roughly corresponds to OCL (Object Constraint Language) assertions that may decorate the various elements of the meta-model, ensuring consistency of the models via well-formedness rules. The pragmatics corresponds to details that could not fit into the previous parts. Example of a pragmatic item is for example how to draw some particular concepts or relations. In UML 2.0 a tentative has been made in the Diagram Interchange RFP [16] to separate the content aspects from the presentation aspects in a given meta-model. This shows the tendency to reduce the pragmatic part by integrating the corresponding aspects into separate meta-models. Usually the pragmatics elements are expressed in natural language informal descriptions.

The MOF also contains features to serialize models and meta-models in order to provide a standard external representation. The XMI standard (XML Metadata Interchange [13]) defines the way serialization is performed. This is a way to exchange models between geographical locations, humans, computers or tools. When a tool reads a XMI serialized model (a UML model for example), it needs to check the version of the meta-model used and also the version of the XMI applied schema.

3.2 Model classification

There are several kinds of models and several possible classifications. MDA relies intensively on the basic classification consisting in splitting models in both categories: PIM and PSM. The former are to be independent of the platform, while the latter are specific to a given one (Fig. 2).

In addition to these two main kinds of models, there is currently a lot of thinking about extending this basic classification. Very often this latter is kept implicit internally in software production tools (CASE tools or IDEs). This is in contradiction with the global goal of the MDA that is to make **explicit** the notion of platform independence and the notion of platform binding. The concept that comes closer to the idea of a platform is the notion of virtual machine. A virtual machine may be built on top of other virtual machines and the relations of abstraction between different virtual machines may be made explicit. There is a lot of insight in the explicit definition of virtual machines that may be reaped out of the research in these topics that occurred in the 80's. When the descriptions of platforms (i.e. PDM for Platform Description Model) and virtual machines are clarified we may then tackle the definition of a PIM (Platform Independent Model), a model containing no elements associated to a given platform. In other times this was simply called a business model, but as for platform models we need to progress now towards a less naive and a more

explicit view. The first idea is that the PIM is not equivalent to a model of the problem. Many elements of the solution may be incorporated in a PIM as long as they don't refer to a specific deployment platform. For example we may take algorithms hints into account. There may be several PIM to PIM transformations before arriving to the state where a PIM may be transformed into a PSM. To express this idea, some use the notion of a CIM (Computation Independent Model), which is a PIM where the problem has not yet been worked out as a solution. This entire ongoing work may lead to an extended classification as depicted in Fig. 3.

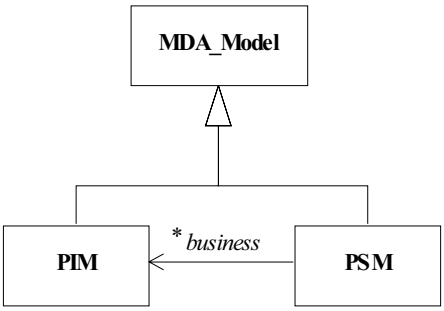


Fig. 2. Basic MDA classification

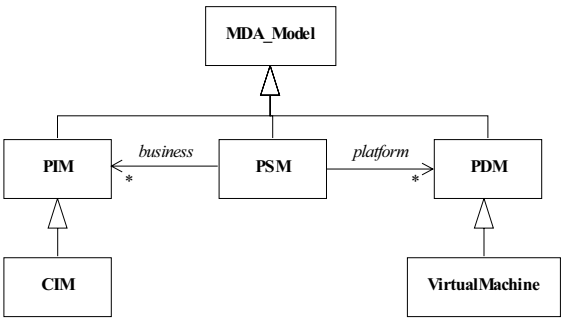


Fig. 3. Extended MDA basic classification

Finally MDA is not the only model-oriented technology[5]. It is therefore necessary to distinguish between MDA models and external models. In order to be more precise about this distinction let us look at the following example. The same Java program may be represented in three different technological spaces: programming language syntax, XML document and MDA. The former two are external models and the latest is internal for the MDA technological space. This is illustrated in Fig. 4 and Fig. 5. The standard source text of a given program, the XML document of the same program expressed in the JavaML DTD for example [11] and finally the model of the same Java program expressed in a given Java meta-model may be converted one from the other by bridges between these technological spaces. However, inside each technological space, there are tools to handle the specific representation. It would be a pity to rebuild all these tools inside the MDA space if they perform well in other spaces. It seems much better to propose well-engineered bridges between the different spaces. This will be described later when discussing MDA import and export services.

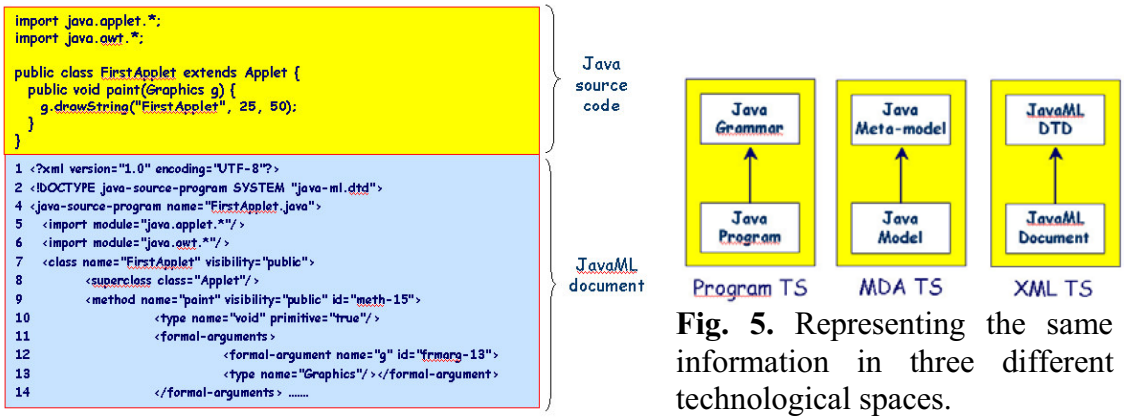


Fig. 5. Representing the same information in three different technological spaces.

[5] There are several other non-MDA model oriented technologies.; to name a few, Model Management [Ph. Bernstein], MIC[Model Integrated Computing], STEP/EXPRESS, etc.

Fig. 4. Java standard source code and JavaML representation

In the context of the MDA models, it is also possible to classify models in function of their focus. For example, Fig. 6 describes four broad categories of models depending on their goal: product models (e.g. a UML model is a product model describing an object-oriented software system); process models (e.g. a SPEM model is a kind of process model); mega-models (A mega-model is a model with other models as elements. It may serve as a global map for the information assets of a company); and transformation models (these latter are described in more details later in the paper).

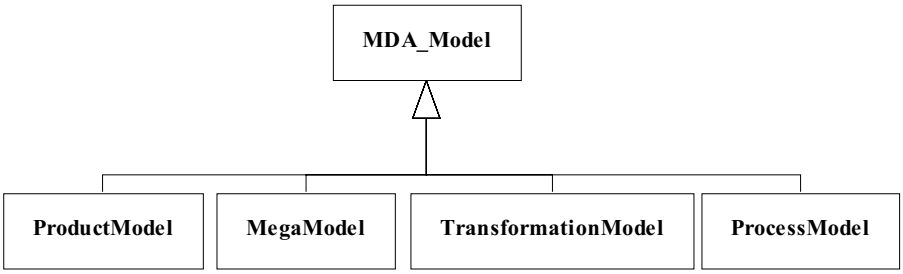


Fig. 6. Sample 1 of model classification

Of course the classification proposed in Fig.6 is far from being complete. It is very likely that well see many other broad categories of models being identified in the MDA scope. There are also multiple other possible criteria for model classifications:

- *Dynamic vs. static* – a static model is invariant while a dynamic model changes over time. This should not be confused with the static and dynamic aspects of systems. The most common situation is a static model of a dynamic system.
- *Executable vs. non-executable* – A Java program may be naturally considered as an executable model. The Action Semantics can be used to give executability to a UML model.
- *Atomic or composite* – an atomic model only consists of basic elements while a composite model contains at least another model. The possibility of composition is defined at the meta-model level. The containment hierarchy for models is distinct from the specialization hierarchy.
- *Primitive or* derived – a derived model may be obtained from other models (primitive or derived) by a derivation operation or a sequence of such operations. A derivation operation is a simple and deterministic model transformation.
- *Essential or transient* – an essential model is a model that is intended to stay permanently in the model repository system. A transient model is disposable, i.e. it has been used for some temporary purpose and has not to be saved. Compilers use for example some intermediate files to carry on their transformations from high-level languages to low level machine languages.

The previous list of criteria is by no means complete. We could mention for example test models, code source model, pivotal models for interchange…

4 MDA workbench suite

The most apparent external elements in an MDA workbench are the precise tools composing this workbench. An MDA tool implements a set of high-level services ensuring model manipulation. In order to make that efficient and to foster interoperability these tools will have to be plugged on the top of a more abstract model framework.

4.1 MDA tool definition

Fortunately in this context we should be able to propose a rather precise definition of a tool: it is an operational implementation of a set of operations applicable on specific models. The meta-models supported by a tool should be exhaustively and explicitly defined. As a consequence of this, the tool may also be referenced in a given process model for automatic (stand-alone) or semi-automatic (in association with a human agent) execution.

An MDA tool implements a set of operations on MOF based models and meta-models. For example Rational Rose, Objecteering or Argo UML implement model creation/browsing/serialization (XMI) but also provide code generation facilities (for a limited set of languages like C++ and Java), etc.

The main characteristics of MDA tools are that they should be interoperable, i.e. they should be able to operate on a virtual MDA platform like the MIF (Fig. 9), collectively implementing complete chains of operations on models. This is one of the main advantages that should come from the usage of standards like MOF, XMI, SPEM, UML, CWM and many others.

The most visible elements in the MDA landscape are the tools used to perform operations on models. Therefore, one can distinguish many different kinds of tools as illustrated in Fig. 7: IDEs (Integrated Development Environment); CASE (Computer Aided Software Engineering); CARE (Computer Aided Requirement Engineering); CAPE (Computer Aided Process Engineering); model transformation tools; code generators…

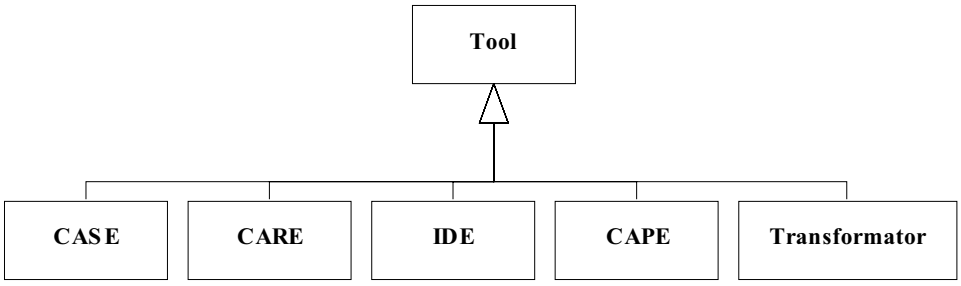


Fig. 7. Different kinds of tools

There are plenty of tools that can be associated to each category. For example *Poseidon* may be considered as a CASE tool, *MIAStudio* as a model transformation tool and *VisualAge* as an IDE. Obviously the real situation is much more complex and we can have for example tools that are at the same time CASE tools and IDEs.

A given tool implements a number of services; this is the ***signature*** of the tool (Fig.8). Each service may be defined by a model and may take input and output models. There are mainly two kinds of models from the MDA perspective: MDA models and external models. Strict MDA models have been discussed above. An external model is any kind of artefact that could generally be described by the word model like a program source code, an XML document, etc. It generally pertains to another TS. An import service may transform an external model into a strict MDA model while an export service does the reverse.

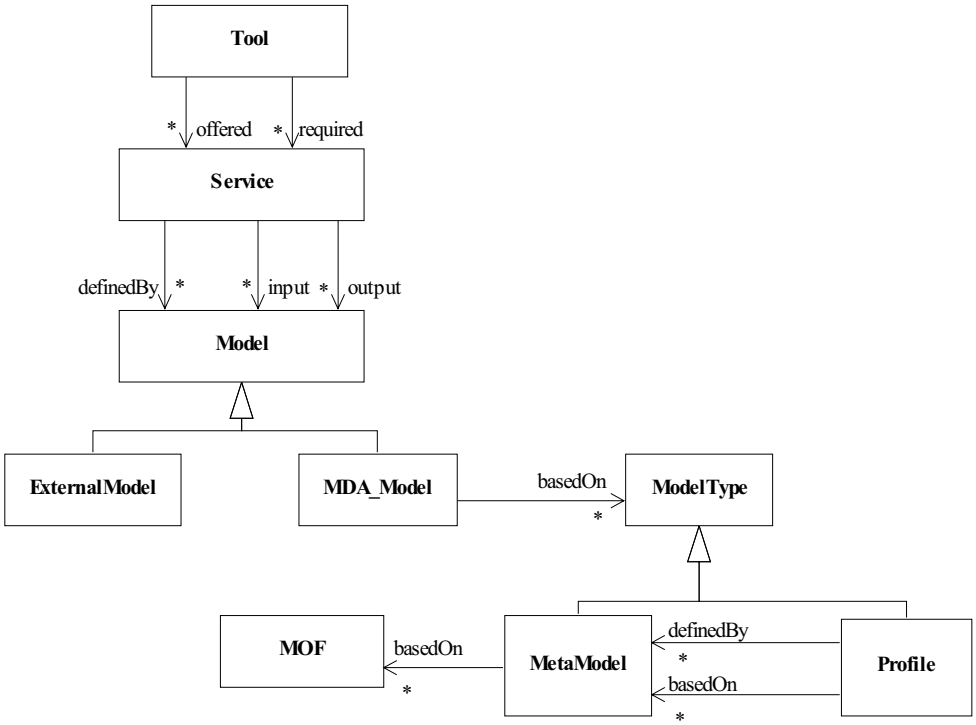


Fig. 8. General organization of an MDA-tool

4.2 Model interoperability framework

In the near future, it is likely that MDA workbenches will consist of several MDA tools integrated on top of general abstract frameworks intended to provide facilities to integrate or/and to make operate together this set of tools. Platforms such as NetBeans/MDR, DotNet/Phoenix or Eclipse will probably offer basic model engineering services. Our intention is then not to concurrent them, but to provide added values upon these basic model management platforms.

Moreover, one of the major goals of our proposal is to address at the same time short term, medium-term and long-term objectives. Instead of talking about a model interoperability bus, we prefer to position our work in the more general context of a generic platform called **MIF (Model Interoperability Framework)**. In addition to the usual model exchange facilities (mainly based on XMI-encoding) possibly augmented with SOAP-level protocols, we hope to be able to take into account the

variety of other industrial model engineering platforms soon to become available (Fig. 9).

In order to achieve this objective, MIF relies on both following tenets. Firstly, it is based on the notion of model as a first class entity [5]. Secondly, it offers total independence towards specific industrial platforms, as stated in the MDA component properties sample described above (P1). The only dependence accepted in the MIF is on open standards. The perimeter of the MIF is then aligned on the MDA™ as defined by OMG. Any use of the word model in the context of the MIF means an "instance" of a MOF-compliant meta-model. If a model refers to a profile, this latter must then in turn refer to a MOF-compliant meta-model. This means that any model is explicitly "typed" by its meta-model.

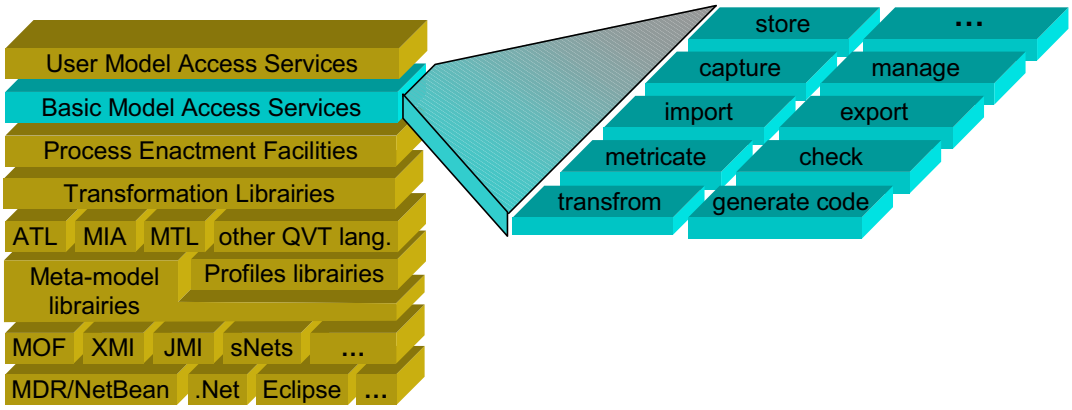


Fig. 9. Overview of the Model Interoperability Framework (MIF)

As depicted on the right side of the Fig. 9, MIF ensures a number of basic model access services. Some of them may remote service accesses, eventually accessible through facilities similar to Web Services. Among the services, we may quote capture, persistency, normalization, alignment, etc.

One of the most important operations on models is the transformation operation. Transformations in the MIF are expressed for example in the ATL language (ATLAS Transformation Language) [2]. ATL will conform to the QVT recommendation when it is ready. There are also other transformation languages that will comply with the QVT standard like MTL defined by INRIA/Triskell, MIA defined by Sodifrance or TRL defined by France Telecom. The notion of a PIT (Platform Independent Transformation) has been defined in [3]. The importance of HOTs (Higher Order Transformations) has been outlined in [2]. We completely stick to these definitions and we'll use them practically in the context of the MIF platform.

While MIF is fully in the scope of the MDA it needs to be as open as possible with other standard organisms (e.g. W3C, etc). For example, in some cases, we may offer a translating service from the QVT compliant ATL to XSLT. One of the originality of the MIF is then to provide generic services to import and export models from non-MDA TS[6].

[6] The MIF proposed solution to import/export external TSs is original but inspired from the Eclipse plug-in system and general Web services. It is not discussed in this paper.

5 Basic Model Access Services (BMAS)

A model may be built, updated, displayed, queried, stored, retrieved, serialized, etc. When most of these operations are applied, there is an implicit or explicit meta-model. A UML CASE tool has very often an integrated built-in UML meta-model. Sometimes it is possible for such a case tool to dynamically adapt to new versions of the UML meta-model. In a similar way, a meta-model may be built, updated, displayed, queried, stored, retrieved, serialized, etc.

Efficiently storing and retrieving a model or a meta-model to/from persistent storage is not always easy to implement, especially when configuration and version management are involved. In many cases using simple file systems after XMI serialization lacks efficiency and does not scale up.

Filtering a model means extracting a specific view on this model. The important question here is how the view is specified and if this operation may be considered as a dyadic operation producing another model. There are many other apparently monadic operations that turn out to be dyadic, if we are able to define the argument as a meta-model or a model. Some examples are measure, verification, normalization, optimization, etc.

Obviously one of the most popular operations on models is code generation. One may convert a UML model into a Java or a C++ programs. Many aspects of this kind of operation should be considered here. In some cases we may consider bidirectionality, i.e. backward as well as forward transformation, with for example transformations of C# code into UML model. However one should not be misled by the apparent simplicity of these transformations. Usually the underlying algorithms are quite complex and much progress will be made in the coming years in this area. If we consider the target language (C++, C#, Java, etc.) as defined by a meta-model (corresponding to its grammar), then we may envision generic transformations and really parameterized tools.

6 Model transformations

The keystone of the BMAS layer of the MIF is without any doubt the transformation service. This point will now be discussed in more details.

6.1 Some basic definitions

Let's consider the transformation *t* transforming a model *Ma* into another model *Mb* as defined by: *t: Ma -> Mb*, where model *Ma* is supposed based on meta-model *MMa,* and model *Mb* on meta-model *MMb*. We note this situation as: *sem(Ma,MMa)* and *sem(Mb,MMb).* As a matter of fact, a transformation is like any other model. So we'll talk about the transformation model *Mt* defined as: *Mt: Ma->Mb.* Obviously since *Mt* is a model, we postulate the existence of a generic transformation meta-model *MMt*, which would be similar to any other MOF based MDA meta-model: *sem(Mt,MMt)*, *sem(MMt,MOF)* and *sem(MOF, MOF)*.

The nice property here is that we may envision the possibility of transformations producing transformations (higher order transformations): *Mx: Mu->Mt*

Obviously the existence postulate for *MMt* is based on the forthcoming outcome of the OMG MOF/QVT RFP 12 on the transformation language.

The schema described above should be completed in several aspects. First one should provide the process model associated to a set of transformations, i.e. a definition of when the transformations are applied, who is applying them, what are the pre-conditions and post-conditions of these transformations and so on.

In some cases the transformation takes some particular form if the source and target meta-models are in the relation of refinement like a CORBA and a CCM meta-model.

We may also understand the need to check, before applying a transformation, the consistency of this transformation by studying the relation between both meta-models.

Furthermore there is a subtle and important aspect that is related to transformation, which is traceability. A transformation operation should produce a traceability model *Mtr* between the elements of the target model Mb and the elements of the source model *Ma*. Obviously this traceability model is based on a given meta-model *MMtr*. There are obvious relations between *MMa*, *MMb*, *MMt* and *MMtr*. The study of these relations is at the hearth of the definition of the MDA.

Presently the main issue in the MDA initiative is to succeed in defining "transformations as models". This means that there should be a generic MOF-compliant transformation meta-model defining some kind of UTL (Unified Transformation Language). The answers to the ongoing MOF 2.0 QVT [12] (Queries/Views/Transformations) should be shortly known and this will have on the MDA technology an impact as least as important as the impact of XSLT got on the XML community.

6.2 Transforming models

Now we must consider that similarly to *Ma* and *Mb*, *Mt* is also a model and thus is written in the language of its meta-model [9]. This is further elaborated in Fig. 10.

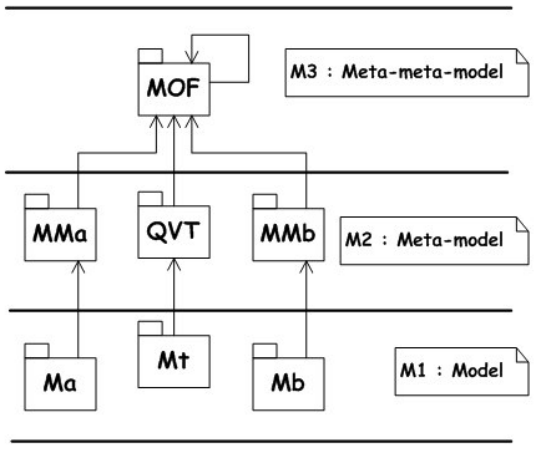


Fig. 10. The place of a transformation language

Fig. 10 postulates that there exists this common and unique transformation language, QVT. A given transformation operation is thus represented as follows in the basic relation: ***Mb ← f (MMa, MMb, Mt, Ma)***. This means that a new target model *Mb* based on meta-model *MMb* is obtained from the source model *Ma* based on meta-model *MMa*, by applying a transformation *Mt* based on the standard transformation language.

6.3 Weaving models

One of the difficult and unanswered questions about the MDA is whether we need to offer services much more complex than transformations. If we look at the classical ***Y*** organization [17] as described in Fig. 11 for example, we see that there are many successive transformations in the three branches of this ***Y***. But the difficult question is about the binding itself. Is this merging operation M reducible to a transformation operation or to a combination of transformations?

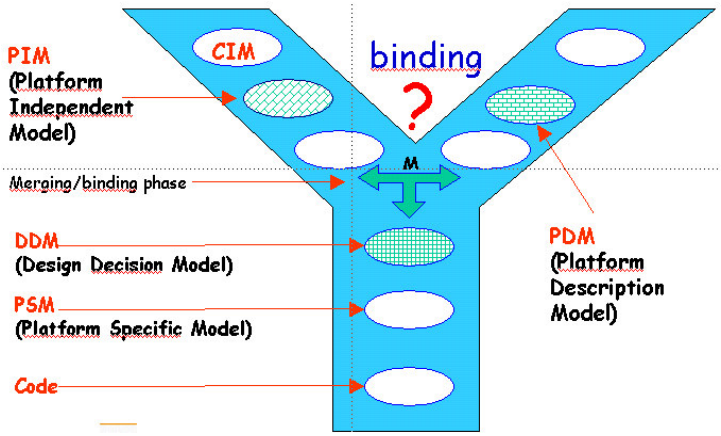


Fig. 11. The Y organization

7 MDA in action

This section will be dedicated to exemplify some of the MDA paradigms previously presented: some examples of MDA component are firstly sketched; and finally an experiment of a full MDA workbench dedicated to Distributed Real-time Embedded Systems development is outlined illustrating the concept presented in section “4 MDA workbench” p.9.

7.1 Sample of MDA components

In order to make more concrete the approach, let us present some specific examples of MDA components here. Once again there is absolutely no objective of exhaustivity in this list intended only for illustration purposes.

The most typical MDA component is a transformation. Such a component refers to a source meta-model and a target meta-model. The transformation is supposedly written in a given language, also referred by its meta-model. Note that the source, target or language meta-models may be abstract meta-models, i.e. the transformation could be applied on specializations of these meta-models. A given engine may perform the transformation if it has the capacity to process the language defined by the specified meta-model. This engine may be considered as a tool implementing a model transformation service. A given engine may have the capability to process several kinds of transformation languages. This can be done by various ways of interpretation or transformation. For example a given engine may discover that it does not have the direct capability to interpret a language, but it knows of another transformation component that could be used to transform the transformation itself so that it may run the output.

The second typical MDA component is a full-fledged meta-model. A meta-model has a name, a unique identifier (URI), a definition, a purpose, a version number, a serialization format, etc. It may be a specialization of another meta-model and it may be later specialized. A meta-model may be atomic or composite. Another related kind of MDA component is a profile, as defined in UML 2.0 and MOF 2.0. A profile makes reference to its base meta-model.

We envision the existence of general Web sites with free available resources. Many general utilities will be accessible, like general or specific transformer components. Just to quote some examples mentioned in [2], we may have standard transformations allowing to move from one version to another version of a meta-model, like from UML 1.4 to UML 1.5. Also, but less typical, we may have transformation allowing to move from one serialization format to another one, like from XMI 1.0 to XMI 1.1 or from XMI 1.1 to XMI 1.2. Combining these transformations will allow to transform a model in UML 1.4/XMI/1.0 into an equivalent model in XMI 1.5/XMI1.0. Other standard transformers will allow moving between a given profile and a meta-model. Another interesting component that should be available on a MDA resource workbench is a promoter [2], i.e. a component able to transform a model into a corresponding meta-model. We have experimented with such a promoter completely written in the ATL language. This comes as a replacement for a popular tool named UML2MOF and available in the MDR/NetBeans environment (http://mdr.netbeans.org/uml2mof/).

Finally there will also be available resources on a MDA component site for a lot of other operations like standard documentation production for given meta-models like UML, standard code generators for various target languages, etc. More advanced components will also be available. For example checking the consistency or other properties of a UML model is generally not a yes or no answer. The result is another model and the definition of the consistency rules will be given by a separate meta-model. All these artifacts will be specializable and composable MDA components available through a standard component management framework.

7.2 An example of MDA workbench for DRES[7]

Engineers of DRES domain are already using model-oriented approaches such as SA/RT, SDL…
However, these approaches are not yet entirely satisfactory, since they still require advanced real-time development skills. The MDA-workbench outlined here and called ACCORD/UML has originally been designed to lighten the daily workload of automotive engineers, by relieving them of need for real-time expertise [18, 19, 20, 21]. To achieve this purpose, it proposes a set of MDA artifacts assisting the users all along the development cycle of their applications. The following items are examples of such components:

- *DRES intrinsically require various points of view* – process definition and tooling associated with UML extensions that provide high-level concepts can formalize the content of the models as well as their interdependency. It is then possible to ensure visibility, on a model, of requirements corresponding to implementation concerns and to easily extract or modify them without changing the rest of the model.
- *DRES implementation choices vary* widely – definition of CDMs (Computation Description Models) and Platform Description Models which allow separation of these elements from the rest of the application model. Dedicated transformations can then take charge of most of the final implementation activities.
- *Performance is often a "sensitive" issue for DRES* – code generation heuristics have shown that it is possible to both manipulate high-level concepts at the model level and ensure effective implementation, thus eliminating the high overheads usually associated with implementing and running these concepts at the execution level.
- *DRES are often critical to testing or validation* – definition of UML extensions eliminates ambiguities and provides full semantics, thus enabling development of application model mapping procedures on formal models used with formal analysis tools to derive test sequences or determine the feasibility of scheduling.
- *DRES generally require highly skilled developers* – generic implementation patterns and architectures are widely used for such systems. They can be introduced in the development process as reusable assets, in conjunction with dedicated transformation procedures facilitating reuse of developer know-how.

Definition and development of a set of MDA components for development, validation, exploitation and maintenance of DRES lies at the core of new large-scale joint research program (the CARROLL program) involving CEA, INRIA and Thales (see www.carroll-research.org).

8 Conclusions

MDA is about an important paradigm change, from objects to models, from hand-coding to model generation, from model contemplation to model transformation, etc. Like Monsieur Jourdain, many people are today discovering or claiming that they

[7] Distributed Real-time Embedded Systems

have been applying MDA principles for decades, and in some cases this may be true. However, with a lot of hype accompanying the model engineering movement, it seems important to clearly state the essence of this new way of designing information systems and conducting software engineering projects. We have expressed the view, in this paper, that the main advantage of MDA is its proposed regular organization for various kinds of information assets related to software systems under construction, under operation or under maintenance.

Considering models as first class entities is a bigger change than it may first appear. It obliges us to make explicit many working habits with very often the consequence of improving the associated process. The idea that any model is associated to one specific meta-model is of paramount importance. It allows us to state that each model represents exactly one aspect of a system and to build on solid grounds when discussing aspect separation, aspect weaving and other operations.

One particular advantage of proceeding in this way is that we are allowed to discuss classification schemas for models and meta-models. We have presented in this paper two such schemas, the former considering product, process and transformation models and the latter dealing with PIMs, PDMs and PSMs. Both classifications are central to the MDA, yet they are not completely defined. Work is still in progress in defining what a transformation model is and making explicit the notion of platform independence and the operation of platform binding. When these ongoing efforts will converge, we will see more clearly the important impact of the MDA.

Two important operations in model engineering are elicitation and transformation. We use the word elicitation to mean capturing the details of a given external system into a precise MDA model. Once a model is built (in the MDA space) it may be operated upon by automatic and semi-automatic tools, for example undergoing successive transformations.

Transformation operations are common to the MDA and to generative techniques. There is a lot of insight and know-how that has been progressively set up in the last fifty years on automatic transformations. Since a program is a model (with its meta-model corresponding to the grammar), program transformations may be seen as a special case of model transformations. There are however many other sources of inspiration when dealing with transformation in the MDA sense (e.g. graph transformation). The DBMS community may provide a lot of algorithms for schema transformation that could be easily adapted to the MDA context. The idea of technological spaces leads us to look constructively at the collaboration between various solutions instead of focusing on their competitive aspects. This allows mixed technological solutions to be more easily assembled and deployed.

We have discussed in this paper a lot of concepts related to the MDA like the notion of signature of a tool, leading to the possible dynamic discovery of the available services in the scope of model engineering. In some cases we are rather satisfied by the provided definitions. In other cases the notions are not yet satisfactorily defined and much more work is needed. Further investigations on high-level identification of abstract APIs and APIs extensions (à la Eclipse) have been undertaken. This approach seems to fit nicely with the notion of tool signature presented in this paper. We hope this initial investigation is helpful in identifying the basic notion of an MDA artifact and may help defining a general MDA platform.

Acknowledgements

Many colleagues have contributed ideas that are presented in this paper and specially the members of the ATL team in Nantes Frédéric Jouaud, Grégoire Dupé, Jamal Rougui and Patrick Valduriez.

We want also thanks the MIRROR team of Thales for their valuable feedbacks and comments on this paper.

References

1. Badros G.J. JavaML: A Markup Language for Java Source Code, 9th International World Wide Web Conference, The Web Next Generation, Amsterdam, May 15-19, 2000.
2. Bézivin J., Breton E., Dupé G., & Valduriez P. The ATL Transformation-Based Model Management Framework, Submitted for publication.
3. Bézivin J., Farcet N., Jézéquel J.M., Langlois B., & Pollet D. Reflective Model Driven Engineering, UML 2003, San Francisco, (November 2003).
4. Bézivin J. From Object Composition to Model Transformation with the MDA TOOLS'USA 2001, Santa Barbara, August 2001, Volume IEEE publications TOOLS'39.
5. Bézivin J., Jézéquel J.-M. & Rumpe B. Basic Principles for Model Engineering, Submitted for publication.
6. D'souza D. Model-Driven Architecture and Integration: Opportunities and Challenges Version 1.1. February 2001, Available from www.kinetiuym.com
7. Gérard S., Terrier F. et al., Using the Model Paradigm for Real-Time Systems Develoment: ACCORD/UML, OOIS'02-MDSD, Montpellier, Springer LNCS 2426.
8. Kurtev Y., Bézivin J. & Aksit M. Technological Spaces: An Initial Appraisal, CoopIS, DOA'2002 Federated Conferences, Industrial track, Irvine, 2002
9. Lemesle R. Transformation Rules Based on Meta-Modeling, EDOC,'98, La Jolla, California, 3-5 November 1998, pp.113-122.
10. OMG/MDA Guide. MGA Guide (Draft Version 0.2). OMG Document ab/03-01-03, January 2003. Available from www.omg.org.
11. OMG/MOF. Meta Object Facility (MOF), v1.4. OMG Document formal/02-04-03, April 2002. Available from www.omg.org.
12. OMG/RFP/QVT. MOF 2.0 Query/Views/Transformations RFP, OMG Document ad/2002-04-10. Available from www.omg.org.
13. OMG/XMI. XML Metadata Interchange (XMI), v2.0. OMG Document formal/03-05-02, May 2003. Available from www.omg.org.
14. Soley, R. and the OMG staff. Model-Driven Architecture, November 2000. OMG document. Available from www.omg.org.
15. Miller J. et al. What UML should be, Communication of the ACM V.45 N.11, 2002.
16. OMG/Diagram Interchange. UML 2.0 Diagram Interchange RFP. OMG Document ad/2001-02-39, October 2001. Available from www.omg.org.

17. Roques P. & Vallée F. UML en action : De l'analyse des besoins à la conception en Java, Eyrolles, 2000.
18. 1. S. Gérard, F. Terrier, and Y. Tanguy. "Using the Model Paradigm for Real-Time Systems Development: ACCORD/UML", in OOIS'02-MDSD 2002, Montpellier, Springer, LNCS 2426.
19. 3. C. Mraidha, S. Gérard, F. Terrier, and J. Benzakki. "A Two-Aspect Approach for a Clearer Behavior Model", in The 6th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'2003) 2003, Hakodate, Hokkaido, Japan, IEEE.
20. 4. P. Tessier, S. Gérard, C. Mraidha, F. Terrier, and J.-M. Geib. "A Component-Based Methodology for Embedded System Prototyping", in 14th IEEE International Workshop on Rapid System Prototyping (RSP'03) 2003, San Diego, USA, IEEE.
21. 5. T.H. Phan, S. Gerard, F. Terrier, and D. Lugato. "Scheduling Validation for UML-modeled real-time systems", in WiP of ERCT 2003, Portugal, Porto.

Safety Challenges for Model Driven Development

N. Audsley, P. M. Conmy, S. K. Crook-Dawkins & R. Hawkins,

Department of Computer Science,
University of York,
Heslington,
YORK YO10 5DD, UK
{neil, philippa, steve, rhawkins}@cs.york.ac.uk

Abstract. New techniques and approaches are emerging for systems development, such as the OMG's "model driven architectures"(MDA). These approaches represent a very distinct shift from traditional approaches, (such as the ubiquitous "V" model). They offer ways of reducing uncertainty by breaking down the process into different areas of concern or "domains", each domain representing a different perspective of the system. This "separation of concerns" is valuable in that it helps develop a mature body of theory for each perspective across several projects and therefore would facilitate improvement. Yet, this fragmentation of the process challenges our ability to deploy conventional safety analysis techniques that are often structured around the systems development model and traceability of system requirements and behaviour. This paper discusses these issues and suggests some approaches to resolving the difficulties.

1. Introduction

Modern systems development has reached a scale and complexity that makes it unrealistic to assume that one company or organisation controls or owns all the technology or capabilities required to deliver the system. The size of systems under development and the breadth of technologies involved mean that corporate or technological barriers inherently fragment processes. To retain cohesion and coherence throughout systems development, common systems modelling approaches need to be used across different domains of knowledge that span several different organisations and certification bodies.

Characteristics pervasive to all domains such as safety and timing analysis must be resolved consistently when constructing safety arguments. Whilst MDA would support improved understanding of domains in general cases, safety theory depends on context and traceability across the entire development process to deploy effective risk management strategies.

This paper explains some specific concerns over safety and model driven development and describes approaches to address them. We believe model driven approaches have a great deal to offer in the development of safety related systems, particularly the opportunity to focus on risk management at a system model level, yet some issues still require further discussion.

The next section considers the general implications of new approaches; section 3 briefly describes the MDA; section 4 assesses the implications of MDA from a safety perspective and draws out a key set of requirements to be addressed; section 5 explores argument construction and suggests some possible solutions; a short example is given in section 6; section 7 contains our conclusions.

2. Motivation for Reform

There are several aspects of emerging development methodologies that challenge the traditional development life cycle models. These are the elements that motivate a move towards MDA approaches. We review each aspect below from the perspective of safety assessment.

Disengaging design analysis from technical volatility

A distinction is made about information available to describe systems during a development process:

1. *Platform Independent Models (PIM)*: contain information that "provides formal specifications of the structure and function of the system that abstracts away technical details"[2].
2. *Platform Specific Models (PSM)*: contains information specific to individual components, such as computing elements/processing units or operating systems. Information within these models may be structured around performance criteria.

Complete safety arguments will draw on both types of information. The notion of platform or system independence will inevitably be challenged when attempting to build these arguments. This is because safety arguments depend on both design principles and supporting evidence showing these principles have been implemented in practice. For example, safety claims about the design may be refined down to a point where they can be discharged by direct reference to evidence – see fig 1.

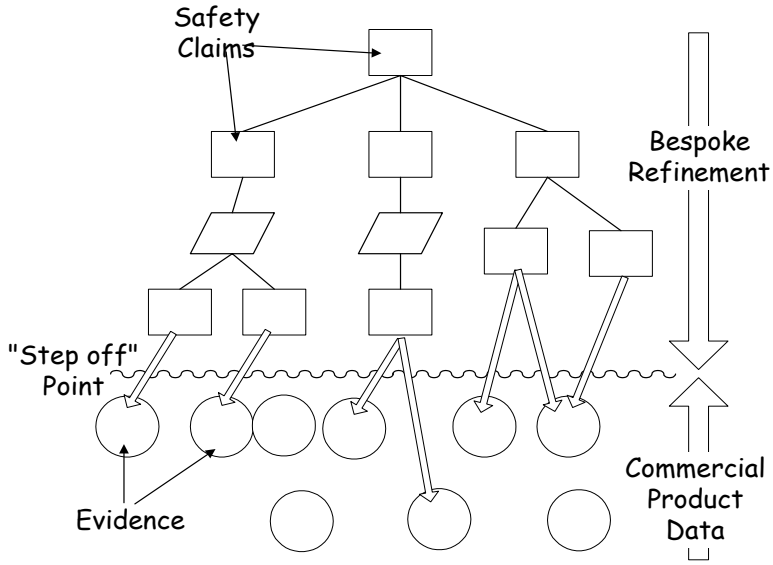


Fig. 1. Separation of System level safety claims from product level data

If the platform independent safety claims can be insulated from changes in the low level technical data then the PIM/PSM distinction could be upheld in the safety argument as well as the design. This is important as the design and the safety argument should be a reflection of one another.

For example, if the design changes, the safety claims may also change, new supporting data may need to be gathered. The claims and the evidence that discharge them need to be considered as separate elements.

Use of "Commercial off the Shelf" products and Supply Chains

Many technologies are developed by smaller, more responsive firms or firms with specific specialities. This establishes a requirement for common and compatible product architectures, where specific concerns are separated into the domain of knowledge or organisation best able to address it. This is a recurring problem for safety critical systems development, one that has often prompted the use of specific contracts with technology vendors, or to take a more dismal view, restrictive disclaimers within end user licences for many competitive products.

Safety in a development Context

Safety requirements and critical safety behaviour should be validated during systems analysis rather than implementation. There is a need to recognise risks during development and communicate "derived safety requirements" across functional, corporate and technological boundaries throughout the development process. To achieve this, each stakeholder in the development should define their understanding of system behaviour within their local domain, allowing inconsistencies to be identified at an earlier stage. It would be important to highlight inconsistencies prior to descending into the complex detail of specific implementations.

Diseconomies of Scale for Software

The amount of software on the current and next generation platforms is growing at a rate that outstrips our ability to deliver this software let alone test and prove properties about it. As platforms become more integrated (such as the use of "integrated modular avionics") a greater proportion of this software is likely to have safety or dependability implications. Yet programming languages focus on the detail of implementation, forcing the need for a more structured approach to system development prior to committing to a software implementation and the inefficiency of testing.

Summary

All these aspects are fundamental pressures on the development of high integrity software brought about predominantly by the need for more effective and efficient systems development approaches. The proliferation of new technologies in very specific problem domains increasingly emphasises the importance of defining an integrating infrastructure. This infrastructure would sit above the level of specific technologies and help to define and deploy coherent and consistent risk management strategies.

3. Model Driven Architecture MDA

Model Driven Architecture (MDA) [1, 2] is an approach that may provide the level of integration required to develop the infrastructure required to integrate information from specific domains to support a coherent risk management process. MDA achieves this by

placing the model at the centre of the process and defining mappings to refine the model based on specific technologies or disciplines. The abstract models (or "platform Independent models" PIMs) that result can then be mapped to specific implementations (or "platform specific models" PSMs) by defining a mapping from the abstract system description to concrete technologies. Fig. 2. shows the basic approach.

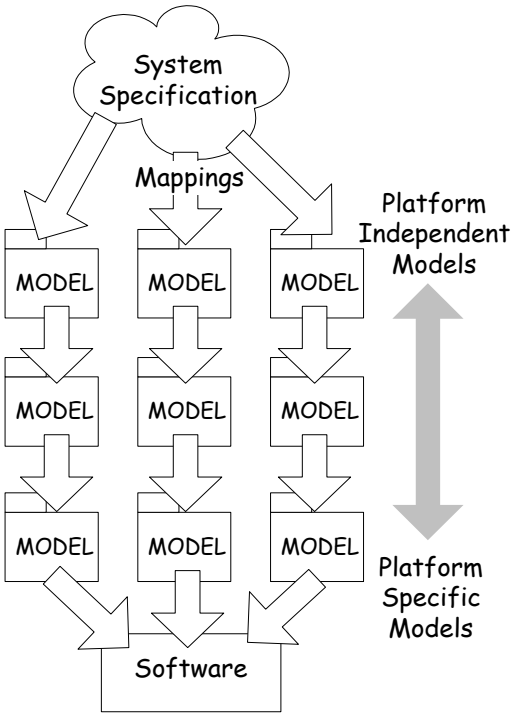


Fig. 2. Basic Illustration of a Model Driven Process

MDA is based on distinct domains. A domain represents a specific perspective on the development – such as a specific body of theory related to the application or a technology. Such domains would need to contain at least two elements:

1. A set of rules for determining if a model was well formed with respect to current theory and knowledge in that area
2. A number of mappings capable of translating a model to other domains whilst preserving the "well-formed" requirement above.

Technological advances and development would be accommodated within the relevant domain area and mappings updated to exploit these changes if necessary. Our argument is that this reduces the scope associated with a change and hence its complexity. Systematic co-ordination of domains may provide a unifying framework for integration of constantly changing technologies, including those developed by external organisations as COTS products. System level arguments and domain co-ordination through mappings therefore share some common objectives.

As each domain is independent of any single development, information can be developed and re-used directly by instantiating a domain to provide the appropriate system model. These domains can be analyzed (and verified) to a degree of precision that could not be supported by individual development projects and this increases the maturity of the

theory and application in the domain area. The literature on MDA refers to this concept as "separation of concerns".

The systems are built by establishing mappings between these well-formed domains. For example - a domain may exist that holds common design principles for nuclear power plant protection systems. This could be translated into a system description in the UML by mapping from the power plant domain to the UML domain using a suitable UML profile, and then translating this design into code by mapping from the UML domain to the Ada language domain. This stepwise refinement is the heart of the MDA principle, allowing greater interoperability. The domain's themselves would be subject to correctness and dependability criteria so that they represent stable, rigourous approaches in their own right. To implement the Nuclear plant protection in (say) C rather than Ada, we could substitute a C domain in place of the Ada domain without necessarily needing to redo safety analysis at the system level, as both the Ada and C domains have previously been shown to represent a correct implementation view through the use of well-formed and verified domains. An example of these techniques is the development of the F-16 modular mission control computer [3].

Our key question is whether this "separation of concerns" leads to fragmentation of the processes that underpin the safety argument process, this is dealt with next.

4. Safety Challenges

Some system characteristics impact across all domains. Safety is a key example because proving specific system behaviour depends on a traceable and predictable method of system refinement from top-level requirements through to implementation. Whilst safety requirements should be established as part of the system model, the implications on these requirements of translating this model between separate domains has not been explored. The initial concerns are:

1. Inherent nature of Safety Requirements: Safety requirements relate to system properties within an intended deployment environment. Whilst "best practice" for safety could be abstracted into a separate domain, the deployment of this practice will have implications for every mapping undertaken to translate the model. Therefore safety theory cannot be "factored out" as if it were simply another technology.
2. Identifying Derived Safety Requirements: Safety requirements exist at every level of abstraction and many only emerge during the detailed system analysis and design phase (hence "derived"). It may not be possible to state the safety requirements completely at the platform independent level. A mechanism will be required to identify derived safety requirements as they occur and update the model as necessary, highlighting any inconsistencies or concerns this raises. The stakeholders in this review process would need to have knowledge and visibility of the nature and sequence of mappings being applied to system models. Little support appears to exist in the MDA model for this type of review process across domains.
3. Traceability: Certification arguments depend on traceability of top-level requirements down to evidence from implemented systems. Whilst MDA is a unifying framework, it doesn't, at present support traceability to a greater extent than recording the sequence of mappings used to refine the model from one domain to the next. There are a potentially infinite number of ways a system model could be refined from PIM to PSM, significantly complicating safety certification arguments without introducing the idea of a controlled process.

4. Constraining Technologies: A major rationale behind MDA is inclusion of new technologies, tools and other "middleware". One of the key issues with safety critical development has been to establish a set of trusted tools, which permit greater precision and rigour in the reasoning about system behaviour. For example, the use of Ada as a programming language as it is more amenable to static analysis than, (for example) C. If we embrace the idea of inclusive new technologies, what frameworks might be necessary to validate the domains and mappings that define these technologies prior to their use?
5. Reinforcing Technical "silos": The results of safety assessment aren't absolute, but often involve a degree of judgement or reasoning by a suitably independent and experienced individual. If key technologies are located within domains – each with their own set of working principles, tools and practices; does this make a safety reviewer's job more or less difficult? Our own experience of projects carried out across divisions with different technical perspectives is a tendency for conflicting interpretations of risk and hazards to emerge which compromise the system level safety case. This is largely down to cultural issues and less clear perception of the overall engineering process.
6. Certification: Extant safety standards (such as DO178B and Defence Standard 00-55) [4],[5] are based around a conventional systems development process with traceability, constraints, hazard logs etc supporting information exchange and recording to support arguments and reason about evidence collected in context. It is unlikely that the same can be achieved though information exchange between autonomous domains without further definition of a systematic and consistent process for reasoning about safety requirements.

If MDA approaches are to be adopted, it will be necessary to consider the implications of the safety lifecycle as part of the process. It will not be feasible to restrict safety assessment to a specific domain. Support for a defined, traceable process will be required to develop the safety argument as the design progresses.

In summary, model driven approaches based on refinement of models across a number of different domains will need to resolve the following "Process Safety Issues":

1. Establish a systematic and coherent approach to risk management common to all domains involved in system development.
2. Define a process for recognising derived safety requirements and resolving them as part of an overall strategy of system safety.
3. Establish a method of tracing safety requirements across domains and resolving any misunderstanding or discontinuities as they emerge.
4. Strategic leadership at a system level with authority to resolve understanding of risk across distinct domains.

This is a minimum set of requirements required to address the challenges posed by the fragmentation of the development process into distinct domains. Before going on to consider how these might be addressed, it is worthwhile to consider why existing traditional (i.e "non MDA") processes will become less effective.

Defining a new process

Existing approaches to risk management [4,5,7,8,9] tend to focus on hazard directed arguments whereby risks are quantified early in the process, using a risk/probability

matrix. Subsequently, a battery of techniques (fault trees, markov models etc.) is used to refine these down to requirements on individual subsystems, and ultimately down to components. This approach requires strict control over all aspects of the process and tends to motivate the delegation of safety requirements, where tacit knowledge and understanding is only made explicit on an ad-hoc basis as these techniques are deployed. For model driven approaches, this knowledge is already explicit in the various domain, yet as there is no pre-defined route from top-level requirements to implementation, a focus on accountability is less tenable. There is a need for a more structured approach that uses the construction of safety arguments based on system models to resolve risk directly, by appealing to argument strategies that connect platform performance to system requirements and exploit knowledge with each domain.

5. Constructing arguments

The platform independent model (including safety claims) must be refined to a level of detail whereby it can be mapped to a set of components as an implementation. In the same way, the safety claims must have been refined down to a point where they can be addressed directly by reference to evidence. Neither safety claims nor the design principles are changed when mapping vertically down to an implementation.

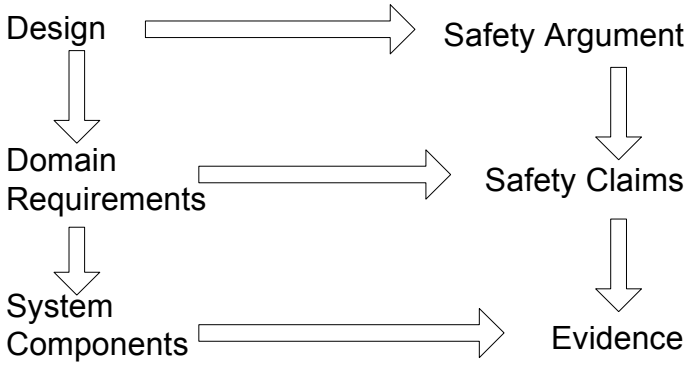


Fig. 3. Connection between design and Safety during system realisation

Platform independent knowledge and platform specific knowledge remain separate so that

1. System knowledge (and the standard components associated with it) can be validated as implicitly correct and can be re-used elsewhere; and
2. The system can be mapped to a new implementation without requiring the system to be re-designed.

Distinct domains would also need to remain independent and distinct to uphold the integrity of the domain as a specific viewpoint. The potential complexities and management challenges to constructing safety arguments with these requirements in mind are illustrated in fig 4.

The safety argument has a tree structure as top-level safety claims (such as “system is safe”) are decomposed systematically into lower level claims (such as “software is written in Ada”). Fig 4 shows how platform independent (claims with clear boxes) and platform specific information (shaded) might be distributed across the safety argument. In addition, some claims have been associated with a specific domain by labelling them. (A,B,C, … X,Y,Z).

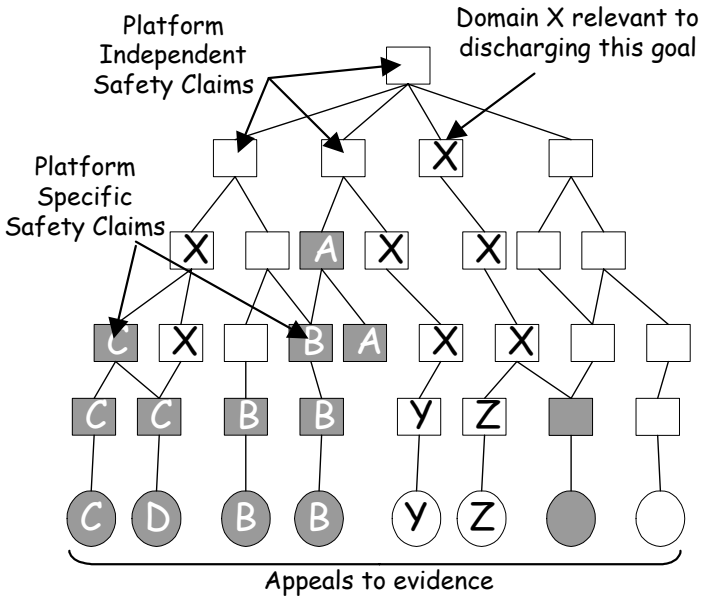


Fig. 4. PIM and PSM elements in a safety argument structure

Given that all the claims must be discharged, how might the suppliers of expertise for each domain come together to deliver a safety case?

Some suppliers of subsystems and/or components might be able to demonstrate compliance to specific claims directly – perhaps taking over whole branches of the safety argument structure to discharge platform specific requirements. For example, if domain C is the sole responsibility of a specific supplier, then it might be possible to outsource a whole branch of the safety argument relating to domain C. Perhaps setting up a supply chain if Domain D is associated with a second tier supplier.

The situation may not always be so straightforward. Consider the implications if domains A or X are the responsibility of external organisations. Domain X is involved with several areas of the safety argument and relates to several other domains that may or may not be within this supplier's abilities to address. Therefore there will be complex information exchange between domains to deliver a safety argument.

Possible Approaches

Approach #1: Independent safety constraints on each template, domain, and mapping set.

The obvious response to the potential threats of a fragmented process is to assess the implications for each and every part of the translation. This includes every model that represents the system within a domain and every mapping that translates models between domains. The approach is illustrated in fig 5.

Whilst this could be considered a "defence in depth" approach with much potential for crosschecking, there are several potential weaknesses in this approach:

1. It would involve re-testing many principles in a local or specific context, providing potential for inconsistencies.

2. It is unlikely to be cost effective.
3. It would require a separate "safety perspective" on every domain, which would be difficult to both define in abstract terms and to manage systematically
4. Consistency checks would be complex – especially if domains are under control of different individuals and evidence is broken into piecemeal parts.
5. Little motivation for common approaches. Each domain is motivated to reason about risk assessment within its only domain in the general case, not the system model specifically. Accountability for specific risks and hazards can be lost or confused, resulting in little increase in capability.

The underlying problem with this approach is complexity and the potential for inconsistencies across the process. Safety analysis primarily relates to system properties, not the simple conjunction of component properties.

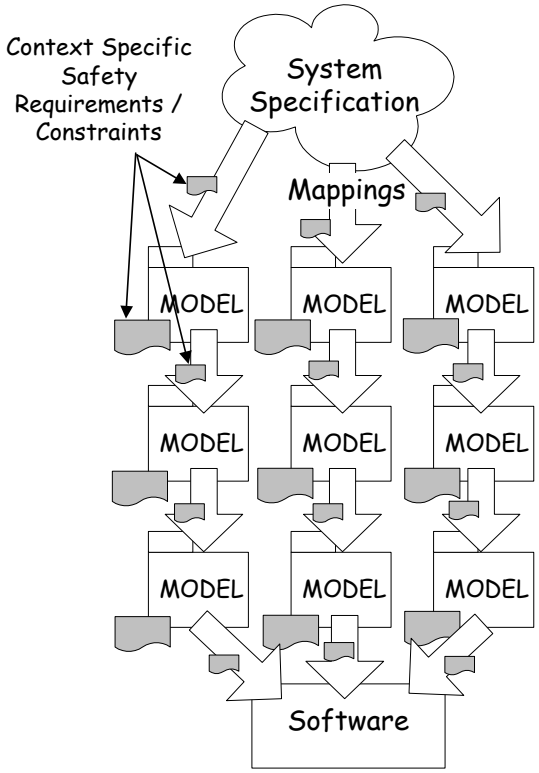


Fig. 5. Approach #1 - Individual safety constraints

Approach #2: Refinement Levels

An alternative approach would emphasise system level properties in the safety case. A common framework can be established by defining a number of refinement levels. These levels connect the different models that make up the system at a common level of abstraction. System level claims can then be validated and discharged across these levels. The claims discharged at each level would come from a deductive process of refining top-level safety requirements. At the beginning of the process, the top-level safety claims would be identified, and then decomposed into lower level claims appropriate for each level of refinement at which the system will be validated as a whole. In this way, the safety argument is refined and traceable back to original requirements and can be shown as consistent with the generated software, see fig 6.

This process would highlight inconsistencies more effectively than approach #1 as the system model is made explicit in the form of safety goals. Since this model is independent of any single domain, then the domain models can be altered without prompting change to the safety argument process in general.

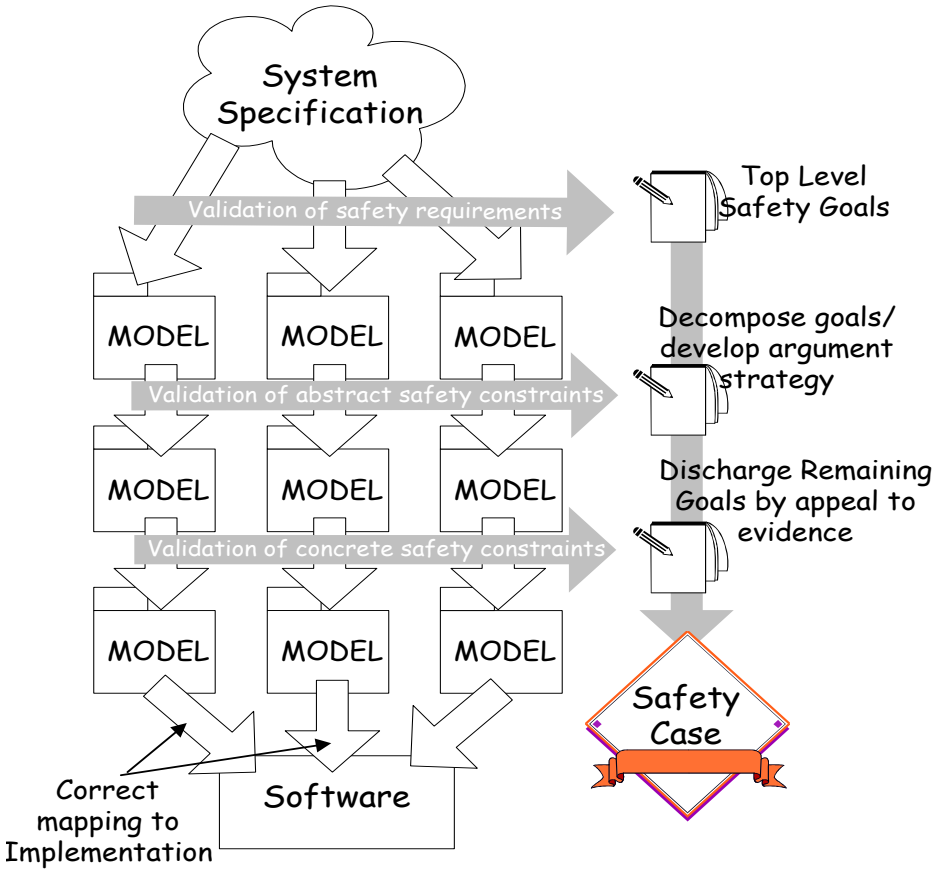


Fig. 6. Approach #2 Imposition of process stages

The weaknesses of this approach are:

1. It depends on domains (and the models derived from them) being defined at common levels of abstraction.
2. Safety assessment inevitably follows a step behind the design process, potentially compromising the capability of the process to drive design decisions based on derived safety requirements.
3. Safety requirements would become complex due to the need to view the system as a set of distinct viewpoints rather than as an integrated whole.
4. Possibility of repeated verification effort associated with the need to perform common tests on model structure across the distinct domains.
5. Due to the "coarse" level of integration between the development and safety processes, derived safety requirements may require the audit trail to begin again at the start, unravelling much of the development effort.
6. System analysis and system safety perspectives are still considered separate, making it difficult to preserve "safety " as a system level concept and focus in on problem areas to which the safety case is sensitive.

7. Reinforces the idea that safety is a separate discipline.

This approach doesn't truly address the fragmentation in the process as the development and safety process have been lashed together based on abstraction levels in the product, rather than more rigourous and sound safety principles. Addressing this requires some way to reason about safety practice within each domain. High integrity development involves the integration of system analysis and safety assessment. This can only be achieved if we provide a systematic process of resolving system-level safety issues within the specific context of each domain

Approach #3: Safety argument patterns

The approaches above are ineffective because they treat safety as "after the fact". The motivation for doing so was to retain the idea of a single, traceable process for refining safety requirements. Yet this is inconsistent with the philosophy of separate domains within MDA. An approach is required that combines the need for consistent traceable arguments with the need to recognise separate domains, each of which contributes to emergent safety behaviour. Kelly [13] argues it is inevitable that safety arguments will need to be broken down into common accepted principles. If improvement is to be made, the ability to record and reuse standard safety approaches is essential. As each development will be different, the approach would require the ability to communicate these safety principles as templates or patterns and show how they are instantiated systematically to build up safety arguments.

1. Define a critical set of key behavioural variables (response times, fault containment, Levels of redundancy) at the system level.
2. For each domain, specify a "safety contract" derived from the top level criteria or identified derived safety requirements and establish the conditions (pre and post conditions) for this to be upheld – including constraints on functional and non-functional behaviour
3. Refine the model by following mappings from the problem domain to the other domains – working towards implementation
4. Pull in safety argument "patterns" as relevant to reason about the model under the local domain and discharge the relevant conditions (see fig 7). An example of a pattern is shown in fig 9.
5. Feedback any derived safety requirements as they emerge for inclusion in the process.

Failure to meet a condition would stop the analysis and require resolution (perhaps an additional safety requirement is derived in response)

The safety argument patterns would provide a definitive resource for describing risk management strategies across domains. Providing a more systematic and rigourous process of reasoning about safety within the domains without this being dependent of the specific product under development. The preconditions provide a traceability mechanism. The impact of derived requirements is also better resolved as they would result in the inability to support a precondition or postcondition and hence be easily identified. The conditions are directly supported by the patterns as refinement and traceability are a result of systematic, repeated use of the argument patterns.

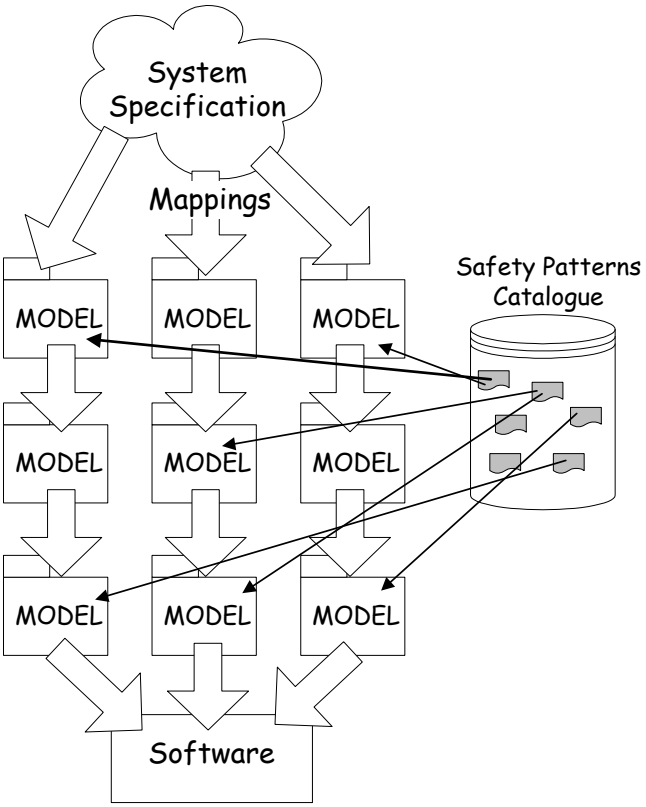


Fig. 7. Defining templates suitable for instantiation across domains

The idea being that a section of the safety argument can be constructed systematically by being explicit about context and dependencies. Fig 8 gives a very basic idea of how the patterns may help to construct sections of the safety argument. The crosshatched areas show where a pattern might be used to resolve claims involving several domains and their contribution to the overall safety argument.

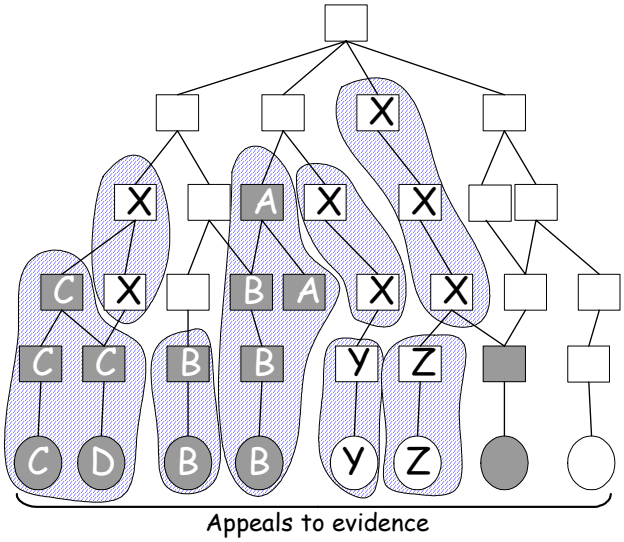


Fig. 8. Breaking up the safety argument on a more manageable basis

The concept is that the patterns fit together to make up the safety argument in its entirety. The conditions of one pattern could be derived from the output of the previously

applied pattern. In this way, the argument is built up systematically across the domains without compromising the overall process. It also defines explicit interfaces based on proven argument strategies, reducing the potential for inconsistencies when functional or corporate barriers are present.

To illustrate the promise of this approach it is now assessed against the "process safety issues for MDA" identified at the end of section 4, and an illustrative example is developed in section 5.

Establish a systematic and coherent approach to risk management common to all domains involved in system development.

The approach is systematic because a set of well formed general patterns are deployed to address specific safety conditions derived from safety requirements and system development. Coherence is supported as the patterns are a common resource and provide a common strategic direction across different domains, functional areas and corporations, whilst permitting each domain the degree of autonomy required innovating or exploit new approaches.

Define a process for recognising derived safety requirements and resolving them as part of an overall strategy of system safety.

The safety requirements are made explicit by the conditions imposed on each domain. These reflect the current state of the safety argument with respect to the system at any point. Derived safety arguments may emerge during the instantiation of argument patterns themselves, perhaps making explicit the tacit knowledge deployed during design work or conventional safety assessment techniques. The intent is that through the use of a systematic process of refining safety requirements in the context of the specific domains and bridges between them. Within the more focussed environment of a specific domain of expertise, there is a better chance of identifying and resolving derived safety requirements than if they were simply raised as project wide concerns.

Establish a method of tracing safety requirements across domains and resolving any misunderstanding or discontinuities as they emerge.

The patterns provide a common method for refining safety requirements into a set of specific conditions for each domain to support. Conditions and patterns can be used to formulate mappings for safety requirements across domains to uphold the integrity of the safety argument. Discontinuities will show up as inconsistencies in the argument, conditions not met, or misunderstanding of pattern use. The process is more manageable because it is more clearly defined.

Strategic leadership at a system level with authority to resolve understanding of risk across distinct domains.

The use of patterns and refinement of safety conditions opens up a common dialogue for discussion of safety requirements. The discussion can focus on a set of safety conditions, argument approaches and techniques to help address risk management systematically. This leads to more mature understanding of how risk management is affected throughout the process and moves away from the chaotic processes we have observed, where (for example) each functional area is tasked to deliver a separate hazard analysis.

This approach not only fits into the MDA approach of deriving models by instantiating domains (in this case the safety patterns catalogue is instantiated to deliver argument

development). It also makes clear the strategy for risk management off-line from any one single development.

6. Small Worked Example

For illustration purposes, a small example follows based on an imaginary terrain avoidance system for a fast military aircraft. The aircraft must be of high performance, but also be stealthy. Terrain avoidance is part of the Navigation system.

Top Level Safety requirement
The system safety requirement is that the aircraft must not penetrate the minimum safe separation (MSS) from terrain. This mitigates against risk due to the hazard "controlled descent into terrain" which, in the terms of Defence Standard 00-56, would represent a critical event, and therefore earn a Safety Integrity Level (SIL) of 4 (the highest such level).

Domain Requirements
Mapping these requirements down into the Navigation, Stealth and Flight performance domains would result in a number of derived requirements to be resolved to achieve a consistent mapping down to implementation for all three domains identified:

1. Navigation domain functional requirement: Maximum avoidance manoeuvre should not exceed 4g.
2. Stealth domain functional requirement: Use of radar avoided or reduced to a minimum. Apertures in the airframe itself to be restricted to 1.
3. Non-Functional requirements inherited from the flight performance domains: Additional weight to be less than 300kg, apertures in the airframe minimised

Refinement of Safety Requirements
The top level safety requirement, which has implications across (at least) the three distinct domains, requires consistent resolution to ensure all the top level requirements are satisfied prior to mapping the system down onto a specific implementation. The integrating element will be a safety argument pattern.

With the derived requirements above, the navigation domain requires a SIL 4 sensor system to detect land profiles that penetrate the MSS and trigger an adjustment to the flight path. The requirement (from the stealth domain) for a single aperture in the airframe would limit the navigation group to a single sensor. Since no single sensor will discharge the SIL 4 requirement, diverse sources of evidence will be required to validate the single sensor. One proposal might be to use a digital map to store terrain data and validate this against real time data from the sensor. The idea being that the broad data in the map can be used to validate that the real time sensor is operating within tolerance levels. If these tolerance levels are breached, a recovery manoeuvre is triggered. Since the two data sources are independent, neither sensor is required to support SIL4. Under Def Stan 00-56 the requirement can drop to SIL 3, allowing each sensor to fail "occasionally" without compromising system safety requirements.

This informal reasoning must be resolved more rigorously. A safety argument pattern for "diverse argument" (as defined by Kelly, [13]) can be instantiated to resolve the safety requirement whilst upholding the derived requirements between the domains. The pattern

is shown in fig 9, and is defined using an extension of "Goal Structuring Notation" (GSN) designed specifically to represent patterns, as it makes the context and basis of claims explicit. Goals are represented by rectangles and are decomposed into a number of subgoals (also rectangles) that represent a set of more specific claims that together discharge the higher-level goal. The basis of the decomposition has been recorded using a strategy (S1) shown as a parallelogram. Contextual information is shown within a lozenge shape. Arrows indicate the direction of decomposition. A hollow circle on an arrow represents an optional decomposition, whereas a filled circle represents multiple decompositions. For example, in fig 9, there may be several sources of evidence Gn that support G1 – hence the ">1" label on the decomposition to the left subgoal.

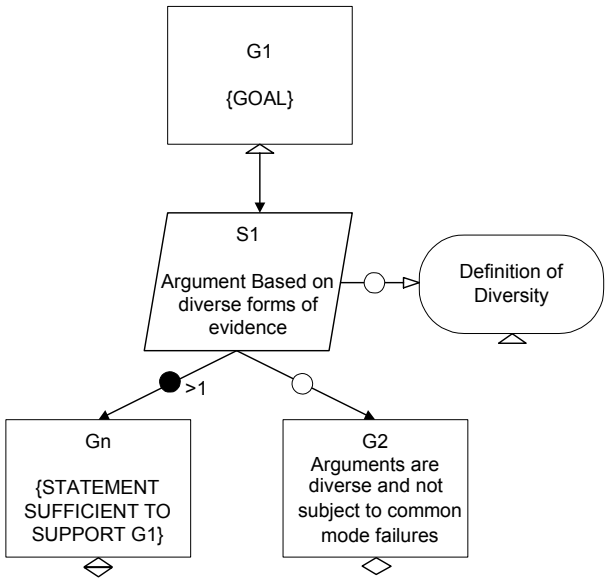


Fig. 9. Safety argument pattern for diverse arguments (from Kelly)

This pattern can then be used to record the reasoning above that discharges the safety argument whilst maintaining a system model consistent across the Navigation, Flight performance and Stealth domains. Fig 10 shows partial pattern instantiation, the diamonds indicate that the claims would need to be developed further in the full argument.

This approach provides several advantages

1. *Focus on Design Analysis*: Product level arguments remain separate from specific technologies. The refinement of the safety argument has been driven by design analysis, not specific technologies.
2. *Support for Technology Transparency*: New technologies could be introduced without compromising the system model or requiring system level re-design
3. *Separation of Concerns*: Those in the navigation domain are able to adopt new approaches without compromising system level models or rationale developed in other domains
4. *Meaningful information exchange*: Platform level reliability requirements can be mapped across domains without prescribing specific technologies or implementations.

These are all aspects of the MDA philosophy that mature and systematic use of safety argument patterns provide during a high integrity development process adopting MDA principles.

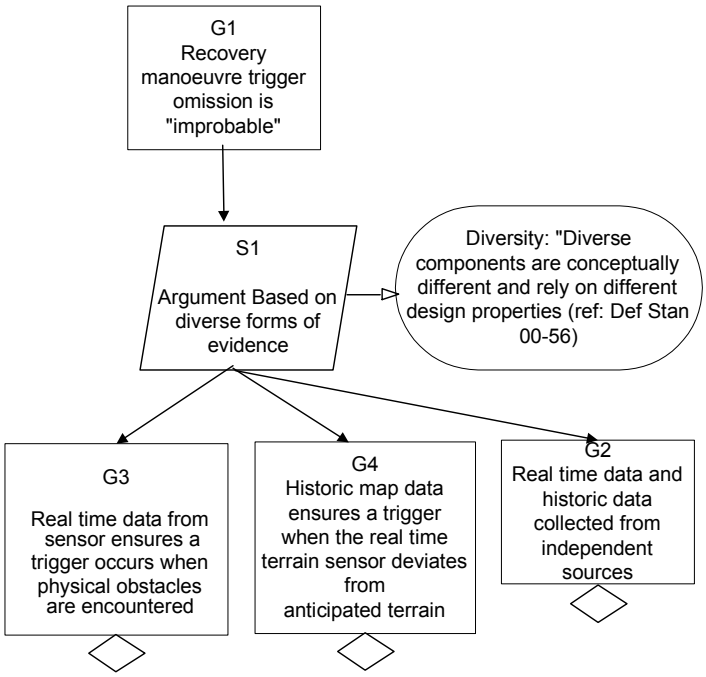


Fig. 10. Partly instantiated pattern

The example shows how the system model can move forward by focused effort within specialised, focused domains. At the same time, a system level argument is developed reconciling derived requirements across domains, see fig 11. Technological advances can be accessed with each specific domain without the complexity of rolling them out across entire projects when the benefits may be restricted to only a few domains. This could be thought of as a supply chain for systems development expertise, each area capable of deploying mature, proven technology into a number of projects.

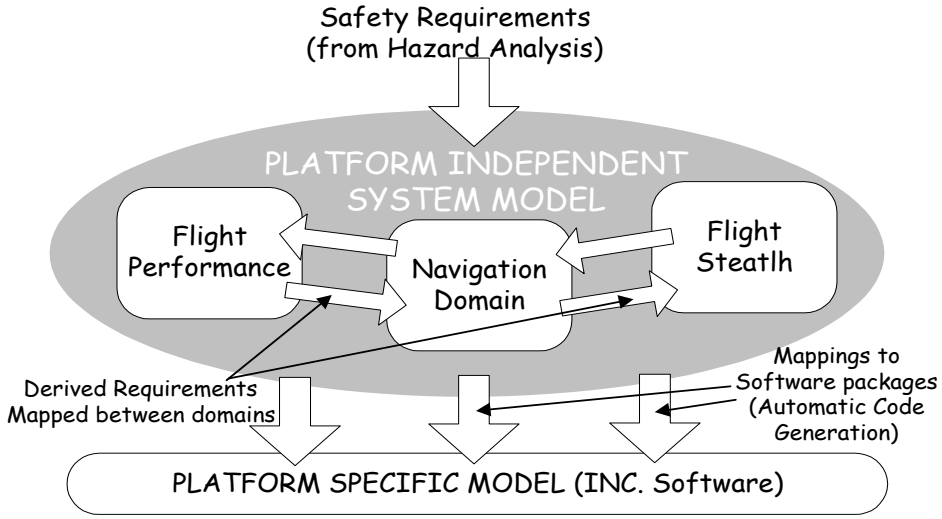


Fig. 11. Overview of suggested process

One difficulty with this approach is identifying and resolving conflicts across domains in a systematic way. For example the safety argument pattern used above (fig 10) doesn't record explicitly the need for a single sensor on the airframe that avoids the use of radar emissions (both requirements from the "stealth" domain). These are implicit in the

refinement approach taken, but aren't relevant to the safety argument directly as it currently stands. They would need to be made explicit in some cases; for example, a new sensor technology in the navigation domain would need to be assessed against these implicit criteria.

Resolving Conflicts

If conflicts are to be resolved in a systematic manner, then some supporting infrastructure or method will be required to record and resolve shared derived requirements explicitly. In [14], Dawkins and Riddle describe a structure for reconciling the general requirements and provisions of a Commercial Off The Shelf (COTS) product to the specific context of a safety critical system in development, see fig 12.

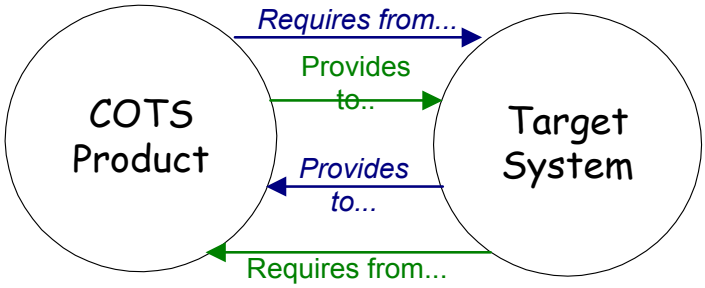


Fig. 12. Reconciling the system-component relationship

This is a bipartite process whereby the needs of the system under development are reconciled to both the functionality offered by the COTS product, and environmental properties required to deliver those functions. The intent would be to adopt a similar policy with respect to reconciling requirements across domains:

Table 1. Simple example of reconciling mappings between domains

Stealth domain:	**Navigation Domain:**
Requires from {Navigation}:	*Provides to* {Stealth}
Few airframe apertures	Few airframe apertures
Low sensor emissions	Low sensor emissions
Provides to {Navigation}	*Requires from* {Stealth}
<null>	<null>

Of itself, this is little more than a matching exercise, the key would be to use this as a device to prompt discussion between the two domains to ensure that the mappings between distinct domains uphold system level requirements and prevent the introduction of discontinuities due to misunderstandings.

Conclusions

Model driven architectures and other modular approaches being put forward for systems development raise fundamental challenges to existing safety processes. Whilst many aspects of system development could potentially be abstracted into separate bodies of

theory and deployed as instantiations of these theories, safety analysis and other system specific attributes cannot be abstracted in this way. The process of building safety arguments depends of a traceable refinement throughout the development process from top-level requirements down to implementation.

New approaches to system development do provide important benefits in the development of safety critical or high integrity systems because they help us to reason more effectively and completely about specific technologies, commercial tools, and middleware and integrate this into a defined, traceable process. These approaches also promote the concept that domains describing these middleware technologies and can be refined and developed to improve the maturity of our understanding of these components and increase our capability to deploy them on high integrity developments without introducing faults.

This paper has defined an approach, based on existing (and mature) work, for a safety process for model driven development. The assessment of three separate approaches has illustrated the problems involved. The safety argument process must be efficient so that it gains support across development teams, and also based on stable principles so that it is coherent and develops maturity of approach. Both of these aspects help to promote an effective risk management process by systematic resolution of safety requirements whilst mitigating the risk of the more fragmented processes.

References

[1] Soley R., "Model Driven Architecture" Object Management Group (OMG) White Paper, Draft 3.2, November 17, 2000 (www.omg.org/mda)

[2] OMG Architecture Board ORMSC, "Model Driven Architecture – a technical perspective", document number ormsc/2001-07-01, July 9, 2001.

[3] Clark L. E., Hogan B.D., Ruthruff T., Kennedy A., "F-16 Modular Mission Computer Application Software", http://www.omg.org/mda/mda_files/New_MDA.ppt

[4] RTCA, Inc. "Software Considerations in Airborne Systems and Equipment Certification (DO-178-B)", RTCA SC-167/EUROCAE WG-12

[5] UK Ministry of Defence, Defence standard 00-55 (PARTS 1 & 2) "Requirements for Safety Related Software in Defence Equipment", 1August 1997.

[6] Kroeger B., "Texas Instruments Future Directions" , Boeing Commercial Aeroplane Group Electronic Component Management Program Users Forum II, March 5, 1997

[7] Society of Automotive Engineers Inc., *Aerospace Recommended Practice (ARP) 4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems*, November 1996, www.sae.org

[8] Society of Automotive Engineers Inc., *Aerospace Recommended Practice (ARP) 4761: Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment*, December 1996, www.sae.org

[9] Smith, D.J. *Reliability Maintainability and Risk: Practical Methods for Engineers*, fifth edition 1997, Butterworth-Heinemann

[10] Pumfrey, D.J., *The Principled Design of Computer System Safety Analyses*, DPhil Thesis, University of York, 1999.

[11] Wirth, N. "Program Development by stepwise refinement", CACM, vol. 14, no. 4, 1971, pp221-227.

[12] Lions, J.L., *ARIANE 5 Flight 501 Failure Report by the Inquiry Board*, Paris, 19 July 1996, http://java.sun.com/people/jag/Ariane5.html

[13] Kelly, T.P. Arguing Safety – A Systematic Approach to Managing Safety Cases, Dphil Thesis, University of York, 1, 1999.
[13] Dell, M, Fredman C, *Direct from Dell*, Harper Collins Business, 2000.
[14] Dawkins, S.K., Riddle, S.,(1999), *Managing and Supporting the use of Commercial Off The Shelf (COTS) components*, Proceedings of the Safety Critical Systems Symposium (SSS), Springer Verlag, February 2000
[15] Fisher M.L. 1991 "What is the right supply chain for your product?", Harvard Business Review, March –April; 1997

Invited talk: UML2 - a language for MDA (putting the U, M and L into UML)?

Alan Moore

Artisan Software Ltd
AlanM@artisansw.com

Abstract. The Unified Modeling Language is probably the most wide spread modeling language the software engineering community has ever seen. The breadth of its use is at least partly attributable to that fact that it has a large number of diagram types, drawn from many sources with, to say the least, ambiguous semantics. This has meant that almost every potential user has found some part of UML that they can use in a way which suits them. UML 2 has followed the same path as UML 1.X in this regard, with more, and more complex diagrams, with both overlapping and imprecisely defined semantics; by induction we can assume that this will further enhance it's breadth of use and popularity.
However, UML has also been touted, with good reason, as a key enabling technology for MDA and in this role the features listed above look like significant disadvantages. This presentation looks at some of the issues with using UML 2 as a language for MDA.

Using an MDA approach to model traceability within a modelling framework

John Dalton[1], Peter W Norman[1], Steve Whittle[2], and T Eshan Rajabally[1]

[1]Engineering Design Centre, University of Newcastle upon Tyne, UK
John.Dalton@ncl.ac.uk,
[2]BAE Systems, Warton, UK

Abstract: The Newcastle Engineering Design Centre (EDC) with BAE Systems have developed a modelling framework representation to support the modelling of complex engineering systems. The purpose of the framework is to allow a range of models (representing aspects of systems engineering) to be integrated so that an overall system can be envisaged. Central to this is the ability to trace properties within the product hierarchy represented by the framework. The purpose of this paper is to highlight the design process involved in the implementation of traceability within the framework. Beginning with some Unified Modelling Language (UML) sketches of the desired tracing capabilities, this has relied very heavily on a Model Driven Architecture (MDA) approach to Executable UML to develop and implement this. This paper begins with a brief outline of the modelling framework, describes the tracing capability required, then finishes with a description of the modelling methods used to implement the tracing methods within the framework.

1 Introduction

One of the great challenges in complex system development is the control of design properties throughout the lifecycle. Modelling plays a key role in this activity but current approaches do not adequately support product integration. In particular there is a failure to provide proper traceability of design properties throughout the product breakdown structure and no adequate management of the impact of uncertainties in modelling activity throughout the lifecycle. The modelling framework research aim is to obtain an understanding of areas in which modelling is used by developing an integrated modelling environment (IME) that can be fully deployed within BAE Systems. Implementation of this approach will allow a more efficient and effective use of modelling capability, permit a consistent record of how systems function, create a traceable record of the design decisions used, and encourage reuse of previous design decisions in new systems.

An important aspect of the design of complex systems is that of traceability. An object oriented approach to an integrated environment can provide the basic platform upon which a tracing methodology can be developed. Such a methodology can be performed on objects (such as properties) within a complex system design framework.

This can benefit the systems design engineer in a number of ways. For example, the mass say, of an engine sub-system (i.e. an emergent property) may be critical to a design. By tracing this property through a design framework, the fundamental low level properties of the system, which contribute can be revealed. Collection of information in this way allows designers to target their efforts at a very early stage in a design. Traceability can also be used to measure and test the sensitivity of properties and estimate the required fidelity of integrated models. Sensitivity traces can also enable the validity of modelling fidelity, i.e. justifying whether low or high fidelity modelling is required to achieve a desired accuracy. As part of the IME work a generic tracing methodology has been developed by the Newcastle EDC. Although applied to properties the methodology can be used to trace a range of entity relationships within a variety of modelling environments.

The research reported in this paper, performed in conjunction with BAE Systems, has produced a working modelling framework within which objects such as properties can be traced. To adequately describe the tracing methodology within the scope of MDA, this paper first describes the integrated modelling environment (Section 2, *Modelling framework*) and the modelling methods used to create this (Section 3, *A description of the framework using UML*). Once established this forms the foundation upon which the tracing method is built. This is first described in Section 4 (*Description of the tracing method*), then the design and implementation of this is presented from a Model Driven Architecture (MDA) point of view in Section 5 (*Modelling the tracing methodology*).

2 Modelling framework

The modelling framework structure is in the form of an abstract container, which comprises of components that represent the models being integrated. The main purpose of the framework container is to:

(i) represent models, their associations, and properties relating to system configuration, and

(ii) hold and exchange information about a system defined by the models it represents.

A simplified view of the modelling framework concept is shown in Figure 1. This shows a number of sub-systems, which collectively represent a product system. Each sub-system contains a number of components. The components are designed to represent an external entity such as a model, or an aggregation of models, where an actual model does not exist but is formed as a collection of components that do represent models. Components contain collections of objects, which deal with the modelling representation. These hold and transfer properties via links with other components and evaluate any data from represented models or other sources external to the framework.

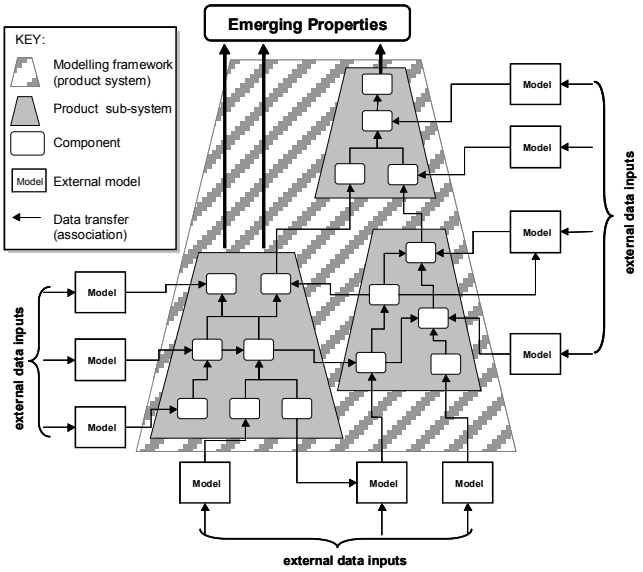


Fig. 1. Conceptual view of a modelling framework. In this case the framework encompasses three sub-systems which contain the components representing system models

Figure 2 highlights the framework view of a relationship between the physical domain and the modelling domain. A system element such as, say a heat exchanger, may exist as either a physical entity or as a proposed design. The important properties of either are extracted and modelled by an engineer in such a way that it exposes the properties of interest. The objects within a component define the type of information that is required. This includes:

(i) functional behaviour demonstrated by a model,
(ii) design definitions, or goals required by the designer, and
(iii) structural information about an external entity being modelled including any additional information that may be required, such as: cost, part numbers, etc.

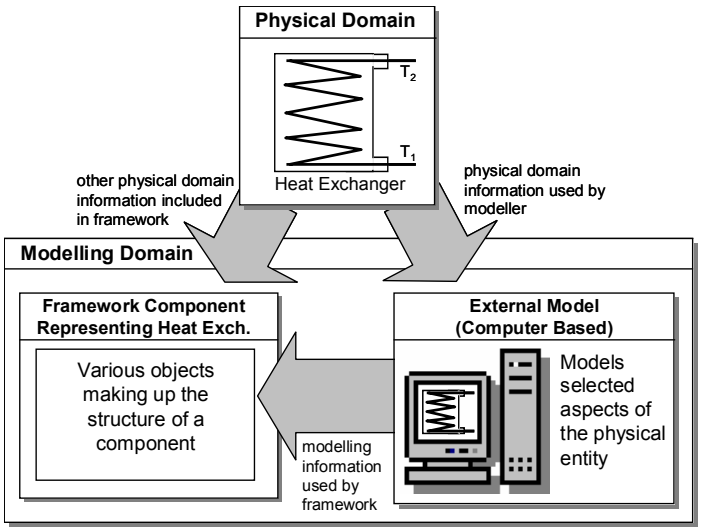


Fig. 2. Relationship between the physical domain and the modelling domain

The relationship between the objects contained by a component requires an understanding of how such entities can be arranged to represent modelling output [11], [10], [2]. Behaviour is defined as: the way that the functionality of a model is achieved. This is functionality seen as a purpose of an entity [4], [6]. Modarres [6] also states that: behaviour is how an object acts or reacts, in terms of its state changes, so as to attain its intended function. The subjective motive of a system or its component parts may be defined by its goals [6] or what it sets out to achieve (or at a more basic level its design definitions). The relationship between functions and goals are described by the following: to attain a goal, one needs a collection of functions to be realised [6]. Goals may also be described as design definitions in that a designer may specify some limit or value that a function must achieve. This may then be used as a measure of success or failure for a function. The relationship between these objects is shown in Figure 3, this highlights that the behaviour of something is how a function or purpose is achieved and that a goal is how a function or collection of functions is measured or tested. Figure 3 shows that extra structural information is available to all other objects within a component.

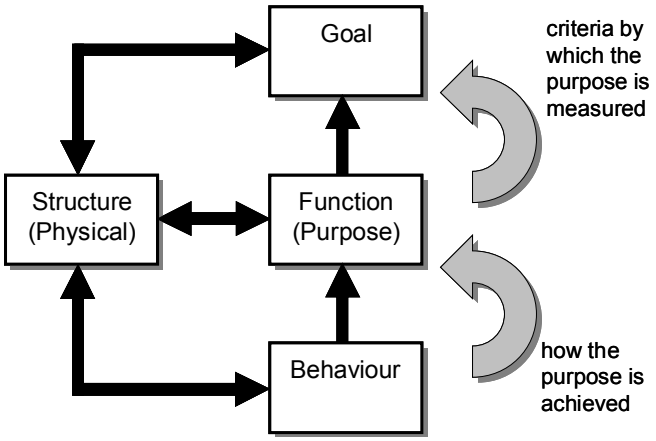


Fig. 3. General relationship between objects contained by a component

The functional, behavioural, goal and structural object, which represent the differing aspects of the modelling domain, all hold collections of common property objects, which in turn contain the physical aspects of the models being represented. These may also hold any descriptions, assumptions and justifications associated with the modelling properties being represented.

3 A description of the framework using UML

Conceptually the modelling framework is an information meta-model, designed to support prototype and established system designs, which are described by means of system properties and the associations of modelling tools that generate and use the properties. In support of this ideal we have constructed the modelling framework as an object oriented information model [9]. To describe the structure of the framework a

number of UML diagrams have been constructed [1], [8]. The basic structure of the framework is briefly described here using a class diagram.

A variety of objects have been discussed in the previous section (i.e. structure, behaviour, functional etc.) to describe the actual methodology of how the framework meta-model works. The simplified class diagram used to describe how these objects are constructed is shown in Figure 4. This illustrates how the objects are formed to compose the modelling framework. Following the figure it is clear that a framework can be composed of zero or more components. Omitting the state class for the moment we can then see that a component can contain zero or more objects (described here as generic but intending to represent the structural, goal, functional and behavioural objects). In turn each of these objects can hold zero or more property objects.

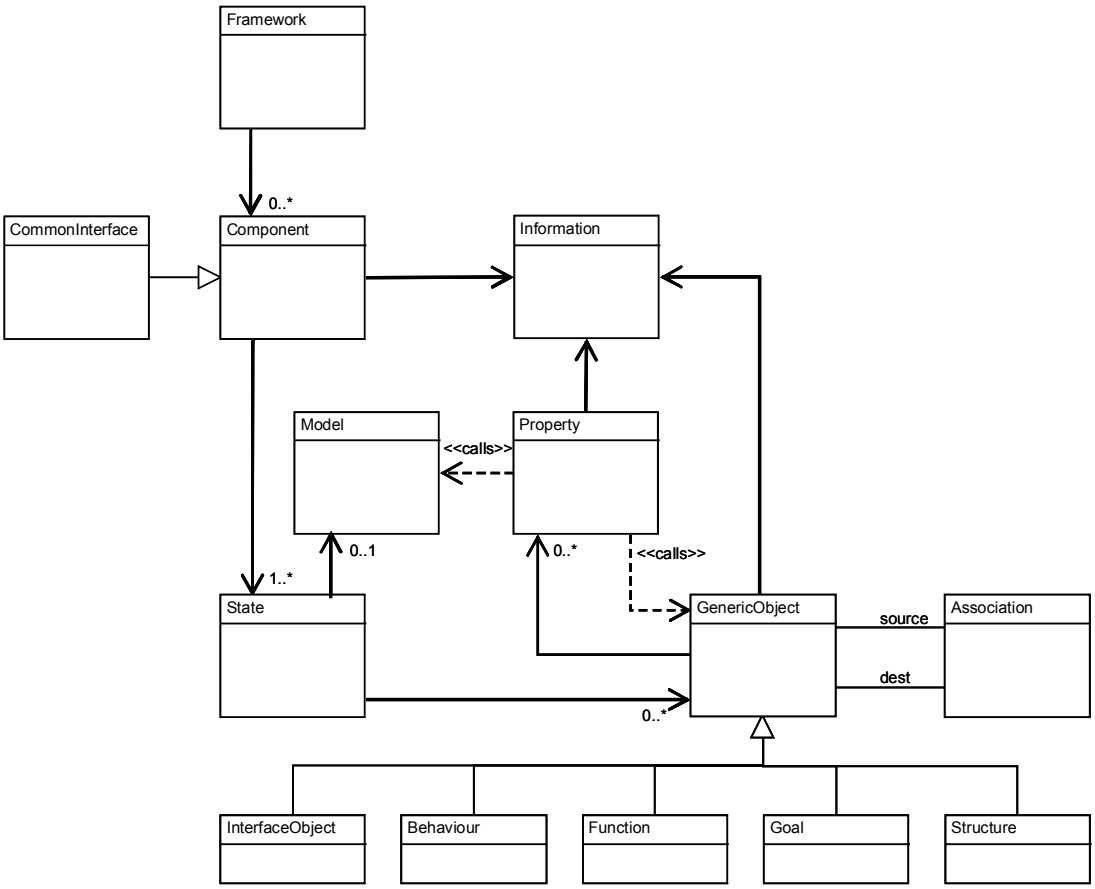


Fig. 4. The main class diagram

The inclusion of a state class allows the contents of a component to be dynamically changed so that it can represent alternate models. For example, in fluid mechanics a component may represent a laminar flow model. Given a change of conditions, and a resulting change of Reynolds number the requirement may then change to represent a turbulent flow model.

The main attributes of the property class are the property identifier and any expression or functional list associated with the property. This is arranged so that a property, force say, may be expressed in terms of its physics, such as:

```
force = mass * acceleration
```

or functionally as in:

```
force = Φ(mass, acceleration)
```

The property class highlights that as well as being referenced from a represented model, a property may also refer to another property (i.e. as a functional parameter) either within the same object, within another object, or (via an interfacing object) within another framework.

4 Description of the tracing method

Simulating a system using an integrated modelling approach such as this permits objects such as properties to be traced. This allows the fundamental low level properties of the system (which contribute to the high level emerging properties) to be revealed. The collection of information in this way allows designers to target their efforts at a very early stage in a design. Traceability can also be used to measure and test the sensitivity of properties and estimate the required fidelity of integrated models. A typical sensitivity trace may show that a low fidelity model (and therefore a cheaper model) is all that is required to simulate a system successfully. Whereas a similar trace may identify where effort is required to improve model fidelity.

To perform any trace requires some relationship between the entities in a system and this relationship may take the form of a syntactic or semantic nature [7]. Pearson [7] identifies a number of types of models within which such a tracing process can operate. These include:

(i) information models,
(ii) process models,
(iii) documentation models, and
(iv) enterprise models.

The modelling framework used here is an information model, although the potential to increase the scope remains an option for the future specifically with respect to the documentation model and the storage and reuse of design decisions. In relation to syntactic traceability, the rigour referred to by Pearson [7] is that of a functional nature, which is how the relationship between properties is described in the framework meta-model. The tracing process described here therefore concentrates on the property object, however it is perfectly feasible that given some other matching criteria (such as with subjective descriptions) other objects can be traced. Two forms of functional tracing have been developed for the modelling framework. Firstly, considering the mass of say an engine sub-system, then tracing the contributing parts of this may be considered a downward trace. An alternative to this is to perform a trace on a low level property upwards through a system to identify which emerging properties it

influences. This section briefly describes both these processes using simple examples, the implementation is discussed in Section 5.

4.1 Downward trace

Figure 5 shows a simple example of a downward trace. The "`Result`" object, shown in the top left of the figure contains an emerging property "`force`". This property is formed as a function of other properties within other connected objects. The result of the trace on this property is displayed using a tree structure on the right of the figure. This displays all the contributing properties down to the lowest level. The tree root is the title of the property being traced. Each branch in the tree is indicative of some functional operation, which is left unspecified, for example:

`y = a + b`, `y = a * b`, and `y = a / b`

are equivalent since in each case we are saying:

`y = Φ(a,b)`

This describes the syntactic relationship between the entities being traced (which in this case are properties).

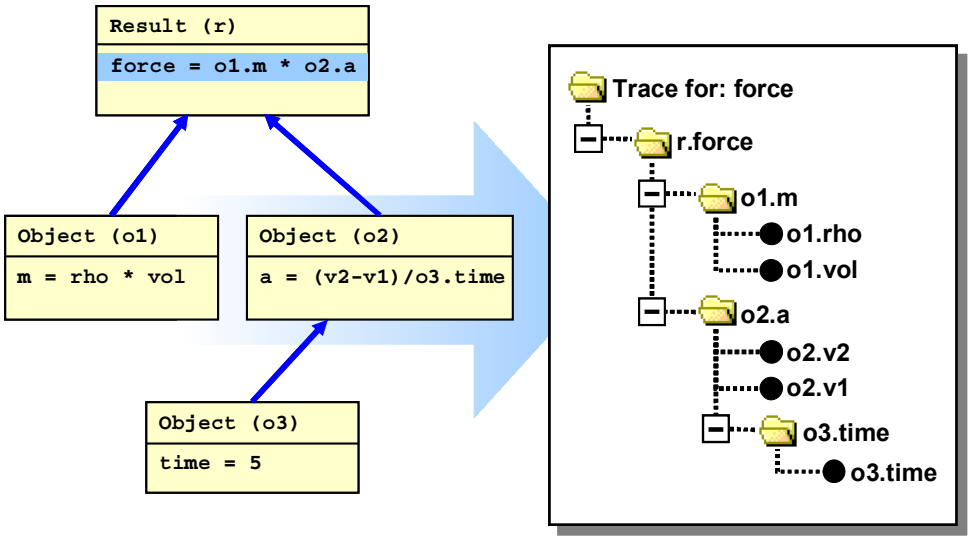


Fig. 5. Downward trace tree

The tree in Figure 5 shows that in "`Result`" the property "`force`" is a function of "`o1.m`" and "`o2.a`". In turn in "`object o1`", "`o1.m`" is a function of "`rho`" and "`vol`". The depth of the tree, i.e. the number of branch levels, may be considered a measure of the system complexity. The rightmost leaves represent the low level properties, that contribute to the traced property. Indeed it can be seen that the low level properties, which make up "`force`" in "`Result`" are "`o1.rho`, `o1.vol`, `o2.v2`, `o2.v1` and `o3.time`".

4.2 Upward trace

If we consider a similar network of objects to the above figure (shown in Figure 6), the purpose of an upward trace is to inform where in an information system the basic low level properties will be used, i.e. where they form (part of) the emerging properties. Figure 6 makes some slight changes to the information network in Figure 5 (highlighting that an information route through a framework need not be a single path). This shows an upward trace for the property "`o3.depth`" in object "`o3`".

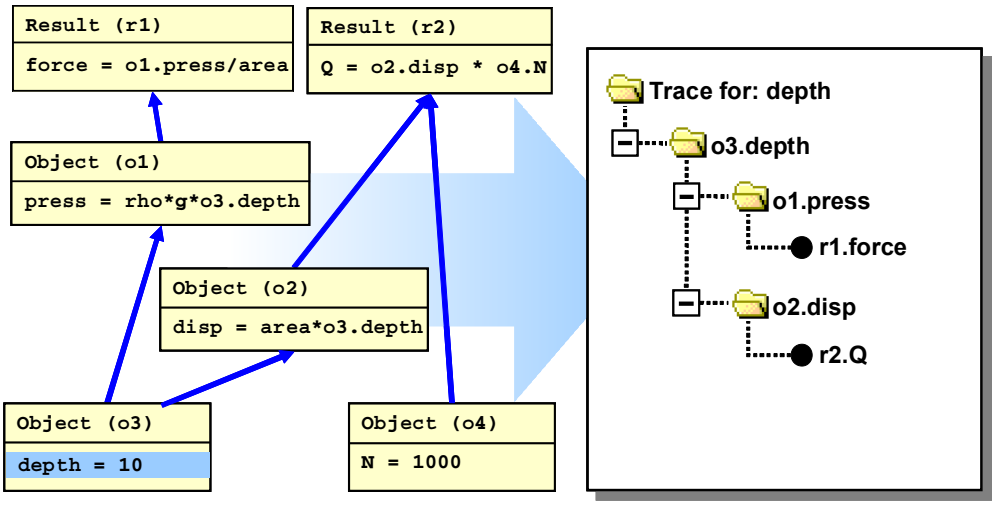


Fig. 6. Upward trace tree

The produced trace shows that the depth property is directly used as part of the other functions contained in objects "`o1`" and "`o2`". The leaves in this case are the end results for "`force`", and quantity "`Q`", i.e. the emerging properties.

5 Modelling the tracing methodology

One of the objectives of MDA [3] is to formulate solutions to problems in a high level abstract language (as high a level as possible [5]). Mellor [5] states that the use of Executable UML is at such a high layer of abstraction and that it is abstract from specific programming languages or software specifications. The specification for Executable UML requires that a set of models are prepared (represented as diagrams), which define the conceptualisation and behaviour of a solution to a problem that allow the solution to be viewed from a number of points of view [5]. These are identified as the three fundamental projections or three basic types of model, which can achieve this.

Table 1. Concepts in an executable UML model [5]

Concept	Called	Modelled as	Expressed as
the world is full of things	data	classes attributes associations constraints	UML class diagram
things have life cycles	control	states events transitions procedures	UML state chart diagram
things do things at each stage	algorithm	actions	action language

Table 1 shows that this approach can be represented by a number of UML diagrams, i.e. class diagrams, state charts and eventually into an action language described as a states procedure. Transferring this view to the tracing process methodology we can consider the stages that are required to perform this. These are identified as:

(i) formulation of a data structure,
(ii) population of the data structure, and
(iii) the tracing process itself, which is manifested as a search of the populated data structure.

The step by step processes identified to make the trace methodology work are shown in the trace process swimlane of Figure 7. Using this approach we can follow the above steps with the contents of Table 2, which compares well with Mellor's [5] view of Executable UML. In this case the data structure that we intend to use was best described using a class diagram. The next step, population of the data structure, is related to a state chart diagram (Table 1). Based on the structure of the class diagram in the first step, a description of the population of this was found to be more clearly explained using a sequence diagram. This type of diagram (used primarily to model the dynamic aspects of a system [1]) shows quite clearly the interactions between the objects used. The final step, modelled as an action diagram was represented by an activity diagram. Although no standard action language is currently available within standard UML [5], this was found to be the most representative.

Table 2. A summary of the procedures described in this section

Step	What it is	UML diagram used	Modelled as
1	data structure	class diagram	classes
2	assemble data from framework	sequence diagram	procedures
3	perform search	activity diagram	actions

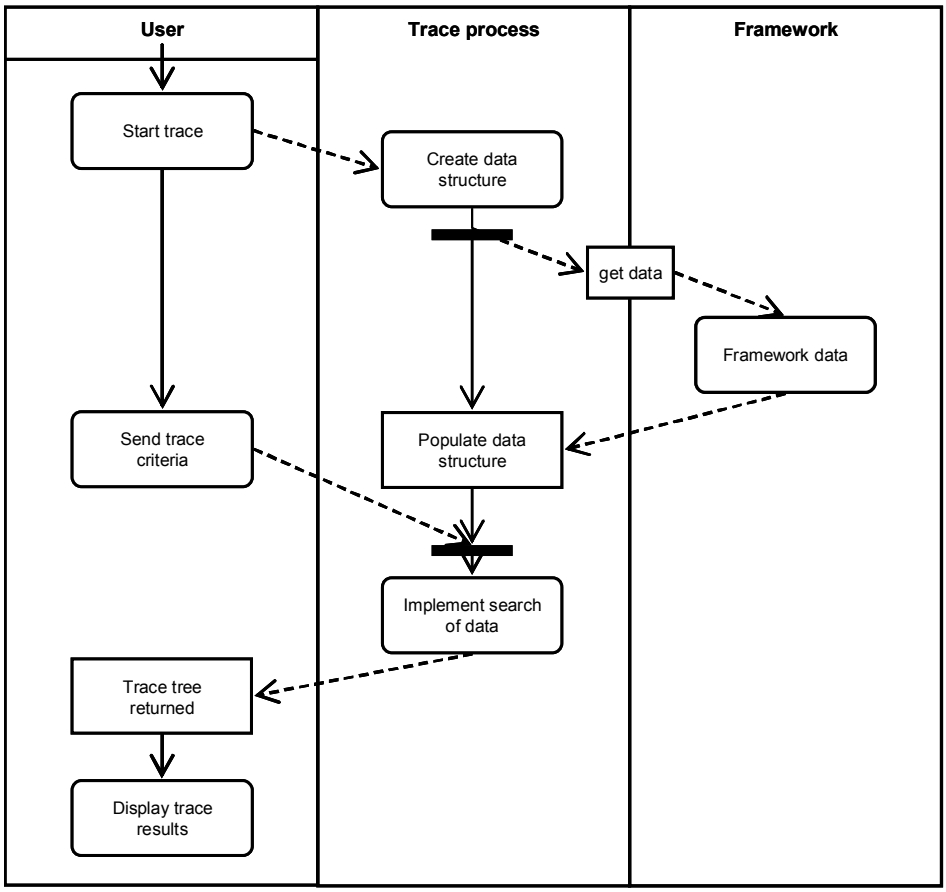


Fig. 7. An activity diagram highlighting the three basic processes (within the Trace swimlane) required for tracing

These three types of diagrams, which we have now identified are used to describe the tracing methodology in such a way that it satisfies the criteria required for Executable UML. These are now described in more detail.

5.1 Class diagram of the data structure

The class diagram describing the data structure is shown in Figure 8. This is essentially a template that will be populated by the properties obtained from the framework objects and upon which the search processes will operate. The primary data container used in this template is a hashtable, which has two main methods, `put()` and `get()`. The put method requires two parameters, which reference objects, these are a key and some content. The method uses a unique key to place a content object within the container. Once stored the `get()` method when given the unique key will retrieve a reference to the stored content object from the container. Although there should be no reason for specifying the type of data storage used here, a hashtable was found to be the most convenient from a practical point of view, since early develop-

ment of this process was undertaken with a view to implementing this using Java. Despite this, the data storage methodology may be thought of as generic regardless of the practical compromises that have been made.

The two classes, which inherit the hashtable characteristics (shown in the figure) are `PropertyTrace` and `TraceRecord`. `PropertyTrace` is the fundamental class used in the tracing process and contains two main methods: `constructData()`, which actually assembles the data and `doSearch()`, which performs the searching process. The data objects placed in this container are an aggregation of a key (`ObjectID`) and some content (`TraceRecord`). The key object is a string that uniquely identifies an object within the framework, in the form of:

```
componentTitle+":"+objectTitle+":"+prefix
```

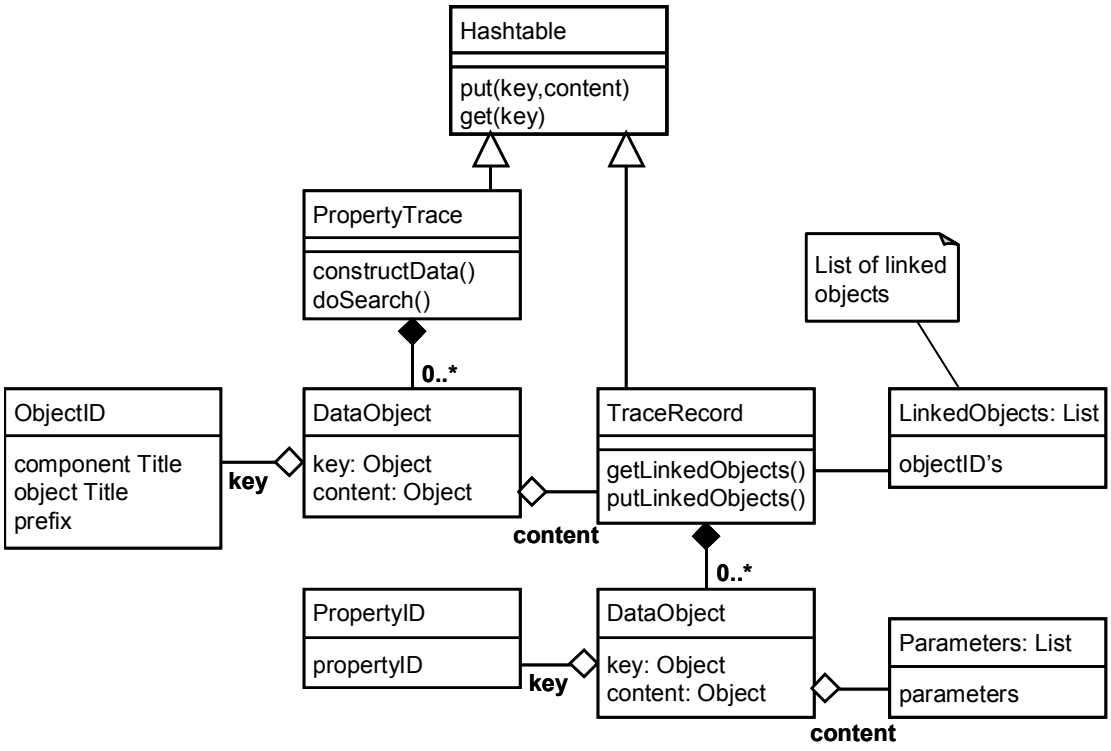


Fig. 8. Class diagram describing the data structure

The content object of this class, titled `TraceRecord` is itself an extension of the hashtable class and is a container for data objects that have a unique key, which references the properties held by a framework object. The content held within this data object is a simple array of parameters attributed to the property, for example (and as we have seen):

```
y = a * b / c
```

can be referred to as:

```
y = Φ(a, b, c)
```

where y is the property and the parameters are a, b, and c. The class diagram in effect describes a nested hashtable, the basic PropertyTrace class holds objects where the key is a unique identifier for all the objects in a framework and these hold a content object TraceRecord. The TraceRecord container holds a list of objects referred to by a key that represents all the properties within a framework object, the content being the parameters, which relate to that property.

5.2 Populating the data structure

The data structure described in 5.1 is populated with the contents of the objects obtained from the framework (in this case the data relating to properties). This process is described using a sequence diagram that progresses against a timeline and illustrates how each instantiated class (and associated objects) are accessed and manipulated. The sequence diagram is shown in Figure 9. Highlighted are the hashtables PropertyTrace and TraceRecord described in Figure 8.

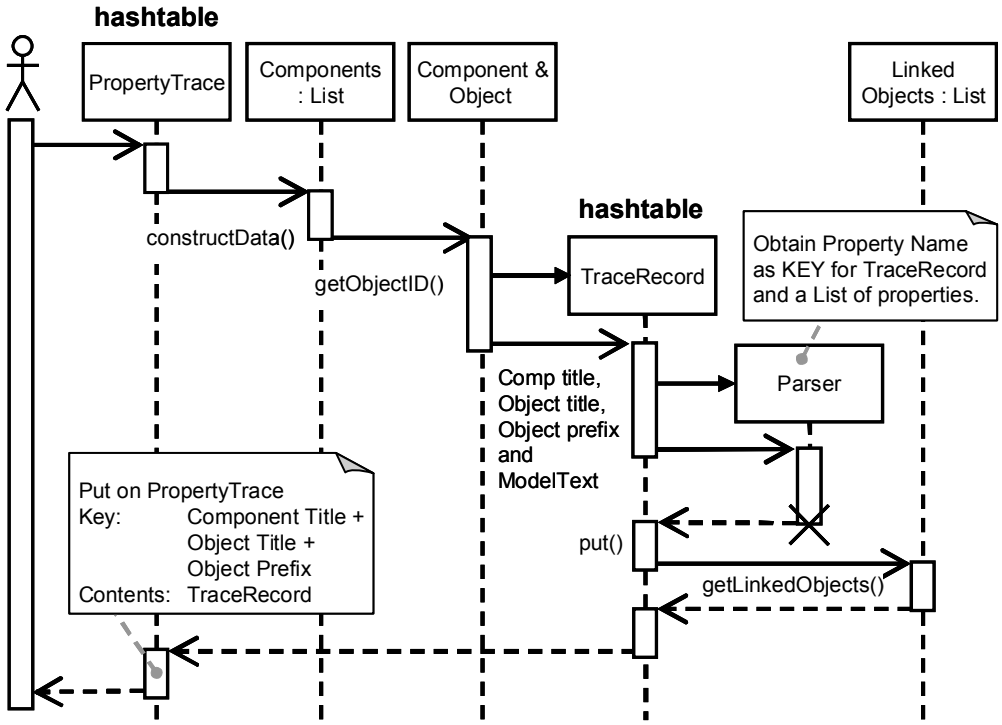


Fig. 9. Sequence diagram, populating data structure

The initial call to an already instantiated PropertyTrace accesses a list of all the components within a framework. The process is a compound iteration through the list of components (each of which contain a list of objects) obtaining the distinct component/object title information, i.e.:

```
componentTitle + ":" + objectTitle + ":" + prefix ,
```

This provides us with sufficient information to construct a key object identifier for storage in `PropertyTrace`. At this point a `TraceRecord` is instantiated and each line of any modelling representation text contained by an object (i.e. the functional relationships) is read. This is passed to a helper object called `Strip&Parse`, which checks each line for suitability and if relevant divides the line into a property identifier `String` and an `ArrayList` of parameters.

Once accomplished, the content object (the `ArrayList` of parameters) is placed into the `TraceRecord` hashtable container using the property identifier as a key. The next step is to access a list of associations. This is an `ArrayList` of all the links within a framework. The purpose of this is to identify any links that point to the current object. A call to the `AssociationList` object returns an `ArrayList` of such links that can then be added to the `TraceRecord` using `PutLinkedObjects()` (Figure 8). The newly instantiated `TraceRecord` can now be used as a content object along with the unique object identifier as a key to place this within the `PropertyTrace` hashtable.

Essentially this entire process shown in the figure scans through all the data and property objects held within a framework and where applicable parses the text held within the objects in a framework and populates the data template described in 5.1. The purpose of arranging the data in this way is so that a recursive search may be performed very quickly.

5.3 The search process

The recursive search process of the data structure that we have now assembled is described in this section with the aid of an activity diagram, shown in Figure 10. This diagram describes the recursive search process when a trace is performed. The diagram is entered (top left) with some trace criteria, specifically the identity of the property object and framework data object, which form the access keys to the data storage container. If this exists the content objects are immediately returned by the hashtable and a tree node is constructed for the trace display. Assuming a syntactic relationship exists between the trace property and other properties within a framework, the trace process then continues searching the data structure recursively.

If the parameters are in the form of a leaf then the search call simply returns. If this is not a leaf then the search will proceed based on the parameters. The property is a function of these parameters so that the next level of search will take place on these parameters as further properties. Eventually a final leaf is determined at which point the process terminates and returns a tree model that can be displayed in the application tool. A sample trace obtained from an implementation of this is shown in Figure 11.

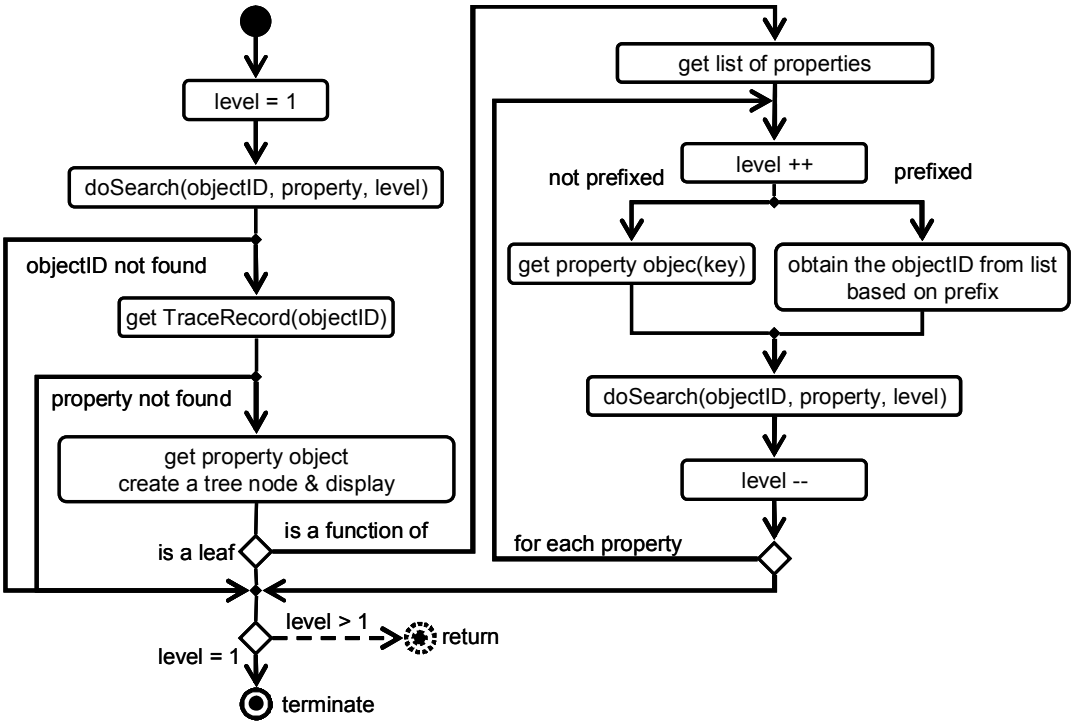


Fig. 10. The trace search process

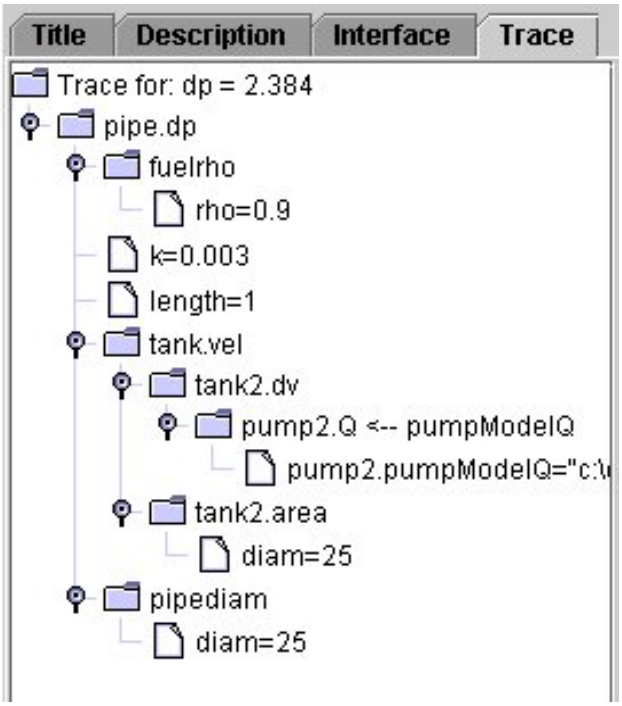


Fig. 11. A sample trace obtained from a implementation of the tracing methodology

6 Conclusions

The modelling methods used to design and implement a tracing capability within a modelling framework have been presented in this paper. These approaches have illustrated some of the advantages of modelling using platform independent methods. The modelling process has been used to rigorously design both the structure and behaviour of a tracing procedure that is flexible and extensible. Currently aimed at a syntactic relation between properties, by modelling the process in this way it is anticipated that this can be extended into areas such as design decisions and semantic relationships. Although not directly involved in a meta-modelling environment, this has shown that by planning, developing and implementing a system using such methods, it will be possible to omit the human-programming procedure from future development. So far the models developed in this project have been used to develop Java code, it is hoped that future work in this field will be to extend the use of Executable UML to the entire modelling framework. Once this is achieved then the complete system can be modelled in such a way that a tool such as: say, the Kennedy-Carter iUML software can be used to produce the object code.

References

1. Booch, G., Rumbaugh, J., Jacobson I,.: The unified modeling language user guide, Addison Wesley Longman (1999) ISBN 0-201-57168-4
2. Chandrasekaran, B., Josephson, J.R.: Representing function as effect, Proceedings of the 5th International workshop on Advances in functional Modelling of Complex Technical Systems, Paris (1997) 3-16
3. Frankel, S.: Model driven architecture, OMG Press, Wiley Publishing Inc. (2003) ISBN 0-471-31920-1
4. Keuneke, A.: Device representation: the significance of functional knowledge. IEEE Expert (1991) 22-5
5. Mellor, S.J., Balcer, M.J.: Executable UML: A foundation for model-driven architecture, Addison-Wesley (2002) ISBN: 0-201-89685-0
6. Modarres, M., & Cheon, S.W.: Function-centered modeling of engineering systems using goal tree-success tree technique and functional primitives. In Reliability Engineering and System Safety 64 (1999) 181-200
7. Pearson, S., Saeed, A.: Information structures for traceability for dependable avionic systems, Technical Report number 567 Department of Computing Science, University of Newcastle (1997)
8. Rumbaugh, J., Jacobson, I., Booch, G.: The unified modeling language reference manual, Addison Wesley Longman (1999) ISBN 0-201-30998-X
9. Sage, P.: Systems engineering, J Wiley (1992) ISBN 0-47-153639-3.
10. Salustri, F.: Function modelling for an integrated framework: A progress report, Proceedings of the 11th Florida Artificial Intelligence Research Symposium, special track on Reasoning about Function (1998) 339-343
11. Subramanian, D., Wang, C.E.: Kinematic synthesis with configuration spaces, Research in Engineering Design, Vol. 7 (1995) 192-213

Services integration by models annotation and transformation

Olivier Nano, Mireille Blay-Fornarino

I3S, Université de Nice-Sophia Antipolis
{nano,blay}@essi.fr

Abstract. Each component platform (Jonas, JBoss, OpenCCM, .NET, Julia) proposes its own software infrastructure. Objects like containers, controllers, and interceptors characterize these infrastructures. A client request flows through these objects to the component. The management of the services offered to the component by the platforms is located in these objects. To integrate a new service in different component platforms, the developer has to understand the infrastructure of each targeted platforms and to interlace calls to the new service with the calls of the services already provided by the platforms. It involves two difficulties: firstly understanding the software infrastructure and secondly being able to compose the new service with the services already included.
So in this paper we propose an approach, based on model, to deal with service integration. And we show the definition of a meta-model to manage service integration in component platforms (such as the one implementing EJB or CCM specification).

1. Introduction

Component programming allows better code modularisation. Two aspects mainly achieve this modularisation. First the use of interfaces and communication protocols to isolate components from each others. Second the separation of the components code and the services code (authentication, transactions, persistency) allow the programmer to concentrate on the component code. He declaratively specifies which services will be added by the platform during the configuration of the application. To handle this service integration, the component platforms define a software infrastructure that will coat the execution of components. Objects like containers intercept messages intended to components and apply to them some treatments. The calls to the services are located in the objects that surround the component.

Each platform (EJB[4], CCM[1], .NET[7], Fractal[25]) proposes its own software infrastructure. As an example let's compare the infrastructure of Jonas[5] (an EJB implementation (fig. 1)) and OpenCCM[8] (a Corba Component Model implementation (fig. 2)). These figures are simplification of the real infrastructure of these two platforms, but still expose the differences about how the services are handled. In the Jonas infrastructure the services are linked sequentially, whereas in the OpenCCM infrastructure, the coordinator manages the sequencing.

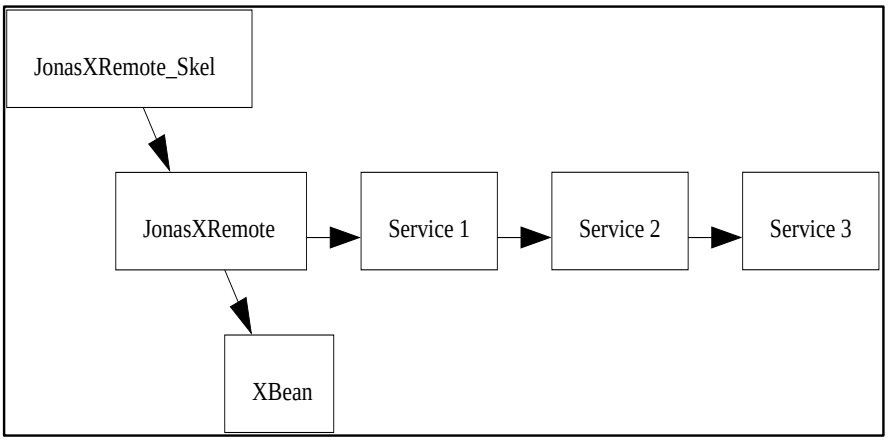


Fig. 1. Message flow in Jonas

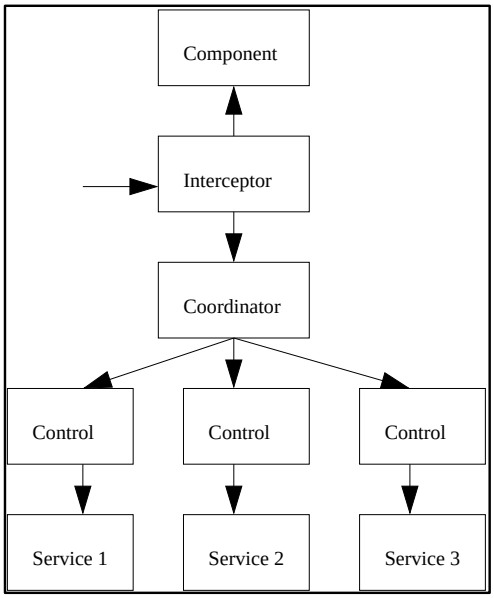


Fig. 2. Message flow in OpenCCM

Even in the case of the implementation of the same general component model the infrastructures differ. For example the two EJB implementations: Jboss[6] and Jonas[5]. Jonas intercepts request to the component with a generated skeleton, whereas in Jboss the interception is done through the use of dynamic proxy on the client side.

This heterogeneity of infrastructures makes the diffusion of new services harder. Indeed to add a new service in a platform, the developer has to know the details of the infrastructure to decide where to insert the calls to the new service.

Each platform provides common services not necessarily implemented in the same way as legacy services. The service developer has then to understand the interlacement of existing services to keep the correct composition semantic.

The code of the interposition objects is often generated (following the contract of the component). In this case the developer has to modify the generator in order to generate the correct interposition code that includes the calls to the new service.

In our approach we consider a service as a set of components and rules specifying how and when to call these components. For example to be protected by a security service, a component has to ask the security manager (which is a component of the service) if a call is authorized or not and takes a decision on what to do next upon the answer of the security manager. We don't need to know the platform to specify this scenario. So we propose to describe service integration in an independent platform manner, compose service integrations and generate adequate code according to the targeted platform (cf. fig3).

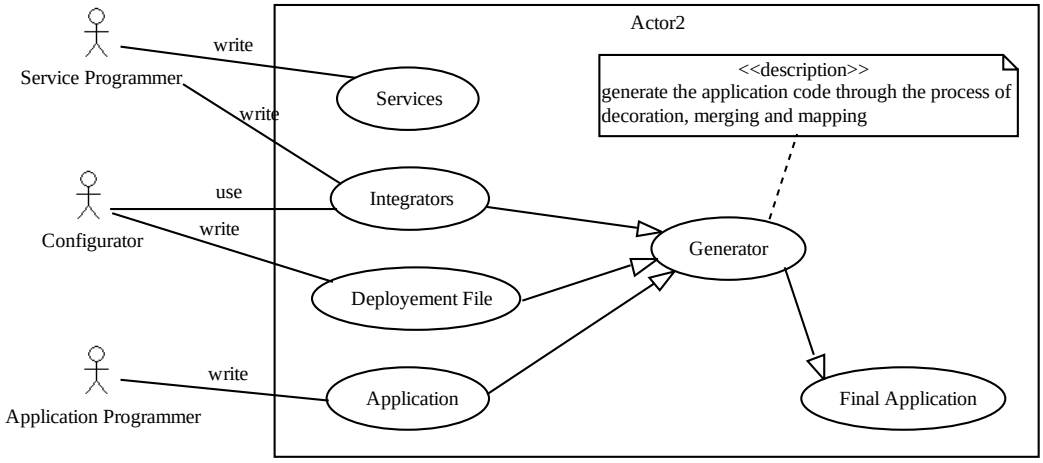


Fig. 3. Service integration process

We would like:

- To describe the integration of service at a higher level independently of any platform
- To compose service integrations at this higher level or detect conflicts independently of the order of declaration of services to integrate
- To project these descriptions of service integrations into real platforms (as it would be done manually) without modifying the infrastructure of the platform.

In order to do so, we need an abstract model of component platform to base our integrators on. Our experience in component platforms showed us that the software infrastructures of component platforms are too different to construct one structural meta-model handling platforms architecture. Moreover service integration often has to deal with communication protocol underlying platforms components. We propose to model the flow (transition) of a message from a client to the component. For this modelling we use a behavioural meta-model. This meta-model is only useful to the description of service integrations. It will not be embedded in the component platforms at projections time. The projection will produce the adequate code according to the targeted platform. This step could be operated as model

transformation between the model of service integration and the platform model, if this one is detailed enough to be able to generate service code.

In this paper we will defend the approach of using meta-modelling to service integration more than the particular behavioural meta-model that we propose in the third section. First, there is a section describing the motivation for this work and the model process we propose for service integration in Section 2. Section 3 details one meta-model and the set of operations we allow on this meta-model to deal with service integration at structural and behavioural level. Section 4 describes the stages of service integrations on examples, and discusses composition and projection of service integrations. Sections 5 provide a look at our perspectives and some conclusions.

2. Model-driven process for service integration

The life-cycle of a "service" may be basically summarized with the three following steps: build-time (specification of the service and implementation of the components), integration-time (definition of how to integrate the service and with which tools. That is to say how to implement the calls to the services in the application) and run-time (the service is called from inside the applications). Since the adoption of the MOF recommendation (Meta-Object Facility) [2] by the OMG in 1997, meta-models are often used to design services, platforms component, and even execution models. But, when dealing with integration, techniques such as Aspect Oriented Programming [12] and generative Programming [31] are used. These techniques are language and platform dependent. The process of integration is then no more driven by models and it implies several drawbacks.

Service Integration based on models

The new UML 2.0 super-structure [3] gives support to components modelling and addresses customisation for modelling of component architectures: J2EE/EJB, .NET, COM and CCM. So, it seems to be a good base to define services integration in an independent way of component platforms. Indeed objects really involves in the management of the requests are not designed and the container don't correspond to a real unique object in any implementation.

Service calls occur at different steps of the execution flow of a request and at different stage of the life-cycle of the component. According to aspect languages, join points are at creation and destruction, at sending and reception time, and so on [12,14,11]. However according to middleware underlying component platforms, other join points occur: marshalling or unmarshalling a message, returning a value, activating a component, diffusion time for event oriented communication, accessing shared data… The way to catch these behaviours is to express service integration as aspects on proxies, interceptor, home, channels, etc… [26]. The aspect is then platform dependent. Even in this case, some join points cannot yet be distinguished such as unmarshalling and accepting a message in a proxy (as in java rmi [32]). For example, it's natural to express that after unmarshalling a message we want to decode it and that when accepting a message we want to be notified. If the two different join

points can't be distinguish, programmer would have to express that decoding must be done before notifying. So, he introduces coupling between services, when it doesn't exist at conceptual level.

In 1995, McAffer proposed to handle the execution flow of a request in an object by means of set of meta-objects, which can evolve according to the targeted object model [11]. We propose to base service integration on meta-UML specification completed with the specification of the execution flow. When no services are yet defined we call this meta-model, the **base meta-model**.

Modeling service integration by Annotating models

Catalysis [27] encourages a strategy in which an initial design is gradually modified to produce a completed one which is a single, fully integrated design. However, service integration modelling cannot follow such a strategy, because service integrations should be defined by separate programmers and don't need to be fully integrated. Some interesting design approaches are rooted in the aspect-oriented programming paradigm such as approaches extending the UML with stereotypes specific to particular aspects (e.g. synchronization [28] or command pattern [29]). In both examples, many of the behavioural details are not explicitly designed in the UML, and composition of stereotypes is not discussed.

The subject-oriented design model [29] supports a design strategy in which pieces (subjects) are identified and designed separately, and may remain separate in the completed design, though related by composition relationships. In particular, composition patterns [22] are used to design cross-cutting requirements, as are services. Composition patterns are based on the combination of merge integration from subject-oriented design and UML templates. Providing flexible separation of concerns gives the developer the opportunity to separate the same concerns throughout the development lifecycle. But this approach doesn't deal with joint points different as the one expected by UML in scenarios, and composition of patterns bound on a same component is not discussed.

So, we propose a mixed approach rooted in the aspect-oriented programming and meta-modelling paradigm. Service integration modeling consists modifying the base meta-model annotating it by rules. Rules are based on operations defined by the meta-model (such as addingFeature, addingControl), and a set of operators such as sequence or conditional to express behavior changing. We call '**integrator'** a service integration description. An integrator is a pattern, describing a set of rules to apply on the behaviour and on the structure of a component to integrate the calls to the service. Integrators may be used to configure an application using bindings between integrator parameters and application entities.

Weaving rules to check consistency of multiple services integration.

According to each service integration, rules should be defined possibly on the same meta-elements. As it's extremely difficult to predict beforehand all possible needs for services, and as we don't want deployer or programmer to be expert in all services, we have to be able to detect conflicts when some semantic inconsistency occurs such as asking for notification of a message when this one is still encoded. In order to achieve automatic composition of services, operators and operations occurring in rules should support merging property i.e. two operations occurring on a same

component should express how they merge when possible, and the result of the composition should at his turn be summarized in an expected set of operations. As service integrations are defined in an independent way, the composition of operations must also respect commutative and associative properties. So, at this step, the process of service integration can be stopped if the merging of integrators corresponding to a user configuration fails. For example, adding two attributes on a same component is accepted if their name is different, otherwise the merging fails. Adding an operation *m* and retracting an operation *p* will result in the same set of operations if the properties are different, otherwise the composition will be refused. The composition of behavior is more complex and will be described in section 4 but it respects commutative and associative properties too.

Model-driven services integration

At the heart of the MDA approach is the question of model transformation. We assume that meta-model of platforms are defined. The designer and programmer are given profiles, UML for Jonas or UML for openCCM. With the help of some facilities provided by the UML CASE tool vendors, they will then, use these dialects of UML to prepare the transformation between a UML design model and the code to generate components [17].

So, in order to generate code integrating service calls, we don't have to deal with code generation but we have to specify the transformation from the meta-model annotated with integrators to platform meta-model. So for each targeted platform a meta-model should be defined designing the objects useful to adapt the component behavior such as the interceptors or controller described in figure 1. This meta-model should have to complete the independent model such as defined in UML 2.0, which don't match the execution flow in term of real objects.

The transformation will not always been possible because some controls introduced by the meta-model cannot be supported by the targeted platforms (cf. section 4).

So, more than a meta-model, we propose an approach of services integration based on meta-modeling. This approach differs from composition pattern oriented design because join points are defined as models and annotation operations are defined by the meta-model and must respect properties towards merging. Our approach for service integration is Model Driven Architecture based [17].

3. Example of meta-model for service integration

Initially we have chosen to focus on component platforms corresponding to component models such as EJB or CCM. To manage structure and behaviour of components, we need a meta-model integrating the both dimensions. We use the structural meta-model proposed in UML2.0 [3] that we extend with meta-objects useful to control and annotate realization of this meta-model. We introduce rules and integrators to deal with component instances, and integratorClass to annotate componentClass (cf. Figure 4).

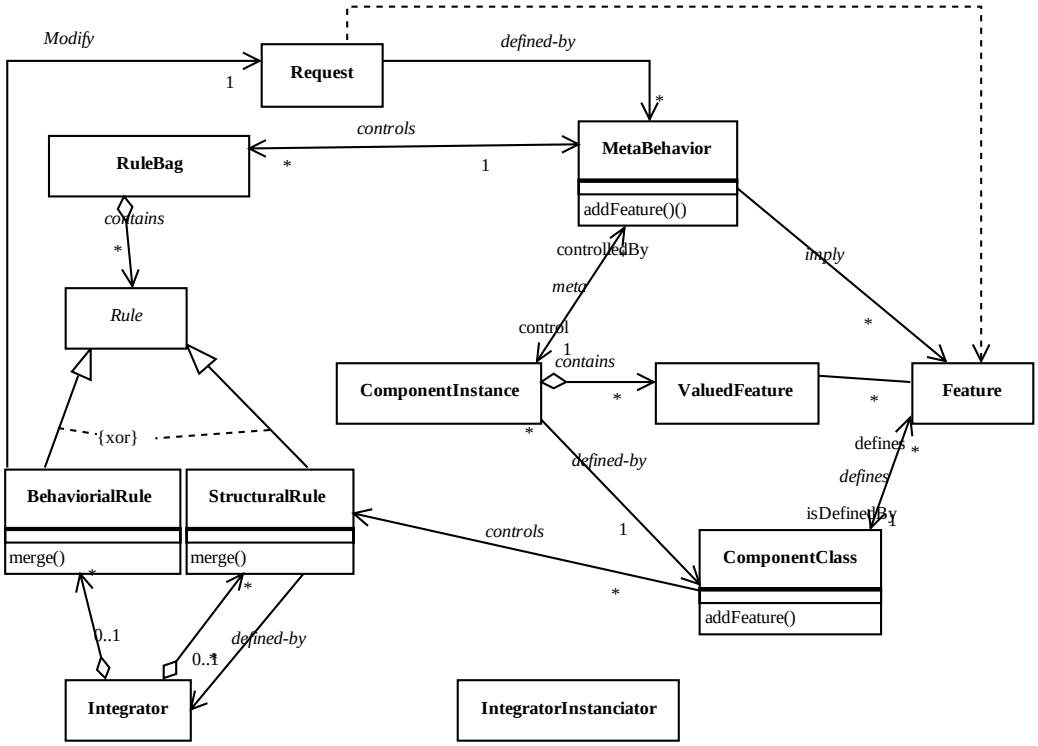


Fig. 4. Structural meta model

As Coda shows with objects, every component instance has a conceptual meta-level. The meta-level is not a single object but rather a set of meta-objects, each of which describes some aspect of the base level of the component behaviour. The mapping between this meta-level and the implementation will depend of the targeted platforms.

In order to describe our approach, the following figure shows the execution meta-model, on which we based our first experimentation. It fits the Remote Procedure Call for a component. The execution meta-model is composed of sequenced meta-objects. These meta-objects represent the different stages of transition of a message from a sender to a receiver.

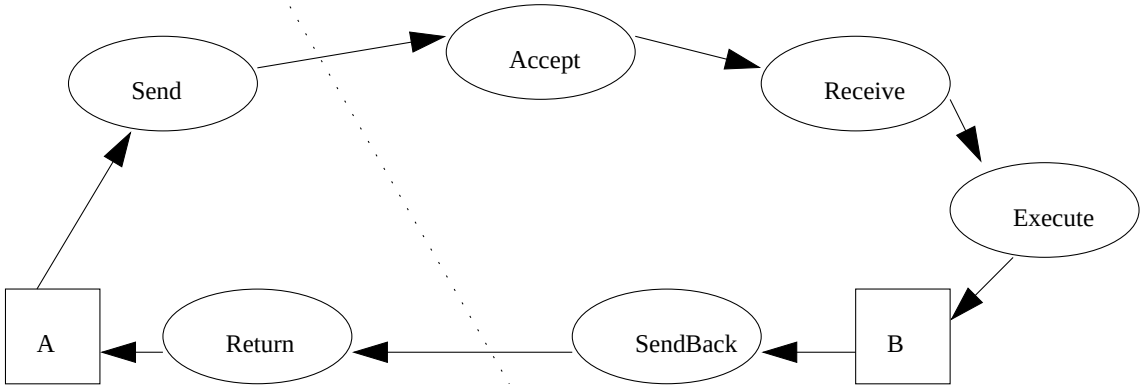


Fig. 5. A configuration of meta-objects defining a basic plan

The execution meta-model defines a basic plan (that can be extended). This basic plan is composed of the following meta-objects: Send, Accept, Receive, Execute, SendBack, Return.

Let us complete this execution meta-model. The meta-objects are linked together with the set of Prolog like rules defined in Figure 6. We use this rules to describe the link between meta-objects in order to detect inconsistency between the extension of the basic plan and the integrators that use the basic plan as reference.

- Send(Message , NetworkRequest) :-
  send(Message, NetworkRequest) ; ! ; Accept(NetworkRequest,)
- Accept(NetworkRequest, Message') :-
  accept(NetworkRequest, Message') ; ! ; Receive(Message',)
- Receive(Message, Message') :-
  receive(Message, Message') ; ! ; Execute(Message',)
- Execute(Message, Message') :-
  execute(Message, Message') ; !; SendBack(Message',)
- SendBack(Message, NetworkRequest) :-
  sendback(Message, NetworkRequest) ; ! ; Return(NetworkRequest,)
- Return(NetworkRequest,Message) :- _return(NetworkRequest, Message)

Fig. 6. Interactions between meta components

Each term beginning with "_" designs the message to evaluate the corresponding behaviour. The same syntax is used in [22] to express evaluation of the initial message. The "_execute" will modify the message to affect the return value, whereas "_receive" in a majority of platforms will not modify the message and has no effective action.

The exclamation mark let us specify when a meta-object finishs its execution. It is then available to accept new messages. It allows us to take into account the modification of interactions between meta-objects forcing to wait for an end of treatment. Omitting it, means that the evaluation will be finished when the evaluation of the rewriting rules will be finished.

The parameters of the Prolog predicates are from the type 'Request'. For now we distinguish two different types of messages. One that is sent through a network and

that is constituted of an array of bytes. And another one that is exchange through meta-objects which contains the signature of the methods, parameters value and the return value.

4. Integrators description and projection

An integrator is represented by a set of role instances and a set of rules. A role is a generic interface describing required operation on a component as defined in UML. Rules describe the transformations to apply to structure and behaviour of components (infrastructure included) to introduce calls to the service (which is also a set of components).

A behavioural rule is composed of two parts. The left part identifies the operation to control and the step of the execution flow to rewrite. The right part is the transformation. Transformations are expressed using operators such as delegation, sequencing, concurrency, and conditionals. We will not give much more details here; we will describe the rules as necessary when used. More on rewriting rules can be found in [10,33]. These rules have some interesting properties like determinist composition with explicit failure case respecting commutative and associative properties. So, the composition result does not depend on the order of the rules.

The structural rules allow the modification of attributes and operations using operations defined on structural meta-level. In order to simplify the rules writing, we have introduced syntactic sugar and we don't use explicit call for example to addFeature() but use a declaration of variables to handle it.

Integrators are per instance basis. They describe the transformation to apply to component instances. Class Integrators define the binding of an integrator to instances of classes. They are used to manage application configuration by final user.

4.1 Using the basic plan to control behaviour: from definition to projection

Let's take a first example: the integration of a notification service. The notification can be made at different stages of the operation's flow. We have chosen to show two different possibilities: the "Notify" integrator that notifies after the reception of a request and the "NotifyCall" integrator that notifies after the acceptation of a message. Other choices (before accepting a request to allow to notify the arrival of request (not yet unmarshalled), before its execution, and so on) are possible.

```
Integrator Notify (Component c1; Channel c){
  c1.* [Receive(m, m')] :- _receive(m, m') ;
                                c.notify(m').
}

Integrator NotifyCall (Component c1; Channel c){
  c1.* [Accept(m, m')] :- _accept(m, m') ; c.notify(m').
}
```

This integrator takes as parameters components conformed to component and channel roles. The role Channel requires a 'notify' operation.

In order to ease the configuration we propose the use of a class integrator. The following class integrator allows the instantiation of variables to feed the instance integrator.

```
Class Integrator NotifyCall (Class C, Class Channel){
      Channel ch = new Channel();
      C.instances -> (ForAll  co : Component do{
        NotifyCall.integrate(co , ch)
      }}
```

In this class integrator we instantiate a channel that we binds by the integrator "notifyCall" to all instances of the component i.e. they share the same channel to push events. If we want that each component instance to publish on his own channel we would have instantiate a new channel for each call to "integrate". Of course depending of the targeted platform the projection will be done only for all instances without being able to integrate a service in only one particular instance.

Configuration: For configuration management reasons and so due to difficulties to identify instances, at present, we propose user to express service integration on component classes and not on specific component. For referencing the channel at projection time, the user will have to complete a configuration files, specifying which class of channel to use and on which component to integrate this service. An example of such a configuration file could be like the one described below.

```
<NotifyCall_Class>
<Component> MyTestComponent</Component>
<Channel>org.omg.CosEventComm.PushConsumer</Channel>
</NotifyCall_Class>
```

Projection: Now that we have described all the steps to describe the service integration in a component let's detail the projection mechanism. As an example we have chosen to describe the projection in the Jonas platform. In Jonas we make the following translation between the model of the concrete platform and the meta-model (cf. fig 7). We don't yet formalize this mapping, but implement it. Here is an intuitive description of this mapping.

- the annotations on `Accept` meta-object will be mapped on code before the call to the *interposition* object in the skeleton
- the `Receive` meta-object as the code before the call to the Bean implementation in the interposition object (called remote object in Jonas)
- the `Execute` meta-object as the Bean
- the `SendBack` meta-object as the code after the call to the Bean implementation in the interposition object.

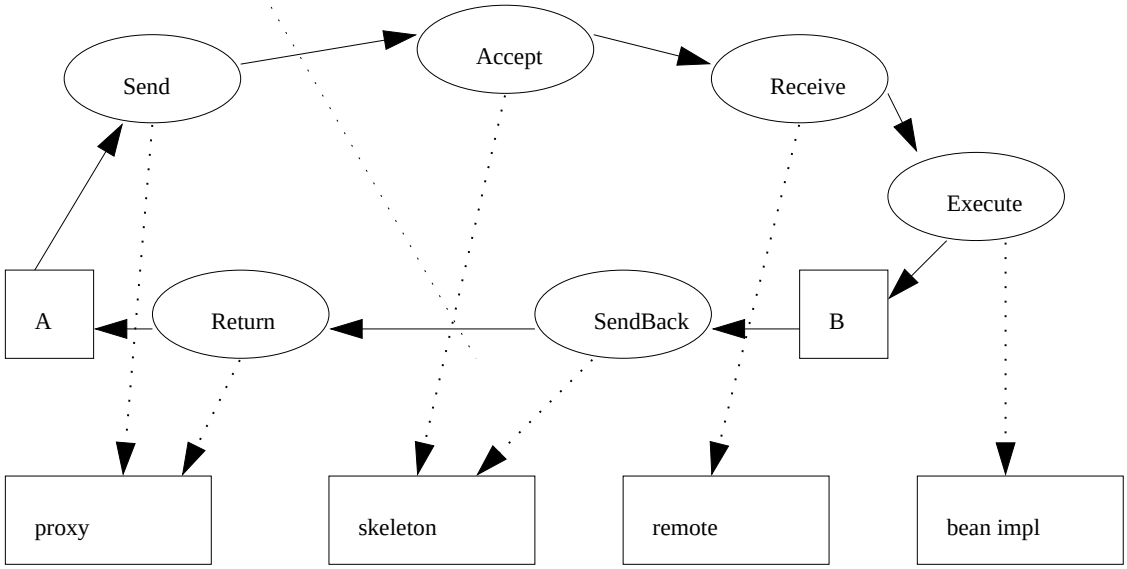


Fig. 7.

4.2 Adding properties: composition of features

It is sometimes necessary to extend the interface of a component to integrate a service. This extension of interfaces let the client query the component about the state of a service.

As an example we show how to integrate call counters that are accessible from outside. The semantic of the call counter is to count every call to the component even the ones that will be rejected. Another counter integrator aims to count every call received by the component (not the rejected ones).

```
Integrator CallCounter (Component c1){
    [c1, private] long nbCall := 0;
    [c1, public, instance] nbCall() :- return nbCall;
     c1.* [Accept(m,m')] :- nbCall++ ; _accept(m, m').
}

Integrator ReceivedCallCounter (Component c1){
  [c1, private] long [1] nbCall = 0;
  [c1, public] nbReceivedCall() :- return nbCall;
  c1.* [Receive(m, m')] :- _receive(m, m'); nbCall++ .
}
```

Abstract operations can't be added using an integrator. Indeed, the need for abstract method should be expressed as a role. Without "implementation" of the

[1] We use CORBA types to deal with heterogeneity of platforms.

operation to add, even simply calling a component of the service, the integration of service can't be performed.

Adding feature with class visibility is forbidden in an integrator too, because integrator deals with component and not with component class. Adding such properties should be possible using class integrator.

Composition: If the "ReceivedCallCounter" and "Notify" integrators are plugged on the same component a merging has to be realized because rewriting rules occur on the same meta-objects. Due to formal rules of merging [10,33], there are no conflicts. It will result in the following rule:

```
c1.* [Receive(m, m')] :- _receive(m, m') ; c.notify(m')
                                          // nbCall++ .
```

If the "ReceivedCallCounter" and "CallCounter" integrators are plugged on the same component, the merge will detect a conflict because two properties having the same name have to be added. Using techniques of renaming such as the ones used for dealing with point of views will then solve the problem. If the public added operations had the same signature, the merge would be rejected, even if the visibility had been different. Indeed, as service integrators are defined by different programmers, in different times of the process flow, it seems very difficult to ask final user in case of conflict to negotiate such as choosing which property overrides another one [22].

Projection: Adding public operations at projection time in Jonas will modify the remote interface of the component. As the added features rely only on properties added by the service and are not relative to the state of the bean, the variables `nbCall` eventually renamed as `CallCounter_nbCall` are projected in the interposition object and the operations are implemented in the interposition object too. On the contrary, if added operations used component properties such as getAllValuedFeatures(), the operation implementation would have been added to the bean implementation. The level of integration of a feature (*component* or *service*) is computed at level of the model.

4.3 Modification of the arguments in the meta-behaviour

Until now, we have defined simple integrators that don't need to modify the meta-model. We will now focus on integrators that need such issue. We can distinguish two kinds of modification on the basic plan: one that modify the arguments of the meta-objects and one that modify the sequencing between meta-objects presented in the following section.

Let's take another example for encoding operation from a client to a component. The following code defines the corresponding integrator.

```
Integrator Encoding (Component c1, Encoding Component cc){
  c1.* [Send(m,m')] :- m"=cc.encode(m);_send(m",m').
  c1.* [Return(m,m')] :- _return(m,m"); m'= cc.decode(m").
  c1.* [Accept(m,m')] :- _accept(m,m"); m'= cc.decode(m").
  c1.* [SendBack(m,m')] :- m"=cc.encode(m); _sendBack(m",m').
}
```

In this example, the value returned by different meta-objects is modified. So this integrator modifies the basic plan.

Composition: These modifications are compatible with the other integrators because they don't share any information, but with the notifyCall integrator. Indeed the value returned by the accept meta-object is modified by the encoding service. Do we have to notify the encoded message or not? We have chosen to not automatically consider modification of the basic plan as delegation and then to notify the encoded message. So, the composition will then result in the following rule:

```
c1.* [Accept(m,m')] :- _accept(m,m");
                       (m'= cc.decode(m") // c.notify(m'')).
```

So decode and notification can be done in any order as soon as the treatment of accept is finished. However as a transformation of basic plan is detected, a warning will be generated at merging time. The service provider can then express modification of the basic plan as delegation, rewriting in EncodingIntegrator:

```
c1.* [Accept(m,m')] :- _delegate(_accept(m,m");
                              m'= cc.decode (m")).
```

The composition of "Encoding" and "NotifyCall" integrator will then generate the following rule:

```
c1.* [Accept(m,m')] :- _delegate(_accept(m,m");
                      m'= cc.decode (m")); c.notify(m').
```

And so, the decoded message will be notified.

Projection: Such modification of the basic plan can result in modifying the 'Message' type. In a first time, we limit such changing in subclassing, that models (and in consequences mapping) support.

In this example, the projection will not be possible to java RMI if encoding modifies the marshalling operation [32]. Because in RMI we can't control the accept stage and only the receive one.

4.4 Modification of sequencing between meta-objects

Some services integration need to modify the sequencing between meta-objects. For example, integration of a security service should be defined as forwarding requests to the `SendBack` objects when a message has to be rejected for a security reason. Such an integrator is defined below:

```
Integrator Security (Component c1; SecurityManager
s){
 c1.* [Accept(m, m')] :- _accept(m, m') ;
      if (s.checkSecurity(m'))
        !
      else
        setException(m',m","SecurityException");
```

```
        SendBack(m'',_)
}
```

Composition: As no other integrators have specified modification of message arguments before the end of `accept` (assuming encoding as been defined using delegation), this service integration doesn't conflict with any other integrator.

If the integrator NotifyCall is merged with Security, the call will be notify even in case of security reject, whereas with Notify integrator, there will not be notification in case of reject.

However, we have simplified in this example, the integration of a security service. Another integration of security service would consist in adding security information such as adding the context call, in the message before sending it. In such a case, a conflict with encoding integrator is detected because the same argument is modified in a non deterministic way. Must the encoding of a message be executed before or after adding security properties? The service provider will then have to explicit an order, defining a new integrator coupling these two services modifying in a consistent way the basic plan.

5. Conclusion and perspectives

Standard approaches of service integrations such as the one based on aspects or meta-programming focus on the wrapping of standard object paradigms. They intend to open the functionality of particular languages facilities and deal with class system and none with component platform architecture. So they are language dependent and they don't respond to the need of service integration on multiple platforms.

So, in the spirit of Model driven engineering, we propose to express service integration on component meta-model designing structural and behavioural aspects. Defining service integration consists in annotating it by means of operations supporting commutative and associative composition. The architecture and approach we propose to handle service integration is largely run-time oriented. This approach allows us to gain a certain measure of platform independence.

We have shown how our approach let us describe services integration based on a meta-model. The meta-objects of the meta-model are linked using Prolog like rules that allow extension of the meta-model. Based on this meta-model we define integrators that are set of rewriting rules and operations. Those rules describe what features to add and what behaviours need to be change in order to make the calls to the services or to extend the meta-model to fit their need (for example changing the order of the meta-objects). We also have shown how composition mechanism allows us to compute composition or to detect conflict. Some elements and problems occurring when projecting integrators have also been discussed. At present time, we are using a structural meta-model of each platform to validate the projection of integrators on these different platforms. Next step is generating real code due to code transformation based on relationships expressed between meta-model for service integration and modelling of platform architectures. Use of languages such as typol, or implementation of QVT should be studied.

We described modifications of the meta-model changing the value of arguments or sequencing between meta-objects. Some services integration modify the basic plan in adding new meta-objects such as management of messages queues. We are studying this point to ensure correct composition of such modifications and detect conflicts.

Until now, we based service integration on run time phases and configuration time when preparing components. This is one of the following steps of our research to allow expression of service integration according to deployment and assembling phases.

The meta-model we proposed as example is a minimal extensions to the meta-UML. However the language to define integrators is not based on UML constructs. This should make definition of integrators not as intuitive as stereotypes to UML designers. So, studying how standard constructs of UML could be used to model integrators is therefore one of our next focus.

Bibliography

[1] OMG CORBA Component Model (CCM) Technical White Paper http://www.omg.org/

[2] OMG *Meta-Object Facility Specification 1.3* http://www.omg.org/

[3].OMG *UML 2.0 Superstructure U2 Group 3rd Revised Submission* - 6 January 2003, http://www.omg.org/

[4] Sun Microsystem Entreprise Java Bean Specification http://java.sun.com/products/ejb/

[5] ObjectWeb *Jonas* http://www.objectweb.org/jonas/

[6] Jboss http://www.jboss.org

[7] Microsoft *.net* http://www.microsoft.com/net/

[8] ObjectWeb *Openccm* http://www.objectweb.org/openccm/

[9] MDA. *White paper, Draft 3.2* http://www.omg.org/mda/papers.htm, November 2000

[10]Laurent Berger *Phd Thesis* Rainbow 2001, University of Nice Sophia Antipolis

[11] J. Mc Affer. *Meta-level programming with CodA* In ECOOP'95. SpringerVerlag, August 1995

[12] Kiczales G., Lamping J., *AspectJ Home Page*, http ://aspectj.org/, 2001.

[13] Gregor Kiczales, et al. *D: A Language Framework for Distributed Programming.* Technical report, no. SPL-97-010, 1997

[14] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin. *JAC: A Flexible Framework for AOP in Java.* Reflection'01, Kyoto, Japan.

[11]Douence R., Fradet P., Südholt M., *A framework for the detection and resolution of aspect interactions*, Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, PA, octobre 2002, p. 173–188.

[16] Breton Erwan & Bézivin Jean. *Towards an Understanding of Model Executability*. In Proceedings of the Second International Conference on Formal Ontology in Information Systems (FOIS-2001), Ogunquit, Maine, USA, October 2001

[17] Breton Erwan & Bézivin Jean. *Model Driven Process Engineering*. In Proceedings of the 25th Annual International Computer Software and Application Conference (COMPSAC 2001), Chicago, Illinois, October 2001.

[18] Breton Erwan & Bézivin Jean. *Weaving Definition and Execution Aspects of Process Meta-Models.* In Proceedings of the 35th HICSS-35 Minitrack Software Technology/Integrated Modeling of Distributed Systems and Workflow Applications, Waikoloa, Hawaii, January 2002.

[19] Jean Bézivin, Olivier Gerbé. *Towards a Precise Definition of the OMG/MDA Framework*. ASE'01, Automated Software Engineering, San Diego, USA, November 26-29, 2001.
[20] Siobhán Clarke. *Extending standard UML with model composition semantics*. In Science of Computer Programming, Volume 44, Issue 1, pp. 71-100. Elsevier Science, July 2002.
[21] Junichi Suzuki and Yoshikazu Yamamoto. *Extending UML with Aspects: Aspect Support in the Design Phase.* In Proc. of the 3rd Aspect-Oriented Programming (AOP) Workshop at ECOOP'99 , Springer LNCS 1743, Lisbon, Portugal, June 1999.
[22] S. Clarke, R. J. Walker. *Mapping Composition Patterns to AspectJ and Hyper/J.* ICSE 2001, Workshop on Advanced Separation of Concerns in Software Engineering.
[23] José Luis Herrero, Fernando Sánchez, Fabiola Lucio, Miguel Toro. *Introducing Separation of Aspects at Design Time*. International Workshop on Aspects and Dimensional Computing at ECOOP, 2000
[24] Wai-Ming Ho, Jean-Marc Jézéquel, François Pennaneac'h, Noël Plouzeau. *A toolkit for weaving aspect oriented UML designs.* AOSD 2002: 99-105
[25] Coupaye T., Bruneton E., Stephani J.-B., *The Fractal Composition Frame-work*, Specification, July 2002, The ObjectWeb Consortium. http://fractal.objectweb.org
[26] Riveill M., Bruneton E., *JavaPod: "an Adaptable and Extensible Component Platform*", RM'2000, Workshop on Reflective Middleware, New York, USA, April 2000.
[27] D. D'Souza, A. C. Wills. *Objects, Components and Frameworks with UML. The Catalysis Approach.* Addison-Wesley, 1998
[28]J.L. Herrero, F. Sánchez, F. Lucio, M. Toro. *Introducing Separation ofAspects at Design Time*. In Proc. Aspects and Dimensions of Concerns workshop at ECOOP 2000
[29] W-M Ho, F. Pennaneac'h, J-M Jezequel, N. Plouzeau. *Aspect-Oriented Design with the UML.* In Proc. Multi-Dimensional Separation of Concerns workshop at ICSE 2000.
[30] Jean-Pierre Briot, *Actalk: a Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment,* in proceedings of ECOOP'89, British Computer Society Workshop Series, Cambridge University Press, pages 109-129, July 1989. 19 pages.
[31] Krzysztof Czarnecki and Ulrich W. Eisenecker, *Generative Programming - Methods, Tools, and Applications,* Addison-Wesley, June 2000
[32] C. Nester, M. Philippsen, and B. Haumacher. *A More Efficient RMI for Java.* In ACM 1999 Java Grande Conference, pages 153--159, San Francisco, CA, June 1999.
[33] Dery A.M., Blay-Fornarino , Moisan S. *Distributed access knowledge-based system: Reified Interaction Service for Trace and Control,* 3nd International Symposium on Distributed Object Applications (DOA 2001), Rome, Italy, September 17-20, 2001

A Metamodel of Prototypical Instances

Andy Evans[1], Girish Maskeri[1], Alan Moore[2]
Paul Sammut[1] and James S. Willans[1]

[1] Department of Computer Science, University of York,
Heslington, York, YO10 5DD
{andye, girishmr, pauls, jwillans}@cs.york.ac.uk
[2] ARTiSAN Software Tools Ltd, Stamford House, Regent Street,
Chelthenham, Glos, GL50 1HN
{alan.moore}@artisansw.com

Abstract. The use of prototypical instances is widespread throughout system and software modelling languages. Despite this, there does not exist an approach to metamodelling this mechanism. This is in part due to the fact that metamodels for modelling languages are often based on program languages where prototypical instances are not commonly used. This paper introduces a generic approach to metamodelling the prototypical instance mechanism. The use of this approach is demonstrated by the augmentation of a small static language with this facility.

1 Introduction

The process of designing a software and systems is an exploratory one. A designer rarely has the ability to identify a precise implementation of a complex system without characterising the requirements in other forms. These characterisation might take the form of abstract views of the connection between parts of the systems (component description, for instance), they may also take the form of a very concrete view describing specific instantiations and scenarios (commonly referred to as prototypical instances) of the system (collaboration diagrams, for example). The exploratory style of design is pervasive throughout modelling [1].

In recent years there has been a move towards rigorously underpinning modelling languages, initially with the Unified Modelling Languages (UML) [2] and more recently with the Model-Driven Architecture [3]. These languages are metamodelled so that (ideally) the semantics of the language can be understood from their models. Although MDA is very much work in progress, existing metamodels of UML are heavily influenced by conventional object-oriented programming languages. As with programming languages, these languages have mechanisms that allow abstraction to be dealt with along the lines of packages and classes. However such languages fail to have a generic model for supporting the specification of prototypical instances.

This paper argues that the ability to specify prototypical instances is fundamental to being able to design systems and introduces a mechanism by which they can be metamodelled. In section 2 we briefly examine the philosophical

rooting of prototypical instances in order to motivate their inclusion in metamodels. Section 3 introduces the mechanism for metamodelling prototypical instances and exemplifies its application to a basic language model. In section 4 we discuss the mechanism and section 5 summarises our conclusions.

2 Background

Representation and classification of *things* has been central to philosophical discourse for many thousands of years. The earliest characterisation was given by Plato who made the distinction between the *ideal* characterisation and instances of the *ideal*. Plato's student Aristotle continued exploring this distinction by using the *ideal* as a basis for the taxonomy of natural phenomena such as plants and animals. This distinction can be seen as the basis for class-based programming languages. Within these a programmer characterises the *ideal* as classes which gives rise to a number of instances (objects) [4].

An obvious deficiency in the approach of Plato and Aristotle is that the *ideal* is often very difficult to identify. This was argued in the nineteenth century by W. Whewell and W.S. Jeyons who *.. emphasised that there are no rules to determine the properties to use as a basis of classification of objects. Furthermore, they argued that classification is not a mechanical process, but requires creative invention and evaluation* [4]. Wittgenstein later demonstrated an alternative approach to that of Plato's which involved categorising *things* according to *family resemblance*. Family resemblance acknowledges that there can be no one true characterisation of anything and we can only talk about the relative relationship between different characterisations. The work of Wittgenstein influenced the research of Eleanor Rosch which became the basis for prototype theory [5]. Prototype theory introduces frameworks for working with, and understanding the relationship between characterisations. The classic example within prototype theory is that a sparrow is a fairly prototypical bird, whereas an emu is not.

The difficulty in characterising the *ideal* representation for instances has also been recognised in the design of software and systems [6]. Within these domains a number of formalisms have been developed which enable the modelling of prototypical instances as a step towards identifying the *ideal* (more commonly referred to as type). Examples of such models are illustrated in figure 1. Figure 1 (a) shows a prototypical instance of a component diagram, this style of notation is extensively used within system engineering. The collaboration diagram shown in figure 1 (b) typifies the style of prototypical instances used in software engineering. These types formalisms often have informal underpinnings and the precise nature of prototypical instances has not being metamodelled.

Although modelling languages such as UML have failed to adequately define prototypical instances, this mechanism has been explored in programming languages research. A number of programming languages have been formed around a prototype mechanism such as Self [7], Omega [8] and Kevo. These languages are notable for their lack of class; instead there is only object which can act as the basis for other objects, which act as the basis for other objects .... The

argument for this kind of program language architecture mirrors the one made in this paper for modelling languages and *people seem to be a lot better at dealing with specific examples first and the generalising from them, than they are at absorbing general abstract principles first and later applying them in particular cases.* [4].

3 Prototypical instance metamodel

In this section we present a metamodel for prototypical instances. The first part gives an overview of the mechanism itself (section 3.1) and the second part demonstrates the augmentation of a language using the mechanism (section 3.2). The models defined in the following section are given in terms of a combination of class diagrams and the Object Constraint Language (OCL) [9] this is a common approach to specifying models and is used to specify both the Meta-Object Facility (MOF) [10] and UML [2].

3.1 Mechanism

The approach we take to metamodelling languages broadly follows that used in [11] and includes a model of both the *type* domain (the *ideal*) and the *instance domain* (instances of the *ideal*) as illustrated in figure 2 (a). One treatment of prototypical instances in the type/instance dichotomy is to add a distinct abstraction *prototype* related to type [12]. A deficiency of this approach is that their is always a fixed type, whereas in reality the type will change as more requirements of the systems are discovered and generalised. The approach we have taken, and illustrated in figure 2 (b), is to combine types and prototypical instances via the *role* mechanism. In this a prototypical instance is determined if its *roleOwner* is not self, and a type is determined by *roleOwner* being self. Consequently a type can be turned into a prototypical instance by giving it a new *roleOwner*. This recursive model also enables a rich descriptions where prototypical instances can also have prototypical instances.

In isolation the *role* mechanism is not particularly semantically rich. A prototypical instance is denotated informally but there is no notion of whether a prototypical instance is valid. In the remainder of this section we briefly examine what it means for a *role* to be valid.

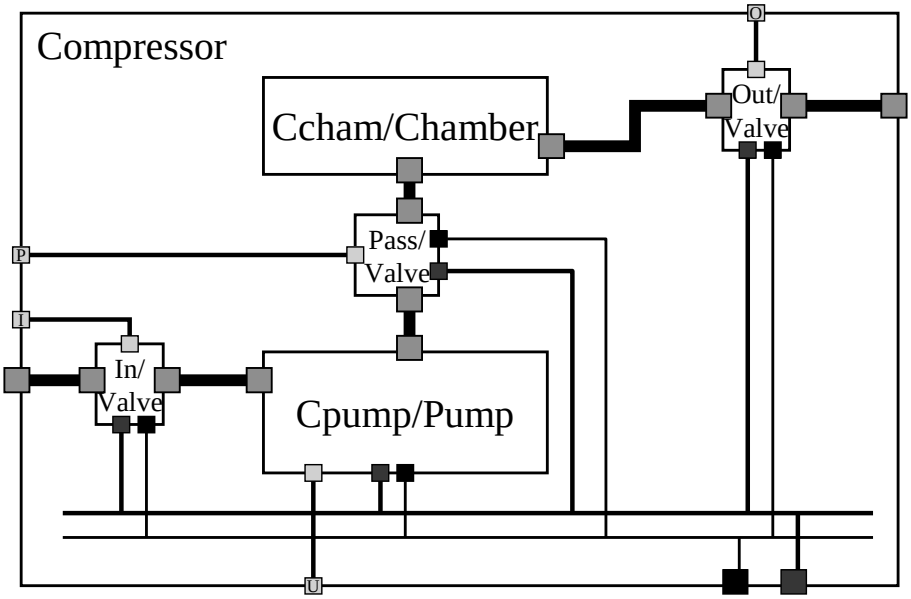
A prototypical instance describes a scenario of its *roleOwner* therefore it gives rise to a subset[3] of the state space (described by instances) of its *roleOwner*. Figure 3 presents a visualisation of this. If we take prototypical instance two (*PI2*) this gives rise to a particular set of instances. For instance this might describe how a vending machine yields a *YummyBar* chocolate. *PI2* is a role of *PI1* therefore *PI1* must realise at least the instances of *PI2* and potentially, but not necessarily, more. *PI1* might describe how a vending machine can yield

[3] Note that this is not a *proper* subset since the prototypical instance may capture the whole state space of its *roleOwner*, indeed this is the case when the *roleOwner* is self.

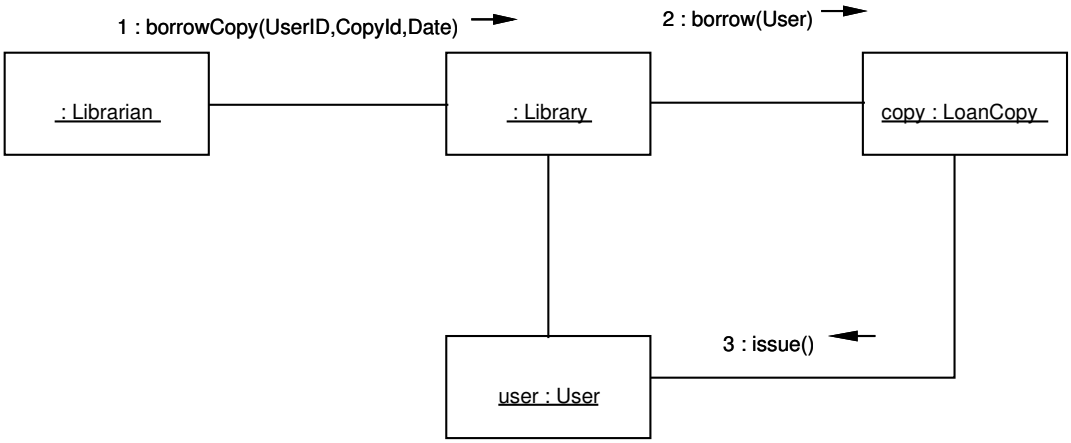
YummyBars and *CocoBars*. *PI1* is a role of a top level *Type* which must yield instances that include all the instances of *PI1*. The top level *Type* in this case might specify a generic chocolate vending machine.

If it was the case that we realised that a generic chocolate vending machine was actually a specific case of a generic vending machine, a new *Type* can be introduced which has as a *role* the original *Type*. Effectively the original type is relegated to being a prototypical instance. Equally, if we decided that we are only interested in vending machines for *YummyBar* chocolate, this can become the new top level type.

The precise nature of the subset relationship is determined by the language abstraction being considered. In the next section we augment a simple language with the *role* mechanism and identify some *role* relationships for abstractions in that language.

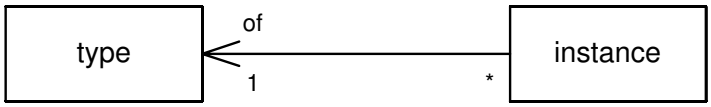


(a) Component scenario

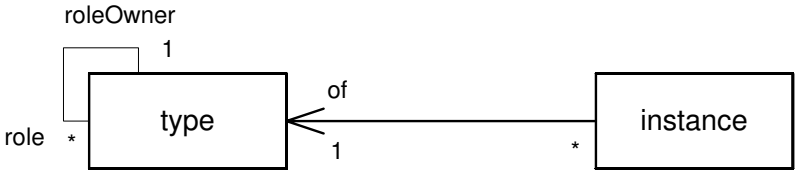


(b) Collaboration scenario

Fig. 1. Examples of prototypical instance models



(a) Type and instance model



(b) Type and instance model augmented with the role mechanism

Fig. 2. Basic model

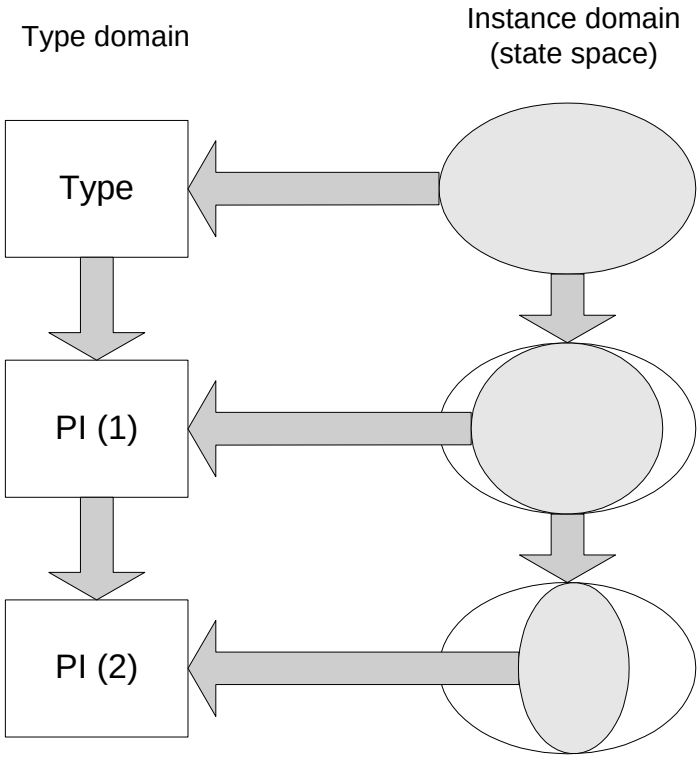


Fig. 3. An overview of the constraint relationship between prototypical instances

3.2 Example

In order to demonstrate the application of the *role* mechanism, we use it to augment the simple static language shown in figure 4. This language describes how classes can own attributes, and how attribute pairs are used to form associations. An important part of applying the *role* mechanism is the identification of the rules that describes when a prototypical instance is valid. The next sub-sections describe the rules for each component of the language.

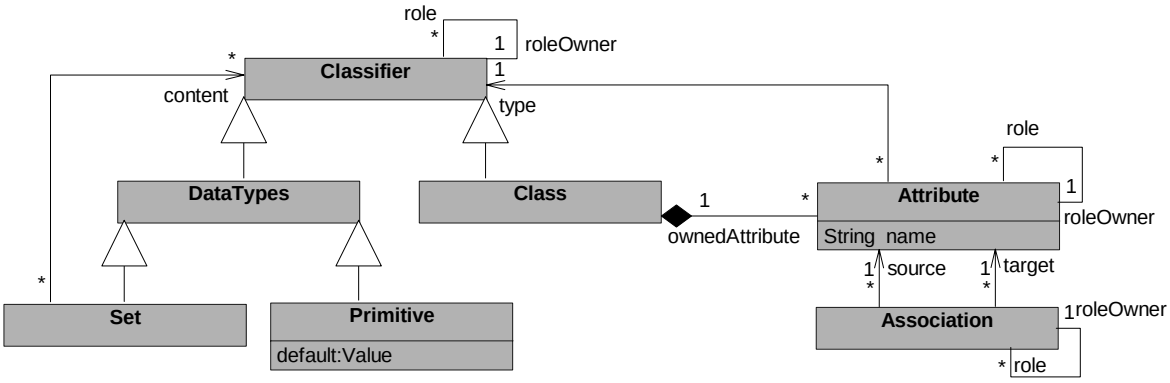


Fig. 4. A simple static language

Data types For the purpose of this example, a naive approach to data types is taken and we do not elaborate primitive types. It is therefore necessary to understand what it means to define a *role* of a *Primitive* type and a *role* of a *Set*. In the case of a *Primitive*, it is possible to say that the prototypical instance of a type can be a value (or a constraint on the range of values the type can assume). Here we take a less complex approach and specify that the prototypical instance must be identical to its *roleOwner* (there is no proper subset):

```
context Primitive
  self.role->forAll(r | r.isKindOf(Primitive) and
    r.default = self.default)
```

A prototypical instance of a *Set* can contain a subset of the elements in the *Set*'s *roleOwner*. The following exemplifies valid *roleOwner*-*role* relationships:

- Set{a,b,c} = Set{a,b,c}
- Set{a,b,c} = Set{a,b}

– Set{Set{a,b},Set{c}} = Set{Set{a,b}}
– Set{a,b,c} = Set{a}

The constraint implementing this rule is described below:

```
context Set inv:
  role->forAll(r |
    r.content->forAll(c |
      self.content->exists(co | co.role->includes(c))))
```

An important aspect of the rules for *role*, such as the one above, is that they force a commuting relationship, between *role*s and *roleOwner*s. The only relationship that can exist between the two models are the *role* associations. This is concretely demonstrated in figure 5 which shows an instantiation of a *Set* and a *role* of the *Set*. The members of *role*'s *Set* must all be *role*s of their owning *Set*s *role* owner.

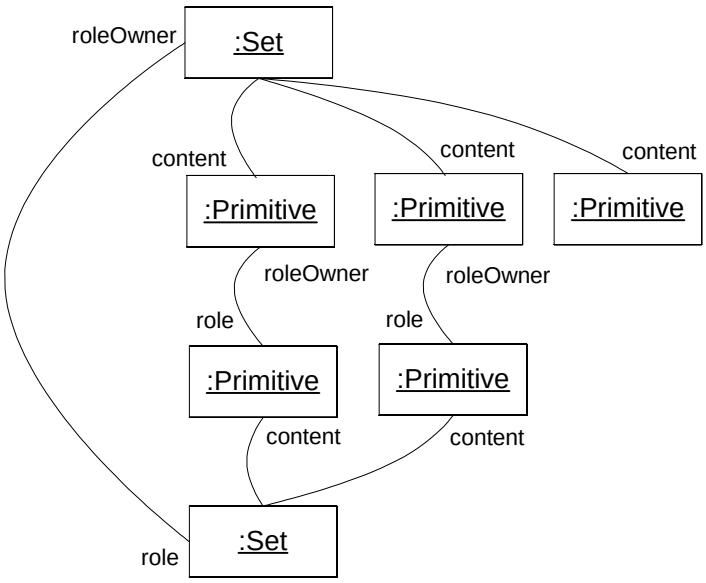


Fig. 5. An example of a valid *role* of a *Set*

Class A prototypical instance of a class may contain prototypical instances of some or all of its *owningRole* attributes, such that the attribute *role*s have identical names to their *roleOwner*, and types which are *role*s of their *roleOwner*'s type:

```
context Class inv:
  role->forAll(r | r.ownedAttribute->forAll(oa |
    self.ownedAttributes->exist(a | a.role = oa
      and oa.name = a.name
        and a.type.roles->exists(t | oa.type = t)))
```

This constraint also ensures that a prototypical instance of a class cannot have additional attributes that are not owned by its *roleOwner*.

Figure 6 illustrates an instantiation of this relationship (omitting types). In this a *mammal* owns an attribute *noOfLegs*, and is a prototypical instance of *animal*. A *fish* has an attribute *noOfGills* and is also a prototypical instance of *animal*. These two prototypical instance determine that *animal* must own attributes which are *roleOwners* of both *noOfLegs* and *noOfGills*.

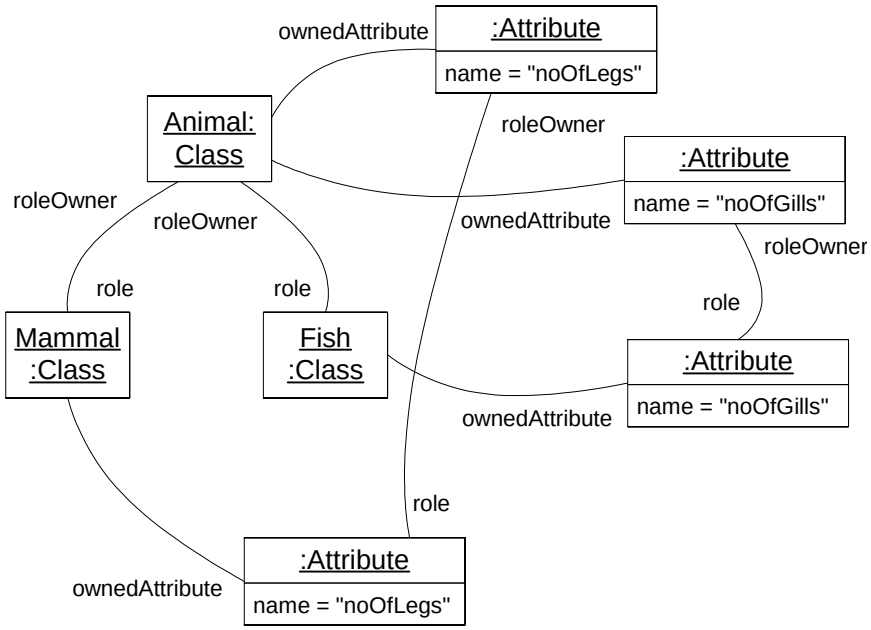


Fig. 6. An example of a valid *role* of a class

Association The straightforward treatment of associations in figure 4 means the *role* mechanism for these is almost completely defined by the *role* mechanism for data types (section 3.2). The only added detail required is to state that the prototypical instances of associations must relate *role*s of their *roleOwner*s attributes:

```
context Association inv:
  role->forAll(r | self.source.role->include(r.source)
    and self.target.role->include(r.target))
```

Figure 7 illustrates a conventional class diagram relating *parent*s to *child*ren, and two prototypical instances of the association. The instance diagram of figure 7 is shown in figure 8. This example demonstrates a typical use of applying the *role* mechanism in the context of associations.

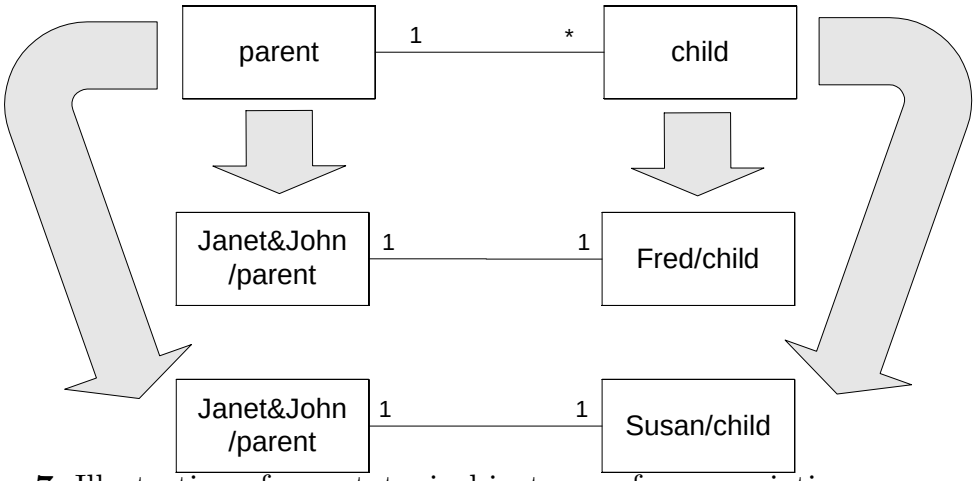


Fig. 7. Illustration of a prototypical instance of an association

4 Discussion

The language described in the previous section is adequate for specifying prototypical instances of static systems. Both configurations described in figure 1 could be translated into our language with relative ease. In the case of the class diagram prototypical instance, this is a one-to-one mapping. The component diagram prototypical instance mapping is achieved by treating components as classes, ports as attributes and connections as associations (along the lines of that suggested in [13]). Note that the translation of connections into associations is only semantically sound because of the simplistic treatment of associations in our model.

By design, the language of the previous section was composed only of static features. However, in order to deal with prototypical instances of languages such as state machines and activity diagrams it is necessary to understand what it means to be a *role* of behavioural features such as an operation. Being able to statically calculate whether one behaviour gives rise to a subset of another behaviour is a less tractable problem than for static language features. The difficulty stems from the often non-deterministic nature of behaviour.

One possible solution to this problem is to define *role*s for static expressions which can then be used to syntactically describe the pre and post conditions of an operation. For example, a pre condition for an operation might be $x > 5$. A prototypically instance of the pre condition might describe $x > 3$ which is clearly a valid *role*. Richer example present more difficult problems and we are continuing to explore this work drawing on extensive research in refinement theory [14].

We have tacitly made the assumption that the types of languages which the prototypical instance mechanism applies, are languages which directly capture

aspects of the software and system. Many modelling languages are also concerned with describing models of the engineering process, for example the mappings between different languages. The Query Views and Transformation proposal is addressing the standardisation of such a mechanism for MDA. Most of the proposals (including our own [15]) suggest that a good approach for dealing with mapping is to use a pattern directed approach. Patterns are effectively prototypical instances and the mechanisms described in this paper could be applied to supporting the specification of patterns.

5 Conclusion

This paper has highlighted the importance of prototypical instances and explored its roots in philosophical discourse. Despite its important, metamodels have failed to provide a treatment of the prototypical instance mechanism. We have presented a generic approach to enabling this style of modelling via the *role* mechanism. The application of the *role* mechanism has been demonstrated by using it to extend a small static language. In order to move beyond static languages we have discussed how it is necessary to explore how *role*s of pre and post conditions, specified as expressions, can be reasoned about.

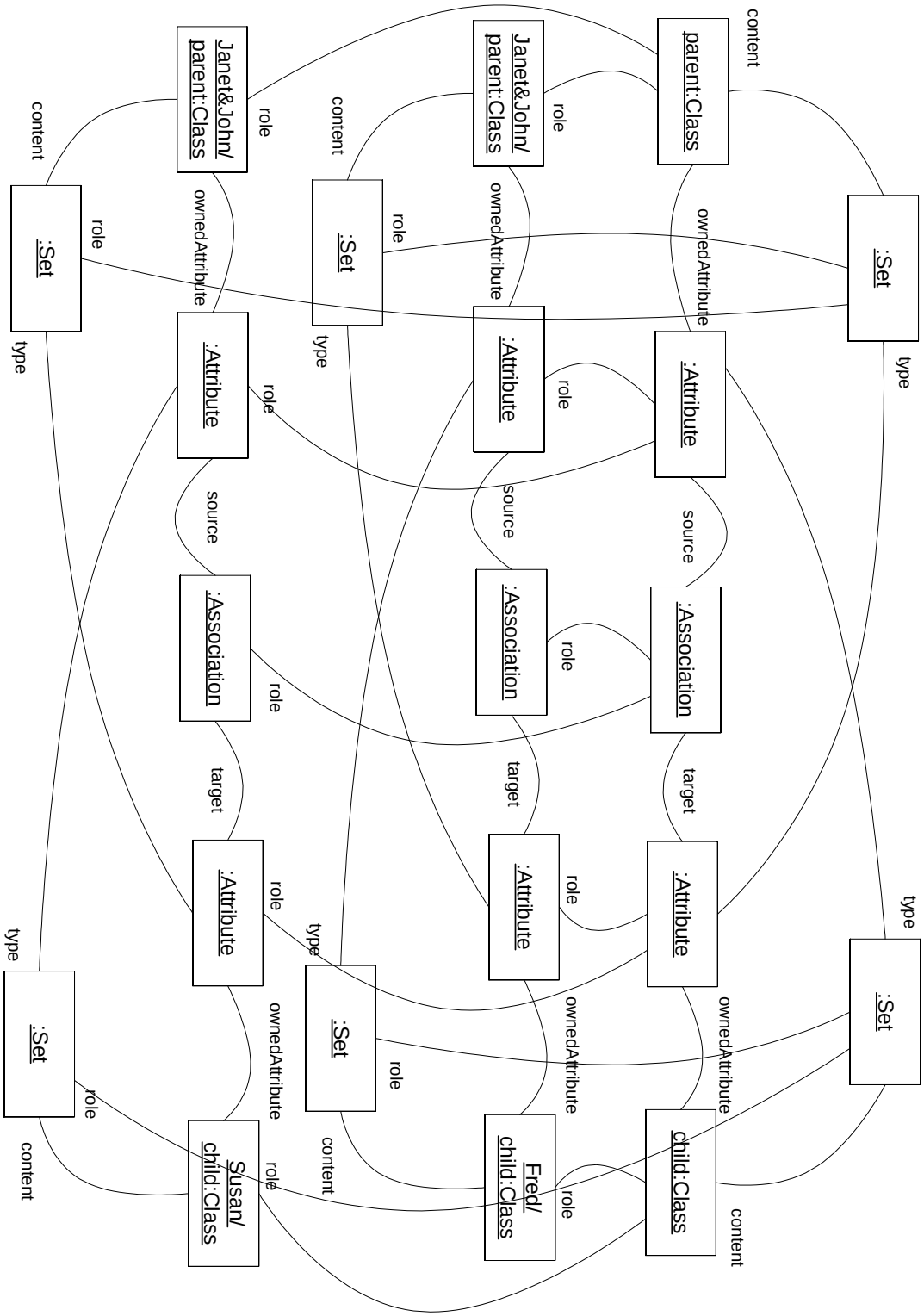


Fig. 8. Instance diagram of figure 7

References

1. D. Budgen, Software Design, 1st Edition, Addison-Wesley, 1994.
2. O. M. Group, Unified modeling language specification version 1.4, http://www.omg.org, ad/01-09-67.
3. D. S. Frankel (Ed.), Model Driven Architecture, 1st Edition, Wiley, 2003.
4. A. Taivalsaari, Class versus prototypes: Some philosophical and historical observations, Journal of Object Oriented Programming (1997) 44–49.
5. G. Lakoff (Ed.), Women, Fire, and Dangerous things: what categories reveal about the mind, 1st Edition, Chicago, 1987.
6. B. B. Kristensen, Object-oriented modeling with roles, in: J. Murphy, B. Stone (Eds.), Proceedings of the 2nd International Conference on Object-Oriented Information Systems, Springer-Verlag, 1996, pp. 57–71.
7. D. Ungar, R. B. Smith, Self: The power of simplicity, SIGPLAN Notices 22 (12).
8. G. Blaschek, Object-Oriented Programming with Prototypes, Springer, 1994.
9. J. B. Warmer, A. G. Kleppe, Object Constraint Language: Precise Modeling with UML, 1st Edition, Addison-Wesley, 1999.
10. O. M. Group, Meta object facility 1.4, http://www.omg.org, ad/02-04-03.
11. 2U, 3rd revised submission to OMG RFP unified modeling language infrastructure version 2.0, http://www.2uworks.org/uml2submission/, ad/03-01-08).
12. B. Pernici, Object Oriented Development, 1989, Ch. Objects with Roles, pp. 75–100.
13. 2U, 2nd revised submission to OMG RFP unified modeling language superstructure version 2.0, http://www.2uworks.org/uml2submission/, ad/02-12-23.
14. C. Morgan, Programming from Specifications, Prentice Hall, 1990.
15. Q. Partners, Initial submission for mof 2.0 query / views / transformations rfp, http://www.qvtp.org, ad/03-03-27.

Metamodelling of Transaction Configurations

Sten Loecher[1] and Heinrich Hussmann[2]

[1] Dresden University of Technology
Department of Computer Science
Sten.Loecher@inf.tu-dresden.de
[2] University of Munich
Institute for Computer Science
Heinrich.Hussmann@informatik.uni-muenchen.de

Abstract. The integration of business logic and infrastructure services is an integral part of the software engineering process in today's component technologies. After investigating the situation regarding transaction management services, we discuss several existing problems and state the requirements for the declarative specification of transaction services. We then argue in favor of a model-based configuration approach for which a conceptual framework is provided based on metamodelling and transformation patterns defined by the Model Driven Architecture (MDA).

1 Introduction

Two primary objectives of software engineering methodology are to provide ways to cope with the inherent complexity of large software systems and concepts to enable better reuse of already existing artifacts. For this purpose, current component technology like Enterprise JavaBeans (EJB) [2] are based on two essential parts. Components are the primary artifacts that contain business logic, whereas the separation of aspects is used to decouple infrastructure services from the former. However, the two aspects have to be integrated eventually. This can be done either programmatically, by incorporating transaction control statements into the source code of the component, or by declarative means, i.e. pre-defined configuration attributes.

The declarative configuration has a number of advantages over the programmatic approach. There is, for example, a clear separation of application-specific and application-independent logic. That allows the independent processing of both aspects by different persons, i.e. domain experts. Moreover, the configuration of transaction services by a pre-defined set of attributes provides a simple, yet precise, and explicit specification of the desired transactional behavior.

However, the declarative approach as currently applied in practice is bound to certain restrictions. These are, for example, narrow configuration capabilities, the late integration of business and application-independent functionality, the poor translation of component-based concepts regarding the application-independent aspect, and the lack of a semantic foundation.

As we learned from software engineering practice, design errors early in the engineering process can result in high lifecycle costs. Therefore, we argue in favor

of a model-based approach for transaction service configuration which has to be accompanied by adequate tool support. We think that metamodelling and MDA provide appropriate means to tackle the subject.

In the remaining sections, a short discussion of the configuration requirements is given and a conceptual framework is sketched.

2 Requirements

To analyze the requirements for the declarative specification respectively configuration of transaction services, two primary domains have been analyzed, namely transaction management concepts and software methodology.

First of all, current component frameworks provide only quite simple transaction management capabilities, i.e. flat transaction models. Experience shows, that the flat transaction model does not fit practical purposes in all cases. This is explained by the long-lived nature of real-live business transactions. It may be argued that today's component technologies are designed for short-lived transactions exclusively. We do not share this point of view, because the component system can be embedded in a context of long-running activities and therefore must be able to deal with issues accordingly. It is therefore likely that future will show the incorporation of advanced transaction concepts into these frameworks. This argument is also justified by research efforts already spent on the subject, e.g. [4]. Consequently, the need will grow for extensive transaction configuration.

Another requirement arises from deliberations regarding the combination of component-based and aspect-oriented concepts. At the moment, both concepts live next to each other rather than being integrated seamlessly. On the one hand, components are required to be delivered with a local specification of all significant properties necessary for a safe deployment into a certain context. On the other hand, current means for the specification of transactional behavior do not allow an appropriate specification of individual components. Therefore, description techniques that respect the principle of locality are still a requirement to be met.

From the preceding consideration also follows the requirement to support the different roles in the software engineering process by adequate and tailored specification capabilities. For example, the component provider does require different specification capabilities than those required by the application developer.

Summarizing the discussion about the requirements we argue that, due to multiple required configuration models and high flexibility, we need elaborated concepts to support the transaction design at several stages in the software engineering process. The next section sketches our vision to serve these requirements.

3 General Approach

We think that metamodelling and MDA provide just the right tools to handle the different requirements stated previously. We therefore developed a conceptual framework that will be explained subsequently.

The framework comprises three core elements, namely a model containing the business logic, models describing configurations, and an integrated model. The model for the business logic is based on the considerations in [3]. We think that the modeling core provided in this study provides just the right starting point for our work. Especially the incorporation of the concept of locality does fit well with our own point of view. Based on the stated requirements, multiple configuration models are conceivable and for some them we are already able to present elaborated solutions. The integrated model is the result of merging and transforming the business logic model and configuration model with respect to the patterns described in [1]. It provides the basis for defining the semantics of configurations and is therefore subject to further analysis and transformation.

All three models discussed so far are platform independent models. Actual configurations for real systems will be the result of according transformations to platform specific models. With this framework, it is not just possible to describe individual configurations but also to compare and to translate different specifications.

4 Conclusions

This position paper states the need for precise and semantically founded description languages to describe transactional behavior respectively configurations for component-based systems. We argued in favor of an approach using metamodelling and patterns defined by the model driven architecture. A general framework has been sketched to illustrate our vision and current work.

References

1. MDA Guide Version 1.0. www.omg.org, May 2003.
2. L. G. DeMichiel, L. U. Yalcinalp, and S. Krishnan, editors. *Enterprise JavaBeans Specification, Version 2.0.* Sun Microsystems, 2001.
3. A. Kleppe and J. Warmer. Unification of Static and Dynamic Semantics of UML: A Study in redefining the Semantics of the UML using the pUML OO Meta Modelling Approach. http://www.klasse.nl/english/uml/uml-semantics.html, 2003.
4. M. Prochazka. *Advanced Transactions in Component-Based Software Architectures.* PhD thesis, Charles University, Faculty of Mathematics and Physiscs, Department of Software Engineering, Prague, 2002.

Invited talk:
Marketing the MDA tool chain

Stephen J. Mellor

Project Technology
steve@projtech.com

A Pattern based model driven approach to model transformations

Biju Appukuttan[1], Tony Clark[2], Sreedhar Reddy[3], Laurence Tratt[4], R. Venkatesh[5]
[1] biju@dcs.kcl.ac.uk, [2]anclark@dcs.kcl.ac.uk,
[3]sreedharr@pune.tcs.co.in, [4]laurie@tratt.net, [5]rvenky@pune.tcs.co.in

[1 3 5] Tata Consultancy Services, Pune, India.
[2 4]Department of Computer Science, King's College London, Strand, London, WC2R 2LS, United Kingdom.

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

[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.

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.

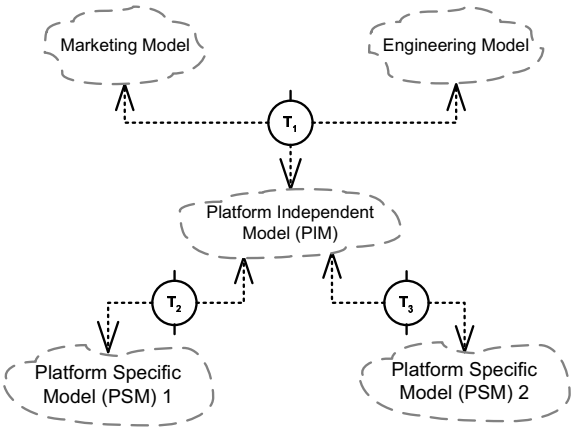


Fig. 1. Transformations and MDA

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.

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. An updated version of this paper based on the revised QVT partners submission is being worked on at the moment.

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.

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

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. Relations are not executable in the sense that they are unable to create or alter a model. They can however check two or more models for consistency against one another Typically relations are used in the specification stages of system development, or for checking the validity of a mapping.

Mappings are transformation implementations. Unlike relations, mappings are potentially uni-directional. Mappings can refine any number of relations, in which case the mapping must be consistent with the relations it refines.

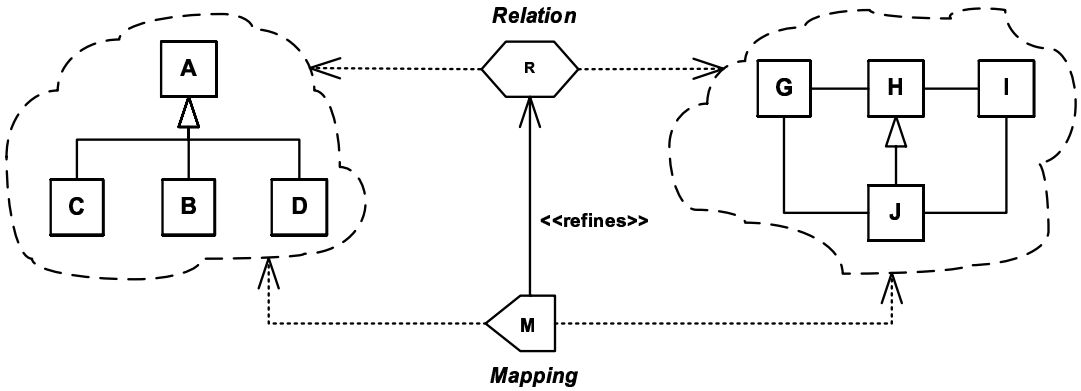


Fig. 2. A high level relation being refined by a directed mapping

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.

As far as possible, the standard languages for relations and mappings share the same syntax and semantics. But, by virtue of the fact that they are different concepts, there are differences between the two. The most obvious difference is that whilst a relation simple consists of a number of domains and an overall constraint, mappings also have an "action body".

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

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

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 Perl-esque 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

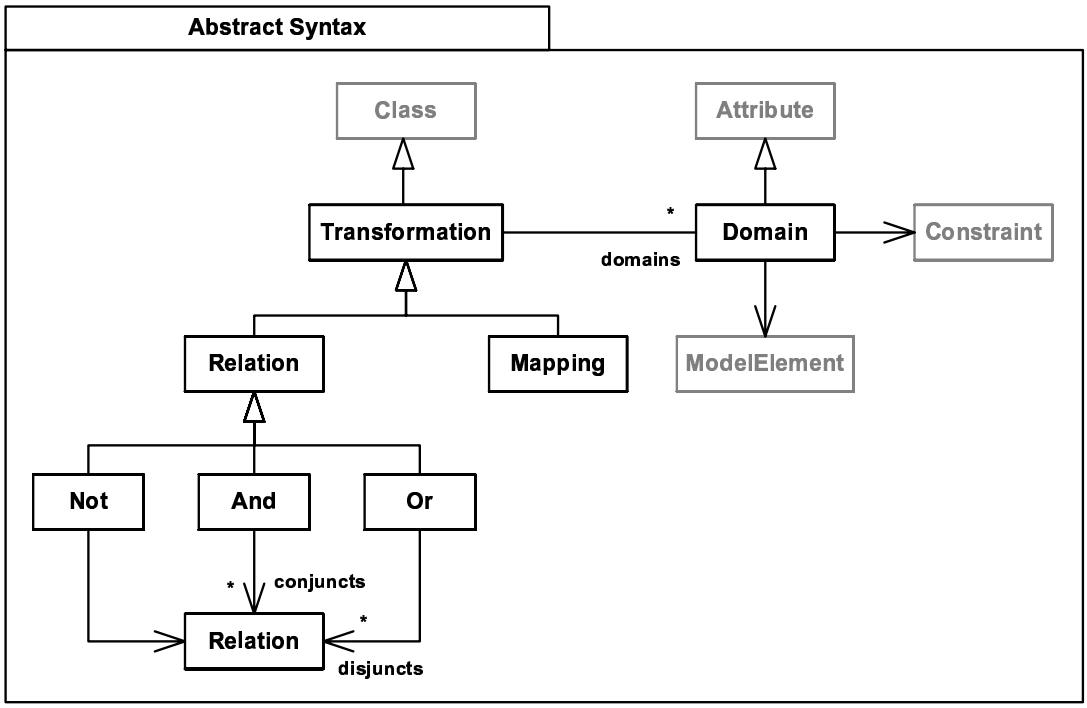


Fig. 4. Infrastructure meta model

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

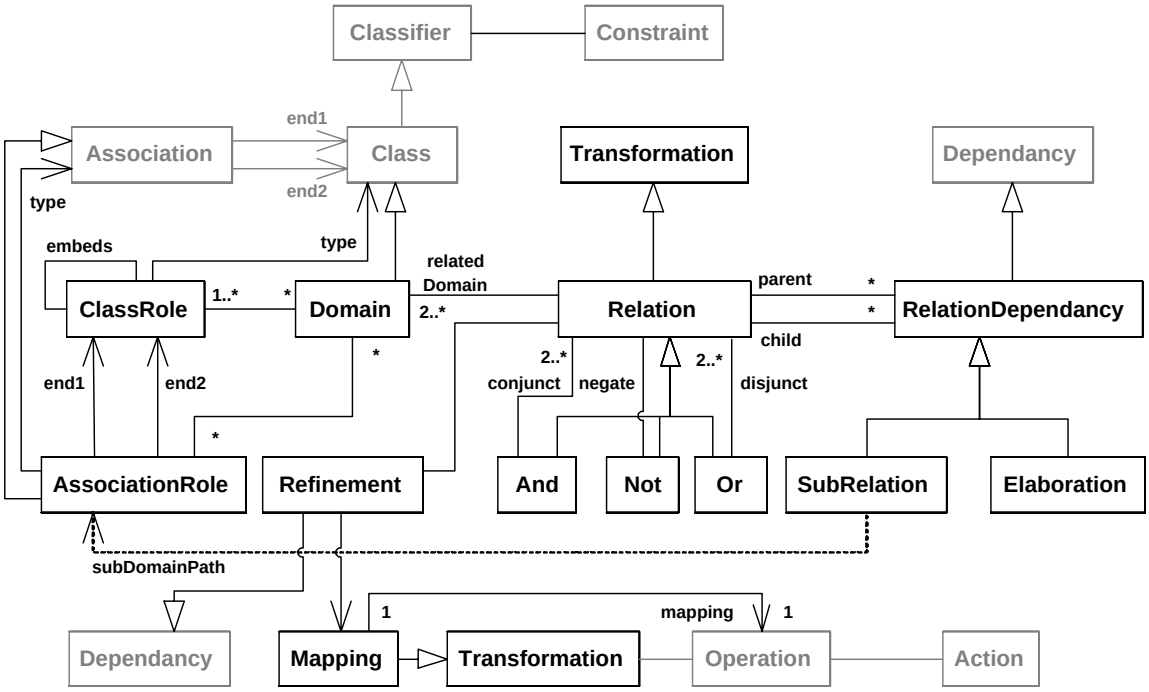


Fig. 5. Superstructure meta model

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.

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, we present an example of transformations between simplified UML models and XML.

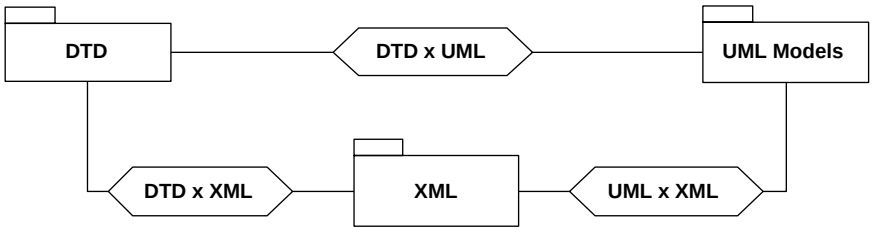


Fig. 7. The overall Transformation framework

Figure 7 shows the overall view of this transformation example. We divide the example into three parts:-

1. The actual transformation of the UML Models to XML represented by *UML x XML*.
2. Verification of the generated XML against the DTD represented by *DTD x XML*.
3. We attempt to capture the relationship between UML Diagrams and DTDs. This is represented by *DTD x UML*.

6.1 The example model

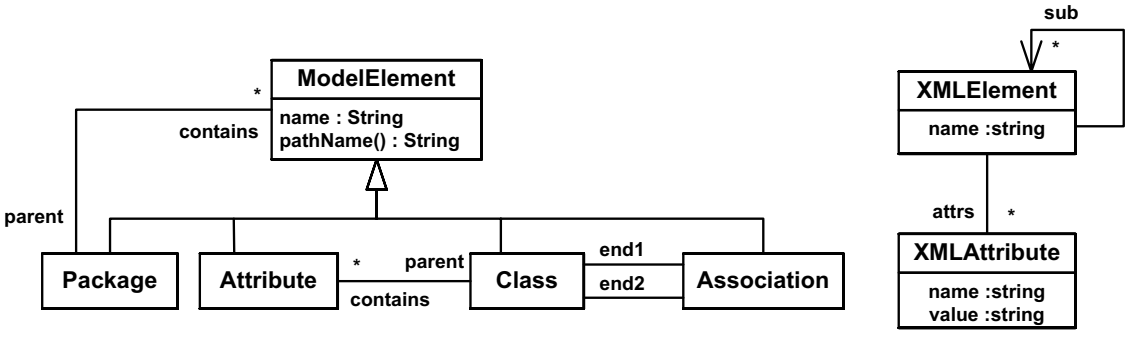


Fig. 8. The example meta-model

Figure 8 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 8 (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

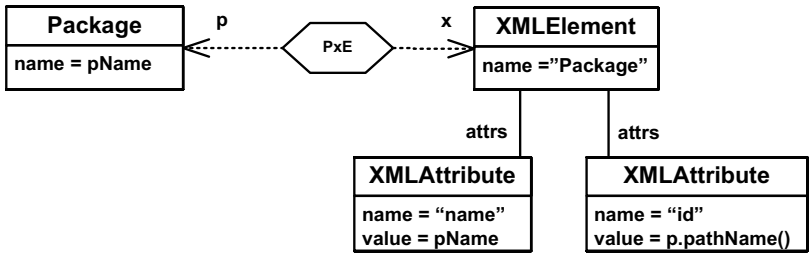


Fig. 9. A UML package to XML relation

Figure 9 shows a relation between the UML `Package` and XML using a pattern language. Although at first glance figure 9 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 9 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.

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`.

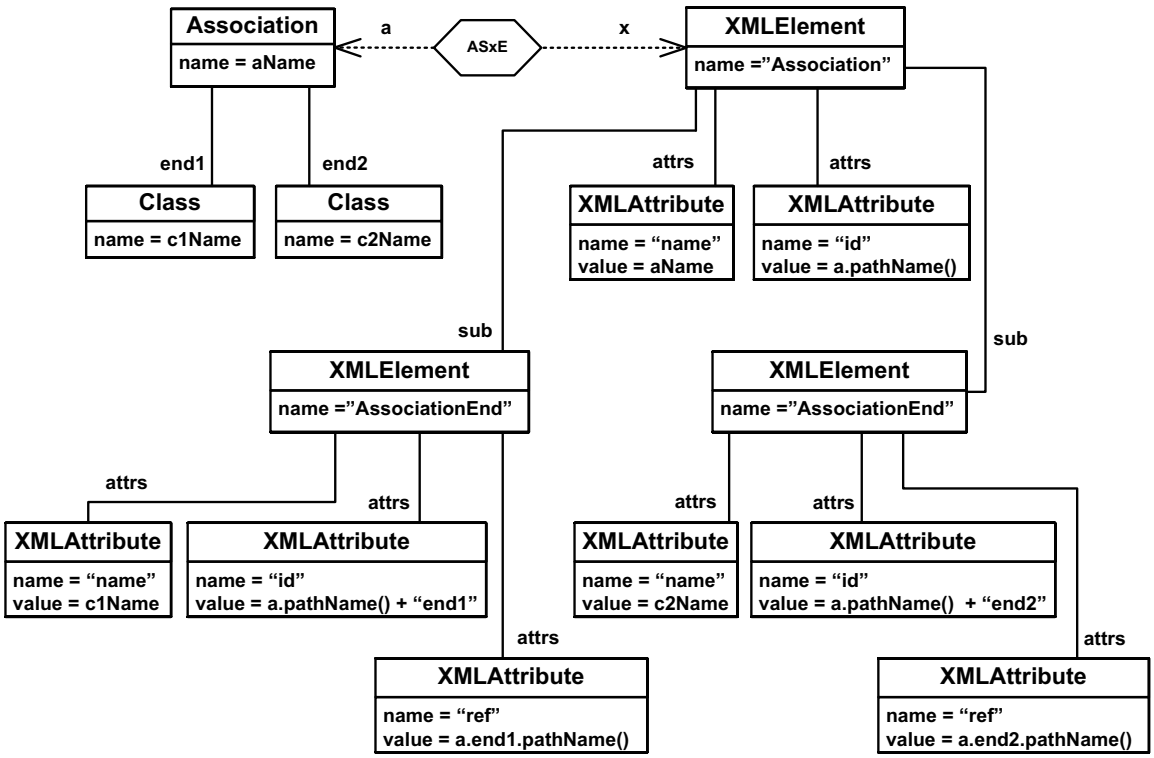


Fig. 10. Transformation of Association

Figure 10 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 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 11 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.

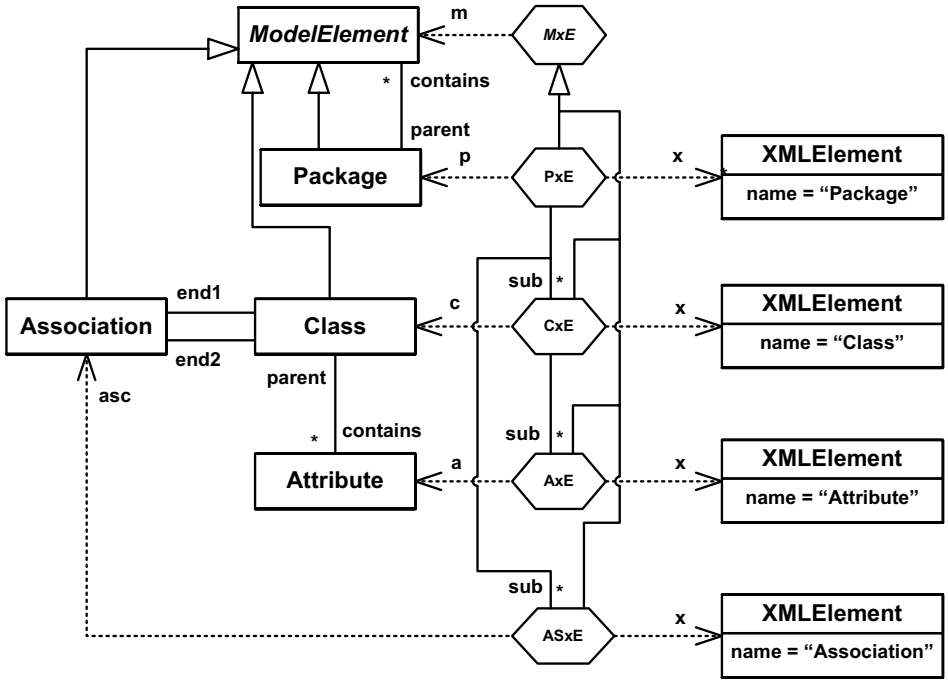


Fig. 11. Transformation composition

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 |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 boilerplate – 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 12 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`.

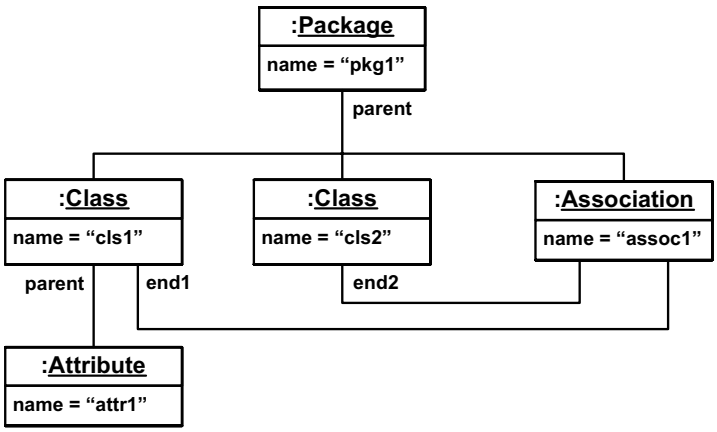


Fig. 12. Object model example to illustrate transformations

Figure 13 shows the complete relations, which combines several of the preceding relations, such as figure 9 and 11, 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>
```

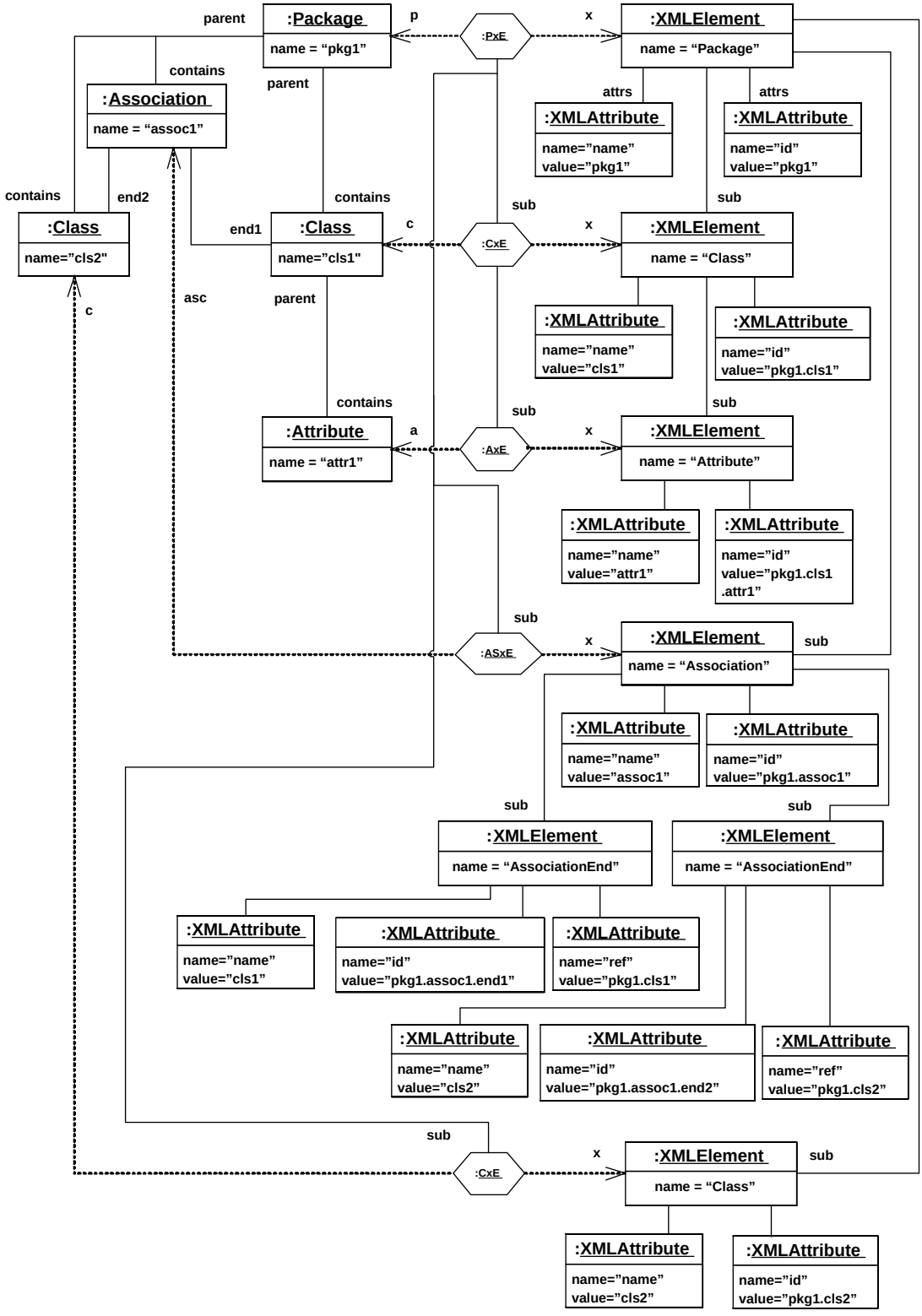


Fig. 13. Complete transformation of the example in figure 12

6.4 Validation of the generated XML against the DTD

A simplified DTD for the Class Diagram in the standard format is shown below:

```
<!DOCTYPE ClassModel [
<!ELEMENT Package
(Package|Class|Association)*>
    <!ATTLIST Package Name CDATA #REQUIRED id ID #REQUIRED>
<!ELEMENT Class (Attribute)*>
    <!ATTLIST Class Name CDATA #REQUIRED id ID #REQUIRED>
<!ELEMENT Attribute>
    <!ATTLIST Attribute Name CDATA #REQUIRED id ID #REQUIRED>
<!ELEMENT Association (AssociationEnd AssociationEnd)>
    <!ATTLIST Association Name CDATA #REQUIRED id ID #REQUIRED>
<!ELEMENT AssociationEnd>
    <!ATTLIST AssociationEnd Name CDATA #REQUIRED
        id ID #REQUIRED ref IDREF #REQUIRED> ]>
```

The DTD (Document Type Definition) represents the skeletal structure of the output generated for an XML representation of the class diagram. Thus, the DTD can be used for type checking of the output XML document to ensure that the generated XML is conforming to the Class Diagram's DTD. This is done by the means of Validation scripts. The scripts can be written in OCL or any other OCL like language.

6.5 Relationship between DTDs and the UML Diagrams

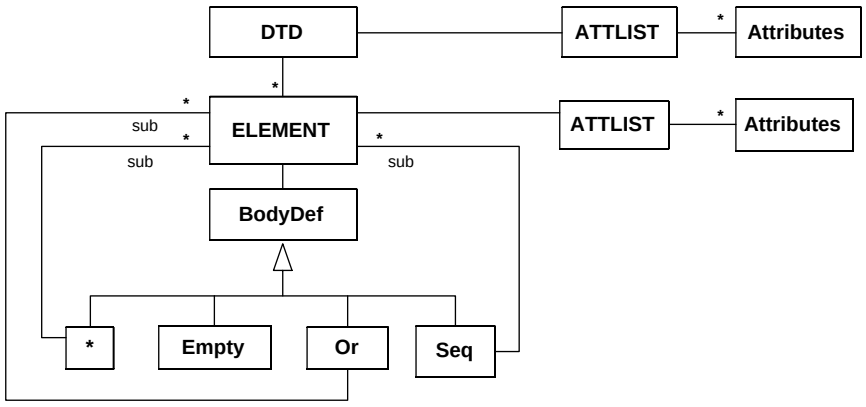


Fig. 14. A Simplified Meta Model for DTDs

Figure 14 shows a simplified meta model for DTDs. For simplicity, we ignore some attributes which are non relevant with this example. Figure 15 shows an instance of the DTD Meta Model shown in figure 14. This is the model representation of the Class Diagram DTD shown in the previous subsection. Herein

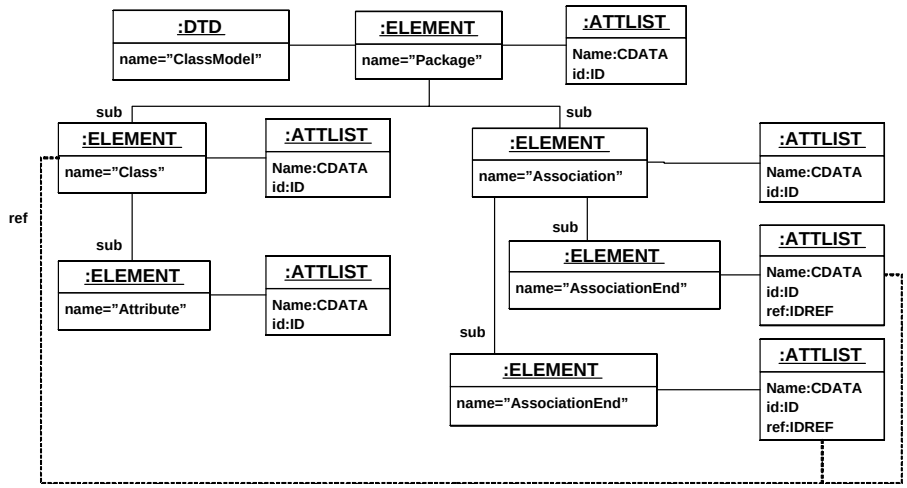


Fig. 15. DTD for Class Diagrams

we define an XML Document named ClassModel having the *ELEMENTS* Package, Class, Attribute, Association and AssociationEnd. Package can contain any number of Elements of type Package, Class and Association. Each Element has two attributes (defined by the *!ATTLIST* keyword) *Name* and *ID* respectively. The Element AssociationEnd has an additional Attribute *ref* which is used to reference the class which forms the AssociationEnd as shown by the dotted association named *ref*.

On close observation of the Class Diagram Models in figure 8 and DTD meta model in figure 14, one comes across a set of relationship between the two models as shown in figure 16. Basically, each Meta object in the class diagram gets transformed into a corresponding ELEMENT in the DTD. The *contains* relation in the Class Diagram becomes the *sub* relation in the DTD. Note the transformation of Associations in figure 16(c). In this case, on transformation, the ATTLIST of AssociationEnds has an additional attribute *ref* which refers to the classes which the association links to.

This gives us a specification which can to used to generate DTDs from Class Diagrams or for that matter, from any UML Model.

6.6 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:
```

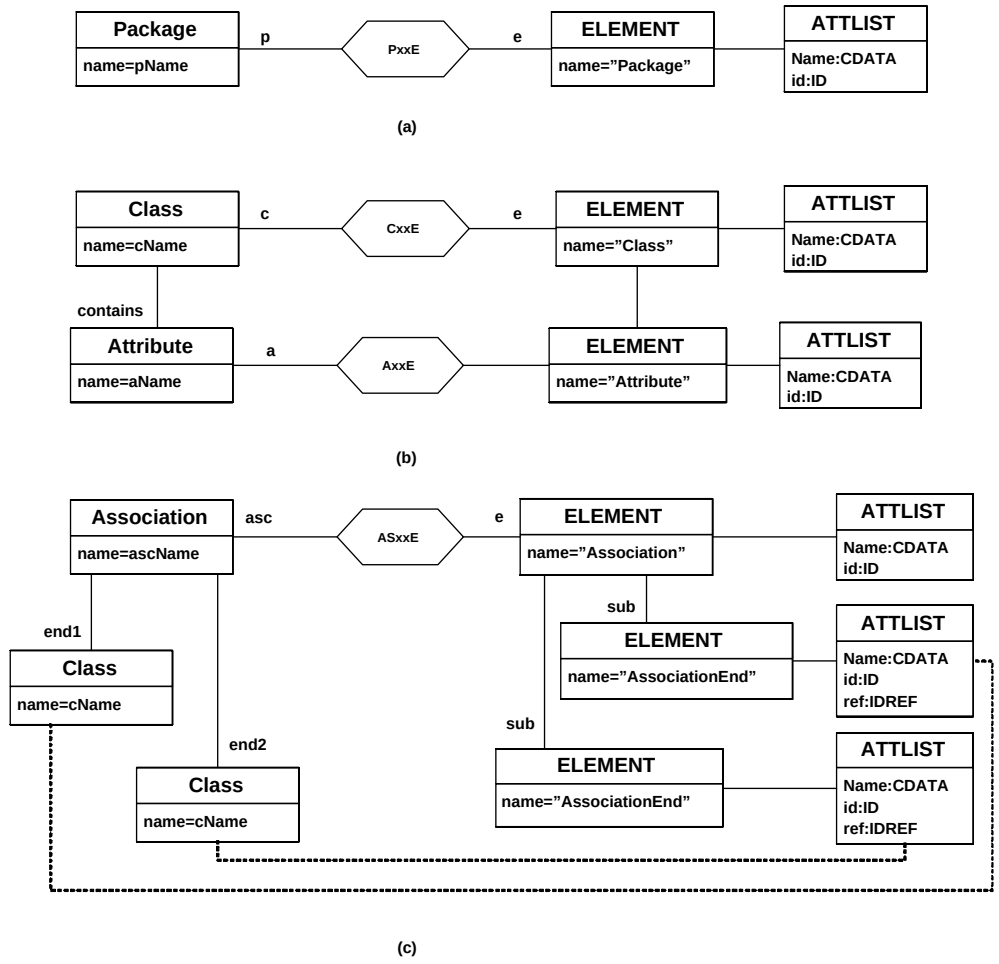


Fig. 16. Transformations of Class Diagrams to DTD

```
        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++.

Another point of advantage is that, since the specification has been defined in terms of relations, it is possible to seek many alternative approaches to implementing this while satisfying the relationship specification. Thus there can be several mapping choices for a given relation spec.

7 Other features

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

7.1 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 11).

7.2 Inter Transformability

Applying the conclusions drawn in subsection 6.5 to the figure 7, it becomes possible that - given any two of the DTD, XML or Class Diagrams, one can verify is correctness with respect to the third. For example, given the Class Diagram and the DTD, it is possible to verify that the XML is representative of the Class Diagram and conforms to the DTD specifications. Or, given the DTD and the XML document, one can verify that the Class Diagram is representative of the XML document.

There is also the possibility of generating the third given any two of them. for example, given the DTD and the XML Document, one can generate the Class diagram using a set of transformation specifications.

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; 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. The authors would also like to thank Dr James Willans and Paul Sammut of University of York and Mr Girish Maskeri of Tata Consultancy Services for their invaluable help with this paper.

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, `http://www.qvtp.org/downloads/1.0/qvtpartners1.0.pdf` (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.

A concrete UML-based graphical transformation syntax : The UML to RDBMS example in UMLX

Edward D. Willink

EdWillink@iee.org
GMT Consortium[1],
www.eclipse.org/gmt
21 October 2003

Abstract.

The increased use of modelling techniques that motivates the Model Driven Architecture requires effective support for model transformation techniques. These are being addressed by the MOF QVT activity with an abstract syntax and/or a concrete textual syntax for transformations. We feel that it should be possible for model driven techniques to be modelled graphically. We therefore present a concrete graphical syntax for a transformation language based on UML and demonstrate the syntax using the working example for MOF QVT submitters.

1 Introduction

The Object Management Group (OMG) has issued a Request For Proposal [14] for a Query / Views / Transformations (QVT) language to exploit the Meta-Object Facility (MOF) [13], which as from version 2.0 should share common core concepts with the Unified Modeling Language (UML) 2.0 [15]. The initial submissions of 8 and revised submissions of 5 consortia have been made, and somewhat surprisingly, only one of them [16] uses a partial graphical representation and another [9] just a graphical context for their language.

This paper describes independent work to provide an Open Source tool to support the OMG's Model Driven Architecture (MDA) initiative [11]. UMLX, a primarily graphical transformation syntax is described that extends UML through the use of a transformation diagram to define how input models are to be transformed into output models. This work has much in common with two of the QVT proposals [10] [16], and it is hoped that it is not too late for some of the ideas in UMLX to influence revised QVT proposals.

We introduce the UMLX graphical syntax in Section 2, in conjunction with the presentation of a UMLX solution to the working problem posed to the MOF QVT submitters; UML to RDBMS transformation. Then in Section 3 we discuss issues not adequately covered as part of the example, before discussing related work in Section 4 and concluding in Section 5.

[1] The author is not directly associated with the OpenQVT consortium of which his day-time employer (Thales Research and Technology Limited, Reading, England) is a part.

2 The UML to RDBMS example

The problem involves conversion of an information model expressed using UML class diagram syntax, and consequently using class and attribute concepts, into a database schema using tables and columns. The incomplete expression of the problem is: "A class maps on to a single table. A class attribute of primitive type maps on to a column of the table. Attributes of a complex type are drilled down to the leaf-level primitive type attributes; each such primitive type attribute maps onto a column of the table. An association maps on to a foreign key of the table corresponding to the source of the association. The foreign key refers to the primary key of the table corresponding to the destination of the association."

[Disclaimer. The author is not an expert in databases, and the UMLX compiler is not yet fully functional, so this example has not been tested. It is hoped that the notation is sufficiently clear for experts to spot errors easily and to correct them with similar ease.]

Information Meta-Models

UMLX operates by establishing a mapping between instances of input and output meta-models, so we must first define the meta-models for the example.

SimpleUML

We use a much-simplified version of the UML meta-model that is sufficient to capture the principles of the transformation.

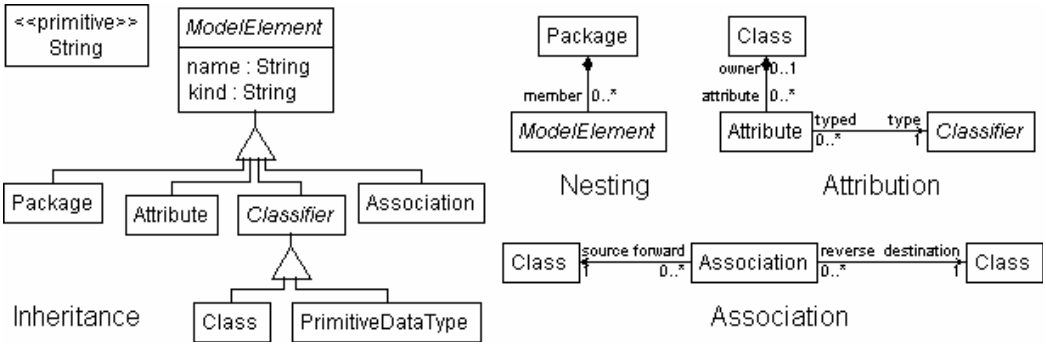


Fig. 1. The SimpleUML meta-model

The built-in `String` is denoted by the `<<primitive>>` stereotype and is used for all attribute values.

In the UML meta-model, all entities inherit from `ModelElement` which provides a `name` and a `kind` attribute. `Package`, `Association` and `Attribute` are directly derived, `Class` and `PrimitiveDataType` are indirectly derived via the abstract `Classifier`.

Overall structure is provided by a `Package` which can contain any `ModelElement` including a `Package` thereby establishing a nesting hierarchy.

`Class`es may have `Attributes` which reference an associated `type`.

`Associations` associate a `source Class` with a `destination Class`.

SimpleRDBMS

We use another simple meta-model for the RDBMS, but re-use the `ModelElement` of the `SimpleUML` model from which the five `DataBase`, `Column`, `ForeignKey`, `Key` and `Table` elements all derive.

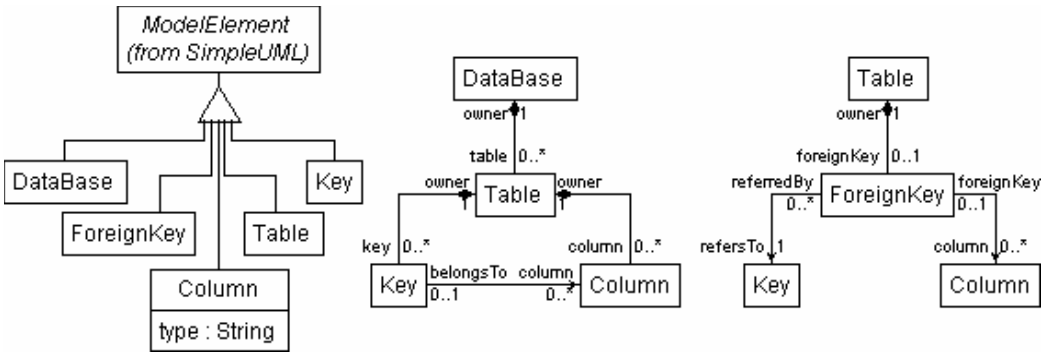
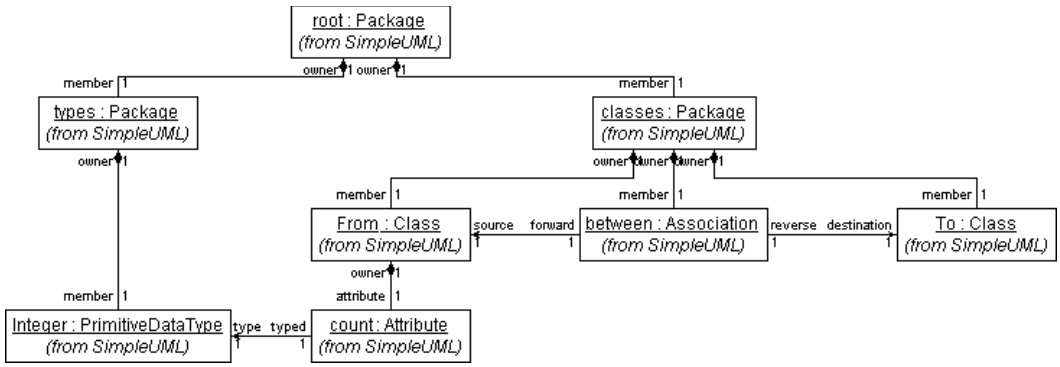


Fig. 2. The Simple RDBMS meta-model

The contents of the `DataBase` is primarily defined by its `Tables` which contain `Columns` and `Keys`. In addition a `ForeignKey` may establish the relationship to the primary key in a different table.

UMLX Concrete Syntax

The UMLX syntax is largely based on the syntax of UML object instance diagrams, with only minor extensions to define transformations. It is therefore helpful to quickly review this less-used variant of class diagram syntax, using an instance of the UML meta-model as an example.



The `root` instance of `Package` contains two nested package instances `root.types` and `root.classes`, of which `root.types` contains the `PrimitiveDataType root.types.Integer`, and `root.classes` contains two `Classes`: `root.classes.From` and `root.classes.To`. The association `root.classes.between` joins them. `root.classes.From` has an attribute `root.classes.From.count` of type `root.types.Integer`.

Note that the instances have their names underlined and that the multiplicities reflect the actual instance multiplicities rather than the formal model multiplicities. It is therefore quite sensible for a `Package` instance to have two children each with the same role name, since each represents a distinct child instance. The instance diagram defines a particular model within the universe of all instance models that comply with the constraints imposed by the class diagram.

Concrete Syntax

UMLX expresses transformations as a translation between a Left Hand Side (LHS) graph representing the input of a transformation and a Right Hand Side (RHS) graph representing the output. LHS and RHS are drawn as UML instance diagrams. UMLX extensions use dashed 'data' flow arcs to declare how the transformation input(s) interface with the LHS, how the LHS relates to the RHS and how the RHS interfaces with the transformation output(s).

Fig. 3. Transformation Input and Output Interfaces

The typically one input and one output are identified by dashed 'data' flow arcs drawn between port icons and instances. The example above associates an input port with external name `from` as an instance of `Class` named `input`, and an output port with external name `to` as an instance of `Class` named `output`. UMLX input models are available for shared reading, but not for writing, hence the input arrow is unidirectional. UMLX output models are available for shared collaborative writing but not reading. The bidirectional arrow indicates the non-exclusive nature of the output instance.

The relationship between LHS and RHS is declared by preservation (keep), evolution (add) or removal (delete) operators.

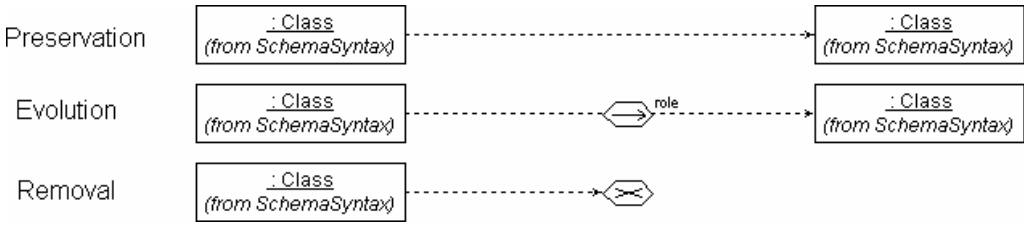


Fig. 4. Transformation Operators

Evolution declares a new RHS instance independent of the LHS. Preservation declares that the LHS composition hierarchy is preserved on the RHS subject to pruning by Removal. More precise definitions of these operations and in particular the `role` are given later.

Transformations are components whose interface is defined by their input and output ports. A transformation instance may therefore be invoked with the LHS providing inputs and the RHS identifying outputs. This is denoted by a lozenge naming the

instantiated transformation and 'data' flow arcs to bind the transformation input and output ports.

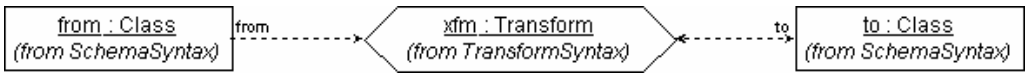


Fig. 5. Transformation Invocation

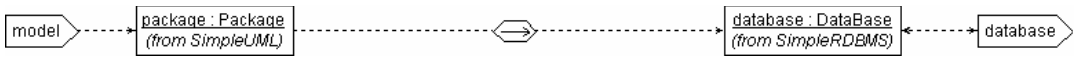
The port names may be placed at the ends of the 'data' flow arcs or on the instance names. A name defined on the arc takes precedence.

Semantics

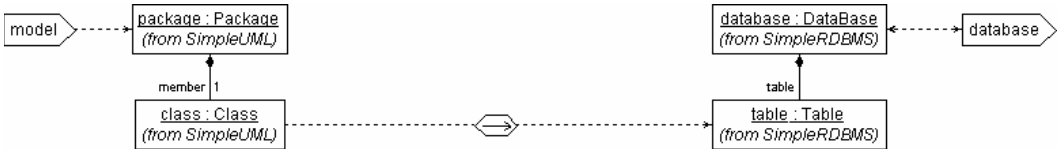
A UML instance diagram defines a particular structure of instances, which we should always refer to as instances. However this can result in rather pedantic descriptions, so we trust that reader will remember that LHS and RHS comprise instances.

The LHS of a UMLX transformation diagram is an instance diagram that defines the context in which the transformation from LHS to RHS occurs. The transformation occurs for each context in which all the constraints imposed by the LHS are fully and maximally satisfied. Standard UML notations are therefore re-interpreted from this perspective.

We will explain this with two trivial examples before proceeding with the real example.



The LHS comprises a single instance `package` which is provided at the `model` input. The `Package` constraint requires that the input be a `Package` (or a type derived from `Package`). The LHS is therefore satisfied whenever an instance of `Package` (but not an `Attribute`) is provided as the input. Whenever the LHS is satisfied, the transformation declares that the RHS and consequently the output is an instance of `DataBase` that evolves from the `package` instance.



The LHS is now slightly more complicated. In addition to a `Package`, a match of the LHS requires each matching `package` instance to have a `Class` `member`. (It is sufficient to specify `member` to identify the `member`/`owner` composition relationship.) If the input `package` instance has three such classes, there are 3 distinct LHS contexts for which all constraints are fully and maximally satisfied, and the transformation is performed three times, once for each context. A distinct `Table` instance is therefore evolved from each `Class` instance, and the evolved tables become part of the shared `database` context.

There is no transformation operator to the `database` output, so there is no declaration defining how the database is created. There is merely the interface constraint that

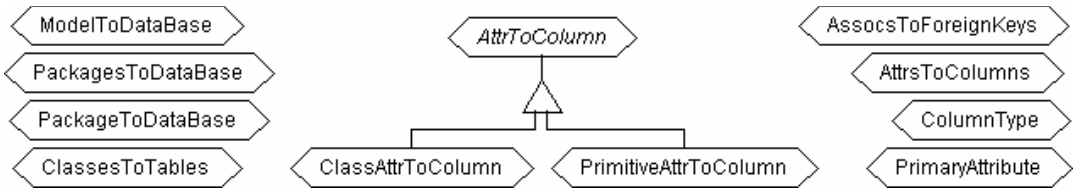
the invoking context must establish a `Database` instance for shared update by this transformation.

Transform Models

Sufficient of the UMLX syntax has now been introduced to be able to explain further issues as they are encountered in the context of the posed example. A full definition of UMLX may be found in [8].

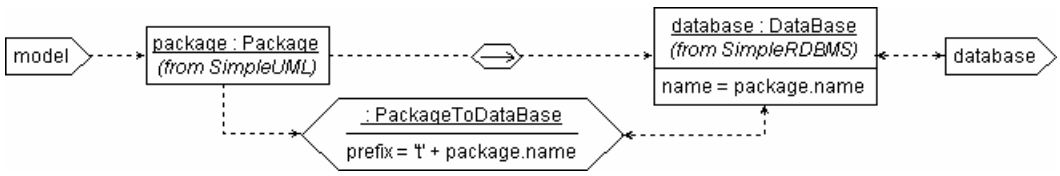
Uml2Rdbms

The `Uml2Rdbms` package comprises 11 transforms, of which two inherit from an abstract transformation. The semantics of transformation inheritance will be discussed when we come to describing those transforms.



Uml2Rdbms.ModelToDataBase

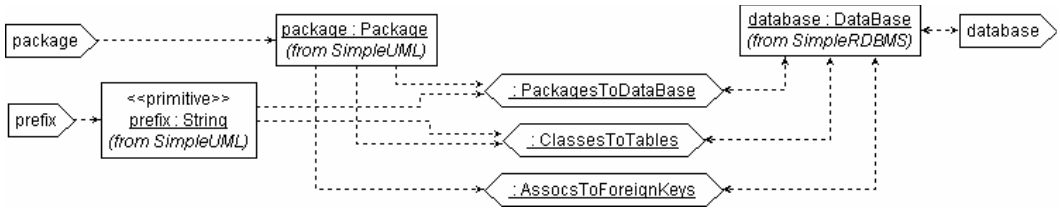
The outer layer of the transformation must establish the external interface and internal context for performing the transformation. The external interface accepts a `Package` at the `model` input port with the external name `model` and internal name `package`, and emits a `DataBase` at the output port with both internal and external name `database`.



The evolution lozenge between `package` and `database` causes a `DataBase` to be created for the incoming `Package`. The embedded OCL expression within the `DataBase` defines the value of `database.name` with the value of `package.name`. Similarly the `prefix` input of the `PackageToDataBase` instance is configured with the value `t_package1`. This will act as the seed for hierarchical table names of the form `t_package1_package2_package3`.

The `package` and `prefix` define the input context and `database` the output context for invocation of the nested transformation `PackageToDataBase`.

Uml2Rdbms.PackageToDataBase



Three distinct transformation activities on a package are expressed by invoking three nested transformations.

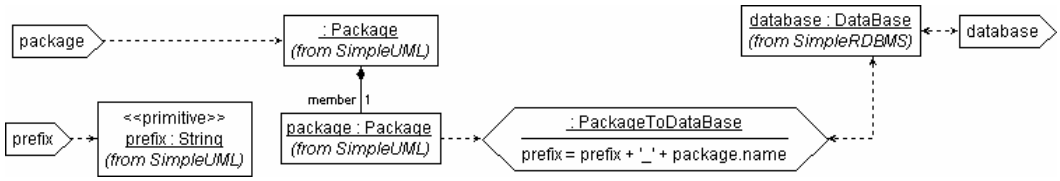
`PackagesToDataBase` is responsible for transforming nested packages to hierarchically named tables.

`ClassesToTables` is responsible for transforming classes (and their attributes) to tables.

`AssocsToForeignKeys` is responsible for transforming associations to foreign keys.

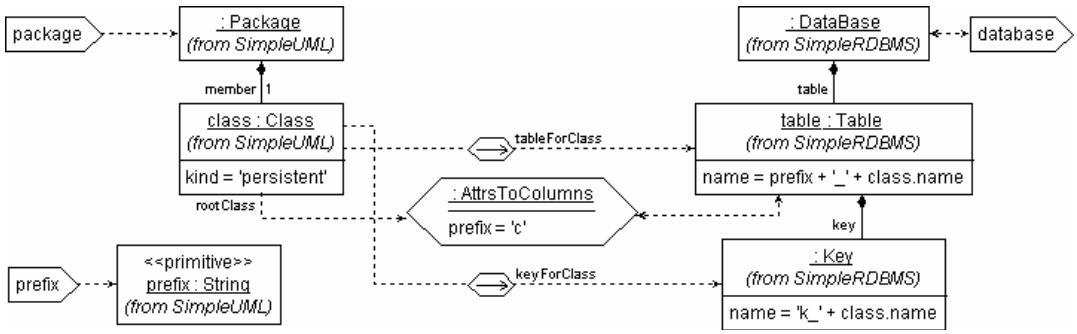
Each of these is invoked by passing the relevant input and output contexts.

Uml2Rdbms.PackagesToDataBase



Transformation of each nested package invokes `PackageToDataBase` recursively for each `Package member` instance of the `package` input. The `prefix` is updated to reflect the extra nesting.

Uml2Rdbms.ClassesToTables

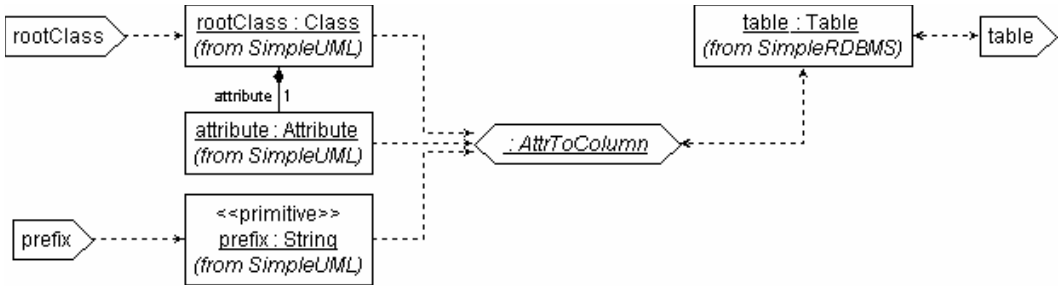


A `Table` and a primary `Key` instance are evolved from each persistent `Class` instance in the `Package` instance. The classes are located as the `members` of the `Package` and are persistent when their `kind` attribute has the value `persistent`. The table has a `name` derived from the package and class hierarchy. The key derives its name from the class alone. Each `class`, `table` pair defines the context for invocation of the nested transformation `AttrsToColumns` which is responsible for

transforming any class attributes into table columns. The column naming is seeded by a `c` prefix in similar fashion to the table naming, so that columns are named `c_attr1_attr2_attr3` as complex attribute types are flattened.

The annotation on the evolutions establishes `{tableForClass:{class}}` as the distinct identity for each `table` instance and `{keyForClass:{class}}` for each `key` instance with respect to the *`class`*. The significance of this will become clear in `AssocsToForeignKeys`.

Uml2Rdbms.AttrsToColumns



Each `Class attribute` needs to be expressed by table columns, but different policies are required for primitive and complex types, so an abstract transformation is applied to each attribute. Pattern matching is a greedy activity applying to everything that can match, and so transformation inheritance is also greedy; every non-abstract leaf[2] transformation is potentially invoked.

Derived transformations may narrow the interface of their base, typically to restrict applicability. And many of these restrictions are amenable to compile-time analysis and so redundant invocation of multiple transforms can be avoided.

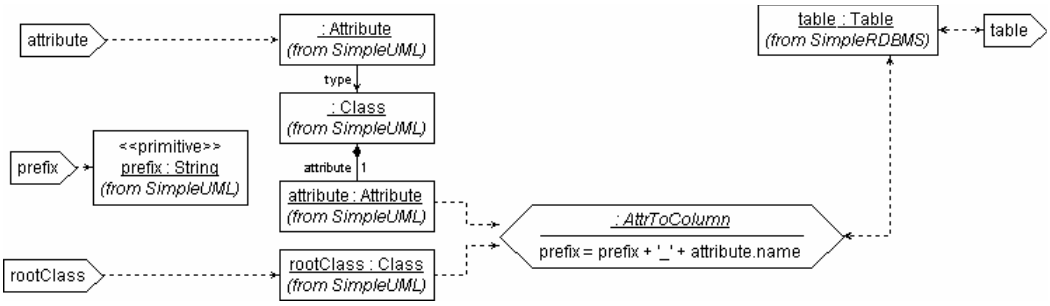
It might seem that this transformation (or `PackagesToDataBase`) is easily folded into its caller; however the unit multiplicity on the `attribute` (or `member`) match would then form a required part of the caller's pattern, and a class without attributes (or a package without nested packages) would then not be transformed.

Uml2Rdbms.AttrToColumn

The abstract transformation defines the interface that may be narrowed by the derived transformations; `ClassAttrToColumn` and `PrimitiveAttrToColumn`.

[2] without further derivation

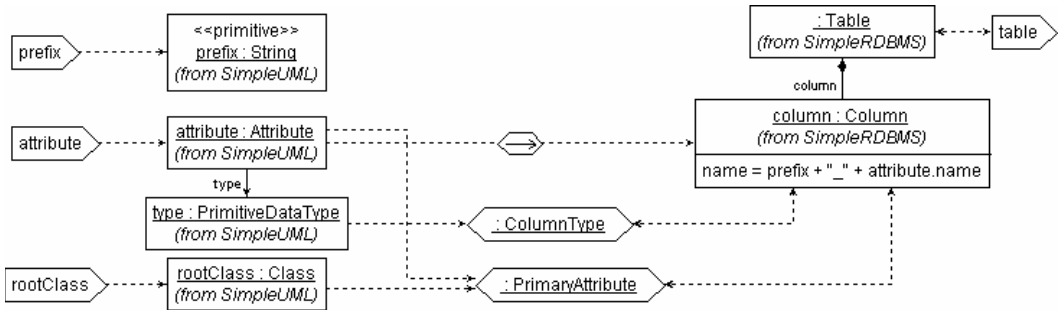
Uml2Rdbms.ClassAttrToColumn



An `attribute` whose `type` is (or is derived from) `Class` invokes a recursion for each `attribute` of the `Class` after extending the naming `prefix`.

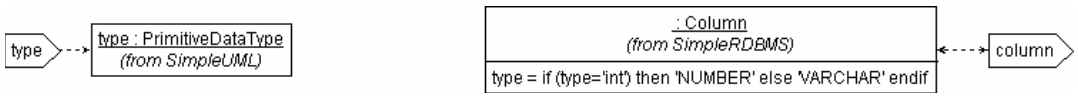
The recursion of `ClassAttrToColumn` therefore drills down through the complex type hierarchy until `PrimitiveAttrToColumn` can resolve the primitive types. Note that the `rootClass` continues to identify the outer class from which the table is being generated.

Uml2Rdbms.PrimitiveAttrToColumn



An `attribute` whose `type` is (or is derived from) `PrimitiveDataType` contributes a `Column` to the table with a hierarchically prefixed `name`. Two further details are resolved by invoking the nested `PrimaryAttribute` and `ColumnType` transformations.

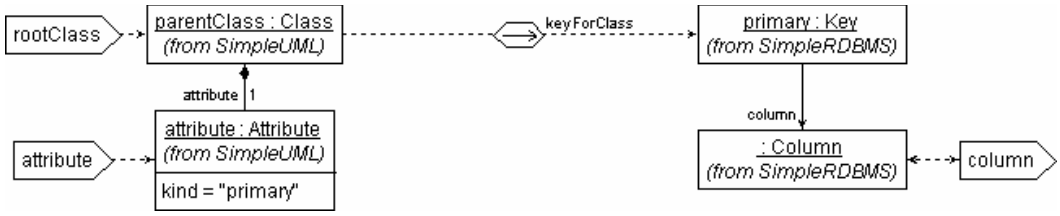
Uml2Rdbms.ColumnType



The column `type` is determined by an OCL expression from the `PrimitiveDataType` class name. (This particular implementation could have been embedded in the invoking transformation, or resolved without resort to an OCL conditional by a pair of further transformations.)

Resolution of the column type is one of the many rough edges in this solution. There is a much more general problem to be solved involving transformation of a specification type system to an implementation type system, which should be defined within the target Platform (Description) Model. The example here just acts as a placeholder for a much more significant transformation suite.

Uml2Rdbms.PrimaryAttribute

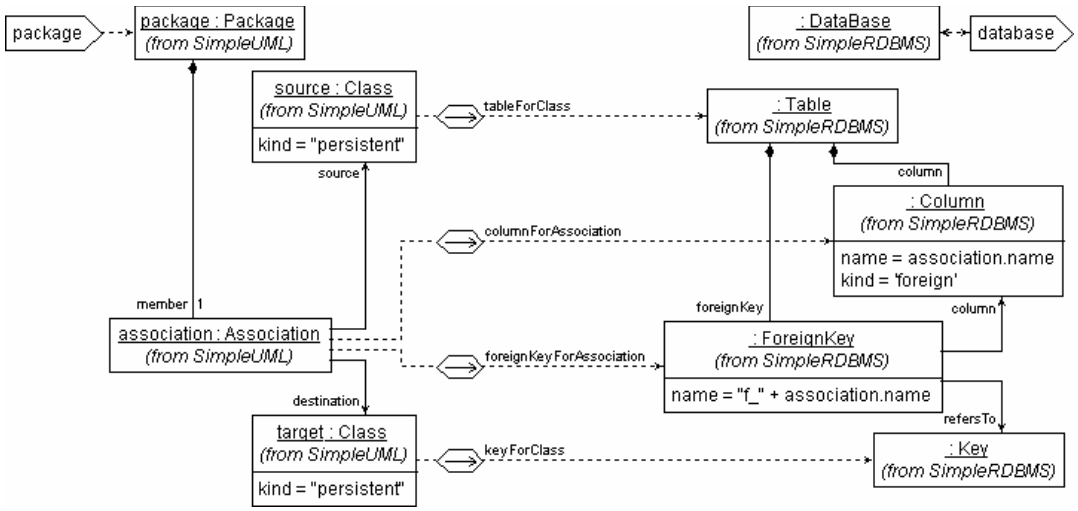


If the `attribute` is annotated as `primary`, the column is incorporated into the primary key, but only if the attribute is an immediate member of its class; attributes of complex types do not contribute to primary keys.

The LHS pattern matches each `attribute` of `parentClass`, but both are bound to inputs, so the LHS matches when the input `attribute` is an `attribute` of `rootClass` which is the class from which the `primary` key originated. The match therefore succeeds only for immediate attributes. For attributes of complex type, `rootClass` and `parentClass` are distinct instances, the composition multiplicity constraint is not satisfied, the LHS does not match and no transformation occurs.

The way in which `keyForClass` ensures that the `primary Key` instance is that created by `ClassesToTables` is described in the next section.

Uml2Rdbms.AssocsToForeignKeys



Finally we have the transformation to create a foreign key instance for each association instance. This transformation occurs 'concurrently' with the class to table transformations and so must collaborate to ensure that each transformation exhibits LHS and RHS consistency: the foreign key instance must refer to the key in the table instance to which the association destination was transformed, rather than some other key or some other table.

LHS Match

The LHS comprises a more interesting structure than the earlier examples. The `Package` context is defined by the invocation. The transformation therefore applies

to each distinct <*association*,*source*,*target*> tuple of instances for which the following constraints are satisfied:

- the *package* instance has base type Package
- the *association* instance has base type Association
- the *source* instance has base type Class
- the *target* instance has base type Class
- the *association* instance is a member of the *package* instance
- the *target* instance is the destination of the *association* instance
- the *source* instance is the source of the *association* instance
- the kind of the *source* instance has the value persistent
- the kind of the *target* instance has the value persistent

The omitted multiplicity on source and destination relationships defaults to unity, which would require only that '*target* is a destination of *association*' however the UML meta-model defines this as an exactly unit multiplicity allowing the stronger test to be made automatically in a practical matching algorithm comprising a one dimensional loop:

```
for each association in package.member
  if association.source.kind is "persistent"
    if association.target.kind is "persistent"
       match found for <association,
              association.source, association.target>
```

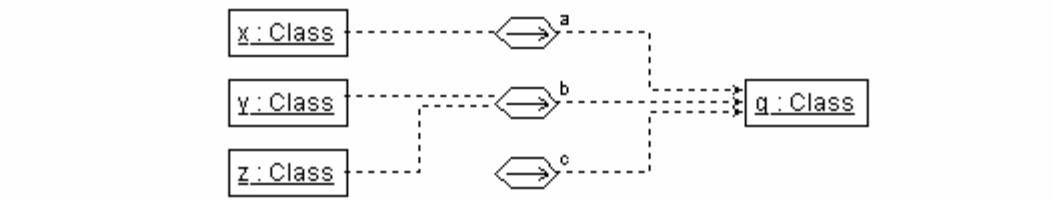
Only four of the nine constraints need validation, since satisfaction of the remaining five is assured for models that conform to the SimpleUML meta-model.

RHS

The RHS defines the required structure to be created for each match of the LHS structure. A ForeignKey uses a Column in the Table for the source Class, and refersTo a Key in the Table for the *destination* Class. The relationship between LHS and RHS instances is expressed by four evolution lozenges from the LHS to RHS.

The consistency problem between the Table instances evolved by the AssocsToForeignKeys and ClassesToTables transformations is resolved by the concept of evolution identity:

Each RHS instance may be the target of zero or more evolution lozenges, each of which may in turn be activated by zero or more LHS instances. In the above examples we have seen only evolution from one LHS to one RHS instance. A more general example is required to fully define the evolution identity concept.



The evolution identity of each RHS instance *q* is established by a two-level signature involving the set of formal names {a,b,c} of each evolution lozenge and the set of

actual LHS instance identities {*x*,*y*,z,#}. # denotes the LHS identity of evolution lozenges without an LHS connection. # is distinct for each transformation invocation and so an evolution from 'nothing' therefore creates a distinct RHS instance for each transformation invocation and corresponds intuitively to a local variable.

The evolution identity for the general example may therefore be written as {a:{*x*},b:{*y*,*z*},c:{#}}. Graphical notations do not readily depict ordering, and so the ordering of evolution terms within the outer {} and of LHS instance identities within the inner {} is not significant.

Returning to the AssocsToForeignKeys transformation, we can now see how {tableForClass:{*source*}} and {keyForClass:{*target*}} align with the {tableForClass:{*class*}} and {keyForClass:{*class*}} identities in ClassesToTables to achieve the required correspondence.

Inheritance

The DSTC solution [10] to this problem observes that a practical UML to RDBMS transformation should handle inheritance. This has been left out of the example in the interests of clarity, brevity and adherence to the set problem. Supporting inheritance would require the SimpleUML meta-model to be extended with a Class to Class inheritance relationship, and an additional recursion from ClassesToTables to traverse the inheritance hierarchy. If occluded attributes are to be mapped to columns, it could be sufficient to just apply another hierarchical naming policy, except that if attributes are visible on more than one inheritance path (C++ virtual base classes) a two-pass algorithm will be required, first to establish the diamond name ambiguities and then to generate the columns once. Alternatively, if occluded attributes are to be suppressed, a work-list of names visible lower in the inheritance hierarchy must propagate with the traversal. UMLX provides solutions to work-lists via the UML multi-object syntax. UMLX provides solutions to multiple passes via sequential composition as explained in the next section.

3 Discussion

UMLX provides a concrete graphical syntax for transformations and consequently has a slightly different perspective from concrete textual syntaxes. It is to be hoped that the move to a QVT standard for transformations can establish an abstract syntax that is compatible with both concrete graphical and textual syntaxes. We will therefore discuss certain aspects of the graphical syntax that may usefully influence an abstract syntax.

Component Composition

The foregoing examples demonstrate the encapsulation of transformations as components supporting hierarchical parallel (concurrent) composition.

The execution context of a transformation component is defined by the models provided at its inputs and expected at its outputs. The input and output ports define

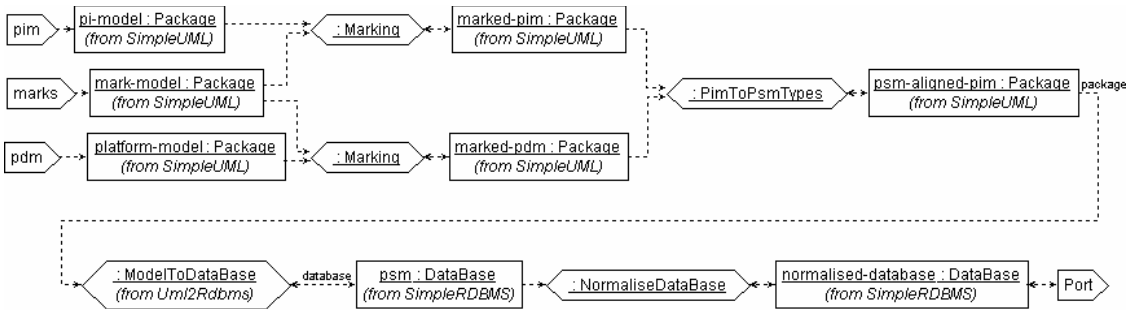
the types of these models. The input models are available for shared reading by many concurrent transformations; the input models may not be written. The output models are available for shared writing by concurrent non-conflicting transformations; the output models are correlated by evolution identities and may not be read. Unsafe concurrent transformations are avoided by a prohibition on transformations with write conflicts and consequently an execution order dependency.

The use of explicit input and output contexts in UMLX differs from many other QVT proposals where the entire input model is available for unconstrained matching. Failure to constrain the input context is adequate for a single transformation, but severely limits scalability to more substantial examples. Even in simple examples, it requires each sub-transformation to fully identify its applicability, whereas the examples presented earlier were able to exploit the context of a parent transformation within its children. The use of explicit inputs enables more than one input to be used thereby enabling transformations to be applied as a mapping between two or more models. This is essential for a suite of MDA transformations from Platform Independent Models to Platform Specific Models, responding to Platform (Description) Models and Mark Models [11].

The example problem is a simple one-model to one-model transformation, so it has only been necessary to use a second input for a trivial string model for a name prefix.

Parallel composition has been demonstrated in order to decompose the example into more manageable sub-transformations. Recursive parallel transformation has enabled hierarchical problems to be readily resolved.

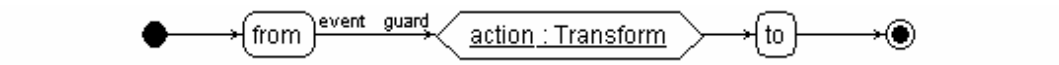
The simplicity of the example avoids the need for a sequential composition. However, as soon we try to exploit the `ModelToDataBase` transformation in an MDA context, it can be seen as just one pass in a multi-pass transform.



We start with PIM, Platform and Mark Models, merge the marks with the re-usable input models, and then invoke `PimToPsmTypes` to resolve the PIM types in terms of the database types defined in the PDM. This eliminates the very inadequate `Uml2Rdbms.ColumnType`. After the `DataBase` has been created, it may be optimised by `NormaliseDataBase`.

Sequential composition is just a generalisation of the one pass LHS, RHS concepts to support multiple passes each with an LHS and RHS. Passes execute in sequence as a consequence of 'data' flow causality; transformations cannot start until all their immutable source models are available.

This simple definition establishes a transformation as a transaction between the state comprising the input models and the state comprising the output models, and so a transformation can be seen as a more powerful way to define a state transition.



Preservation

The UML to RDBMS example is a total model translation, and so it makes extensive use of the Evolution operator to create new RHS instances.

In multi-pass transformations, it is generally undesirable to require a distinct meta-model for each pass; it is often appropriate for the RHS meta-model to be a sub-set or super-set of one or more of the LHS meta-models. One transformation pass may serve to normalise unduly complex elements of the source model, or annotate source elements in an analysis pass prior to a computation pass. In either case, a substantial portion of the LHS instance context needs to be preserved in the RHS context.

UMLX therefore provides the alternate Preservation operator to create a deep copy of an LHS instance, where 'deep' refers to the hierarchy established by composition relationships in the meta-model.

There is an apparent conflict between a need to apply transformations to update a model, and the UMLX prohibition on modifying an input model. However, an update can be realised by creating a copy of the input model in which the required changes are made, and then using the copy as a replacement for the input. This declarative model for creation of the update ensures that the copy can be realised efficiently as an in-place update without dependency on update order.

Removal

Preservation of a complete child composition hierarchy is useful for transformations that perform total preservation with annotation, but excessive for transformations that do refinement. UMLX therefore provides a Removal operator, which may be applied to a LHS instance to suppress its participation in the deep copy for a preservation.

Explicit Preservation and Removal operators may define the behaviour of salient LHS instances. The operators apply implicitly throughout the child composition hierarchy. A model refinement may therefore be specified by Preservation at the root of a model, Removals at unwanted instances and Evolutions for additional instances.

There is an implicit Removal at the root of each input model in order to satisfy the intuition that the RHS is created rather than updated.

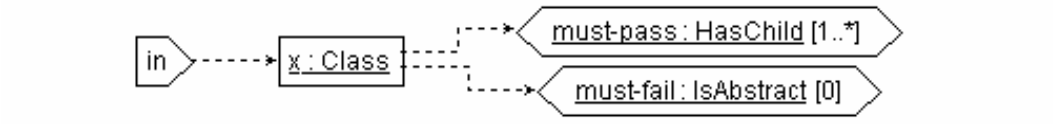
Multiplicity

The earlier examples have only made use of unit relationship multiplicity, in which either the only possible resolution of the relationship constitutes the match, or each possible resolution constitutes a distinct match.

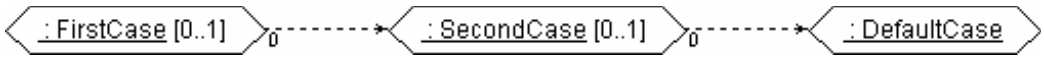
Other values of relationship multiplicity are also useful.

A zero multiplicity requires an absence of matches and so can be used to test for an inverse condition.

In conjunction with multi-objects, a multiplicity of greater than one can match multiple objects. The most useful cases are an unbounded multiplicity, such as `0..*`, which selects all available matching objects, or `*..*` which further requires that all possible objects match. A more explicit multiplicity such as `2` is useful in the limited context where it really is desirable to explore each combination of two out of perhaps five candidate matches.



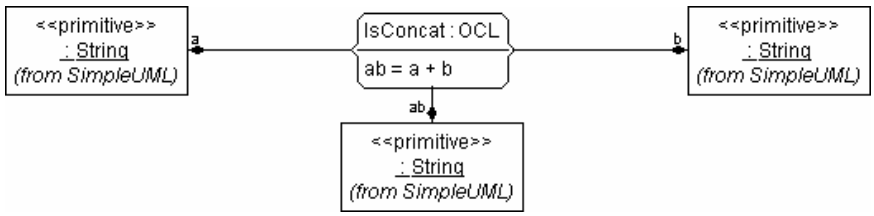
The earlier examples have also only used the default `0..*` multiplicity for invoked transformations. Again other values are useful, in particular a `0` or a `1..*` multiplicity, on a transformation with only an LHS, enables parts of a complex LHS to be factored out using predicate sub-transformations or helpers.



A cascade of transformations can impose limits on the permitted number of matches found within a transformation, and may use a zero multiplicity guard to establish an if-then-else tree, in which subsequent transformations are enabled by 0 matches of a predecessor.

Constraints

The apparent simplity of UMLX lies in its ability to re-use standard UML notation to express constraints in a compact graphical form. This is successful for many of the commonest requirements, but cannot be achieved for all possible constraints, at least not without going far past the threshold at which a textual syntax is clearly superior to a graphical one.



UMLX therefore provides a slight elaboration on a UML constraint that allows arbitrary OCL expressions to constrain a LHS match.

Graph Theory

The concepts of Evolution, Preservation and Removal operators correspond to Add, Keep and Delete in Graph Theory, and since the operators are drawn directly from

LHS to RHS, there is no need for an interface diagram or for instance labels to correlate left, interface and right diagrams. The topological equivalence means that Graph Theory has many interesting results that may be used to prove the soundness of transformation compositions and in particular the validity of optimisations that eliminate common matches (amalgamation) or exploit parallel and/or sequential independence to create composite or distributed transformations. However, it is slightly difficult to categorise UMLX as either Single or Double Push Out (SPO[4] or DPO[5]). The very diverse constraints, arising from UML type and multiplicity syntax, establish gluing conditions that suggest an equivalence to SPO, but the prohibition on execution order dependence and the introduction of an evolution identity, to establish identities for concurrent transformations, provides some of the stronger properties of DPO.

It is certainly possible to define UMLX transformations that do not correspond to DPO, but it is also possible to define transformations that do, and such transformations are reversible. UMLX therefore has a forward arrow within the evolution lozenge with a view to supporting a bidirectional arrow as a checkable assertion of reversibility.

UMLX does not treat nodes and arcs equivalently. Evolution, Preservation and Removal define node behaviour. Arc behaviour is parasitic. Arcs are preserved wherever the pair of nodes at their ends are preserved. Arcs are evolved for each arc explicitly drawn on the RHS. All other arcs are removed; in particular all associations to a preserved composition hierarchy are eliminated rather than left dangling. Where this would violate a multiplicity constraint in a meta-model, the illegal transformation should be detected by the transformation compiler.

UMLX support

An editing environment for UMLX has been configured by exploiting the meta-modelling capabilities of the GME [7] tool from the ISIS team at Vanderbilt.

A compiler and an execution engine for UMLX are being developed as part of the Generative Modelling Transformer project hosted at www.eclipse.org/gmt. This compiler is designed using UMLX, initially as a graphical pseudo-code for transformations manually implemented in NiceXSL[3], a user-friendly front end for XSLT[17]. Compilation is already sufficient to provide useful syntax validation for LHS and RHS compliance with their meta-models and transformation compliance with interfaces. Compilation and execution are currently sufficient to compile and execute model transformations involving concurrent and sequential transformation hierarchies. Once non-trivial OCL evaluation is in place, it should then be possible to reclassify the graphical pseudo-code as the primary source code. Work may then begin on transform optimisation and direct code generation to portable environments such as Java or efficient environments such as C.

It is hoped that this will provide an Open Source framework with which arbitrary XMI [12] to XMI or text transformations can be supported and upon which libraries of MDA transformations can be developed, ultimately providing support for transformation from domain-specific languages in front of MDA and flexible code generation and synthesis following MDA. Such an environment should allow researchers to concentrate on augmenting a particular stage (aspect) of the transformation chain,

[3] Available from http://www.gigascale.org/caltrop/NiceXSL.

without having to replicate or compete with other transformations elsewhere in the chain.

4 Related Work

Gerber et al [6] have experimented with a variety of different transformation languages, and while favouring XSLT, they clearly have their reservations as their code became unreadable. Their experiences have influenced their QVT proposal [10], 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 UMLX example presented here is based primarily on their equivalent solution.

The QVT partners' submission [16] draws an interesting distinction between bi-directional mappings and uni-directional transformations. Their LHS and RHS graphics is similar to UMLX, but without the multiplicities, and they rely on text to define the relationship between LHS and RHS.

The XMOF [9] solution to this example makes use of graphics, but only to establish a context for some nicely symmetrical textual assertions. However the use of graphics to define the coordination of contributions detracts from the modularity and consequent re-usability of those contributions.

The ISIS group at Vanderbilt has pursued the concepts of meta-modelling through the GME tool [7]. A preliminary paper on a Graphical Rewrite Engine [1] inspired the development of UMLX. The evolution to GReAT is described in [2] together with a good discussion on the significance of multiplicities in a UML context. GReAT lacks a clear distinction between LHS and RHS and introduces a non-UML graphical control syntax for imperative sequencing of transformations, whereas UMLX extends UML concepts to achieve almost the same degree of control declaratively.

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.

5 Conclusions

A concrete graphical transformation syntax for UMLX has been presented that makes very substantial re-use of existing UML syntax.

A solution has been presented to the working example from the MOF QVT mailing list. The solution presented is complete with respect to the posed problem, but has many rough edges with respect to a comprehensive solution. By using standard UML notations for typical constructions and constraints, the solution requires very little information in text fields and no non-graphical text. This contrasts with some of the 'graphical' QVT perspectives where over half of the information is non-graphical.

The example demonstrates how encapsulation of a transformation as a component supports parallel and sequential composition, recursion and how multiple inputs and

outputs can be accommodated as will be required to support PIM to PDM mapping for MDA.

The close relationship with Graph Theory has been identified as a profitable source for provably sound transformation optimisations.

6 References

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

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

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

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

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

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

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

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

[9] Compuware Corporation, Sun Microsystems, "XMOF Queries, Views and Transformations on Models using MOF, OCL and Patterns", OMG Document ad/2003-03-24.

[10] 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.

[11] OMG, "MDA Guide Version 1.0.1", OMG Document omg/2003-06-01, http://www.omg.org/cgi-bin/doc?omg/2003-06-01.

[12] 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

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

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

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

[16] 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.

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

Metamodeling Relations - Relating metamodels*

Jan Hendrik Hausmann

University of Paderborn, Computer Science Institute
Paderborn, Germany
hausmann@upb.de

Abstract. Formalizing the relation between the different artifacts in a software development is the basis for transformations and consistency-maintenance. The Model Driven Architecture (MDA) proposes this kind of automated support for a multi-model approach to software development, but a technique for specifying these relations is not yet established. In this paper, we define an extension of a metamodeling language for specifying mappings between metamodels. The language allows mappings to be expressed, independent of transformation direction and platform-specific implementations, and supports partial definitions. A concrete, visual syntax for this language has been previously proposed. This paper focuses on its metamodel definition.

Keywords: Formal semantics, meta modeling, UML extensions, relations

1 Introduction

Modeling has become an integral part of the development of software systems, and the Unified Modeling Language (UML)[Obj01b] has become the standard language to express most of these models. The prevalent problem of this model-based approach to software development is that it is hard to keep all the different models and other artifacts synchronised. On the code level, frequent builds and releases guarantee that all parts of the system work together. Version management tools keep track of the different development versions. For models this kind of support is not yet available. Since many models (of different kinds) can be used in the development of a single system, developers are faced with the problems that transformations between the models have to be performed manually, inconsistencies between models remain undetected, and the relations between the models on different levels of abstraction or from different development stages and the current code build are quickly lost. The code is regarded as the only consistent system representation and all further development of the system will thus be based upon the code. Valuable information contained in the models will get lost and will eventually have to be recreated by re-engineering efforts.

The Model Driven Architecture (MDA) [Obj01a] is a proposal by the OMG to improve this situation. MDA focuses on the question of how to employ UML

* This work was supported by the Segravis EU Research and Training Network

models in a system development. It proposes to use different models (PIM and PSMs) at different stages of the development and to employ automated transformations between them. Finally, the resulting code should be automatically generated from a suitably refined model. This approach requires that the implicit 'glue' between the different models used in the development is explicitly defined. Using explicit relationships between the different models and the code, transformations can be performed that generate new artifacts or propagate changes throughout the models. The vision of MDA is to integrate the models in a way that they form a consistent part of the system description and can thus be reused for system evolution (e.g. changes to the requirements, or a transition to a new deployment technology). The notion of mappings is "one of the key features of the whole approach" [Obj01a], but a standard technique has not yet been established. The OMG has issued a Call for Proposals on this topic called "Queries, Views and Transformations" [Obj02]. Initial submissions to this call have been received. Our approach is a further contribution to this debate.

In this paper, we propose a declarative way of defining model relations based upon concepts presented in [AK02] (see section 2 for details). We regard all models created in the course of software development as expressions in (possibly different) modeling languages which in turn are defined using a suitable metamodeling language such as the OMG's Meta Object Facility (MOF)[Obj01b]. Even program code can be seen as a special case of a model since the abstract syntax tree of a programming language can be expressed as a metamodel. Our idea is to introduce special constructs into the metamodeling language that connect related elements from different metamodels. Special predefined features and the ability to specify OCL constraints provide rich facilities to express properties of this relation. A visual notation for the specification as well as for the instance level of these relations has been introduced in [HK03], see Sect. 3 for a summary. In this paper, we concentrate on the definition of abstract syntax (Sect. 4) and semantics (Sect. 5) of relations, which we also express as metamodels. By integrating these new concepts into MOF we provide the facilities for defining mappings between different languages (e.g. between a UML PSM model and Java code or between a UML analysis and a UML design model). The metamodel definition employs the constructs we are defining; thus we have a true metamodeling approach as the core of the language is used to define itself. Comparisons to related approaches and some remarks on implementing and extending the concepts presented here can be found in the concluding section.

2 Concepts

Relations and transformations are closely connected issues. Relations define which pairs of models belong together or conform to some notion of compatibility. Transformations are employed to construct a target model from a given source model in a way that source and target model are compatible according to the relation defined. We can distinguish between forward and reverse transfor-

mations and between complete transformations and incremental reconciliation operations. All of these transformations are needed in the context of MDA.

A basic difference in tackling this problem is whether to take a descriptive approach and specify the relations or to take an operational approach and specify the transformations. The latter has the obvious advantage that it can be implemented (if it is not tool based in the first place) and can thus be efficiently employed to help practitioners. Examples for such transformations can be found in code-generating and re-engineering tools, in model transformation approaches based upon algorithmic, or upon rule-based structures.

Yet, we believe that our descriptive approach is able to provide many advantages unattainable by transformation approaches: A relation is *abstract* and thus universal. It is neither tailored toward a special direction of transformation nor toward full or incremental transformations. All of those transformations can (in principle) be derived from the same relation. This derivation is not always possible because a relation definition can be *partial*, i.e. given one side of the relation it may not be possible to construct a corresponding element on the other side uniquely. Handling this kind of loose relations is an important feature as software development is a partially creative process which may be structured but not completely formalized.

Relations are *local*. While they may match structures instead of single elements, they are still applicable in a local context and do thus enable reconciliation. Transformation-oriented approaches often use complex control structures and global optimizations to ensure efficiency. This prevents reversing the transformation direction or employing the same mechanism for reconciliation. Rule-based approaches are better suited in this regard.

Relations are *visual*. There is a great need for people to understand structures and connections on a level that is beyond the details provided by code. This has given rise to the UML and its predecessors. In the bigger context of visualizing the connections between models this kind of notation is not yet available. Anecdotally, lots of colored arrows in presentations indicate the need of people to visualize this kind of connection. A model-based approach to the definition of relations can provide this kind of visualization.

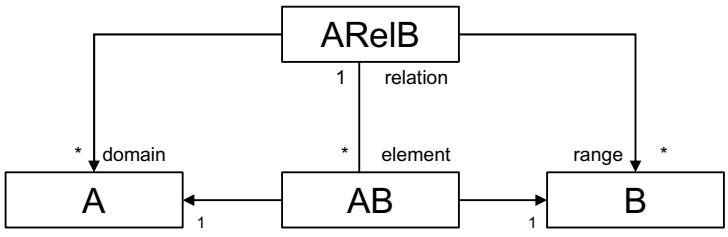


Fig. 1. Pattern for modeling Relations

In [AK02] the basic concepts for modeling relations in UML have been laid out. There, a pattern has been introduced (see Fig. 1) that separates the defin-

ing relation and the pairs of elements it contains. Properties of the relation and its pairs can be defined by specifying OCL constraints. Special emphasis has been put on the need for nested relations. These are what really distinguishes the approach. It is a common case that models comprise nested structures (e.g. compositions). When mapping those nested structures, a completeness criterion with respect to the container is needed. Consider the example in Fig. 2. If we

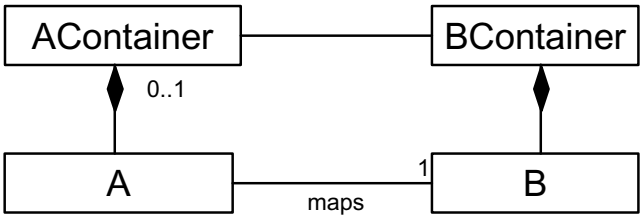


Fig. 2. Modeling Relations using Associations

want to map the structure of AContainer to BContainer we need to be able to express that all elements of one AContainer (i.e., all objects composed in an instance of AContainer) are mapped to elements of the respective BContainer. With associations we can only require that all As are mapped to some B by specifying a multiplicity of 1. We cannot restrict this to As nested in the concrete AContainer nor can we guarantee that the associated Bs reside in the corresponding BContainer. The pattern from [AK02] addresses this problem by defining a relation (ARelB) that captures the elements from the domain and the range, and a pair class (AB) that captures the fact that two elements are related. While this pattern provides the means to express nested mappings, it comprises some applicability problems. Every time a relation according to this pattern is built, two new classes are created and all OCL definitions that accompany the pattern have to be replicated with respect to the new names of the classes. The resulting structure is quite complex, and it is hard to separate the original model classes from the classes capturing the relation. These problems stem from the fact that the concept of relations was introduced on the model level in that approach.

Extending the metamodeling language appropriately would avoid these problems. The basic idea is to introduce Relation as an element of the metamodel, thus providing it as a language feature. All instances of this Relation feature would be separable from the standard language elements. Defined constraints would be obeyed by all instances of Relation automatically. To obtain this new language element one has to provide an abstract syntax, well-formedness conditions, semantics, and a concrete syntax. The following sections will provide these definitions, starting with the concrete syntax.

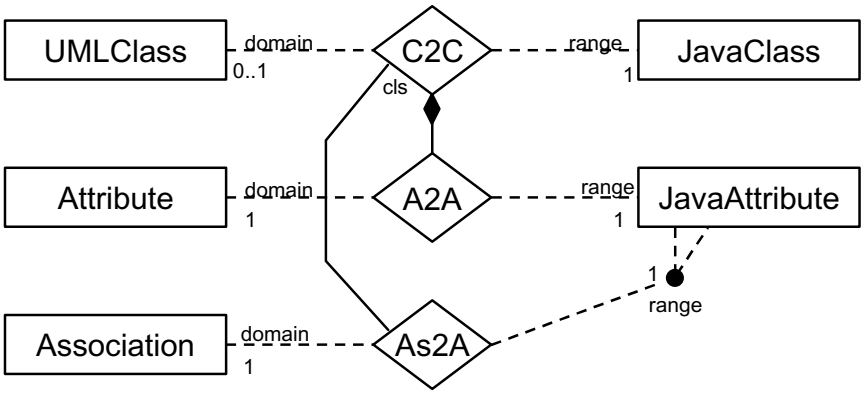


Fig. 3. Example of the Relation notation

3 Concrete Syntax

A concrete syntax for our Relations approach has been presented in [HK03]. Here, we only summarize the main features. Relation specifications extend UML class diagrams. They are used to define the relations between different model elements. Figure 3 shows an example of relations in a simple UML to Java mapping (a much more extensive version of such a mapping is provided in [HK03]). Relations are expressed using a dashed line connecting the related elements. Similar to UML associations, the ends of the relation can be adorned with multiplicity constraints. Hence, a UMLClass maps to exactly one JavaClass but additional JavaClasses without a corresponding UMLClass may exist. Since the relation A2A is nested in the C2C relation (indicated by the composition notation), each JavaAttribute of a JavaClass related to a UMLClass must have a corresponding definition on the UML side (multiplicity 1). A diamond shape in the middle of the relation captures the name of the relation and is used as an anchor for associations to other relations. These associations may express the nesting of a relation inside another or the possible referencing of one association from another. This referencing is needed to avoid redundant definitions. In the example, the relation for mapping associations (As2A) needs information about the mapping of the UMLClasses it connects to ensure that its target attributes are typed correctly. On the range side of the As2A mapping a tuple is depicted by a black dot. This notation is used to combine several objects to a more complex structure. Here, it indicates that a UML association is represented by two Java attributes pointing at each other's classes.

4 Abstract Syntax

The model for the abstract syntax of a modeling language containing relations is given in Fig. 4. This metamodel is aligned to the recent UML 2.0 infrastructure proposal of the U2P group [U2P03][1]. In particular it extends the notion of

[1] This submission is currently recommended for adoption as UML 2.0 by the OMG

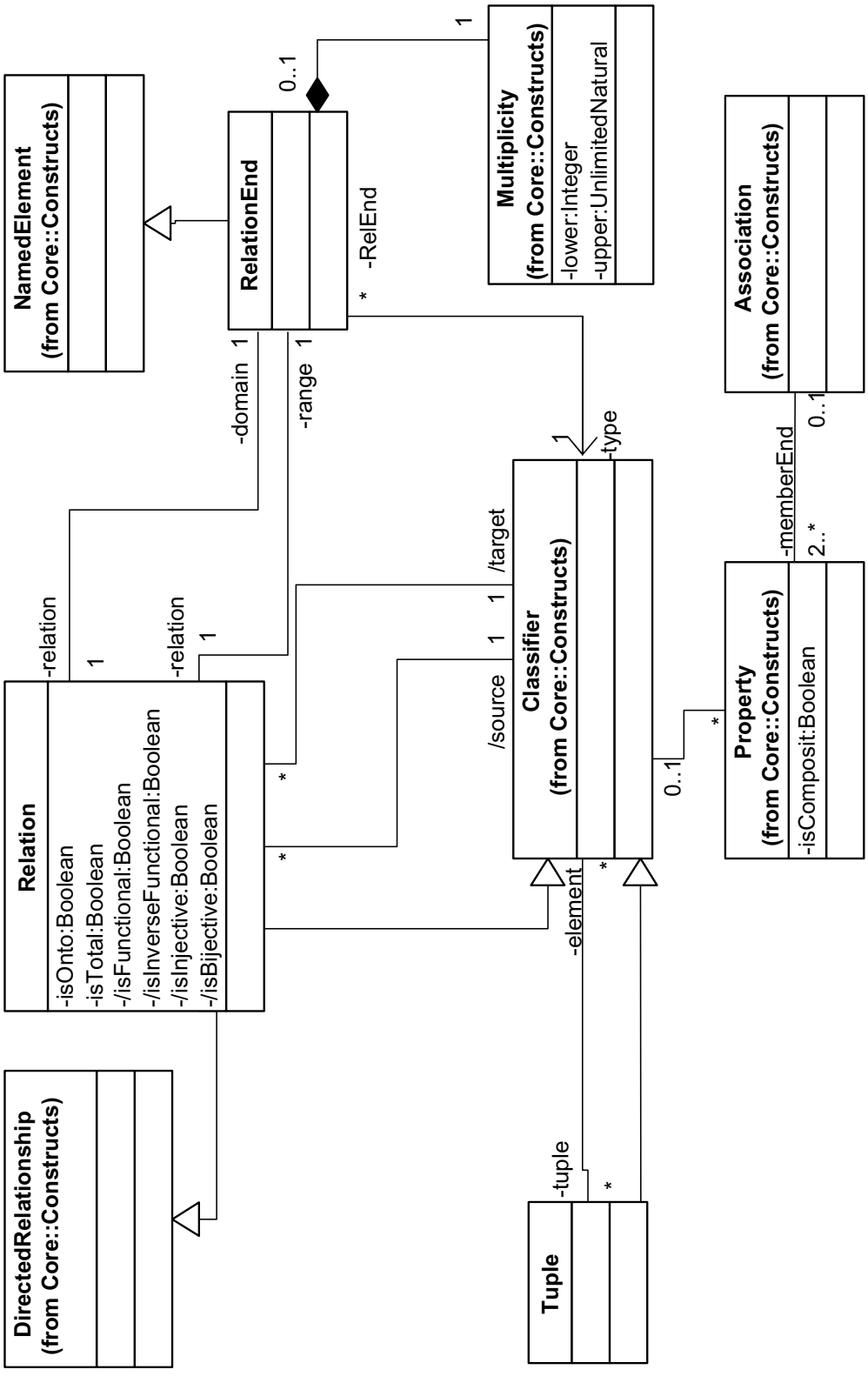


Fig. 4. Metamodel defining the abstract syntax

class diagrams by importing some elements from the Core::Constructs package (Classifier, Association etc.) and combines them with new elements needed to express our relation concepts.

Relation. Relation is the main class of the new model. This class captures the definition of a connection between two model elements. It therefore refines the concept of a DirectedRelationship from the Constructs package. Since relations have instances and contain OCL constraints for them, they inherit from Classifier. Similar to associations, relations are connected to two ends which contain information about the way the instances of the relation can be constructed. While the domain and range ends only indicate the type of the connected elements, OCL constraints may be used to further refine these sets. The source and target links are redefined from DirectedRelationship and indicate the classifiers connected to the domain and range RelationEnds.

```
context Relation
  source=domain.type
  target=range.type
```

For mathematical relations there are well-known properties like *is functional*, *is total* etc. When modeling relations we encode this information in the multiplicity constraints at the RelationEnds (e.g., an upper limit of 1 on the range end of a relation indicates that every domain element may participate at most in one pair of the relation; this corresponds to the definition of functionality). Thus, the metaclass Relation only has derived attributes for easier access. Further properties like *is bijective* can be combined out of these basic features.

```
  isFunctional = (range.multiplicity.upper=1)
  isInverseFunctional = (domain.multiplicity.upper=1)
  isTotal = (domain.multiplicity.lower=1)
  isOnto = (range.multiplicity.lower=1)
```

Note that in contrast to [AK02] these definitions are not derived from the actual instance of the relation but are definitions on the specification level. The attribute isOnto thus states that an instance of the Relations *should* cover all range elements but situations may arise (e.g. by adding new elements to the model) in which the constraint does not hold. This may trigger inconsistency detection and reconciliation mechanisms (see e.g. [HHS02]).

RelationEnd A RelationEnd is similar to an AssociationEnd as known from the current UML definition [Obj01b]. It contains a rolename by which the relation can address the Classifier that forms the type of the RelationEnd. RelationEnds contain multiplicities. These specify in how many Pairs of a Relation a domain or range instance may or must participate. Note that these specifications are not global like with associations but rather dependent on the concrete instance of the relation. This is necessary to provide the facilities for nested relations. It is also the fundamental difference to AssociationEnds[2].

[2] In the UML 2.0 submission AssociationEnds have been integrated in the more general concept of Property, see [U2P03] for details.

Tuple A tuple is a necessary addition to the metamodel to enable the mapping of structures. It is a classifier itself and may thus participate in relations. A tuple comprises an ordered set of elements. We do not provide additional details here because Tuple is just an auxiliary construct which is, e.g., also defined in [Bol02].

5 Semantics

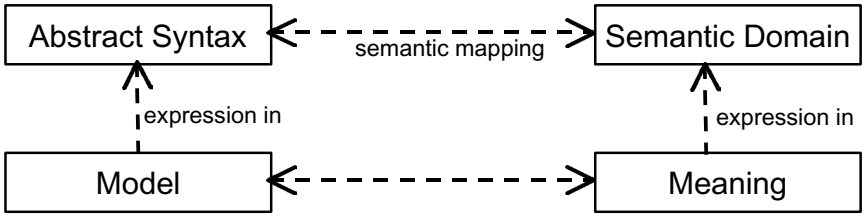


Fig. 5. Conceptual overview of denotational metamodeling

For the definition of the semantics we follow a denotational metamodeling approach as outlined in Fig. 5. We want to supply each model with semantics (i.e. a meaning, expressed in some formalism). For this, we relate the definition of the models (the metamodel) to elements of a metamodel expressing a semantic domain. The following subsections contain this semantics definition in three steps: first we define a UML model with well-formedness constraints that describes the semantic domain, then we explain (informally) how the semantic mapping between abstract syntax and the semantic domain is supposed to work, and finally we formalize this mapping using relations. By using our own approach to explain our semantics we get a recursive approach like the definition of UML's abstract syntax by UML class diagrams. This mapping also provides a nice example to demonstrate the application of our approach.

5.1 Semantic Domain

The model of the Semantic Domain is given in Fig. 6. It comprises three classes: Instance, RelationInstance and Pair. Instance is a generic class that captures the fact that an element of the semantic domain corresponds to some defining element (its type). A RelInstance is an instance of a Relation. It contains the set of all domain and range elements (according to the specification given in its defining Relation). A RelInstance contains a set of pairs. These pairs must be unique in the RelInstance. A Pair can form the scope of a RelInstance (though never for the one it belongs to).

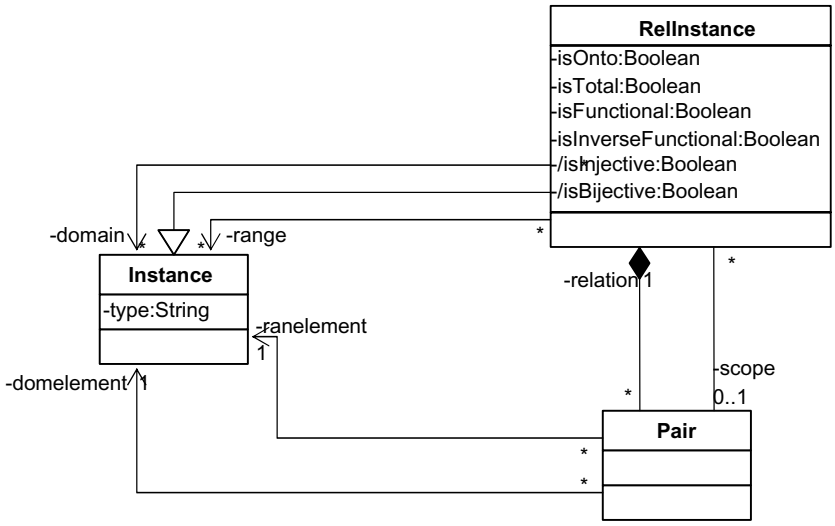


Fig. 6. Metamodel defining the semantic domain

```
context RelInstance inv:
    pair->forAll(x,y|(x.domelement=y.domelement)
     and (x.ranenelemt=y.ranelement) implies x=y)
    not pair->includes(scope)
```

A Pair embodies the combination of elements from domain and range included in the RelInstance. Each Pair is linked to its containing RelInstance and one element from the domain (domelement) and the range (ranelement). It has got to be ensured that only elements from the sets defined by the RelInstance are connected.

```
context Pair inv:
    relation.domain->includes(domelement)
    relation.range->includes(ranelement)
```

5.2 Informal Semantics

A semantic mapping has to provide the connection between the concepts expressed in the abstract syntax and the semantic domain. We will explain these connections first and then proceed to formalize them. The tricky bit here is that Relations as we understand them do not conform to the usual concept of instantiation where a classifier describes a set of instances that conform to its specification. Rather, a set of patterns made up from a RelInstances and Pairs contained in the RelInstance is defined by the Relation. The specification present in the Relation has an impact on different parts of this pattern. The domain and range sets of the RelInstances conform to the description provided by the Relation. The Pairs contained in the RelInstances conform to additional constraints provided by the Relation. The set of all Pairs in one RelInstance conforms to the

multiplicities specified at the RelationEnds. Composition associations between Relations will result in scope links between a Pair that stems from the RelInstance of the containing Relation and a RelInstance conforming to the nested Relation.

5.3 Formal Semantics

The informal semantics given in the previous section described the mapping between the elements of the abstract syntax and the semantic domain. Both were defined as models, using elements provided by MOF. We could thus regard this semantic mapping as an application scenario for the technique we are proposing.

Fig. 7 provides the mapping between the abstract syntax and semantic domain. Note that only parts of the metamodels are shown here to avoid cluttering the figure. The most basic relation is C2I, the Classifier to Instance mapping. This mapping expresses the general semantics of classifiers which says that each classifier describes a set of instances. The instances have a type attribute that indicates the classifier they conform to.

```
context C2I
  -- Classifier to Instance
  domain= Classifier.allInstances
  range= Instance.allInstances
  inv: ranelement.type=domelement.name
```

The relation capturing the semantics of Relations is R2RI, Relation to RelationInstance. It contains several constraints to ensure that the specifications in the abstract syntax are correctly related to the elements of the semantic domain.

```
context R2RI
  -- Relation to RelInstance
  domain= Relation.allElements
  range= RelInstance.allElements
  inv: domelement.isOnto=ranelement.isOnto
   domelement.isTotal=ranelement.isTotal
   domelement.isFunctional=ranelement.isFunctional
   domelement.isInverseFunctional=ranelement.isInverseFunctional
```

The pairs conforming to a RelInstance must obey the multiplicity constraints imposed by the defining Relation (we only show one of these constraints, the others are constructed similarly).

```
   ranelement.domain->forAll(i| ranelement.pair->select(p|
    p.domelement=i)->size>domelement.range.multiplicity.lower)
```

To check, whether the domain and range objects are correctly typed, an access to the C2I mapping is necessary. This is denoted by the associations between R2RI and C2I which can be accessed from R2RI using the rolenames

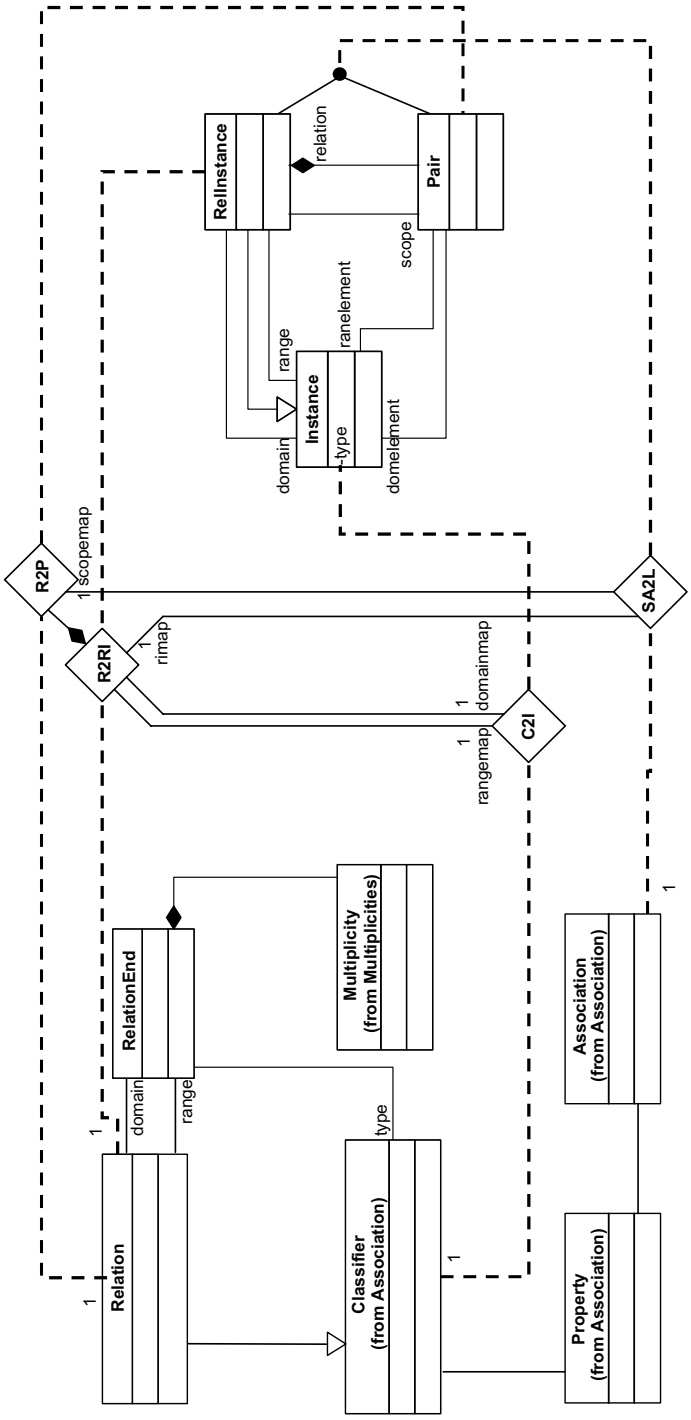


Fig. 7. Mapping between the model of the abstract syntax and the model of the semantic domain

domainmap and rangemap. Since invariants in a relation will be evaluated in the context of a Pair (of the R2RI relation), the navigation self.domelement will result in a Relation and self.ranelement in the corresponding RelInstance.

```
domainmap.domelement->includes(self.domelement.domain.type)
rangemap.domelement->includes(self.domelement.range.type)
domainmap.ranelement->includes(self.ranelement.domain)
rangemap.ranelement->includes(self.ranelement.range)
```

The relation R2P connects Relations and Pairs. It expresses that the invariants of the Relation constrain the way the Pairs in its RelInstance are built. Thus this relation can only happen in the context of an R2RI mapping. The containing Relation can always be addressed from the contained relation using the rolename scope.

```
context R2P
  -- Relation to Pair
  domain=scope.domelement
  range=scope.ranelement.pair
```

To give a formalization of the constraint evaluation, the abstract syntax and semantics domain of OCL expressions have to be defined as metamodels. This is accomplished by the recent OCL 2.0 submission [Bol02]. Yet the structures defined there are quite complex and using them here would require considerable explanations without providing much insight to the relation technique. They are thus omitted here, a forthcoming technical report will fill this gap.

The fourth relation connecting abstract syntax and semantic domain is called SA2L, ScopeAssociation to Link. A composition between two Relations in mapped to a Pair and a RelInstance.

```
contex SA2L
  -- Scope Association to RelationInstance-Pair Link
  domain= collect(a:Association|(a.property.isComposite
   ->includes(true) and (a.property.type=Relation))
```

The Pair contained in the tuple forming the range must be the scope of the RelInstance. By referring to an R2RI and an R2P mapping it is ensured that only instances of the correct relations are connected.

```
  inv: tuple.relInstance.scope=tuple.pair
   scopemap.domelement.property->exists(p|p.isComposite=true
     and p.association=self.domelement)
   scopemap.ranelement->includes(self.ranelement.pair)
   rimap.domelement.property.association->includes(self.domelement)
   rimap.ranelement->includes(self.ranelement.relInstance)
```

6 Conclusions

6.1 Summary

In this paper we have defined elements of a metamodeling language that are able to capture relations. The abstract syntax and semantic domain have been expressed using class diagrams and well-formedness constraints in OCL, i.e., as a metamodel. The new elements can be used to define mappings between languages that have themselves been defined using a metamodeling approach. An illustration of the language extension in use has been provided by defining a semantics mapping of the language itself. This is a rather special kind of mapping. A more practically based example can be found in [HK03], where the relation between UML class diagrams and structures of the JAVA language are defined. This is a form of platform independent model to platform specific model mapping.

6.2 Implementing the Approach

We are in the process of defining tools that incorporate the notion of Relations and are able to generate (two-way) transformations out of them. The Kent Modeling Framework [KMF] is a generator that builds modeling tools from metamodel definitions. These tools are able to automatically check OCL constraints and can thus enforce the conditions imposed by Relations. Rule-based transformation facilities are currently being added to this framework. Rules to detect and reconcile inconsistencies have been proposed in [HHS02]. A closely related effort by David Akehurst [Ake00] proposes to use a modular transformations engine based on instances of the observer pattern.

6.3 Related Work

Relations have been used in computer science for a long time and in different fields (see e.g. [BKS97]). Input for extending our approach can be gained from these different approaches. On the more theoretical side, many results exist how to reason about and derive new information from a given set of relations. A system that builds upon those results is e.g. the RelView system [BBH+99]. This system does not focus on the relations of models but contains interesting ideas on the visualisation of relations and their instances. In comparison to [CRZ+01,BHP00], which also set out to explicitly define model mappings, we provide much richer notations and concepts. By doing work toward reconciliation using mappings, we will also consider the idea of basic operations on mapping classes as in [BHP00]. Our work may be seen as an answer to the challenge issued there: “Find appropriate representations of model mappings that trade of expressiveness of semantics with the goal of keeping operations generic across a wide set of model types”. A similar challenge is formulated in [PW02]. Recent works employing XML-based technologies are XLinkit and MTrans [NCEF00,PBG01]. XLinkit focuses on (consistency) relations between

different models and provides a checking mechanism that points out inconsistencies. The work is targeted toward web-based systems. MTrans defines a textual translation language that describes XSLT transformation rules for transforming models between development phases. The work focuses on executability of the specification, reconciliation and reverse transformation issues are not addressed.

6.4 Future Work

We believe that introducing Relation as a concept in modeling yields many benefits and should receive further attention. In the short term, we need to implement the approach so that we can test it on more sophisticated examples. Of particular interest in this work is the degree to which we can generate executable transformations, in one or other direction, from a mapping definition expressed using our language. Initial experiments are quite positive. We also wish to explore the problem of reconciliation in the context of two-way mappings [KS03].

The approach evolved from identifying a pattern for modeling mappings using standard techniques–class diagrams and OCL, to extending the standard techniques for expressing mappings more concisely. We expect that there are further patterns for modeling mappings that build on the original base pattern, and we expect to discover these as we attempt to model more sophisticated examples. It will be interesting to see if it is possible to further enhance the notations to capture these new patterns, or whether we will have to revert to a more general purpose device, such as package templates [MWC+02], for capturing the patterns.

Acknowledgements

This paper is based on work carried out during a stay at the University of Kent at Canterbury. The generous funding for the stay was provided by the European Research and Training Network Segravis. I am deeply grateful for the supervision and cooperation of Stuart Kent. I would furthermore like to thank Dave Akehurst and Alexey Cherchago for checking the metamodels and the OCL constraints and providing helpful insights.

References

[AK02] D. H. Akehurst and Stuart Kent. A relational approach to defining transformations in a metamodel. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, volume 2460 of *LNCS*, pages 243–258. Springer, 2002.

[Ake00] David H. Akehurst. *Model Translation: A UML-based specification technique and active implementation approach.* PhD thesis, December 2000. (unknown variable note).

[BBH+99] R. Behnke, R. Berghammer, T. Hoffmann, B. Leoniuk, and P. Schneider. Applications of the RelView system. In Lakhnech Y. Berghammer R., editor, *Tool support for system specification, development and verification*, Advances in Computing Science. Springer-Verlag, 1999.

[BHP00] Phillip A. Bernstein, Alon Y. Halevy, and Rachel A. Pottinger. A vision for management of complex models. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(4):55–63, 2000.

[BKS97] Chris Brink, Wolfram Kahl, and Gunther Schmidt, editors. *Relational Methods in Computer Science*. Advances in Computing. Springer-Verlag, Wien, New York, 1997. ISBN 3-211-82971-7.

[Bol02] Boldsoft et al. Response to the UML 2.0 OCL RfP, version 1.5, 2002.

[CRZ+01] Kajal T. Claypool, Elke A. Rundensteiner, Xin Zhang, Su Hong, Harumi Kuno, Wang chien Lee, and Gail Mitchell. Sangam — a solution to support multiple data models, their mappings and maintenance. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):606–606, 2001.

[HHS02] Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Extended model relations with graphical consistency conditions. In *Proceedings of the Workshop on Consistency Problems in UML-based Software Development.*, pages 61–74. Department of Software Engineering and Computer Science, Blekinge Institute of Technology, 2002.

[HK03] J.H. Hausmann and S. Kent. Visualizing model mappings in UML. In *Proc. of the ACM Symposium on Software Visualization 2003*, 2003. to appear.

[KMF] The Kent Modeling Framework. `www.cs.ukc.ac.uk/kmf`.

[KS03] Stuart Kent and Robert Smith. The bidirectional mapping problem. In *Uniform Approaches to Graphical Specification Techniques (UniGra 2003)*, 2003.

[MWC+02] Girish Maskeri, James Willans, Tony Clark, Andy Evans, Stuart Kent, and Paul Sammut. A pattern based approach to defining translations between languages. In *Presented at the fourth workshop on Rigorous Object Oriented Methods (ROOM4)*, 2002.

[NCEF00] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a Consistency Checking and Smart Link Generation Service, 2000.

[Obj01a] Object Management Group. Model driven architecture - a technical view, 2001.

[Obj01b] Object Management Group. UML specification version 1.4, 2001.

[Obj02] Object Management Group. MOF 2.0 query / views / transformations RfP, 2002.

[PBG01] Mikal Peltier, Jean Bzivin, and Gabriel Guillaume. MTRANS: A general framework, based on XSLT, for model transformations. In *Workshop on Transformations in UML (WTUML)*, 2001.

[PW02] O. Preiss and A. Wegmann. Strengthening MDA by drawing from the Living Systems Theory. In *Proceedings of UML2002 Workshop in Software Model Engineering (WiSME@UML2002)*, Dresden, October 2002.

[U2P03] U2Partners. U2 Partners' UML 2.0: Infrastructure, 3rd revised submission. `http://www.omg.org/cgi-bin/doc?ad/03-01-01`, 2003.

Towards Model Transformation with TXL

Richard Paige and Alek Radjenovic

Department of Computer Science, University of York, Heslington, York YO10 5DD, UK
{paige,alek}@cs.york.ac.uk

Abstract. We demonstrate the use of the transformation tool TXL in representing, but particularly for implementing efficient transformations between languages. The approach shows how to write and reuse language definitions, express rules for transformation based on patterns, and outlines how transformations can be developed in an agile way, compatible with the practices of test-driven development. The intention is to be able to use TXL as a behind-the-scenes technology for implementing efficient, scaleable transformations of models.

Introduction

The Model-Driven Architecture (MDA) initiative [6] has at its core the concept of a transformation. Transformations are either *specifications* or *implementations* that input models and either produce new, related models, or relate the input models to existing models. These transformations enable model integration, the mapping of platform-independent to platform specific models, reverse engineering, and platform migration. Transformations can be defined in terms of meta-models of languages. A concrete example of a transformation is an auto-code generator as is present in tools such as Rational Rose or Artisan. Given models described in profiles of UML, auto-code generators produce models in Java, Ada 95, and other languages. These transformations are implementations, as they are written in a programming language, make use of data structures embedded in the modelling tool, and are operational in nature. Transformations based upon the use of visitor patterns are also seeing widespread use, particularly for dynamic tool integration [7]. Transformations that rely on XML or XMI are often implemented using the XSLT package. And there is also a growing movement towards transformation modelling, e.g., via MOF or dialects of UML, or via QVT [5].

The MDA also focuses on separating specification of system functionality from specification of implementation on a specific technology platform. The intent is thus to allow developers to create systems entirely with abstract models, and to insulate developers from specific implementation technologies, data structures, and algorithms, thus enabling reuse of models across different platforms and different underlying implementation techniques.

A number of key issues need to be resolved in order for this aspect of MDA to be fully applicable to building industrial-strength applications.

- Most transformations are hand-built and are implemented directly in a programming language, based on internal representations of models (e.g., in a graph library); reference to the meta-model of the modelling languages is often left implicit. Transformations built in this way are error-prone, difficult to maintain or reuse, and are difficult to understand. They also require developers to have some knowledge of internal representations.
- There is little reuse of transformations when transitioning to new environments. For example, an auto-code generator for a tool such as Rational Rose has to be custom-made, and will not in general be usable in another tool or in constructing new generators for different languages.
- Transformations are often not standalone, and frequently are embedded in a model-building tool. Industrial-strength transformations are often implemented in expensive proprietary tools that cannot be easily customized or modified.
- It is difficult, if not impossible, to check transformations for correctness outside of testing, given that the implementations are often unavailable.
- Changing transformations is error-prone and expensive: even a small change to the transformation can require re-compilation of a tool, and may invalidate existing models. As well, changing transformations usually requires knowledge of implementation details – e.g., AST representations and visitor algorithms. Moreover, if one or more of the languages involved in the transformation happen to change, then the transformation itself is invalidated and, in general, will need to be entirely reworked.
- Transformations must be efficient and it must be possible to tune their performance to meet the requirements of industrial projects, where large-scale models, including legacy subsystems, must be dealt with.
- The process of *building* transformations is often incompatible with the practices and principles of agile development [14], which are of growing importance in systems engineering.

In this paper, we demonstrate a specification-based technique and tool for describing and implementing general language transformations; we illustrate how to use the technique for transformations involving UML models. The technique is based on the TXL system [3], which has been designed for handling transformations involving large-scale amounts of data. Novelties with TXL include its ability to shield users of the transformations from implementation details (e.g., data structures) and its use of *abstract patterns* for defining the transformations.

We show how to represent language definitions in TXL, and how to describe transformations using rule sets. The approach is independent of any CASE tool and data structure, and is compatible with agile development; we illustrate the latter by outlining a test-driven development process [1] for TXL specifications. Moreover, changing the transformation does not require changing the language definitions or representations, and small changes to language definitions can easily be

accommodated with small changes to the transformation rules. The approach also supports transformation reuse: one can extend or redefine parts of an existing transformation to produce new ones.

We commence with a brief review of related work and introduce the TXL system. We show how to use TXL for model transformation, based on an example for mapping a profile of UML into Java. The profile represents UML models using a simplified subset of the XMI specification for UML; the transformation thus operates at the level of the UML meta-model. We discuss the advantages and disadvantages of the approach, and comment on its performance, particularly at an industrial scale.

Our intention is to position TXL as a potential back-end technology for representing and implementing efficient industrial-scale transformations. It would be interesting to carry out experiments to determine whether or not the TXL approach is a suitable front-end technology for modellers to use in describing transformations – from the perspective of usability and expressiveness – but this is not the focus of this paper.

Related Work

There has been much work on transformations for UML in the CASE tool industry. Leading tools such as Rational Rose and Rhapsody – and many others – support auto-code generation in a number of programming languages (e.g., Ada 95, C++, Java) and document interchange formats (e.g., XMI, DOM). These transformations are invariably implementations and are not reusable; moreover they are difficult if not impossible to customize by their users. The approach is not immediately compatible with the aims of MDA.

The QVT Partners initial submission to the QVT RFP [5] proposes a technique for modelling transformations in a manner compatible with UML and MDA. Transformations are modelled in a graphical language, with an executable dialect of OCL used to capture patterns for transformation. Transformations can be both mappings and relations; the latter can be used in either direction (i.e., they are reversible). Preliminary tool support is also available, via technology from Xactium. QVT aims to provide modeller accessible technology for describing and constructing transformations.

The Eiffel Studio tool from ISE [9] provides a simple user-accessible approach to defining mappings from the Eiffel language. The tool provides an interface revealing Eiffel meta-elements, e.g., class, cluster, attribute, routine, invariant. Users of the tool can associate text tags and strings with each meta-element, expressing how to map the meta-elements into a target language. Thus, users can define new mappings in a straightforward way, without having to build auto-code generators. The mappings are unidirectional and require some understanding of Eiffel syntax in order to write them. They are not reusable, and will need to be changed if the Eiffel syntax is modified in

any way other than extension. As well, complex mappings are difficult to express with this approach, since it is based on simple string replacement.

The Template method design pattern has been proposed as a useful mechanism for implementing transformations within CASE tools [10]. The pattern provides the means to separate the *process* of transformation from the *details* of the textual or graphical rewriting. The process can be reused for different transformations, while a developer of a new transformation must provide a concrete implementation of textual rewritings, usually in terms of actions applied across an abstract syntax tree. This approach has been implemented in the BON-CASE tool and has shown to be practical for implementing auto-code generators reasonably quickly [10].

Porres [12] suggests using a UML-aware scripting language for representing UML models and defining transformations, as part of the System Modelling Workbench. It requires using Python and does not support graphical manipulation of models. But it is similar to the work proposed in this paper as it represents UML models using XMI, and represents transformations textually using an OCL-like language. Our work represents transformations using patterns, which may be simpler and can often lead to fewer rules being needed to specify a complete transformation.

XSLT is a standard language, from the W3 Consortium, for transforming XML documents into other XML documents; it is particularly useful for tool interchange. It is a complex language, with part of this complexity due to the generality of XML. It is strong in its support for pattern and string matching – a significant similarity with TXL. However, XSLT is directly targeted to transforming XML documents, whereas TXL is a general purpose transformation tool.

More generally, work on graph transformation has been applied to UML mappings, e.g., in the Fujaba project [13].

Overview of TXL

TXL, due to Cordy et al [3], is a general transformation tool designed particularly for efficient processing of large datasets. It has been applied successfully in the Canadian banking industry for design recovery [2] and re-engineering a number of different banking IT systems in order to deal with the Y2K problem. This involved transforming massive programs – in a variety of different, sometimes obsolete languages – quickly, efficiently, and oftentimes while a version of the program continues to run. Industrial experience with using TXL, particularly for design recovery and source-level transformation, involves more than 4.5 billion lines of code in a variety of languages and representation formats.

TXL is grammar-based and supports grammar extension and reuse. It is also ideally suited to agile development techniques, which aim to produce code quickly

and reliably, with only minimal use of modelling. We discuss agile development in more detail later.

TXL works by accepting an input text (e.g., in XMI or ASCII) and constructing a parse tree. This tree is then transformed, based on a specification of transformation rules, into a new tree, which is then unparsed to form output text. Thus, a TXL specification consists of a grammar specification (containing syntactic well-formedness rules for both the input and output languages) and a suite of structured transformation rules.

Figure 1 shows an example of a TXL grammar specification for a very simple program language for expressions. TXL supports an extended BNF-like notation for sequencing, including **[repeat X]** (for a sequence of zero or more X's), **[repeat X+]** (one or more X's), **[list X]** (a comma-separated list of zero or more X's), and **[opt X]** (zero or one X's).

```
define program
     [expression]
end program

define expression
        [term]
      | [expression] + [term]
      | [expression] - [term]
end define

define term
     [primary]
   | [term] * [primary]
   |  [term] / [primary]
end term

define primary
        [number]
      | ( [expression] )
end primary
```

Figure 1: A TXL grammar specification for expressions

Lexemes and basic tokens (e.g., numbers, identifiers, etc.) can be specified using a standard regular expression notation, and individual characters can be grouped together using a prefixed prime in order to create anonymous lexemes on the fly.

Grammars are not usually built from scratch; it is common to import an existing grammar and to use TXL's facilities for grammar overriding. For example, one might import the grammar for C++ and redefine its **statement** set of production rules to allow XML mark-up, as follows.

```
include "Cpp.Grammar"
redefine statement
     …
|    <[id]> [statement] </[id]>
end redefine
```

The **include** statement imports an existing TXL grammar for C++ and the redefine statement adds a new production rule for marked-up statements; the ellipsis is an

instruction to TXL to retain all other production rules unchanged. This approach to grammar overriding has also been used successfully to carry out semi-parsing with TXL, wherein specified sentential forms encountered during parsing are ignored.

The input-to-output transformation in TXL is specified using a set of rules on non-terminals (which, recall, specify rooted sets of sentential forms). Each rule specifies a target type (i.e., a non-terminal in the source grammar) to be transformed, a *pattern* (an example of the target type that we're interested in transforming), and a *replacement.* Consider the following example rule, which specifies how to transform statements in C to statements in Pascal. This particular rule provides a pattern to match *while*-loops. Each rule has a name (e.g., **map_while_statements**) and specifies a replacement pattern that applies to a particular non-terminal (in this case, **statement**).

```
rule map_while_statements
    replace [statement]
          'while E [expression] {
                S [repeat statement]
          }
    by
          'while (E) 'do 'begin S 'end ;
end rule
```

The pattern specifies that a while-loop in C consists of the token **while** (indicated by the prefixed prime), followed by an expression **E**, followed by the body of the loop in braces; the body consists of zero or more statements **S**. **E** and **S** in the above rule are variables of type **expression** and **statement**, respectively. The pattern is replaced by a while loop with the body in **begin-end** wrappers. We would need to write additional rules for transforming different C language statements, but they would follow the same template, requiring new patterns and new replacement texts. There are generalisations of rules, particularly for passing parameters. Parameters are typically used to build transformed results from several parts, and roughly correspond to temporary variables in Yacc. This enables more complex transformations to be implemented, above straightforward textual replacement.

TXL supports *rules* (which search their scope for the first instance matching their pattern) and *functions*, which are typically used to apply several rules to a single scope. Functions do not search; they attempt to match their entire scope to their pattern, transforming it if it matches. For example, the following function transforms one occurrence of the pattern N1+N2.

```
function resolve_addition_expression
    replace [expression]
          N1 [number] + N2 [number]
    by
          N1 [add N2]
end function
```

The TXL repository contains a number of grammars for existing widely used languages, such as C++, Pascal, Java, XML 1.0, and others, which can be adapted or used to produce new grammars easily.

As was mentioned earlier, TXL is well suited to agile development, particularly styles of development driven by testing, e.g., TDD as suggested by Beck [1]. A useful approach to building TXL transformations is to first write a set of test cases, treating these as a specification of the TXL transformation rules. The TXL specification is built incrementally, and run against the test cases as construction proceeds. In general, we write the simplest possible transformations that make the test cases run; tuning comes later. Studies by Cordy et al have shown that TXL specifications tune very well; speed-up factors in the range of 10-100 times are common [4].

Finally, TXL has been used successfully for design recovery, particularly for systems for which minimal source code is available, or for which architecture and design documents have been lost. The paper [2] illustrates this approach and explains how TXL has been used successfully in the banking industry for design extraction. In particular, the utility of TXL for design recovery initially led us to consider its application for model transformation.

Model Transformation via TXL

While TXL has, up to this point, been used primarily for the transformation of languages and design recovery, there is nothing to prevent its use for transforming models, particularly, but not exclusively, to implementations. Such a use will require the meta-model of the modelling language to be represented using a set of TXL grammar specifications. There are a number of advantages of using TXL to specify and implement model transformations.

- It is extremely efficient, having been designed to process large datasets with large grammars (measured in terms of the number of sentential forms in the grammar).
- It supports grammar reuse, thus enabling the design of new transformations from existing language definitions and existing transformations.
- Existing models can be used as test cases for building TXL specifications in an agile way, as discussed in the previous section.
- Changing transformation rules will not require any changes to the language definitions, nor to any tools that are used to produce the models.
- Since languages are specified independently, through grammars, there is the opportunity for modular re-design and maintenance of language definitions.
- It is possible to check the transformation rules for correctness given that they are specified in a structured, precise way, independent of any implementation. While a formal semantics for TXL transformations does not as of yet exist, one could be produced and used as the basis for checking the validity and consistency of transformations.

We posit that TXL could be used to support elements of the MDA initiative. However, there are some elements of the initiative, and of related work, that TXL does not currently address. These include:

- *Relational mappings*: TXL transformations are unidirectional, i.e., from an input to an output, whereas as suggested by the QVT Partners submission [5], relational or bi-directional mappings are useful in order to enable reverse engineering and program understanding. In order to build a relational mapping in TXL, rules for each direction must be constructed. A more automatic mechanism would be useful.
- *Graphical Modelling*: TXL is grammar-based, and as such the rules and languages are not modelled in a language such as UML or QVT. However, such grammars can easily be used to express meta-models in a textual format. Given the meta-model of UML expressed as an XMI DTD, a TXL grammar for that meta-model can be produced in a straightforward manner, thus enabling TXL to process textual representations of models. It seems possible to consider writing a TXL specification to map the UML XMI DTD specification into TXL's input format (though given the significant ambiguity in TXL's input syntax, and the looseness of XMI DTDs, this could be challenging). As such, it may be more palatable to developers of transformations to use TXL as a back-end technology for implementing transformations, with a graphical language such as QVT as the front end.

We address these issues further in the discussion and conclusions.

Example: Transforming a profile of UML to Java

In order to illustrate how we might use TXL to implement model transformations, we show how to transform parts of a profile of UML to Java. Our purpose with this example is not to show a complete transformation; rather we show excerpts in order to demonstrate some of the issues that need to be resolved so as to apply TXL successfully in this domain, and to illustrate some of the strengths associated with using TXL.

The transformation makes use of XMI as a text-based representation of UML models; there is nothing in the general approach to using TXL that requires us to use XMI. The reader should consider this as a representative way of using TXL to model and implement UML-based transformations.

We assume that we have profiled UML to be suitable for implementation in Java, thus restricting the primitive types, relationships, and stereotypes that are available. We otherwise do not restrict which UML diagramming elements are present in the profile, other than that class diagrams must be included. We assume that we have available a suitable UML CASE tool that can generate XMI specifications of UML

diagrams that conform to the UML DTD available from the OMG. This DTD will form the basis of our TXL specification of UML.

To use TXL for language transformation, we take the following steps.

1. *Construct working grammars for each language.* A grammar for Java exists in the TXL repository. The UML DTD is available from the OMG; we continue to implement it in TXL (so we show snippets below).

2. *Uniquely rename grammar non-terminals.* It is possible that grammars constructed independently will share some non-terminal names; these must be resolved. The typical approach is to prefix non-terminal names with characters indicating the source of the non-terminal, e.g., **`Java_expression`** versus **`UML_expression`**. Note that this is a manual process, but it could be automated.

3. *Identify corresponding non-terminals.* We decide which non-terminals in the source and target languages should be used as the basis for transformation. In meta-model terms, these non-terminals correspond to meta-elements that need to be related by the transformation. The rules that specify the transformation will be written in terms of these non-terminals. For illustration purposes, we will base our simple transformation example in terms of non-terminals representing packages, classes, attributes, operations, and assertions, though, as we shall see, we get some transformations for free because of how TXL specifies grammars.

4. *Integrate the grammars for the source and target language.* The original grammars remain untouched, but a new combined transformation union grammar is formed. The general structure of the transformation grammar has one **define** statement for each identified non-terminal. As well, **redefine** statements are added to the original grammars to ensure that both untranslated and translated forms of the identified non-terminals can be accepted in each context. For example

```
define class
    [Java_class] | [ Foundation_Core_Class ]
end class

define package
    [Java_package] | [Foundation_Core_Package]
end namespace


define attribute
    [Java_attribute]
|   [ Foundation_Core_Attribute]
end attribute
```

The first **define** statement states that a class in the transformation grammar is either a Java class or a UML class (and similarly for the other definitions). We also need to redefine the UML DTD grammar such that a **`Foundation_Core_Class`** can be either a UML class or a Java class, since an instance of either can appear during the transformation process. This is a simple matter and requires writing simple, short redefine statements like the one below.

```
redefine Foundation_Core_Class
    …
|   [ class ]
end redefine
```

The process of building these defines and redefines is manual but could be automated, once the user identifies corresponding non-terminals. TXL could be used to automate the process. Finally, the target of the transformation must be specified. We call this a model, which can either be represented as a UML model or a Java program.

```
define model
    [ModelManagement_Model ] | [ Java_program]
end model
```

Transformations will be defined on the grammar element **`model,`** which is now the start symbol of the union grammar.

5. *Build mapping rules.* Each mapping rule will be targeted at some meta-element of exchange, e.g., package or class. The rule will express the relationship between one pattern expressible in the source (UML) and its transform in the target language (Java). We show several examples. The first is the rule for transforming packages. In order to illustrate these rules, we must show some details of the grammars themselves; the UML DTD is quite complex, so we make some simplifications here to avoid getting bogged down in details. However, making a transition from the simplified DTD here to the OMG standard should not be difficult (and could, in fact, be implemented in TXL).

```
rule map_packages
    replace [package]
          < 'Foundation.Core.Package >
          < 'Foundation.Core.Package.name>
          I [id]
          '</ 'Foundation.Core.Package.name >
          P [repeat package_contents]
          '</ 'Foundation.Core.Package >
    by
          'package I '{ P '}
end rule
```

The rule is surprisingly simple. The strings prefixed with a ` are lexemes consisting of more than one character in TXL (we could define them in TXL's **token** subsection but include them here for readability). A UML package is identified by the XMI header, and then its name represented by the variable I is specified. The variable P specifies the contents of the package. This is transformed to the Java package statement shown after the **by** clause. In general, additional rules must be written to transform the package contents; but the grammar rule for package contents will just be a definition containing classes (or interfaces, more generally) and other packages. These will be transformed by the rules for mapping packages and classes, respectively, so an additional rule is not needed.

A similar rule can be written for classes.

```
rule map_classes
    replace [class]
          < 'Foundation.Core.Class >
          < 'Foundation.Core.ModelElement.name>
          I [id]
          '</   'Foundation.Core.ModelElement.name
>
          C [ repeat class_contents]
          '</ 'Foundation.Core.Class >
    by
          'class I '{ C '}
end rule
```

The only difference from the package rule will be in the XMI headers that are used in the pattern, and in the rules that need to be written to transform class contents (which will be operations and attributes instead of package contents like classes and other packages).

Now we consider the transformation of class contents, via additional rules. We do not need a specific rule to transform the contents of a class; rather, we need specific rules to transform attributes and operations. That operations and attributes are included within a class is captured by the grammar itself, and thus when the contents of a class are encountered rules will be applied directly and automatically to the attributes and operations within the class.

```
rule map_attributes
    replace [attribute]
    < 'Foundation.Core.Attribute >
    < 'Foundation.Core.StructuralFeature.type >
    T [type]
    '</ 'Foundation.Core.StructuralFeature.type >
     < 'Foundation.Core.ModelElement.name >
          I [id]
```

```
    '</ 'Foundation.Core.ModelElement.name>
    '</ 'Foundation.Core.Attribute >
    by
          'public T I ';
end rule
```

Rules for transforming types will be captured elsewhere (and will be straightforward to express).

```
rule map_operations
    replace [operation]
    < 'Foundation.Core.Operation >
    < 'Foundation.Core.StructuralFeature.type >
    T [type]
    '</ 'Foundation.Core.StructuralFeature.type >
    < 'Foundation.Core.ModelElement.name >
     I [id]
    '</ 'Foundation.Core.ModelElement.name>
    ( P [list parameter] )
    '</ 'Foundation.Core.Operation >
    by
          T I ( P );
end rule
```

Additional rules can be added for dealing with assertions that are attached to classes and operations. Assume that the UML profile includes OCL as well as stereotypes for invariants and pre- and post-conditions. Assume as well that a contract package for Java (e.g., iContract [15]) is being used. Consider a class invariant; this would require additional grammar clauses, based on the OCL and Java expression languages, plus an additional rule as follows.

```
rule map_invariant
 replace [invariant]
 <'Foundation.Extension_Mechanisms.Stereotype >
 < 'Foundation.Core.ModelElement.name>
 'invariant
 '</ 'Foundation.Core.ModelElement.name>
 A [ repeat single_state_assertion+ ]
 '</'Foundation.Extension_Mechanisms.Stereotype >
  by
 '/*@invariant A '*/
end rule
```

We distinguish between single-state assertions – i.e., those that do not make use of the keyword **@pre** – and double state assertions since only the former may appear in invariant clauses. Additional transformation rules on expressions and assertions must now be written, but these are typically straightforward to do. The only

challenge with rules for assertions will be with OCL quantifiers that appear in UML models; these quantifiers may not all be representable in Java. We could deal with this problem when constructing the UML profile, i.e., establish that the assertions written in UML and OCL are transformable to Java; or, we could instruct our transformation rules to simply omit transformations for quantifiers, and to leave these as comments – i.e., semi-parse the model. All approaches are reasonable and straightforward to deal with using TXL.

To wrap up the transformation process, a single-step function on models must be declared, as follows.

```
function main
    replace [model]
          M [model]
    by
          M [map_packages]
            [map_package_dependencies]
            [map_classes]
            [map_class_relationships]
end function
```

This function simply states that to transform a model **M** we transform its packages, classes, dependencies, and relationships using rules like those specified above.

Discussion, Future Work, and Conclusions

TXL provides a robust, platform independent mechanism for specifying and implementing large-scale model transformations. Such transformations will be applicable to large models, and even heterogeneous models that integrate components in a variety of languages. TXL's facilities for language definition reuse (via import of grammar definitions), modification of language definitions, and proven industrial track record make it attractive as a means of supporting elements of the MDA initiative. What remains to be studied in further detail is TXL's suitability as a technology for transformation builders to use directly. A graphical modelling approach such as QVT may be more attractive as the means by which to design and build transformations, whereas a grammar-based technology such as TXL may be better suited to *implementing* efficient transformations that have been developed via other means. This suggests the need to define and implement formal links between QVT and TXL.

There are several elements of TXL that we intend to address in the future that aim to improve TXL's support for MDA goals, and to explore the links between QVT and TXL.

- *Link to visual modelling techniques.* TXL's grammar-based specification of language definitions is powerful, flexible, and convenient from the perspective of reuse. A link with QVT would improve TXL's utility in terms of addressing MDA concerns. This could be accomplished by making use of the QVT Partners' work and meta-modelling TXL's language definition itself; a transformation between MOF and TXL could then be defined (at the meta-level) thus enabling TXL specifications to be generated, e.g., using a tool such as XMT. While this approach would be the most abstract, an alternative, concrete approach would be to generate TXL specifications automatically from meta-models constructed using a suitable tool, e.g., Eclipse with plug-ins.

- *Multiple view transformation.* So far we have considered only single models (class diagrams with associated contracts) in our TXL transformations. The integrated UML meta-model includes a number of different views, e.g., sequence diagrams and state charts, and integrating them into the TXL transformations will be required. This will be accomplished by expressing in TXL those parts of the UML DTD related to these additional views; there is nothing unduly challenging to this process. An interesting element will be to attempt to use TXL to carry out lightweight view consistency checking using functions. While this seems possible, it may be very difficult to structure the XMI representation of models in a flattened way so that TXL can carry out the checking in a single pass.

- *Link with architectural modelling.* Architectural description languages such as AADL [16], and architectural extensions of programming languages such as ArchJava [11], provide abstract constructs for representing connectors, components, and their semantics. They are of increasing use and interest particularly in the high-integrity systems domain. Once our TXL implementation of the UML DTD is complete, we plan to implement an architectural description language in TXL as well and then will investigate the link between UML and the architectural models. This will provide a way to automatically transition between architectural models and UML models.

- *Links with reasoning and analysis techniques.* We can use TXL to enable transformations to languages that support formal reasoning and analysis. We have recently produced a TXL specification for PVS (effectively capturing PVS's meta-model) and intend to use this to support UML-to-PVS mappings, particularly for heavyweight consistency checking.

- *Larger case studies.* We need to apply the approach to transformation case studies with a particular eye towards performance analysis.

This work has many similarities to that of the QVT Partners. Their work specifically aims to create systems and transformations using models, and shields developers from having to know specific implementation technologies, e.g., ASTs and graph representations. TXL, by comparison, represents transformations using specifications of rule sets; it is thus at a different level of abstraction than the QVT

Partners' work. It is our view that the best use of TXL in this domain is to implement efficient transformations behind the scenes, perhaps by interfacing with the QVT Partners' work as an implementation of their transformations. This remains to be explored in the context of the larger case studies that we discussed above.

In the end we believe that TXL will provide a clean, robust, efficient mechanism to specify and implement a variety of transformations within the framework of the MDA. A mechanism like TXL – that provides efficient, scaleable, extensible transformations – will be necessary within the framework of the MDA. Whether the TXL toolset itself will or should be visible to developers directly is an open issue that can be resolved only by case studies and further investigation.

References

1. K. Beck, *Test-Driven Development*, Addison-Wesley, 2003.
2. T. Dean, J. Cordy, K. Schneider, and A. Malton. Using Design Recovery Techniques to Transform Legacy Systems. In *Proc. IEEE International Conference on Software Maintenance 2001,* IEEE Press, November 2001.
3. J. Cordy, I. Carmichael, and R. Halliday. *The TXL Programming Language (Version 10)*, Queen's University, Canada, 2000. Available at `www.txl.ca`.
4. J.R. Cordy, T.R. Dean, A.J. Malton, and K.A. Schneider. Software Engineering by Source Transformation – Experience with TXL. In *Proc. IEEE First International Workshop on Source Code Analysis and Manipulation*, Florence, November 2001.
5. QVT Partners. *QVT Initial Submission to OMG RFP*, March 2003. Available at qvtp.org.
6. OMG. *Model-Driven Architecture (MDA)*. Document ormsc/2001-07-01, July 2001. Available at www.omg.org.
7. K. Stirewalt and L. Dillon. A Component-Based Approach to Building Formal Analysis Tools. In *Proc. International Conference on Software Engineering 2001*, ACM Press, May 2001.
8. S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. *PVS Language Reference Manual*, SRI International, November 2002.
9. ISE Inc. *Eiffel Studio 5.2 System Documentation*, March 2003.
10. R. Paige, L. Kaminskaya, J. Ostroff, and J. Lancaric. BON-CASE: an Extensible CASE Tool for Formal Specification and Reasoning, *Journal of Object Technology 1(5),* August 2002.
11. J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language Support for Connector Abstractions. In *Proc. ECOOP'03*, LNCS, Springer-Verlag, July 2003.
12. I. Porres. A Framework for Model Transformations. In *Proc. Workshop on Integration and Transformation of UML Models* (co-located with *ECOOP'02),* http://www-ctp.di.fct.unl.pt/~ja/wituml02.htm.
13. U.A. Nickel, J. Niere, J.P. Wadsack, and A. Zündorf. Roundtrip Engineering with FUJABA. In *Proc. Second Workshop on Software-Reengineering (WSR),* Bad Honnef, Germany, Fachberichte Informatik, Universität Koblenz-Landau, August 2000.

14.A. Cockburn. *Agile Software Development*, Addison-Wesley, 2001.
15.R. Kramer. *iContract: the Java Design-by-Contract Tool*. Available at http://www.reliable-systems.com/tools/iContract/iContract.htm.
16.B. Lewis and P. Feiler. *Avionics Architectural Description Language Tutorial*, Toulouse, October 2002. Available at http://www.axlog.fr/R_d/aadl/seminar.html

A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard

Tracy Gardner[1], Catherine Griffin[1], Jana Koehler[2], and Rainer Hauser[2]

[1] IBM Hursley Development Laboratory, MP 188, IBM Hursley, Winchester, SO21 2JN, United Kingdom, {tgardner|catherine_griffin}@uk.ibm.com
[2] IBM Zurich Research Laboratory, CH-8803 Rueschlikon, Switzerland, {koe|rfh}@zurich.ibm.com

Abstract. Model-to-model transformation is a key technology for OMG's Model Driven Architecture™. The need for standardization in this area led to the MOF 2.0 Query/Views/Transformations Request for Proposals (RFP) from OMG. The RFP elicited eight submissions.
This paper makes the following contributions: Terminology for queries, views, and transformations is introduced based on the terminology used in the submissions, but edited for consistency. A set of common transformation scenarios is described, motivated by the authors' practical experience with transformations. The submissions are reviewed, compared to each other, and their highlights discussed. Based on the review and the experience of the authors in developing model-driven transformations, recommendations for the final standard are presented.

1 Introduction

OMG's Model Driven Architecture (MDA) [6] is a software development approach in which models are the primary artifacts. Abstract models are refined to more concrete models, eventually resulting in platform-specific models from which executable artifacts (such as code and configuration files) can be generated. MDA differs significantly from earlier uses of modeling languages such as OMG's UML™[5] in which the primary purpose of models was to aid understanding and communication. The main difference with MDA is that the models are the key part of the definition of the software system. Rather than the models being handed over to programmers to implement, all or much of the structure and behavior of a system is captured in models, which are automatically transformed into code (and other platform artifacts). Knowledge of the platform is encoded into transformations, which are reused for many systems rather than redesigned for each new system.

In MDA, automated transformations play a key role. It is important that transformations can be developed as efficiently as possible. A standard syntax and execution semantics for transformation is an important enabler for an open MDA tools chain. On April, 24, 2002, the OMG issued a Request for Proposals (RFP) for MOF 2.0 Query, Views, and Transformations (QVT) [7] to address a technology part of the OMG Meta Object Facility MOF 2.0 pertaining to the main issues in the manipulation of MOF models:

1. Queries on MOF 2.0 models,
2. Views on MOF 2.0 metamodels,
3. Transformations of 2.0 MOF models.

The RFP has elicited 8 submissions, many submitted jointly by a number of organizations. These submissions total several hundred pages, making it a time-consuming task to assess them adequately. This paper makes the following contributions: In Section 2, we define terminology based on the usage in the submissions but edited for consistency. In Section 3, a set of common transformation scenarios based on the authors' experience with model-to-model transformation is introduced. The authors are primarily implementers of transformations [1, 10] rather than implementers of transformation execution languages and environments. Section 4 provides an overview of the submissions and reviews them based on the RFP requirements and additional benchmarks. This section also identifies highlights of the proposals from the viewpoint of the authors. We conclude in Section 5 with a set of recommendations for the final QVT standard and give a brief outlook on current work in Section 6. The goal of this paper is not to provide yet another QVT proposal, but to capture the best of the existing proposals combined with the practical experience of the authors.

2 Terminology

The QVT RFP introduces some terminology which we clarify here. Further terminology is introduced in the submissions themselves. In this section, we provide a unified set of definitions for QVT -related terminology, which enables the proposals to be compared. We begin with definitions of the terms *query*, *view*, and *transformation*, which are fundamental to the RFP.

Query A query is an expression that is evaluated over a model. The result of a query is one or more instances of types defined in the source model, or defined by the query language. An example of a query over a UML model might be: *Return all packages that do not contain any child packages.* The result would be a collection of instances of the Package metaclass. A further example of a query over a UML model might be: *Does a particular attribute in the source have public visibility?* The result would be a Boolean value.

The Object Constraint Language (OCL) [5] is an example of a query language. Queries can also be constructed using a UML Action Semantics (as defined in UML 1.5 or UML 2) [5].

View A view is a model that is completely derived from another model (the base model). A view cannot be modified separately from the model from which it is derived. Changes to the base model cause corresponding changes to the view. If changes are permitted to the view then they modify the source model directly. The metamodel of the view is typically not the same as the metamodel of the source.

Views are typically not persisted independently of their source models (except perhaps for caching). Views are often read only. Where views are editable a change made

via the view results in the corresponding change in the base model. It is therefore necessary for an editable view to have a defined reverse mapping back to the base model. A view may be partial, that is based on a subset of the source model. A view may be complete and have the same information content as the source, but reorganized for a particular task or user. A query is a restricted kind of view. Views are generated via transformations.

Transformation A transformation generates a target model from a source model. Transformations may lead to independent or dependent models. In the first case, there is no ongoing relationship between the source and target model once the target has been generated. In the second case, the transformation couples the source model and target model.

A transformation may be *top-down*, in which case the target model is not modified after generation; changes are always made to the source model and propagated to the target via the transformation. Transformations may be *one-way* (unidirectional), in which case additional information may be introduced to the source model after the application of a transformation. Repeated application of the transformation should not overwrite any information introduced to the target model. Transformations may be *two-way* (bidirectional), in which case each model may be modified after the application of the transformation; changes must be propagated in either direction. In some cases, changes may have been made to both models. If this is permitted then there is the possibility of conflicting changes having been made. In this case, it is necessary to detect such conflicts, but it may not be possible to resolve them automatically. A transformation in which the target model replaces the source model is referred to as an *update* transformation.

When discussing bidirectional transformations we adopt the terms *left-hand model* and *right-hand model* to reflect the symmetry of the relationship. Both models act as source and target for transformations.

A view is a restricted kind of transformation in which the target model cannot be modified independently of the source model. If a view is editable, the corresponding transformation must be bidirectional in order to reflect the changes back to the source model.

The RFP requested a declarative approach to transformation rather than an imperative approach. A number of the proposals challenged this requirement so it is useful to introduce the terminology relevant to both approaches. The following two definitions have been taken from the Free Online Dictionary of Computing [4].

Declarative A general term for a relational language or a functional language, as opposed to an imperative language. Imperative (or procedural) languages specify explicit sequences of steps to follow to produce a result, while declarative languages describe relationships between variables in terms of functions or inference rules and the language executor (interpreter or compiler) applies some fixed algorithm to these relations to produce a result. The most common examples of declarative languages are logic programming languages such as Prolog and functional languages like Haskell.

Imperative *Any programming language that specifies explicit manipulation of the state of the computer system, not to be confused with a procedural language.*

For the purposes of comparing the proposals, we also introduce the category of a *hybrid* transformation in addition to pure declarative and pure imperative approaches.

Hybrid A combination of declarative and imperative constructs to define transformations. Typically a declarative approach is used to select rules for application and an imperative approach is used to implement the detail of rules that are not completely expressed declaratively. This issue is further examined in Section 4.

The following terms are used in the definition of transformations.

Rule Rules are the units in which transformations are defined. A rule is responsible for transforming a particular selection of the source model to the corresponding target model elements. A transformation is specified via a set of rules. Composition mechanisms for rules may be defined. A rule may contain a declaration and/or an implementation. A pure declarative rule will contain only a declaration, a pure imperative rule will contain only an implementation, and a hybrid rule will contain both.

Declaration A declaration is a specification of a relation between elements in the left-hand and right-hand models. A declaration may contain sufficient information to fully describe the transformation from left to right (unidirectional), the transformation from right to left (unidirectional), or both (bidirectional). Alternatively, a declaration may only be able to constrain the left and right sides to determine when an associated implementation should be invoked. Note that in a bidirectional transformation, elements from the right and left sides are available when the declaration is evaluated.

Implementation An implementation is an imperative specification of how to create target model elements from source model elements. An implementation explicitly constructs elements in the target model. Implementations are typically directed, i.e., they operate from left to right or from right to left. However, implementations that can operate in either direction are possible and permitted.

Match A match occurs during the application of a transformation when elements from the left-hand and/or right-hand model are identified as meeting the constraints specified by the declaration of a rule. A match triggers the creation (or update) of model elements in the target model, driven by the declarative and/or implementation parts of the matched rule.

Incremental Transformation If individual changes in a source model can lead to the execution of only those rules which match the modified elements, then the transformation is said to support *incremental* transformation.

3 Common Transformation Scenarios

The authors of this paper are involved in two closely-related projects that apply model-driven transformations. In the following, we discuss common scenarios that a transformation language must address based on our practical experience of implementing transformations.

Simple transformations A simple transformation is one that transforms single elements in the source model into single elements in the target model. Quite often, the source and target models have essentially the same structure. Many of the examples used in the QVT proposals are of this kind. A typical example is a transformation from UML classes, attributes and operations to Java classes, fields and methods. This type of transformation is straightforward to implement imperatively, and should be amenable to declarative approaches.

Expressions Transformations may have to handle string expressions in the source or target model. Where a metamodel for the expression language exists, expressions can be treated as instances of that metamodel and require little special support. However, in many cases it may be unnecessary to fully parse the expression, and some simple support for transforming text is sufficient.

The authors of this paper have worked on transformations with BPEL4WS [3], the Business Process Execution Language for Web services, as the major target. The transformations we have implemented from UML to BPEL4WS [1] and from business view models to BPEL4WS and Adaptive Entities [10] have several examples of expression handling, including transforming source expressions into equivalent target expressions, into elements, or into attribute settings on elements.

Naming Another common situation requiring text handling occurs when the source and target models have different restrictions on naming, or different naming conventions. For example, UML allows many characters in names that Java does not allow. This must be handled in some way when generating Java from UML. One approach is to automatically mangle names in a standard way to make them valid. Any references to those names that occur elsewhere must also be mangled for consistency. Alternatively, the Java naming restrictions can be enforced on the UML model.

Complex transformations This type of transformation builds structures in the target model which do not directly correspond to any individual element in the source model. The transformations may be based on complex algorithms and heuristics.

The transformations we have implemented convert unstructured activity graphs or process graphs into structured BPEL4WS activities. They partition the graphs into subgraphs, based on the control flow and other criteria, and build a well-structured BPEL4WS process. This sort of transformation can be difficult to describe declaratively.

Regeneration and reconciliation Having used a transformation to generate a target model, it is likely that the user will want to modify the generated output. When subsequently the source model is also changed, it would be desirable if the transformation can be reapplied while maintaining any changes the user has made. Trying to maintain user changes may lead to conflicts, which must be resolved—perhaps by asking the user what should be done.

Transformation from partial source models It is often useful to be able to generate a partial target model from a partial source model. For example, in the UML to BPEL4WS transformation described in [1], behavior is described using activity graphs, which can be transformed into executable BPEL4WS. During top-down development, an activity graph can be generated that contains named activity nodes with control flow but does not yet have all details of the actions within the activities elaborated. Such a model contains sufficient information to be able to generate a skeleton of a BPEL4WS document, which is useful for modelers who are familiar with BPEL4WS. It is also useful in scenarios where users are permitted to add information to either a UML model or its corresponding BPEL4WS document.

Resilience to errors The occurrence of an exception during transformation execution should not halt the transformation, i.e., instead of simply aborting, it should be possible to generate a partial model. Rules that are not affected by the error in the source model should be executed as usual, resulting in a partial target model. This approach allows multiple errors to be detected in a single pass. The requirement of transactional behavior of transformation rules is directly related to this feature.

M-to-N transformations It cannot be assumed that there is a one-to-one correspondence between source and target models. The transformations mentioned above take in a source model and generate multiple target models from three different metamodels (BPEL4WS, WSDL and XSD in one case, and BPEL4WS, WSDL, and SACL—a metamodel to specify state machines—in the other case). Note that, even if it is possible to split such a transformation into multiple one-to-one transformations, this may be an inefficient implementation. Support for one-to-many transformation is particularly valuable in cases where the resulting models will be interdependent and must refer to elements created during the transformation. Transformations combining models that represent various aspects of a problem are likely to be many-to-one. In the general case, many-to-many transformations must be supported.

4 Summary of the Submissions

The QVT standard is considered essential to make Model-Driven Architectures a success. The following general requirements were formulated:

– The proposals should be precise and functionally complete, but also minimalistic. Compliance points should be specified and existing standards should be reused whenever possible.

– The proposals should be compatible or clearly specify the changes and/or extensions they make with respect to existing OMG specifications.
– The proposals should be implementation independent, address security issues where needed and specify the degrees of internationalization.

The MDA Technical Perspective states that relationships between models should be defined at the same level of abstraction as their metamodels defined in MOF. Given that all models will be represented in MOF, a single transformation language for all MOF models is possible and should be formulated in the proposals. Mappings to any non-OMG language should be obtainable by defining a MOF metamodel for such a language. Transformations are defined as mappings and a unique transformation language is considered to play a role similar to the role XSLT plays for XML representations. Queries are required to filter and select elements from a model similar to XPATH that is used to select elements from an XML model. Views are considered as models derived from other models. Although the RFP initially calls for views on metamodels (see [7] and Section 2), the remaining RFP document discusses views of concrete models that reveal specific aspects of a modeled system, not the metamodel.

Apart from these general requirements that are applicable to almost any standard, more specific requirements are formulated that address QVT -specific issues:

– Proposals should define a language for transformation definitions that can be declaratively represented in MOF, i.e., a transformation is a MOF model itself. The language must be expressive enough to express all required information to automatically generate a target model from a source model. The transformation language should also allow one to formulate and create a *view of a meta**model*, but in general, all mechanisms should operate on *model*, i.e., *instances* of metamodels defined in MOF 2.0.
– A transformation should support the propagation of incremental changes occurring in one model to the other model. Although singleton sets of models are the main focus, it should also be possible to transform multiple models into each other.

The optional requirements summarized below are also very interesting and was given considerable attention in many of the submissions:

– Transformations should be defined symmetrically and go beyond the simple source-to-target approach.
– Transformations should be able to implement updates, i.e., the target model is the same as the source model.
– Inverse transformations should be definable such that $T^{-1}(T(M)) = M$.
– The proposed languages should support the traceability of transformation executions. Transformations with a transactional character should be definable (commit, rollback), which would prevent an invalid (not well-formed) model resulting from a transformation that has failed during execution.
– A transformation language should support the reuse and extension of generic transformations. It should provide mechanisms to inherit and override transformations and the ability to instantiate templates or patterns.

– The use of additional transformation data not contained in the source model but that parameterize the transformation process should be possible.

The OMG Action Semantics specification [5] is mentioned as already having a mechanism for manipulating instances of UML model elements. Submitters are thus also requested to discuss how their proposal relates to the UML Action Semantics. In response to the RFP, 8 proposals were submitted and are listed at the OMG web site [7]:

1. Adaptive Ltd. (in the following abbreviated with ADAPTIVE)
2. DSTC/IBM (abbreviated with DSTC)
3. Compuware Corporation/Sun Microsystems (SUN)
4. Alcatel/Softeam/TNI-Valiosys/Thales (THALES)
5. Kennedy Carter (KC)
6. TCS, which comprises Artisan Software, Kinetum, Kings College, and the University of York (TCS)
7. Codagen Technologies Corporation (CODA)
8. Interactive Objects Software GmbH/Project Technology (IO)

In the following, we will briefly summarize the various proposals and highlight the strengths and weaknesses given the following criteria:

– Ease of technical adoption, i.e., could we, as implementers of transformations, apply a proposed solution to solve our transformation problems,
– Scalability of the proposed solution in terms of size and complexity,
– Completeness of the proposal and readability/self-containedness of the documentation.

We also considered the *benchmarks for mapping approaches* as they were developed in [8], because these also express some of our own requirements:

– Bidirectional mappings should be supported.
– Both directions of the mapping should be captured in one definition.
– It should be possible to define rich well-formedness conditions on mappings.
– Mappings should be easy to understand and write.
– Mapping definitions should facilitate the construction of tools, which
  - given a source and a target model and a mapping, decide whether the source/target pair is a valid instance of the mapping,
  - provide automated support for reconciliation of dynamically changing models to prevent inconsistencies from occurring,
  - allow one side of a mapping to be partially be generated if a complete generation is not possible in a fully automatic manner,
  - interoperate with modeling tools.

We would like to mention that we found many submission documents to be incomplete and sometimes so unclearly formulated that we cannot guarantee that the following review exactly matches the intentions of the submitters. To the best of our efforts, it is based on our understanding of the submissions that we were able to derive from the available documents.

4.1 Queries

All proposals regard queries as a required necessary part of a transformation, i.e., a query determines when and how a transformation (rule) is applicable to a model (or a set of models) and how the result of the transformation will be built. Several submissions propose the use of the OCL 2.0 language: IO, SUN, TCS.

IO defines queries as a means to perform model analysis. Executing a query can be considered as a specific transformation task, since it only returns elements from the model. Note that this understanding is more limited than the definition we spelled out in Section 2. IO also discusses that queries must be able to cross model boundaries when used during a transformation, i.e., a query must be able to refer to the source and target model as well as to a transformation's execution history.

THALES proposes an extension to OCL called TRL. A query can return not only elements from the queried model as an answer, but can also return a composite type (e.g., a tuple or collection) or a more complex type defined by some metamodel.

The CODA submission proposes an extension to XQuery and XPATH, called MTDL, as the query language. It is defined similar to a relational data base query. A SelectStatement contains a PathExpression containing several PathSteps, each of which can refer to a source or target model transformation path to allow "the navigation over source (or target) elements from the current element". Within the context in which it is written, we understand this as being similar to the IO proposal of referring to the execution history of a transformation.

KC proposes the UML Action Semantics [5] for queries and to use its Action Semantics Language ASL [11] as a concrete syntax. In our opinion, this amounts to using any arbitrary programming language as a query language. We would therefore not agree with the KC statement that this submission proposes a fully declarative solution, but consider it an imperative solution.

The ADAPTIVE proposal contains no proposal for a query language, but only addresses the problem of views.

The DSTC submission regards queries as specific transformations and builds on the fact that the transformations are themselves represented as models. The query model is thus a subset of the transformation model, which is based on F-logic [9]. The submitted document does not explicitly state which subset of the transformation model constitutes the query model, but the transformation model contains a Query object, which associates Pattern Definitions and inherits a Variable and Pattern Scope. Queries are considered to return existing elements from a model, not to construct new elements.

Figure 1 summarizes our understanding of the submissions along two important dimensions that we identified. On the X-axis, we distinguish whether the query is selective, in the sense that it can only return elements from the queried model, or constructive, in the sense that it can return other elements/values as well. On the Y-axis, we distinguish whether the query language is fully declarative, is declarative but allows reference to the execution history, or is imperative. Since we classified KC as imperative, we also believe it is constructive due to the flexibility of the Action Semantics.

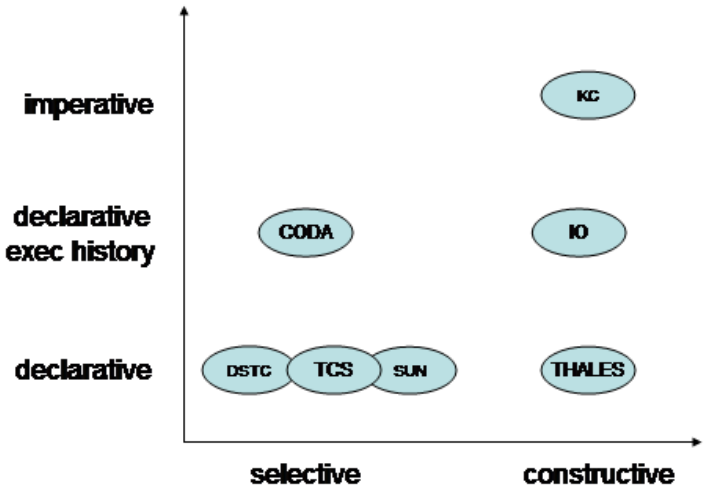


Fig. 1. A classification of the submissions for queries.

4.2 Views

Most of the submissions link views very closely to queries and/or transformations. The only exception is the ADAPTIVE submission, which proposes a portal-based approach. Unfortunately, however, the document has formatting problems and appears very incomplete, containing only eight, partially empty pages, so we cannot say much about it. Two angles of understanding of a view can be observed[3]:

- A view is produced as the result of a query, i.e., a view is the visualization of the query answer (CODA).
- A view is the result of a transformation (THALES, KC, TCS, IO, SUN, DSTC).

Consequently, the proposals carry over their solutions to transformations and/or queries to the problem of creating a view of a model. The CODA proposal defines views as the result of queries, which are represented in MTDL. Transformations can be done on the model itself or on any view of it. THALES defines views as a projection on a parent model created by a transformation. Views, like queries, are expressed in the TRL language. KC again proposes the use of the UML Action Semantics. A view is considered a transformation in which the target is completely derived from the source. SUN regards views as specific transformations represented in XMOF. In the TCS submission, a view is a projection on a parent model created by a transformation. IO defines a view as a specific transformation ("an abstraction") where viewpoints will be created by model editors that operate only on a subset of a metamodel. The DSTC submission also considers views as specific transformations, but emphasizes an important difference:

> *"The only difference between a transformation and a view is the underlying implementation. For a transformation, the target extent is independent of source*

[3] However, this difference is marginal in the sense that all proposals regard queries as an integral part of a transformation.

extent; its objects, links and values are implemented by storing them. For a view, the target extent remains dependent on the source extent; its objects, links and values are computed using the source extent. The definition of transformations and views is the same (the specification of source and target models and the relationships between them)."

We see the following important difference in the understanding of views—compare this also to our definition of views in Section 2: *Is the view linked to the model or does it have an independent existence and can it be manipulated without necessarily changing the model from which it was created?* Only the DSTC proposal addresses this issue in a clear way and states that views remain physically linked to the model, i.e., any change in the model will also occur in the view if this view contains the changed part of the model. From the other approaches, we could not derive a clear answer to this question. The CODA proposal even seems to imply that models and views can be manipulated independently of each other.

4.3 Transformations

In a nutshell, all proposals (except ADAPTIVE) adopt a unifying solution to queries, views, and transformations. In four submissions, exactly the same language is proposed to solve all three aspects: KC proposes the UML Action Semantics, THALES proposes TRL (an extension of OCL 2.0), DSTC proposes F-Logic [9], CODA proposes MTDL (an extension of CMOF).

The other submissions (IO, SUN, TCS) propose using OCL 2.0 [5] for queries. The queries are used inside transformations to determine when a transformation is applicable. The transformation languages are separate definitions comprising different elements and formalisms. Following below, we evaluate the various proposals along our previously introduced criteria:

Self-containedness Most documents are not really self-contained. KC and DSTC essentially refer to other languages described elsewhere. All other proposals show the metamodels of their languages, but usually omit a detailed description of the semantics. The examples given are either highly simplified or nonexistent. Therefore, in many cases, one can only obtain a very limited picture of how a transformation would be represented and executed.

Scalability We distinguish scaling behavior in terms of the size of transformation models and scaling behavior in terms of the complexity of the transformations that can be expressed, cf. the scenarios discussed in Section 3. Declarative proposals that assume a uniform rule base (DSTC, IO) can be assumed to scale worse than proposals that introduce a structured transformation base (TCS). The complexity of the transformations that can be expressed is determined by the expressivity of the transformation language. KC's proposal to use the UML Action Semantics yields the most expressive solution since it allows an escape to arbitrary programming languages. Similar code escapes are provided by IO, SUN, and TCS. In contrast to this, THALES, CODA, and DSTC propose declarative, decidable languages, which are less expressive, but should have

advantages in terms of tooling support (e.g., deciding consistency of transformations may only be possible for them, not for the others).

Many approaches assume that the transformation rule set can be structured to some extent. The most detailed proposal for a structuring of the rule set comes from TCS. Rules in the TCS submission have a state and they can specialize other transformations or be defined as composites. A rule can transform between an arbitrary number of domains (classes, associations, packages), but there must be unique names for all attributes and domains. The proposal envisions a hybrid representation for the transformation rules, see Figure 2.

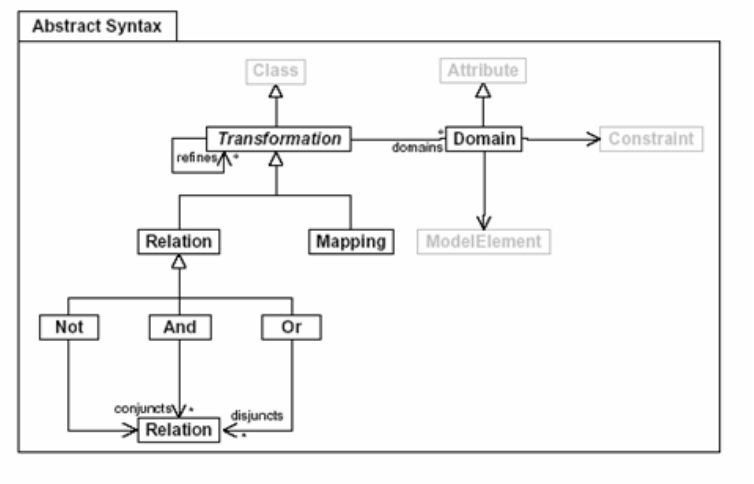


Fig. 2. Transformation rules as relation-mapping pairs in the TCS submission.

The declarative part is called a *relation*, whereas the implementational part is the actual *mapping* that takes place. Relations are multidirectional and nonexecutable. They can be composed of other relations using NOT, AND, OR subrelations and so-called *elaborations*, which, for example, replace an abstract relation by a set of detailed relations. Mappings are executable and can be described in some Actions Semantics Language. A mapping can refine any number of relations, i.e., the same implementation can be reused in several declarative rule definitions.

Transformations are organized into an ordered sequence of steps that define the operational part, cf. Figure 3. Each step is specified by a set of transformation tasks, with each task being defined as a set of relation-mapping pairs. Transformation tasks can be marked as transactional. The traceability of a transformation is achieved via logging the transformation steps.

Simplicity Whether a transformation definition is easy to understand and write depends also on personal preferences. Assuming that transformations will be written by programmers, any programming-like solution using Action Semantics could be assumed to be (fairly) simple. The declarative IO proposal already reports problems in maintaining and completing large rule sets, but not in writing a single rule. Pure logic-based

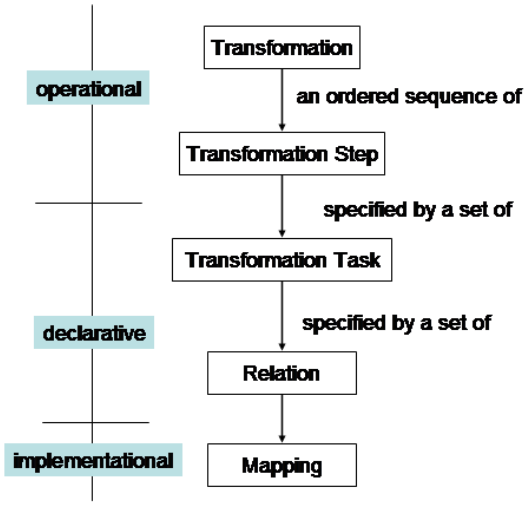


Fig. 3. Structure of a transformation in the TCS submission.

languages (such as proposed by DSTC) will be harder to use by those not experienced in using the formalism.

Bidirectional mappings can be provided in terms of transformation definitions and in terms of transformation executions. The definition of a transformation is usually given as a set of rules, i.e., a single rule is the basic unit of transformation that can be defined and executed. Figure 4 categorizes the proposals along two dimensions: whether an imperative, hybrid, or declarative language is used to write transformation rules, and whether the transformation rules are executable in only one direction (from source to target), which we call unidirectional, or from both directions (from source and target), which we call bidirectional. Some approaches envision mixed types of transformation rules within the same transformation model, i.e., some rules are unidirectional, while others are bidirectional. We classified those approaches as uni/bi.

Various proposals discuss whether the source or target model should drive the execution of a transformation. Note that when rules are bidirectionally executable, the execution direction must be determined, e.g., by selecting one rule side (pattern) that should be matched and the other rule side that should be derived, see for example the SUN submission. The DSTC proposal discusses aspect-driven transformations, which would not be driven by specific MOF elements in the source or target model, but rather by some semantic, representation-independent concept. Although aspect-driven transformations are an interesting idea, it remains open how they would be represented in the QVT standard. Although we got the impression that some proposals favor one side to drive the transformation, others allow the driving side to be arbitrarily specified.

Furthermore, we found assumptions in the submissions that refer to the cardinality of the source and target model sets. There are approaches that assume multiple source models from which one target model is produced, while others envision a single source

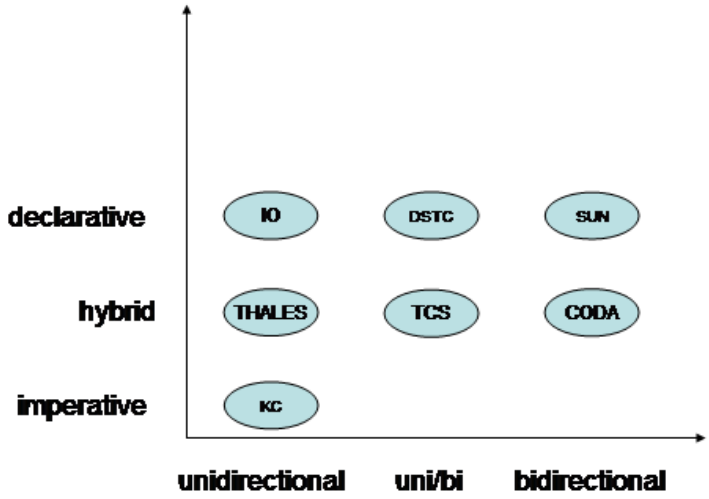


Fig. 4. A classification of the submission with respect to the proposed nature of the transformation languages and their possible execution directions.

model transformed into various targets, or in the most flexible case, many source models can be transformed into many target models. Figure 5 summarizes our findings.[4]

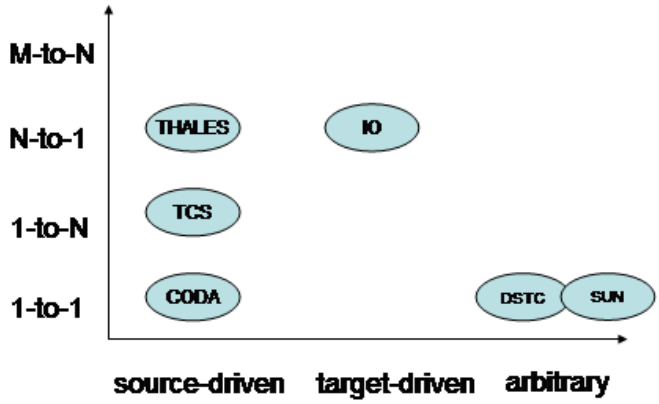


Fig. 5. Drivers and input/output-cardinality of transformations.

Ease of Adoption Given our impression of the current state of the proposals, we assume it to be more or less equally hard to adopt any of them. At the one extreme, one could

[4] ADAPTIVE and KC have been omitted, because they provide no information on this aspect of transformations. In the case of CODA, we are not sure whether it is really source-driven and some unsecurity also remains in the case of SUN and THALES, which could be more general than is apparent from their submission documents.

simply apply one's preferred programming language to adopt the KC proposal, i.e., the transformations that we have implemented in Java already today could be considered as an adoption. However, this is of course not the intention of the KC submission. At the other extreme, we would place the DSTC proposal, of which we think that it can only be fully exploited after having understood the hundred pages long paper [9] or if good tooling support is made available.

Rich Conditions The answer to this criterion corresponds directly to the proposed query language, since all submissions use their query language to formulate conditions in transformations.

Tooling Aspect Many claims are made, but it is hard to tell whether they are met without undergoing a comprehensive evaluation of the tools, which are not easily accessible. The important requirements of model reconciliation, failure handling, consistency checking and model integration are widely discussed, but we could not find a proposal so far that would convincingly demonstrate how these issues are resolved in a tool. The IO submission, for example, defines and checks a constraint before the query in a mapping rule is executed and before the target model is actually modified. Many proposals admit that these issues, although important, are not yet addressed by their submission. Many of the proposals also argue that defining symmetric or inverse transformations or considering them something special is not very meaningful and that the practical relevance of inverse transformations remains unclear. Instead of defining inverse transformations, some proposals define pairs of complementary transformation rules, see for example the IO submission. Security aspects were discussed, but we did not find many concrete proposals. The problem with updating a model is usually considered to be a standard transformation problem and we did not see specific solutions to handle updates of the same model except that one may distinguish between *update rules*, which modify a model, and *create rules*, which generate a new model, e.g., in the THALES submission.

5 Recommendations

This section introduces a set of recommendations for the final QVT standard. The recommendations are based on the highlights of the initial responses to the RFP and the authors' experiences in developing model-to-model transformations.

Support a hybrid approach to transformation definitions: In the experience of the authors, a declarative approach is useful for specifying simple transformations and for identifying relations between source and target model elements. However, many transformations are not straightforward. This is especially true when transforming between languages at a similar level of abstraction, such as horizontal transformations and transformations to middleware platforms that support high-level abstractions. It may not be possible for the target audience of transformation languages to construct complex transformations using a fully declarative approach. An imperative language is preferable for the definition of complex many-to-many transformations that involve detailed model analysis. The following quote by Adam Bosworth [2] supports our recommendation:

Alan Kay is supposed to have said that simple things should be simple and hard things should be possible. It has been my experience over 25 years of software development that for most software products, simple things should be declarative and/or visual and hard things should be procedural. Declarative languages have an unfortunate tendency to metastasize because people need to do things that are hard. When they grow in this way, not only can most people not use these new features, they find the entire language more daunting.

Provide a simple declarative specification language: Also in line with the above quote from Bosworth, we recommend that the language used for declarative specification be simple. A graphical concrete notation for the language is likely to be of value to some user communities. Simplicity is somewhat subjective, but as a guideline, the capabilities of the declarative language should not go beyond the point at which a capable modeler/programmer would find an imperative specification more straightforward to construct, comprehend, and maintain.

Use declarative queries only: Do not allow to link a query to a specific runtime execution of a QVT session. Keep the query language fully declarative. Allow it to return not only elements from the queried model, but also answer types defined by some metamodel. In the simplest case, a query can return a Boolean value even if Booleans are not part of the queried meta model. It should be discussed whether a query statement should be decidable over a model. In this case, the full expressivity of programming languages should not be allowed, but languages such as OCL should be preferred. The recommended space is shown in Figure 6.

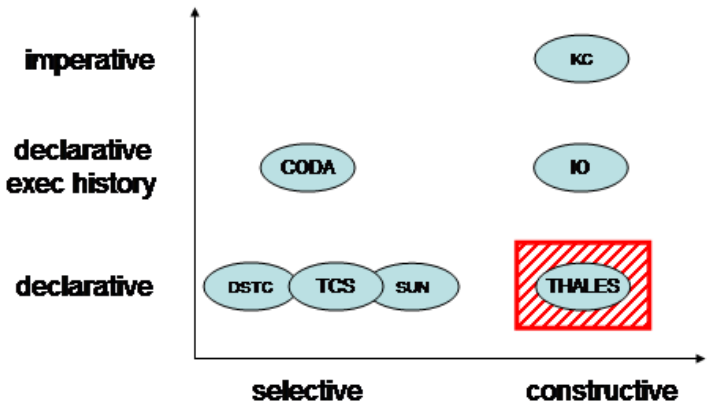


Fig. 6. A recommended space for query languages.

Provide an abstract syntax for the transformation language: The transformation language should have a defined abstract syntax for composition, declarative, and imperative parts. Where possible these should be based on existing OMG standards.

The proposal may specify an example concrete syntax, but should permit other declarative and imperative languages to be plugged in via transformations to the abstracts syntax specified in the proposal.

Adopt common terminology: In Section 2, we put together a set of definitions that provide a unifying view on the major concepts occurring in the QVT space. We recommend that a final standard precisely define a common terminology. In general, we assume that the final standard specification will be a complete, self-contained document.

Use the Action Semantics as an interchange format: Imperative parts of rules should be exchanged via UML Action Semantics. The UML 2.0 Action Semantics will be appropriate within the time scale of this RFP. This will provide a standards-based interchange format for imperative specifications of transformation behavior while permitting the use of particular concrete syntaxes that are appropriate for use in particular development environments. Such concrete languages could include textual or graphical concrete syntaxes for UML Action Semantics, e.g., the ASL [11], imperative languages specifically designed for the specification of rules (such as the TRL language proposed in the THALES submission), or programming languages such as Java where a mapping to UML Action Semantics is provided.

Support symmetric rule definitions: It should be possible to describe symmetric rules that can be executed left-to-right, right-to-left, or in a reconciliation mode where both source and target models exist and either may have been modified. Symmetric rule definitions facilitate the development of bidirectional mappings, avoid the duplication that occurs when a rule and its inverse are defined separately, and provide a useful starting point for reconciliation. Symmetrically defined rules may contain direction-dependent implementation parts that are used when the rule is executed in the corresponding direction. Figure 7 shows the space that we recommend for the representation of rules.

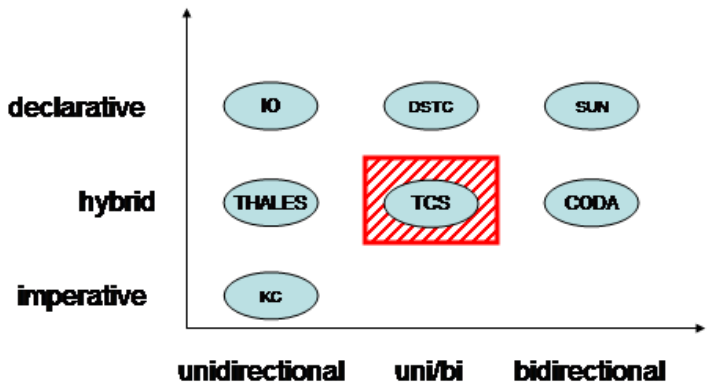


Fig. 7. A recommended space for rule representations.

Support composition and reuse: It is often valuable to be able to construct a complex transformation from multiple intermediate transformations, either because some of the subtransformations already exist, because the intermediate results are of interest, or to decompose the problem into simpler steps. The proposal must support composition and packaging mechanisms to support the development of large transformations and systems constructed from multiple transformations. These mechanisms should come from UML. A transformation definition should be an executable UML model. The generalization and templating capabilities of UML should be considered for rules and transformations. Furthermore, the transactional behavior of composed transformations is another issue that deserves deeper exploration.

Support complex transformation scenarios: The transformation scenarios described in Section 3 should be supported by the adopted standard. The requirements have all arisen from practical experiences with the implementation of transformations. Figure 8 shows the space for rule executions that we recommend to offer the largest possible flexibility in supporting complex transformation scenarios.

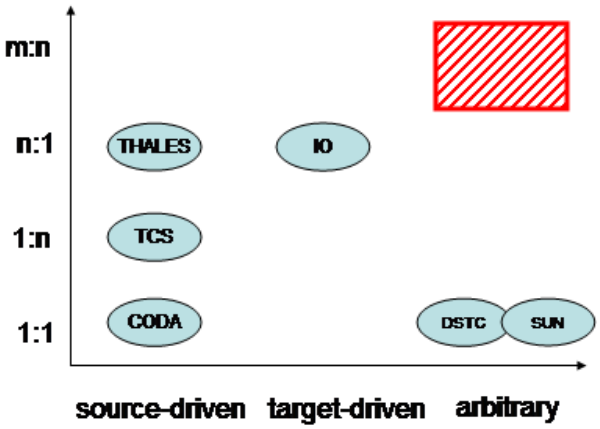


Fig. 8. A recommended space for rule executions.

Provide complete examples: The final proposal should include a number of complete nontrivial transformation specifications. These should preferably be standards-based transformations of value to the MDA community. Good examples could be a reversible mapping between (a subset of?) Java and UML 2.0 Action Language or an implementation of a MOF 2.0 package merge (a many-to-one mapping).

Establish requirements on transformation executions: We found the problem of errors during a transformation and the optional requirement of transactional transformations to be important in order to achieve robust transformations in practice. A standard should clarify this issue and provide a solution. Furthermore, the question of how large transformations will be handled should be addressed, i.e., how will transformations work in two directions, how will incremental changes of models and “life

model synchronizations" be supported, and how will associations between source and target model elements be established?

Emphasize the tooling aspect: We found the definition of use cases useful to derive requirements that a tool should satisfy and to show how the standard can support the scenarios described in the use cases. Usability and ease-of-use aspects seem to be of critical importance. Another important issue seems to be the consistency and completeness problems that should be further clarified: How does a transformation designer know that his rule set is consistent and will produce a valid target model? How does he know that his rule set is complete and will produce a complete target model? Are the results of rule executions deterministic or can different outcomes occur? If yes, how would we deal with that? What would it mean?

6 Conclusion

This paper has provided a complete review of the submissions to the OMG's Request for Proposals (RFP) for MOF 2.0 Query, Views, and Transformations (QVT) with respect to the requirements stated in the RFP and the requirements of the authors of this paper who anticipate being users of the language to be defined in the final QVT standard.

We have introduced a terminology to clarify the major concepts occurring in the QVT space based on the usage in the submissions but edited for consistency. A set of common transformation scenarios is described. It specifies features that the final QVT standard must support in order to enable the implementation of transformations of value to the authors. The submissions are classified along various benchmarks and criteria that we considered to be of particular importance such as the expressivity of the query language, the scalability of transformations, the simplicity of transformation definitions, and the ability to flexibly execute transformations. We have also discussed nonfunctional issues, in particular the usability of the proposed language. If the QVT standard is to be widely implemented, the adopted transformation language must be usable by the target audience.

Our current work focuses on the development of an architecture to implement QVT - based transformations based on the experience gained while conducting the reviewing work described in this paper. We also further evaluate the applicability of F-logic to the transformations we are interested in, because it is at the heart of the proposal supported by IBM.

The authors hope that this paper will act as a useful overview of the submitted proposals and as a transformation implementers' view on the requirements for a successful MOF 2.0 Query, Views, and Transformations standard.

References

1. J. Amsden, T. Gardner, C. Griffin, S. Iyengar, and J. Knapman. UML profile for automated business processes with a mapping to BPEL 1.0. IBM Alphaworks http://dwdemos.alphaworks.ibm.com/ wstk/ common/ wstkdoc/ services/ demos/ uml2bpel/ docs/ UMLProfileForBusinessProcesses1.0.pdf, 2003.

2. A. Bosworth. Data routing rather than databases: The meaning of the next wave of the web revolution to data management. In *Proceedings of the 28th VLDB Conference*, http://www.cs.ust.hk/vldb2002/VLDB2002-proceedings/, 2002.
3. F. Curbera et al. Business process execution language for web services. www-106.ibm.com/developerworks/webservices/library/ws-bpel/, 2002.
4. FOLDOC. FOLDOC free on-line dictionary of computing. http://wombat.doc.ic.ac.uk/foldoc/.
5. The Object Managemant Group. MDA specifications. http://www.omg.org/mda/specs.htm.
6. The Object Managemant Group. OMG model driven architecture. http://www.omg.org/mda.
7. The Object Managemant Group. OMG MOF 2.0 query, views, transformations request for proposals. http://www.omg.org/techprocess/ meetings/ schedule/ MOF_2.0_Query_View_Transf._RFP.html.
8. S. Kent and R. Smith. The bidirectional mapping problem. *ENTCS*, 82(7), 2003.
9. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.
10. J. Koehler, R. Hauser, S. Kapoor, F. Wu, and S. Kumaran. A model-driven transformation method. In *Proceedings of the 7th International IEEE Conference on Enterprise Distributed Object Computing (EDOC)*. IEEE Press, 2003.
11. I. Wikie et al. ASL language level 2.5. Technical report, Kennedy Carter, 2003.

Invited talk: Executable Meta-Modelling - How to turn MOF into a Programming Language

Tony Clark

Xactium Ltd
tony.clark@xactium.com

Abstract. UML is a static notation for describing dynamic systems and MOF is a static notation for describing UML. Behaviour is expressed in UML on static models by interpreting them in a particular way. However, since UML has no dynamic meaning, it is not possible to interact with a model in any dynamic sense. In many cases, the utility of UML would be increased if we could execute it. This talk will discuss how UML can be turned into a programming language by adding simple execution primitives to MOF. The resulting MOF based meta-programming language can be used to define a family of meta-interpreters each of which provides a dynamic semantics for corresponding modelling languages.

Eclipse as a Platform for Metamodelling Tools

Catherine Griffin

IBM UK Laboratories, Hursley Park, Winchester, SO21 2JN, UK
catherine_griffin@uk.ibm.com

Abstract. Eclipse is an open platform for tool integration built by an open community of tool providers. The OMG, IBM, Borland, and other MDA tools vendors are members of the eclipse.org consortium. Eclipse would be an ideal platform for the next generation of metamodelling and MDA tools.

1 What is Eclipse ?

Eclipse is an open-source development project dedicated to creating a world-class tools platform, a framework for building highly integrated development tools.

Eclipse platform development is organized in four main projects:

- Eclipse – the base Eclipse platform and Eclipse development tools
- Eclipse Tools – common tools components
- Eclipse Technology – research and education activities
- Eclipse Web Tools Platform – tools for web application server development

The Eclipse project produces the Eclipse software developer kit (SDK), which is a Java integrated development environment built on the Eclipse platform, and provides a complete development environment for Eclipse-based tools.

Most of the Eclipse platform is licenced under the Common Public License, which encourages open-source development while allowing tool developers flexibility and control over their software technology.

Development of the Eclipse platform is overseen by the Eclipse consortium, eclipse.org. Members of the eclipse.org Board of Stewards are software development tools vendors committed to build products on the Eclipse platform. There are currently around 40 eclipse.org members, including IBM, Borland, Oracle, SAP, TeleLogic and the OMG. The OMG joined eclipse.org in December 2002. This quote is from Richard Soley, Chairman and CEO, OMG:

> Openness is the key to widely adopted, interoperable and portable implementation. I am delighted to represent OMG on the Eclipse Board in order to continue to be part of the open source movement that OMG so strongly believes in. Eclipse and other key open source systems often implement OMG standards – including the Model Driven Architecture, which is represented in the Eclipse Modeling Framework. I look forward to contributing to Eclipse as a member of the Board

For more information, go to http://www.eclipse.org

2 Eclipse Modeling Framework

The Eclipse Modeling Framework, usually known as EMF, is an open-source lightweight implementation of OMG MOF integrated with the Eclipse platform. It includes metamodel generation from UML models, XMI 2.0 serialization, reflection APIs, and customizable code generation of metamodel and editor implementations.

The EMF metamodel was originally based on MOF 1.4, but the more complex concepts were removed to make it easier to implement and use, so to avoid confusion, the EMF metamodel is called *Ecore* rather than MOF.

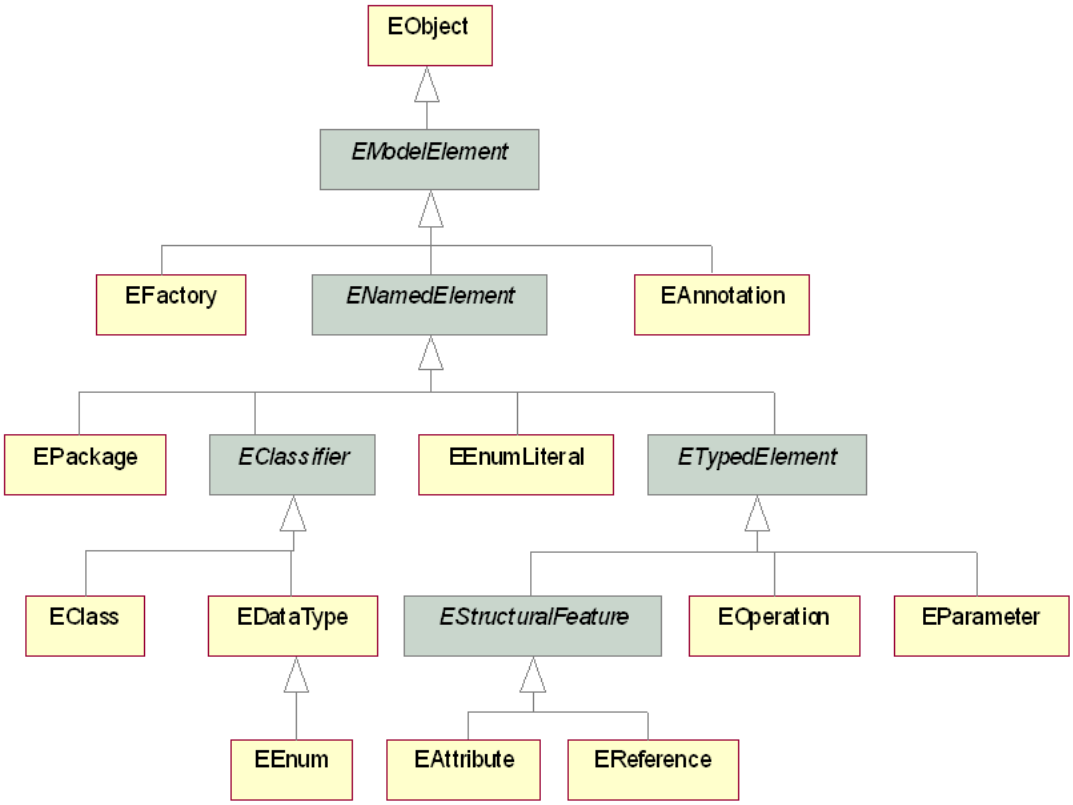


Fig. 1. Ecore Class Hierarchy

The designers of EMF have contributed to the MOF 2.0 Core specification, and as a result the EMOF subset of MOF 2.0 is very similar to Ecore, although there are some minor differences.

EMF has been developed principally for use in Eclipse-based tools and is not based on the concept of a metadata repository. Usually, EMF models are persisted as XMI 2.0 or XML files in the Eclipse workspace.

EMF is developed under the Eclipse Tools project. The Hyades test framework (another Eclipse Tools project) is implementing the OMG's UML testing profile using EMF, and it has also been used in many IBM products over several years.

3 Extending the Eclipse Platform

Eclipse-based tools consist of *plug-ins* - extensions to the Eclipse platform. The Eclipse plug-in architecture allows plug-ins to define *extension points* that other plug-ins can extend. This architecture is fundamental to understanding Eclipse – the Eclipse platform is itself built entirely out of plug-ins on top of a small core runtime.

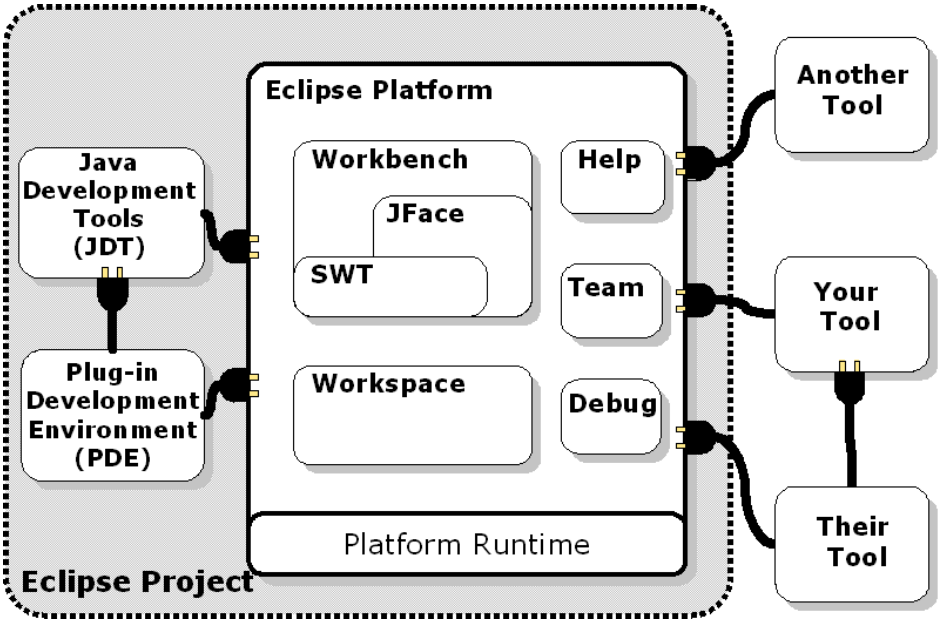


Fig. 2. Eclipse Platform Architecture

Developers building tools for the Eclipse platform start from a proven base of common infrastructure, allowing them to develop better tools more quickly.
More importantly, they can extend that infrastructure with their own frameworks, so that other developers can build on top of what has already been done rather than re-inventing the wheel.

4 A Platform for Model Driven Architecture

The Eclipse project demonstrates that an open platform with a recognized set of common infrastructure components supports integration, accelerates development and promotes acceptance.

Realizing the vision of MDA will require not just new tools, but a new integrated development environment, bringing a model-centric approach to the whole application

lifecycle. To be useful, MDA tools must be integrated with each other and with existing tools. The simplest way to enable this level of integration is to base tools on a common platform, using common infrastructure components.

There are a growing number of MDA related Eclipse-based tools, both commercial and open-source, but as yet the level of integration between them is low because of a lack of common infrastructure. Standards-based, best of breed metamodelling infrastructure in a common platform would accelerate the development of new tools, enabling more rapid innovation and progress.

A new Eclipse Technology project, GMT (Generative Model Transformer) is aiming to develop tools for model transformation and code generation, and together with EMF, this could be the starting point for an MDA tools platform. More projects like this are needed to provide a foundation of common MDA infrastructure to be re-used in tools.

5 Conclusion

By building tools based on Eclipse, developers can contribute components to a common platform for MDA.

The whole MDA community will benefit because different groups can specialize in the areas where they have the most experience, duplication of effort will be avoided, and tools will be based on a commercial-quality common infrastructure.

Tooling Metamodels with Patterns and OCL

D. H. Akehurst, O. Patrascoiu

University of Kent at Canterbury
{D.H.Akehurst, O.Patrascoiu}@kent.ac.uk

Abstract. Computing is moving to a new paradigm where models are first class artefacts. Metamodelling is one of the foundations of this future. However, it is all very well to have metamodels and languages with which to define them (i.e. MOF), but what should we do with them once they are defined? One task should be to populate the model described by the metamodel definition and ensure that the well-formedness constraints are correctly specified; another task may be to create a tool based on the metamodel. In order to enable experiments with variations in the metamodel an automated approach to building such tools is required. Judicious use of patterns can facilitate automatic generation of such tools. The ability to auto-generate a tool from a metamodel definition facilitates experimentation and general confidence in the definition of the metamodel. The tool generated can be subsequently used as the foundation for a more functionally rich hand-coded tool.

1 Introduction

Models are becoming 'primary artefacts' in modern computing. The Object Management Group's (OMG) Model Driven Architecture (MDA) [1] strategy envisages a world where models play a more direct role in software engineering and production. Metamodelling is a key facility in this new era; it provides the facilities to support the design of models, i.e. a metamodel is a model of a model. The OMG has defined the Meta Object Facility (MOF) [2] as a language for defining metamodels; but what do we do with a model specification (an expression in MOF) once it is defined and what support can be provided to aid the specification of models?

- Can we check that a population meets the well-formedness constraints?
- Can we implement the specification?
- Can we provide tools to support the defined model?

The aim of this paper is to demonstrate that, the use of programming patterns and the provision of support for the Object Constraint Language (OCL) [3], enables us to say yes to all of these questions.

OCL provides the Unified Modelling Language (UML) with a navigation and constraint expression facility. By implementing this language over the metamodel specification language MOF, a sub set of UML, we provide a facility to check the well-formedness constraints that typically form part of many model specifications. In addition the OCL implementation provides a very useable *query language* [4] for exploring metamodel populations.

Modelling raises the level of abstraction at which computing is done; programming patterns enable us to map modelling abstractions onto today's 'executable' languages. Traditionally higher level constructs are mapped onto lower level ones using a particular pattern. Much of this has in the past been about behavioural constructs

mapping to a pattern of lower level (assembly instruction) constructs. However, modelling is more of a structural facility; hence we form structural patterns in addition to behaviour patterns. Such patterns are becoming common place, although often implemented by hand. One of the most well-known is that of accessors and mutators as implementation patterns for class properties. Many tools support such patterns in their code generation facilities.

The mapping from metamodel onto implementation can be seen as an MDA exercise requiring a mapping from MOF (PIM) to a programming language (PSM). However, as there is as yet no standard specification language for transformations and more importantly, as the PSM is a programming language, we feel that specifying the mapping in a manner that resembles the target programming language expression, is more appropriate. There are many tools that will support code generation from UML models, e.g. Rational Rose [5], Poseidon [6], Together [7].

However, what we are proposing here is something more than simple code generation. We support generation of a complete framework useable for populating, manipulating and exploring the specified model (or metamodel).

To fully realise the power of modelling in an implementation we need to support complex patterns; such as Visitors, Factories, Repositories, support for bi-directional Associations (opposite ends should always refer to each other), and possibly many others. To achieve this in a fully flexible manner it is necessary to provide a tool that enables the specification of the required patterns in such a way that the instantiation of the pattern takes its parameter values from the specified model.

We already have a language for navigating metamodel specifications – OCL. This language facilitates the extraction of template parameters from the specified model; to provide a full template language we need to add mechanisms to:

a) compose the string values that define the implementation code
b) generate files and directories for storing the implementation code
c) call sub-templates with appropriate parameters

Ideally, an appropriate template language would be designed and specified (or an existing one altered) that incorporates OCL as an expression language. However, as an interim solution, we have found that a few minor extensions to our OCL implementation provide us with the required functions.

By defining templates for a number of interacting patterns we are able to generate, from the specification of a model, a tool that supports:

- Building and manipulating model populations
- Viewing model populations
- Evaluating OCL constraints and expressions over the population
- Provision of persistent storage as XMI [8], HUTN [9] (or other formats).

This is demonstrated through the use of a tool called the Kent Modelling Framework (KMF) [10], developed and used at the University of Kent at Canterbury; the tool has been under development and use for over five years, supported by a number of different research projects. The tool has been used as part of those projects to generate support tools for a variety of different models, such as: the UML metamodel (versions 1.4, 1.5, 2.0); the OCL (abstract syntax model) version 2.0; a metamodel (discussed in [11]) based on the Reference Model for Open Distributed Processing (RM-ODP) [12], as part of the DSE4DS project [13, 14]; and diagrammatic language models, as part of the Reasoning with Diagrams project [15].

In addition, due to the use of OCL as the template language we are able to generate an implementation of KMF using itself.

The rest of this paper is organized as follows. Section 2 described the patterns generated by the latest version of our KMF tool. Section 3 describes how we use OCL as a template language. Section 4 discusses how the generated code is fitted together into a useful support tool for the original model specification. Section 5 discusses existing related work to that presented in this paper. Section 6 concludes the paper with a summary and discussion of future work.

2 The Patterns

The generation of our modelling tools from a metamodel is achieved through the use of a number of interacting programming patterns. Information is taken from the metamodel specification and used to populate the parameters of each pattern. Each of the major concepts from the metamodel, class, attribute, association and package, are used to instantiate a different set of patterns; described in the following sub sections. We currently use Java as the target implementation language; it would be easy to adapt the templates to additionally target other languages.

Patterns have been recognised as a useful programming technique for a number of years. In particular the book by the "gang of four" [16] discusses a number of widely used patterns. Most of our patterns are taken or adapted from those documented in this book.

2.1 Classes

Each metamodel class is implemented using a pattern of interface and implementation class. An interface is constructed to represent each class and the type hierarchy of the class; a key value of this pattern is that the implemented type hierarchy will support multiple inheritance. The name of the interface matches the name of the metamodel class. The implementation class is named after the metamodel class, but with an added extension to the name – such as 'Impl'.

2.2 Attributes and Associations

Each attribute of, and association end that is navigable from, a particular metamodel class is implemented by adding appropriate accessor and mutator methods in both the interface and implementation class. These method signatures follow the standard java pattern as shown below:

```
public <Type> get<CapitalisedName>()
public void set<CapitalisedName>(<Type> <name>)
```

The implementation of attributes and associations differ. Attributes are implemented simply by providing an appropriately typed private variable, which is returned or assigned in the accessor and mutator bodies.

Two alternative patterns are used to implement an association end. The choice between patterns is based on whether or not the association end (or its counter part at the other end of the association) is marked as 'navigable'.

If only one end is marked as 'navigable' then the same implementation pattern is used as that for attributes; the class navigated from will contain a private variable returned and assigned by the accessor and mutator.

If both ends of the association are navigable then we must ensure that setting one end will also set the other end appropriately. A variety of programming techniques could be used to achieve this; the simplest being for the mutator implementation to set the opposite end as well as setting this end, i.e.:

```
public void set<CapitalisedName>(<Type> <name>) {
  this.<name> = <name>;
  if (<name>.get<OtherName>() != this)
    if (<name>.get<OtherName>() != null)
      <name>.get<OtherName>().set<Name>(null)
    <name>.set<OtherName>(this);
}
```

This mutator first sets the private variable for this end of the association. The next conditional test is to eliminate a non-terminating recursive loop, without it, each end of the association would attempt to set the other end, which would set the other end, which would set the other end…etc. The second test determines if an existing object is already using the new object as one of the association ends, if so we should remove it (i.e. set it to null). Finally, the other end of the association is set to reference this object as the appropriate end of the association.

This pattern is fine for Association ends with multiplicity of one (or zero-to-one), however, when we have a collection of objects at an association end, something more complex is needed. There are two approaches to implementing this type of association; one is to provide 'Add' and 'Remove' methods on the class that references the collection; the other option is to use accessors and mutators which get and set collection objects.

To ensure the referential integrity of opposing association ends with the first of these implementation approaches, a similar technique can be used to that for single object association ends; the Add and Remove methods can be implemented so as to set the opposite ends of the association.

Performing the same actions when the second approach is used is more complex. Standard Java (or other language) collections are used to implement the model collections and the mutator will allow any collection that implements the collection interfaces to be used as an argument. Our approach to implementing the referential integrity is to provide a separate object that implements an associationEnd. Objects that take on the role of an associationEnd within a model implementation must implement an AssociationEnd interface, the functions of which are delegated to the separate associationEnd object. The AssociationEnd interface is shown below:

```
public interface AssociationEnd {
  Object getOtherEnd(String propertyName);
  void setOtherEnd(String propertyName,
                   Object value,
                   String otherName);
}
```

The accessor and mutator are qualified by a string value in order to distinguish between multiple associationEnd roles supported by an object. The mutator, in addition, takes a value for the opposing association end name as a parameter, so that

the referential integrity actions can be caused. The implementing (delegate) object for this interface performs actions similar to those discussed above, with some variation based on collection objects. Setting a collection object causes a new wrapper collection to be created. This delegates all collection functions to the original collection, but intercepts add and remove methods so as to cause additional actions for setting the other end of the association.

2.3 Packages

There are three patterns instantiated from metamodel packages – Factory, Repository and Visitor. Each of these patterns addresses the classes owned by the metamodel package and addresses any sub packages.

Factory

Based on the traditional pattern for a factory object [16], we provide a variation that scales more easily to large models. The idea of the pattern is essentially to provide an interface for constructors of the classes in a model (Java does not facilitate constructors in an interface specification). A factory interface contains a create method for each (non-abstract) class in the model, the implementation of the factory supports generation of an object of the appropriate class, the returned object being referenced by the implemented interface type (rather than the actual implementation class). We additionally provide a generic create method that takes a (model) class name as parameter.

Our original implementation provided a single factory for a whole model; however we discovered that this approach was not scaleable, causing problems with very large models; and meant that we had to treat the whole model as an entity, where as sometimes we wished to deal with a single package (and sub-packages) of the model. An alternative that we have investigated is to construct a factory implementation class (and interface) for each metamodel class; however, we found that this was unnecessary.

Our preferred evolution of the factory pattern is to provide a factory for each package. These factories are linked in a hierarchy that matches the package structure; any factory can be used to create objects from its own package or any sub-package.

We create an implementation Factory class for each package; the class implements and extends a common Factory interface and implementation which provide common behaviour. The template for the generated class is as follows:

```
public class <pkg_name>Factory extends FactoryImpl {

  public <pkg_name>Factory() {
    <for each subPackage>
     <spkg_name> = new <subFactory_name>();
  }

  <for each subPackage>
   public <subFactory_name> <spkg_name> = null;

  <for each class in this package>
   public <class_name> create<class_name>() {
     return new <classImplName>();
   }
   public void destroy<className>(<className> object) {
     <for each attribute or associationEnd>
```

```
        <if a.multiplicity.oclIsUndefined() then>
          object.<mutator_name>(null);
        <else>
          object.<accessor_name>().clear();
        <endif>
    }
}
```

Repository

A repository provides at its basis a similar function to a factory; it enables the creation of objects (in fact the repository uses the factory to provide this part of its implementation). However, the repository keeps track of all objects created and facilitates operations such as: saving its set of objects – to provide persistence for the model population; returning the set of all objects in the model that conform to (are instance of) a particular type; or deleting objects.

As with the factory pattern, we provide repositories on a per package basis, which are linked in accordance with the package hierarchy. Each Package registers a repository for its sub packages and registers a population for its own classes. The template is shown below:

```
public class <pkg_name>Repository extends RepositoryImpl {
  public <pkg_name>Repository() {
    super.setFactory( new <pkg_name>Factory(log) );
    <for each subPackage>
     super.registerSubRepository("<spkg name>",
              new <spkgFullName>.<spkg_name>Repository());
    <for each class in this package>
     super.registerElementType("<cls_name>");
 }

 public void saveXMI(java.lang.String fileName) {
   super.saveXMI(fileName, new <pkg_name>VisitorImpl());
 }

 public java.lang.String toString() { return "<pkg_name>";}
}
```

The saveXMI method calls the generic saveXMI method, passing a bespoke package visitor. This visitor encodes the manner in which the model elements and their parts should be traversed in accordance with the original model specification.

Visitor

The visitor pattern is one that supports traversal of an object graph that forms the population of a model. The standard visitor pattern provides an implementation of a technique known as ‘double dispatch’. This enables a particular method to be called based on the runtime type of two objects (as opposed to the basic method call that depends on the runtime type of a single object). The two objects are typically: one that is the object being visited (known as the host); and one that is providing a particular piece of behaviour (known as the visitor).

Implementation of the standard visitor pattern provides a visitor interface. This interface typically contains a visit method for each object type for which it provides behaviour. Implementations of the interface visit methods are called indirectly by

calling an accept method on the host object, which takes the visitor implementation as a parameter, this accept method subsequently calls the visit method on the passed visitor using itself as a parameter. The behaviour of *visit* methods typically perform some actions specific to the purpose of the visitor and call accept methods on the objects that form the next hosts in the traversal order.

As with the factory and repository patterns, providing such a visitor for the whole model is not scaleable, or useable if we wish to provide a visitor implementation for only a portion of the model. Thus we also split our implementation of the visitor pattern by package and link them according to the package hierarchy.

2.4 Visitor Implementations

We have found that much of the implementation of tools to support a particular metamodel is provided by implementations of the visitor interface that navigate the model population according to attributes and the various types of association end – *composition*, *aggregation* and *none*. The specific behaviour actions of the visitor implementations can be characterised by grouping the actions into:

- those related to the host
- those related to the attributes
- those related to each type of association end
- those related to linkage between the host and the attributes or association ends

To support this set of visitor implementations, we have defined an implementation pattern for a model navigation visitor. This visitor implementation takes as an additional argument, an object that carries the actions as described in the list above (the visitActions). The interface for this is as follows:

```
public interface VisitActions {

  Object hostAction( String hostTypeName,
                     Object host,
                     Object data,
                     Visitor visitor );

  Object attributeAction( String modelPropertyName,
                          String modelPropertyTypeName,
                          Class implPropertyType,
                          Class collType,
                          Object implPropertyValue,
                          Object data,
                          Visitor visitor);

  Object associationEndAction( String modelPropertyName,
                               String modelPropertyTypeName,
                               Class implPropertyType,
                               Class collType,
                               Object implPropertyValue,
                               Object data,
                               Visitor v );

  Object aggregateEndAction( String modelPropertyName,
                             String modelPropertyTypeName,
                             Class implPropertyType,
                             Class collType,
                             Object implPropertyValue,
```

```
                                Object data,
                                Visitor v );

  Object compositeEndAction( String modelPropertyName,
                                String modelPropertyTypeName,
                                Class implPropertyType,
                                Class collType,
                                Object implPropertyValue,
                                Object data,
                                Visitor v );

  Object linkAttribute( Object propValue,
                        Object hostValue,
                        Visitor v );

  Object linkAssociationEnd( Object propValue,
                                Object hostValue,
                                Visitor v );

  Object linkAggregateEnd( Object propValue,
                           Object hostValue,
                           Visitor v );

  Object linkCompositeEnd( Object propValue,
                           Object hostValue,
                           Visitor v );

}
```

The actions for each implemented visit method are defined to be those that call methods from the visitActions object interspersed with actions that appropriately navigate the population in accordance with the metamodel definition. E.g.:

```
Object visit( <HostType> host,
              VisitActions actions,
              Object data ) {
  Object node =
           actions.hostAction("<HostType>",host,data,this);
  actions.linkAttribute(
    actions.attributeAction("<HostType>.<attName>",
                            "<AttType>",
                            <AttType>.class,
                            <CollectionTypeOrNull>,
                            host.get<AttName>(),
                            data,
                            this
    ),
    node,
    this
  );
  actions.linkAggregateEnd(
    actions.aggregateEndAction("<HostType>.<endName>",
                               "<EndType>",
                               <EndType>.class,
                               <CollectionTypeOrNull>,
                               host.get<EndName>(),
                               data,
                               this
    ),
    node,
    this
  );
}
```

3 Using OCL as a Template Language

As previously mentioned, we make use of OCL as an (interim) template language. To achieve this we have extended OCL in the following ways:

1. Addition of the '+' operator for String values; this is evaluated as a concatenation of the two string arguments.
2. Addition of facility to construct any (model) object. Achieved using the full type name of the object, with constructor arguments contained in following braces '{…}'. This is similar to the syntax for constructing collection objects.
3. Provision of a File class. This object requires a file name as a constructor argument (directories are constructed if necessary); there are two methods, read and write, which either get the file contents (as a String) or write a String argument to the file.
4. Provision of an Expression class. This class enables evaluation of and OCL String value as an OCL expression; arguments are passed to the constructor and evaluate method to provide the environment (free variables) for the expression.
5. Addition of a new query 'context' for OCL expressions that facilitates multiple context variables.

The first of these additional features enables more succinct composition of string values; the second and third features enable us to create files and directories; and the combination of the second, third, fourth and fifth enable us to call sub templates with appropriate parameters.

The templates, shown in the previous subsection, map to a particular pattern of OCL expression. This pattern starts with a query context, defining the parameter object types (free variables) for the expression. Then a number of let statements are given, which define the specific template variables, based on the parameters. The template text is mapped to a series of String and variable concatenations; and finally the concatenated text is written to a file.

To illustrate this, the OCL template (query expression) for constructing package visitors is given below:

```
context
   self : uml::Model_Management::Package,
   properties : uk::ac::kent::cs::kmf::browser::Properties

query:

let
 root_dir = properties.get('root_generation_directory'),
 dir = ''+root_dir+'/'
       +
       Expression {
         String,
         TupleType( self:uml::Foundation::Core::Namespace,
                    sep:String ),
         uk::ac::kent::cs::kmf::util::File {
            properties.get('templates_directory')
            + '/GetFullName.ocl'}.read()
       }.evaluate( Tuple{self=self, sep='/'} ),
 pkg_name = self.name.replaceAll('[^0-9a-zA-Z_]','_'),
 file_name = ''+dir+'/'+pkg_name+'Visitor.java',
 pkg_fullName
```

```
   = Expression {
         String,

         TupleType(self:uml::Foundation::Core::Namespace,
                   sep:String ),
         uk::ac::kent::cs::kmf::util::File {
            properties.get('templates_directory')
            +'/GetFullName.ocl'}.read()
     }.evaluate( Tuple{self=self, sep='.'} ),
 subPackages = self.ownedElement->
    select(e|e.oclIsKindOf(uml::Model_Management::Package)),
 classes:Set(uml::Foundation::Core::Class)
   = self.ownedElement->
       select(e|e.oclIsKindOf(uml::Foundation::Core::Class)),
 nonAbstractClasses = classes->select(c | not c.isAbstract ),

 result_str =
'
package '+pkg_fullName+';

import uk.ac.kent.cs.kmf.patterns.Visitor;
import uk.ac.kent.cs.kmf.patterns.VisitActions;

public interface '+pkg_name+'Visitor
  extends Visitor
{
'+
nonAbstractClasses->collect( cls |
  Expression {
    String,
    TupleType(
      self:uml::Foundation::Core::Class,
      properties:uk::ac::kent::cs::kmf::browser::Properties),
    uk::ac::kent::cs::kmf::util::File {
       properties.get('templates_directory')
       +'/Class__VisitorMethodSignature.ocl'}.read()
  }.evaluate( Tuple{self=cls, properties=properties} )
+';
'
)->including('')->sum()
+'
}
'

in
 if result_str.oclIsUndefined() then
  'Error generating Visitor interface for - '+pkg_fullName
else
  uk::ac::kent::cs::kmf::util::File{file_name}
   .write(result_str)
endif
```

To aid reading this expression the strings defining the generated code are highlighted in bold and the expression parameter variables and defined template variables are highlighted in italics. The expression illustrates all of the five additions to the OCL language.

4 Providing Tools Based on the Patterns

The set of implementation classes generated by these patterns could be used in a variety of applications. An application that we find to be particularly useful is that of

a Browser. The browser is used to create populations of the model and to evaluate OCL constraints over that population.

4.1 Browser

A browser can be created by piecing together various implementations of Visitor Actions, reference to a Repository, our OCL implementation, and a number of actions for invoking operations on these component parts.

A generic browser implementation has been written that can be used in conjunction with any model and set of classes generated using the templates described above. XMI and HUTN visitor actions are written to facilitate saving a population (others could be just as easily written). A generic XMI reader is provided, enabling a previously saved population to be restored. A set of visit actions that construct a JTree has been written; this provides a tree view on the population of a repository. These parts are linked together in a common application along with our implementation of OCL. An image of the generated browser can be seen in **Figure 1**. The panel on the left shows a JTree view of the metamodel components and the instances of those components in the current population. The Console panel on the right shows (in this image) that the metamodel components have been registered with the generated repository. The OCL evaluation panel shows an Invariant, to be evaluated in the context of the class (with name attribute set to "Library") highlighted on the left.

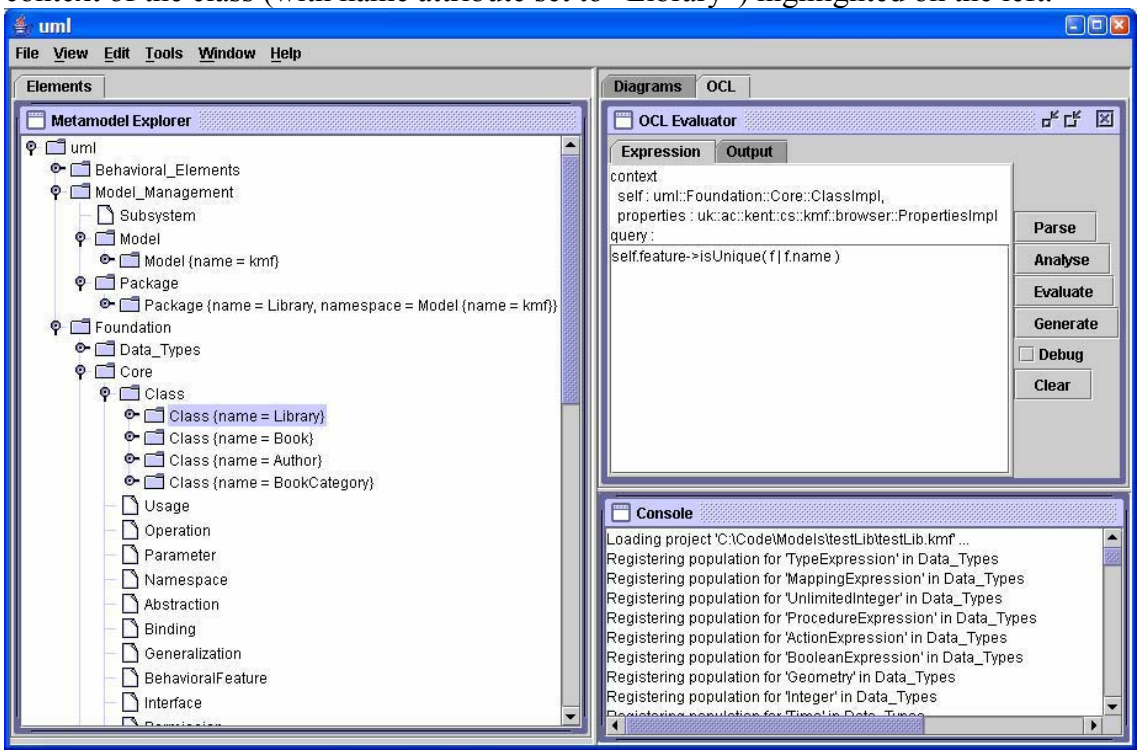


Figure 1 – Generic KMF Browser, browsing a population of UML 1.5 metamodel

5 Related Work

Existing tools such as MetaEdit+ [17] and the Eclipse Modelling Framework (EMF) [18] go some way towards providing similar types of tool to KMF. However they fall short in certain areas. EMF does not support constraints or query

expressions. MetaEdit+ is a more general tool for developing domain specific languages and also does not support OCL.

MetaEdit+ does not directly support generation from class diagrams; it uses its own concepts for metamodel specifications, Graph, Object, Property, Relationship and Role, which are similar to a very small subset of MOF. Metamodels are entered through a series of property boxes, rather than using a visual notation such as class diagrams. The metamodel specification facility enables the definition of a visual language for entering populations of the metamodel (i.e. expressions/specifications in the domain specific language). UML is provided as one of the example domain specific languages supplied with MetaEdit+; however it does not provide an environment that we find easily useable for editing UML models. In addition there does not seem to be a mechanism for adding support for OCL. MetaEdit+ is a commercial tool with development and maintenance support.

Frameworks such as the Eclipse Modelling Framework provide a similar level of generation facility to that provided by our KMF. However, the current release of EMF does not support OCL; we have been working with IBM to provide a version of our OCL library that operates with their EMF generated code. This work has been very successful and we have succeeded in providing facility to:

a) directly evaluate OCL constraints over a population of an EMF model; and
b) generate java code from an OCL expression that when compiled will evaluate the expression.

UMMF - UML Meta-Model Framework [19] is an open source framework written in Perl; it can be used to generate class and interface templates for programming languages Perl and Java; and it will import from XMI versions 1.0 and 1.2.

6 Conclusion

The primary facility offered by KMF, not offered by other tools is the OCL evaluation functionality. In addition, KMF accepts, as input, standard XMI, generated from any appropriate modelling tool (unlike MetaEdit+). Also, KMF provides a fully flexible and user adaptable mechanism for generating code from the provided metamodel specification (possible in MetaEdit+, EMF supports a single, fixed, Java implementation). The KMF tool is based entirely on the concepts and languages of the OMG, making use of XMI, HUTN, MOF, UML and OCL.

This paper has illustrated the significance of patterns as a means to aid the automatic production of tools that to support the specification of a metamodel. The implementation of OCL is used to check well-formedness constraints on populations of the model. In addition by using the OCL as the basis for a template language we have demonstrated that code can be generated from template specifications. The generated code implements standard coding patterns, which are put together to form component parts of a modelling tool.

We show the patterns that we have used within the KMF project and show how we have varied from the standard patterns in order to make the scaleable. We also show the template expressions that will generate a Java implementation for these patterns.

Other work started in [20] and [21] is being continued in order to extend the generated patterns so that they will support an implementation of model transformations. We are looking for a suitable template language, based on OCL, to use instead of directly using our variation of OCL. Additionally, we feel that

providing an implementation in an aspect oriented language such as AspectJ [22] may provide useful facilities for linking the interacting patterns.

Acknowledgements

David Akehurst acknowledges support of the EPSRC project "Design Support for Distributed Systems" (GR/M69500/01) and its investigators J.Derrick and A.G.Waters. Octavian Patrascoiu acknowledges support of the EPSRC project "Reasoning with Diagrams" (GR/R63509/01) and its investigator P.Rodgers.

References

1. OMG: Model Driven Architecture (MDA). Object Management Group, ormsc/2001-07-01 (2001)
2. OMG: Meta Object Facility (MOF) Specification, Version 1.4. formal/2002-04-03 (2002)
3. OMG: Response to the UML 2.0 OCL Rfp (ad/2000-09-03), Revised Submission, Version 1.6. Object Management Group, ad/2003-01-07 (2002)
4. OMG: Request for Proposal: MOF 2.0 Query / Views / Transformations RFP. Object Management Group, ad/2002-04-10 (2002)
5. IBM: Rational Rose. http://www.rational.com (2003)
6. Gentleware: Poseidon UML tool, version 1.4. www.gentleware.org (2003)
7. Borland: Together. http://www.borland.com/together/index.html (2003)
8. OMG: XML Metadata Interchange (XMI), v2.0. Object Management Group, formal/03-05-02 (2003)
9. OMG: Human-Usable Textual Notation (HUTN) Specification. Object Management Group, ptc/02-12-01 (2003)
10. KMF-team. Kent Modelling Framework (KMF).[Online]. Available: www.cs.kent.ac.uk/projects/kmf
11. Akehurst, D. H., Derrick, J., Waters, A. G.: Addressing Computational Viewpoint Design. In: Proc. EDOC 2003 (2003)
12. X.901-5: Information Technology - Open Distributed Processing - Reference Model: All Parts. ITU-T Recommendation (1996-99)
13. DSE4DS-team. Design Support for Distributed Systems (DSE4DS) Project Home Page.[Online]. Available: http://www.cs.ukc.ac.uk/projects/dse4ds/index.html
14. Akehurst, D. H., Bordbar, B., Derrick, J., Waters, A. G.: Design Support for Distributed Systems: DSE4DS. In: Proc. 7th Cabernet Radicals Workshop (2002)
15. RWD-team. Reasoning with Diagrams (RWD) project.[Online]. Available: www.cs.kent.ac.uk/projects/rwd
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
17. MetaCase: MetaEdit+. http://www.metacase.com/ (2003)
18. IBM: Eclipse Modeling Framework. http://www.eclipse.org/emf/ (2003)
19. Stephens, K. UMMF - UML Meta-Model Framework.[Online]. Available: http://kurtstephens.com/pub/uml2code/current/htdocs/
20. Akehurst, D. H.: Model Translation: A UML-based specification technique and active implementation approach. University of Kent at Canterbury (2000)
21. Akehurst, D. H., Kent, S.: A Relational Approach to Defining Transformations in a Metamodel. In: Proc. The Unified Modeling Language 5th International Conference (2002) 305-320
22. AspectJ-team: AspectJ. http://www.eclipse.org/aspectj/ (2003)