Computer Science at Kent

Second European Workshop on Model Driven Architecture (MDA)

with an emphasis on Methodologies and Transformations

September 7th-8th 2004
Canterbury, UK

Proceedings

Edited by
D.H.Akehurst

Technical Report No. 17-04
September 2004

Contents

Preface

The Model-Driven Architecture[1] (MDA) is an approach to IT systems development fostered by the Object Management Group (OMG). It is based on forming a separation between the specification of a systems essential functionality as a platform independent model (PIM) and the realisation of the system using more detailed and specific platform specification (PSM).

The MDA approach to the development of distributed IT systems will affect the current methods and techniques employed to manage the development process. It is recognized that specifying the mappings from transformations from a PIM to a PSM is a key enabling aspect of the MDA approach. This is substantiated by OMG's current Request for Proposals (RFP) on techniques and facilities to enable transformations.

In this workshop we explore how the MDA approach affects or impacts on methodologies for system development, and explore the techniques available for specifying transformations, in particular taking a look at tools (or potential tools) for supporting such specifications and methodologies.

This workshop is following on from two previously successful workshops:

- Metamodelling for MDA held in York, November 2003, and
- First European Workshop on Model Driven Architecture with Emphasis on Industrial Application held in Enschede, March 2004.

This two track, two day workshop on Methodologies and Transformations provides the opportunity for in depth discussion regarding each topic whilst allowing interaction between experts in each area.

The first day is dedicated to setting the scene, involving presentations on some of the accepted submissions. Based on the topics covered by the submissions, specific problems in the areas of transformations and methodologies are identified.

The second day of the workshop is targeted at "doing some work" (after all this is a 'work'shop) and the delegates divide into groups for smaller scale discussion on the selected problems. The goals of the discussion groups are clearly defined and each group is expected to report back on the results of the discussion The results are included in this proceedings.

[1] Model-Driven Architecture, MDA, UML, XMI, and their corresponding logos are registered trademarks or trademarks of the Object Management Group, Inc. in the United States of America, in the European Union, and in other countries.

Keynote

If model transformation is the answer, what was the question?

Tracy Gardener, IBM

Abstract

This talk sets out the problem domain for model transformation and introduces a set of use cases for transformation including model differencing, pattern expansion, model merging and weaving, alternate views and generation of platform specific artifacts from a platform independent model. The talk also discusses where model transformation fits into the development process and who we can expect to be building and using model transformations.

Biography

Dr Tracy Gardner has a PhD in the area of programming/modelling language design which was a winner of the CPHC/BCS Distinguished Dissertations award 2000. Tracy has been a user of OO modelling languages since 1993 and has worked with UML from the beginning. While working for the UK Office for Library and Information Technology (UKOLN) Tracy was involved in two collaborative EU projects in the digital libraries domain (the latter part of DESIRE and the early part of RENARDUS). Tracy has spent time as a practitioner of model-driven development, using the UML-based Rational Rose Real-Time product while working for Marconi Telecommunications Ltd. Since joining IBM in 2001 Tracy has been involved in model-driven component technologies for business integration.

Dr Gardner's current work is on applying Model-Driven Development to the Business Integration domain; she was the main contributor to a UML profile for automated business processes with a mapping to BPEL4WS and is now collaborating on IBM's response to the OMG's Business Process Definition Metamodel and MOF Queries/Views/Transformations RFPs. Tracy has presented on model-driven development at a number of industry conferences (including the OMG MDA? Implementers' Workshop, Enterprise UML 2003) and will also present at the 1st European Conference on Model-Driven Software Engineering. Tracy Gardner is also a member of the program committee for the second OMG MDA? Implementers' workshop. She has recently been involved as a reviewer for UML 2.0 (for the Activity modelling chapter) and as a subject matter expert for a UML 2.0 professional certification exam.

Methodologies

Enabling Model Driven Product Line Architectures

Toacy C. de Oliveira [0], Ivan Mathias Filho [1] , Carlos J.P. de Lucena [1],
Paulo Alencar [2], Donald D. Cowan [2]

[0] Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul, Av. Ipiranga 6681 , Porto Alegre – RS , Brazil

[1] Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rua Marquês de São Vicente 225, 22453-900, Rio de Janeiro, Brazil

{toacy, ivan, lucena}@inf.puc-rio.br

[2] University of Waterloo, 200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1.

{palencar, dcowan}@csg.uwaterloo.ca

Abstract
Product Lines Architectures and Model Driven Architectures are techniques to achieve reuse of software assets. Combining these approaches facilitates the specification of platform independent models that can be used at stakeholders level to indicate computation independent functionality and at developers level to assist the generation of applications by means of transformations. In this scenario, our work proposes the use of Model Driven Product Lines by integrating the Features Model, which is a high-level specification approach, and Object Oriented Frameworks, in a way that elements in the two models can be represented and related. Moreover we propose a semi-automatic approach to obtaining the final application code (the actual product), using a description that guides application developers during the product derivation process.

1. Introduction

Software development industry is typically driven by business opportunities. In such scenario, the ability to increase market share and decrease time-to-market are important issues that strengthen two aspects present in software development processes: i) the capability to use past experiences and; ii) the ability to achieve independence from the underlying hardware and software platforms. Those aspects are related to reuse in different levels of abstraction and raise important questions. How to achieve reuse from high-level specifications? How to relate specifications found in different levels of abstraction? How to automate the generation of software artifacts from high-level specifications? How to guarantee some properties along the development process?

To cope with these issues, Software Engineering practitioners and organizations, such as OMG, have developed ways to systematize the construction of software assets, for instance Product Line Architectures [1] and Model Driven Architectures [18], which can be combined [19] to achieve large scale reuse by means of structured and configurable representations of platform independent software assets.

Product Line Architectures (PLA) are designs for families of applications [1]. They intend to facilitate the realization of software artifacts (products) from a highly configurable and adaptable representation that contains variable and constant aspects about an application domain. Therefore a PLA achieves reuse combining the bits and pieces that represent the domain assets, typically frameworks and/or components, in subsets (products) that meet business opportunities [8]. On the other hand, Model Driven Architectures (MDA) advocate the use of high level models and transformations to produce platform independent assets that can be used to generate an application[18]. As a result, the combination of PLA and MDA approaches allows reuse in two dimensions: horizontal and vertical. Horizontal reuse can be achieved when defining different applications (i.e., products) from a common set of assets. Vertical reuse can be attained when defining such products in a platform independent manner.

In this context, this paper presents ongoing work [15][20][16][13] that intends to assist the use of Model Driven approach when developing Product Line Architectures, using a set of high-level notations such as Features Models [9][10] and UML diagrams[17], and a language to specify horizontal and vertical reuse transformations to product specific design.

The paper is organized as follows. Section 2 describes an overview of the approach. In Section 3 we depict the underlying technologies. Section 4 contains an example of the approach. Section 5 contains the Conclusions and Future Work.

2. An Overview of the Approach

Our approach intends to combine Model Driven Architecture (MDA) and Product Line Architecture (PLA) so that software can be developed in a platform independent manner and generated from high level models. To accomplish such goal, we combine Features and Framework Models so that software products, i.e., frameworks instances, can be obtained in a semi-automatic way by means of assisted transformations from Features Models to application models and code. For that reason, we describe a PLA as an artifact composed of three types of documents: a Features Model, an Annotated Framework Design, and an Instantiation Script (Figure 1).

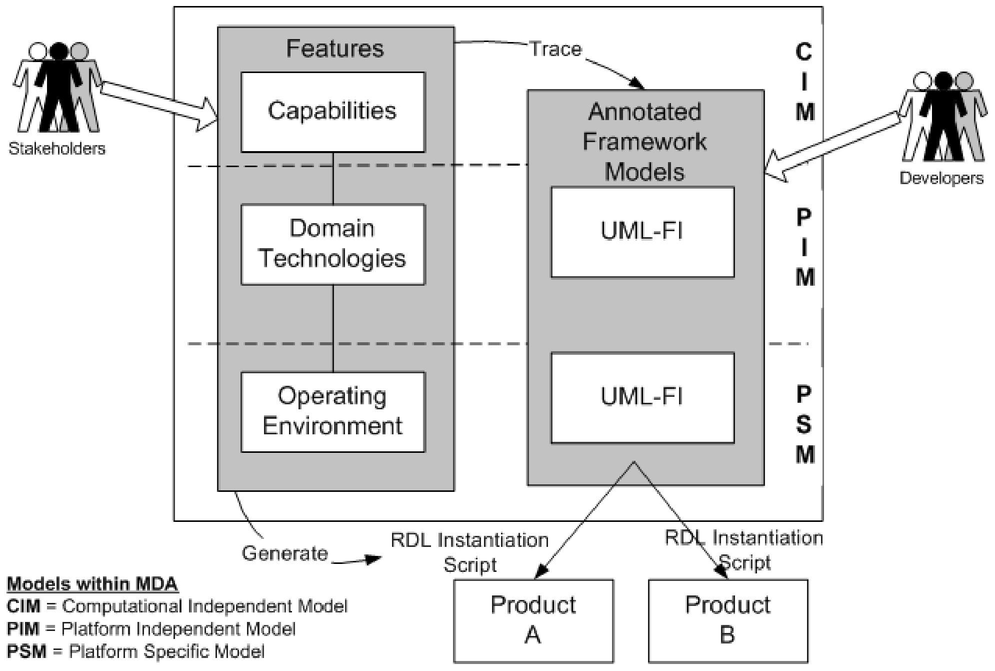


Figure 1 - Approach Overview

The Features Model is used as the original formulation but with a complete integration with the UML metamodel [4]. The integration of the Features Model with the UML language proved to be a good strategy to precisely describe the mechanisms used to associate the characteristics of an application domain with the design elements that realize such characteristics. This occurs because the selection of a given feature will trigger a series of reuse actions that will adapt the design of a framework taking into account the requirements of a specific application of a domain.

The annotated framework design is the representation of object-oriented framework models using UML. Actually, in order to represent all flexible characteristics embedded in the framework models, we have developed and adopted UML-FI (UML for Framework Instantiation), which is a UML profile designed to emphasize the flexible elements in a OO design (the hotspots).

The last document is an Instantiation Script that represents the instantiation process that is responsible to transform UML-FI models to UML models representing the target application. With this script, a software developer can be guided on how to accommodate the product specific needs into the framework's hotspots. In order to achieve a succinct process description we have developed RDL (Reuse Description Language), which is a script language that allows the specification of order and state dependencies between well known object oriented programming activities that are commonly used when instantiating an incomplete design.

To integrate the three types of documents, Features, Framework Models and the Instantiation Script, we have created a mapping technique that relates the flexibility represented by optional and alternative features in the Features Model, to extension points (hotspots) in the framework design. In this context, the framework instantiation begins with the stakeholders selecting the characteristics (Capabilities, Domain Technologies and Operating Environments) that they want to include in a specific produce (Figure 1). The selected characteristics, together with the annotated class models, will generate an instantiation script containing all necessary steps to instantiate an application with the desired characteristics.

To finish the proposed approach it is necessary to execute the generated instantiation script with the help of a RDL execution environment, which will adapt the original framework design, inserting some design elements to adapt the hotspots. It is worth mentioning that this last step is not entirely automatic and that it will be necessary for the application developers to provide some additional information, such as class names, attribute names and types, methods names, etc.

To finalize this overview it is important to make one additional remark: the documents used in this approach are XML-based. According to our past experience [14][16], XML models are suitable for program manipulation and can be translated to formal models that can be validated. Moreover, the use of XML lets us perform structural analysis in the design [12] in order to discover signs of best practices violations [2][11], structural regularities conformity [12] and refactoring smells [5].

3. Underlying technologies

3.1. The Features Model

The Features Model [10] is a suitable approach for representing system's characteristics at different levels of abstraction. Its ability to capture the "capabilities (services, operations, attributes, etc…), domain technologies (methods, theories, etc…) and operating environment (HW platform, O/S, DBMS, etc…) of an application family", and organize their structural relationships by means of a graphical notation, facilitates system understanding for non-software development practitioners. Furthermore, Features can be interpreted from a Software Development Process perspective, as the domain specification that delineates the design that must be realized in the underlying application (see Section 4 for an example).

3.2. Design Representation - UML-FI

Our approach focuses on instantiation of object-oriented models (Framework Models) and so we need to express the nature of hotspots and the related instantiation activities in terms of OO programming techniques. To provide such representation, we have developed UML-FI (UML for Framework Instantiation), which is a profile for UML. This representation uses stereotypes (i.e., an UML extension mechanism) to indicate, at the design level, the object oriented activities that should be performed.

With UML-FI it's possible to indicate most basic object oriented programming activities such as class extension, method redefinition and value assignment (useful for Blackbox framework [3] instantiation) by means of annotations (stereotypes) in class diagrams (see Figure 2).

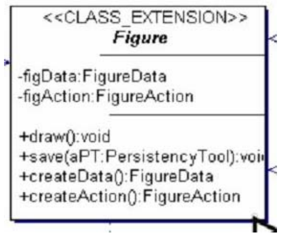


Figure 2 – Class Extension Stereotype

.

UML-FI is also able to represent the concept of "Pattern Instantiation". Pattern Instantiation specifies a group of correlated actions that can be seen as one as is the case of Design Patterns [6].

In addition, hotspots can also be mandatory or optional. Therefore the reuser has the ability to decide if the associated design element will (or will not) be present in the final application. UML-FI indicates optional and mandatory aspects as a tagged-value named **presence** that can assume the values OPTIONAL or MANDATORY (Figure 3).

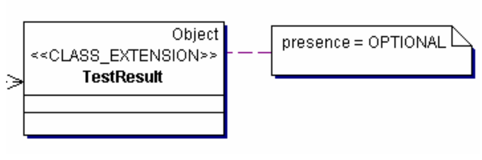


Figure 3 – Optional representation.

3.3. Instantiation Representation - RDL

The representation provided by UML-FI models does not guarantee that a valid framework instance is produced. Some information, such as what patterns to apply, was deliberately omitted to avoid graphical complexity. In addition class diagrams cannot provide the sequence of instantiation actions that should be applied to obtain the final product. To make this missing information and sequencing explicit, we have developed a representation that adds this information to the instantiation process specifications. We call this representation

RDL (Reuse Description Language). RDL is a domain specific language that enables framework developers to express how framework instantiation should be performed by listing the instantiation tasks in a detailed script-like document. We have adopted a language to represent the process with a view to making it to easy to read by an ordinary reuser.

In order to organize RDL statements, we have adopted the concept of cookbook and recipes [7]. Cookbooks contain a set of recipes. Recipes can be traced to functions in an imperative programming language and contain the instantiation code itself (see Section 4 for an example).

4. An Example

The instantiation process consists of the selection of a group of related Features that implement the desired characteristics and the further adaptation of the framework's original design, based on the selected features, to produce a framework instance. Thus, the adapted design contains the mandatory elements of the original design, plus the elements included due to feature selection, plus the elements that the framework reusers include during the instantiation process (the product increments).

In order to produce the framework configuration, we claim that the reuser must be guided through an interactive and semi-automatic process that captures the design elements that will represent the design increments. This process is specified in terms of a RDL script that will be generated from the Framework's Feature Model.

In Figure 4, the features that will be included in the final application are shown in dark gray. For example, the feature **Figure** is related to a UML-FI element, the class **Figure**, by means of an association table. This class indicates that the reuser should create a subclass from **Figure** class to incorporate product specific needs.

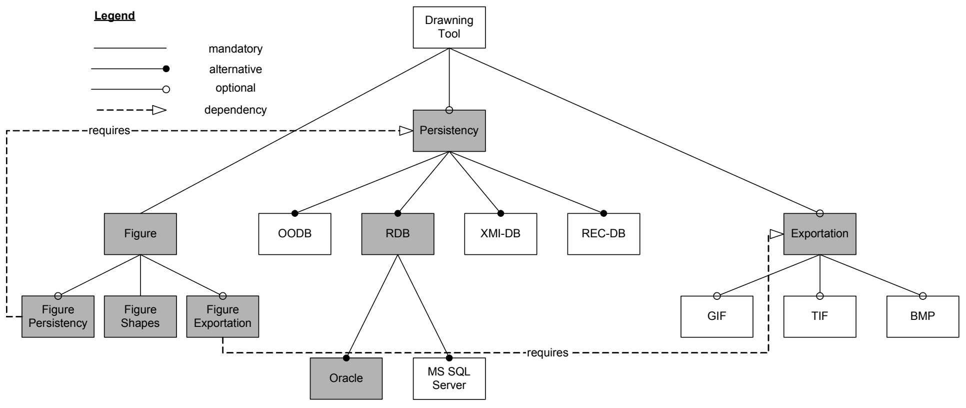


Figure 4– The Features Diagram of an instance of the DTFrame Framework.

Based on the selected Feature (Figure), and its trace to the UML-FI design element, it's possible to generate a RDL script that specifies the required Class Extension activity as shown in bold in Listing1.

```
COOKBOOK DTFrame
       RECIPE MAIN;
              LOOP
                     V1=CLASS_EXTENSION(Figure);
                     METHOD_EXTENSION(Figure,V1,save);
                     METHOD_EXTENSION(Figure,V1,createAction);
                     METHOD_EXTENSION(Figure,V1,createData);
              END_LOOP;

       END_RECIPE;
END_COOKBOOK;
```

Listing 1 The Resulting Script.

5. Conclusions & Future Directions

This paper presents a detailed description of a framework instantiation process that tackles the problem of deriving specific products from an architecture designed to meet the requirements of a family of applications. The approach intends to represent such architecture with a set of models that can be used at the stakeholder level, to provide high-level reasoning, and at a developer level, to facilitate product derivation.

The presented approach is in conformity with the MDA recommendation for flexibility in implementation. The use of framework technology meets the flexibility in implementation providing hotspots to adapt a software product to a specific platform. This is what we call Vertical Reuse: the application functionality is not extended; only new platforms are aimed. Moreover, the frameworks can also meet the objectives of Product Lines Architectures as new functionalities are added to a software product attaching specific components to the hotpots. This is what we call Horizontal Reuse: the production of applications that meet distinct subsets of the requirements of a domain.

As future work we intend to enhance the xFIT (XML-based Framework Instantiation Tool) tool to integrate the models adopted. We also plan to use Ontology Techniques to provide consistency among the terms used to describe Product Line Architectures.

References

1. Batory, D., Cardone, R., Smaragdakis, Y.,Object-Oriented Frameworks and Product Lines,Proceedings of the First Software Product Line Conference,p227--247, 2000.
2. Beck, K. Smalltalk Best Practice Patterns. Prentice Hall, 1997.

3. Fayad, M.E.; Schmidt, D.C., Johnson, R., 1999. Application Frameworks. In: Fayad, M.E. Schmidt, D.C., Johnson, R. (Eds.), Building Application Frameworks – Object-Oriented Foundations of Framework Design, John Wiley, New York, New York.

4. Filho, I.M., Oliveira, T.C., Lucena, C.J.P., 2002. A Proposal for the Incorporation of the Features Model into the UML Language. In: Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS2002), Ciudad Real, Spain.

5. Fowler, M et al.. Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts.
7. Johnson, R.E., 1992. Documenting Frameworks Using Patterns, ACM SIGPLAN Notices, vol. 27, n. 10, September, 63-76.
8. Chastek, G., Donohoe, P., Kang, K., Steffen Thiel, Product Line Analysis: A Practical Introduction, Technical Report CMU/SEI-2001-TR-001
9. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., 1998. *FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architecture*. In: Annals of Software Engineering, vol. 5, 143-168, Kluwer Academic Publishers, Dordrecht, Holland.
10. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1993. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.
11. Meyers, S. , Effective C++, Addison Wesley, 1992
12. Minsky, N. H., Law-governed regularities in object systems; part 1: An abstract model. *Theory and Practice of* Object Systems (TAPOS), 2(1), 1996.
13. OLIVEIRA, T. C., LUCENA, C. J. P., COWAN, D. D., MATHIAS , I. F., ALENCAR, P. Feature Driven Framework Instantiation In: Ecoop, 2003, Darmstad. Workshop on Modeling Variability for Object-Oriented Product Lines. , 2003.
14. Oliveira, T.C., Alencar, P., Cowan, D. , Towards a declarative approach to framework instantiation Proceedings of the 1st Workshop on Declarative Meta-Programming (DMP-2002), September 2002,Edinburgh, Scotland, p 5-9
15. Oliveira, T.C., Alencar, P., Cowan, D. Filho, I.M., Lucena, C.J.P. , Software Process Representation and Analysis of Framework Instantiation, IEEE-Transactions in Software Engineering, March 2004 p145-159.
16. Oliveira, T.C., 2001. Uma Abordagem Sistemática para a Instanciação de Frameworks Orientados a Objetos, Ph.D. Thesis, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil. (In portuguese).
17. UML specification found at http://www.omg.org/technology/documents/formal/uml.htm

18. MDA specification found at http://www.omg.org/mda
19. Deelstra, S., Sinnema, M., Gurp, J. , Bosch, J. Model Driven Architecture as Approach to Manage Variability in Software Product Families. Workshop On Model Driven Architecture: Foundations And Applications June 26-27, 2003, University of Twente, Enschede, The Netherlands
20. Filho , I. F., Oliveira, T. C., Lucena, C. J. P. A Framework Instantiation Approach Based on the Features Model. Journal Of Systems And Software. , to appear.

Costs and Benefits of Multiple Levels of Models in MDA Development

João Paulo Almeida, Luís Ferreira Pires and Marten van Sinderen

Centre for Telematics and Information Technology, University of Twente
PO Box 217, 7500 AE Enschede, The Netherlands
{almeida, pires, sinderen}@cs.utwente.nl

Abstract. In Model-Driven Architecture (MDA) development, models of a distributed application are carefully defined so as to remain stable in face of changes in technology platforms. As we have argued previously in [1, 3], models in MDA can be organized into different levels of platform-independence. In this paper, we analyze the costs and benefits of maintaining separate levels of models with transformations between these levels. We argue that the number of levels of models and the degree of automation of transformations between these levels depend on a number of design goals to be balanced, including those of maximizing the efficiency of the design process and maximizing the reusability of models and transformations.

1 Introduction

The development of a distributed application can be regarded as the process of building a realization of the application that satisfies user requirements. In most traditional development cultures, application developers are instructed to produce intermediate models to facilitate bridging the gap between requirements and realization. These intermediate models are mainly regarded as a means to obtain a realization of the system, with different models addressing different design concerns. The ultimate product of the development process is the realization, which can be deployed on available implementation technologies (platforms). Any intermediate models produced during the development processes are considered means and not ends.

In the case of Model-Driven Architecture (MDA) development [8], however, intermediate models that are used to produce the final realization are also considered final products of the development process. These models are carefully defined so as to remain stable in face of changes in platform technologies, and are therefore called platform-independent models (PIMs).

In MDA development, models can be organized into different levels of platform-independence [1]. Models at a particular level of platform-independence can be realized into a number of platforms. When multiple levels of platform-independence are adopted, successive (automated) transformations may be used that lead ultimately to platform-specific models (i.e., models at the lowest level of platform-independence with respect to a particular definition of platform).

An indispensable activity in early stages of MDA development is to determine which levels of models and which (automated) transformations are necessary. This activity is part of the *preparation phase* of the MDA development process [4]. In the preparation phase, (MDA) experts define the metamodels, profiles and transformations that are to be used in the *execution phase* by application developers.

The organization of the execution phase in terms of levels of models depends on a number of design goals to be balanced, including those of maximizing the efficiency of the design process and maximizing the reusability of models and transformations. In this paper, we analyze the factors that should be considered in order to determine the organization of the execution phase. We claim no conclusive or concrete guidelines on the use of different levels of models and transformations. We rather aim at setting the stage for further discussion on this very important issue for MDA development.

The concept of abstract platform we have proposed in [1, 3] supports us in the discussion. An abstract platform is an abstraction of infrastructure characteristics assumed for models of an application at a certain level of platform-independence. An abstract platform is represented through metamodels, profiles and reusable design artifacts [3]. For example, if a platform-independent design contains application parts that interact through operation invocations (e.g., in UML [10]), then operation invocation is a characteristic of the abstract platform. Capabilities of a concrete platform are used during platform-specific realization to support this characteristic of the abstract platform. For example, if CORBA [5] is selected as a target platform, this characteristic can be mapped onto CORBA operation invocations.

This paper is further structured as follows: section 2 discusses how the automation of transformations between two levels of models can be justified; section 3 considers the use of intermediate levels of models, and section 4 provides some concluding remarks.

2 Introducing Automated Transformations

During the execution phase of an MDA project, an application developer derives models at a lower-level of platform independence from models at a higher-level of platform independence. In order to increase the efficiency of the application development process, the developer may use automated transformations to bridge between different levels of models.

A requirement to the automation of transformation is the specification of transformation in the preparation phase. Full automation of transformation between two levels of models requires the transformation specifier to define rules to transform all possible source models into appropriate target models. The transformation specifier must fully understand the relation between source and target metamodels, and express these rules in a suitable transformation language, supported by a transformation tool. For these reasons, transformation specifications should be produced by a knowledgeable (MDA) expert.

When transformation is automated, the creative design activities that would normally be executed manually by a designer are generalized and moved to the specification of the transformation itself and to the application of marks (marking). The costs of defining an automated transformation between two related levels of models *A* and *B* must be compensated by reusing the transformation. The following conditions contribute to the reuse of the transformation:

- *the number of applications built using models at level A and targeting B is high*, i.e., the (abstract) platform at level *B* is popular for targeting applications that can be expressed in terms of (abstract) platform at level *A*;
- *changes in application requirements are frequent*, but these changes do not affect the stability of the (abstract) platform at level *A*;
- *the development process is cyclic, and the number of design iterations is high*, i.e., the model of the application in *A* is modified several times during the development. In this case, manual manipulation of models would have required manual propagation of changes applied at level *A*.

The bottom-line is that the cost of building an automated transformation between levels *A* and *B* must be lower than the costs of manually deriving models at level *B* (from designs at level *A*) over (a long period of) time. Therefore, the stability of the (abstract) platforms at level *A* and *B* should be considered. The stability of the (abstract) platform at level *A* allows more applications to be developed in terms of this platform and the stability of (abstract) platform at level *B* is required to reuse the transformation, since transformation from *A* to *B* is specific to the platform at level *B*.

It is possible that models obtained manually and automatically differ significantly with respect to relevant qualities. These qualities should be considered when justifying automation. For example, depending on the transformation, automated code generation may result in implementations of lower time performance. When this is the case, this can be reflected in cost estimates by lowering the cost of manual coding to account for the benefits of obtaining implementations that perform better. Automated code generation may also lead to improving the correctness of implementations. In this case, cost estimates should include the costs incurred by testing, both for testing the transformation and testing the code obtained manually.

3 Introducing Intermediate Levels of Models

We envision two different extreme approaches to organizing the development process with respect to platform-independence levels:

i. *an approach with minimal use of levels of platform-independence*, in which one level of platform-independent models and one level of platform-specific models are related (through a fully or partially automated transformation), and;
ii. *an approach with exhaustive use of intermediate platform-independence levels* and several (fully or partially automated) transformations between these models.

We argue that a combination of these extreme approaches is the most effective way to handle the problem. In the sequence, we consider the costs and benefits of introducing an intermediate level of models between two arbitrary levels, a source level and a target level. This allows us to consider the full range of combinations of the extreme approaches (i) and (ii), since the recursive introduction of intermediate levels eventually leads to an exhaustive use of intermediate levels. In the discussion, we distinguish between fully or partially automated transformations.

3.1 Fully automated transformations

Figure 1 depicts the alternative situations which we contrast for fully automated transformations: (a) a situation in which a transformation *T* produces models at level *B* from models at level *A*, and (b) a situation in which a transformation T_1 produces models at level *X* from models at level *A*, and a transformation T_2 produces models at level *B* from models at the intermediate level *X*. The arrows in Figure 1 represent the execution of a transformation.

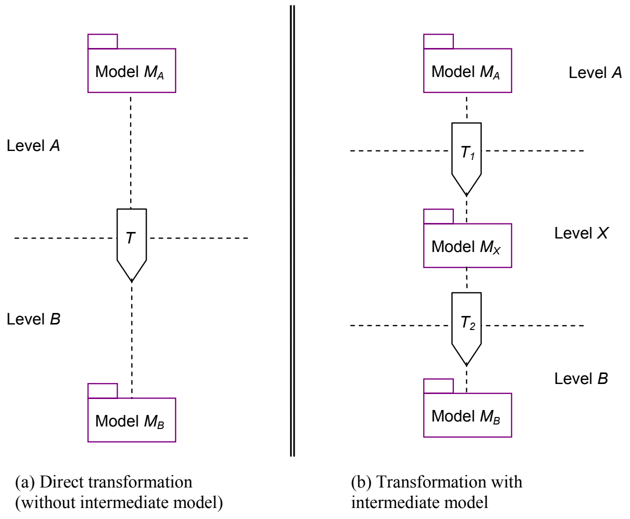


Fig. 1. Direct transformation and transformation with intermediate model

Considering solely the effort spent in the preparation phase to specify the transformations in situations (a) and (b), we cannot formulate a general rule to decide whether an intermediate step should be introduced. In some cases, it may be easier to define two transformations using an intermediate model, and, in some other cases, direct transformations may be easier to define.

Nevertheless, it is possible to draw some general conclusions on the consequences of introducing intermediate levels of models for the reuse of transformations. In this respect, an intermediate level of models may be beneficial since:

1. it may be possible to *reuse the transformation from source models to intermediate models*, even if the original transformation from intermediate models to target models cannot be reused (e.g., because of platform change); and,
2. it may be possible to *reuse the transformation from intermediate models to target models* in new projects, since there may be transformations from different source levels to the intermediate level.

A transformation between levels *A* and *B* is specific to the (abstract) platform of level *B*. Therefore, the stability of (abstract) platform at level *B* is required to reuse the transformation. Introducing an intermediate level of models may serve to factor out parts of the transformation that are less platform-specific, capturing unstable transformation *X* to *B* separately from stable transformation *A* to *X*. For example, consider that the level *A* consists of models in an application-domain-specific language [2], and that level *B* consists of middleware platforms, such as CORBA/CCM [5, 9] and Web Services [14, 15]. Instead of defining a transformation directly from *A* to *B*, one may consider the introduction of EDOC CCA models [11] as intermediate models at level *X*, capturing a transformation from the domain-specific language to a solution that is more stable than middleware platforms. Additional transformations that do not have to consider the specificities of the domain-specific language can be used to transform the EDOC CCA models to CORBA/CCM or Web Services PSMs. Clearly, this solution requires the stability of the intermediate level *X*, in the example, EDOC CCA models. This solution is depicted in Figure 2(a).

A transformation between levels *A* and *B* is also specific to the (abstract) platform of the source level *A*. Introducing an intermediate level of models may also lead to the reuse of the transformation from the intermediate model to the target model. For example, consider that the level *A* consists of models in different application-domain-specific languages, and level *B* consists of Web Services. Introducing an intermediate level *X*, e.g., populated with EDOC CCA models allows us to reuse the general-purpose EDOC to Web Services transformation. This transformation is not "contaminated" with application-domain-specific issues. Again, this solution requires the stability of the intermediate level *X*. This solution is depicted in Figure 2(b). While we have presented the two solutions separately, they could be combined, as depicted in Figure 2 (c).

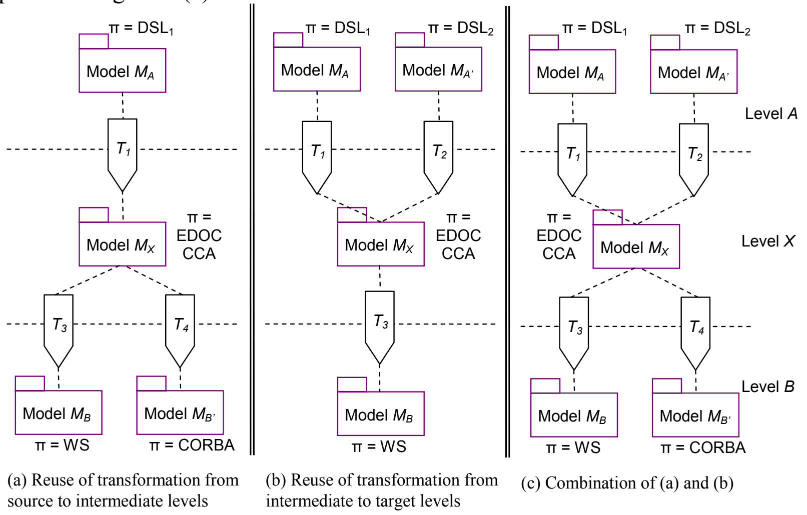


Fig. 2. Reuse of transformations due to introduction of intermediate level of models

In order to justify the introduction of the intermediate levels of models *X*, the abstract platform of the level *X* must be suitable for a large number of applications that can be described at level *A* and realized on platforms at level *B*. In our example, the consequence of this observation is that the abstract platform at level *X* should be independent of application domains at level *A* and independent of technology platforms at level *B*. In addition, standardization of this abstract platform may be necessary to increase the opportunities for the reuse of transformations to and from the intermediate level. The EDOC CCA is an example of such an abstract platform, allowing the description of distributed application in terms of components and their interconnection in terms of messages exchanged through ports.

The same pattern of transformation reusability can be observed when considering the transformation of EDOC CCA models at level *X* to models at the level of programming languages such as Java. In this case, level *B* in Figure 2 can be regarded as an intermediate level in the transformation, consisting of CORBA and Web Services-specific models. These models are transformed into Java interfaces, stubs and skeletons through standardized transformations [7, 12]. These transformations are executed through tools such as the one available in [13] and the ones listed in [6].

3.2 Partially automated transformations

It may be necessary to introduce an intermediate level of models between a source and a target level when no automated transformation can be defined directly, or when automated transformations produce results that do not satisfy non-functional requirements. By introducing an intermediate level of models, intermediate models can be elaborated upon, e.g., incremented, modified, combined with additional models and marked. The intermediate level can be regarded as a means to systematically lowering the degree of automation, and introducing opportunities to insert design decisions in the transformation from source to target models.

For example, let us consider again level *A* consisting of models in application-domain-specific languages, level *X* consisting of EDOC CCA models and level *B* consisting of CORBA/CCM and Web Services-specific models. This situation is depicted in Figure 3. In this example, marking EDOC CCA models manually is a means to specify properties that are not stated in source nor intermediate models and that may be required for the realization of the application on a target middleware platform. These properties may be requirements on the replication of components to satisfy availability requirements, requirements on the potential location of components in the distributed environment to satisfy time performance requirements, requirements on the persistency mechanisms required, etc. These requirements refer to specific components in the EDOC CCA models and cannot be specified meaningfully at level *A* or derived directly from EDOC CCA models.

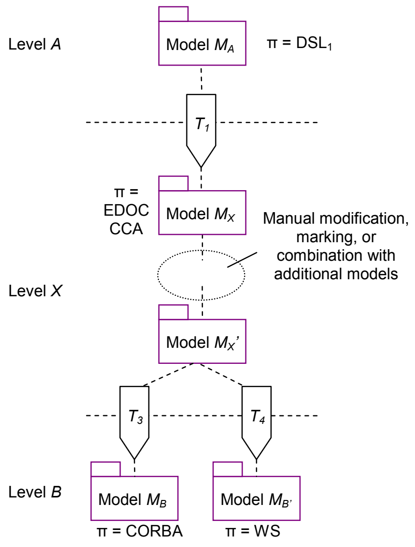


Fig. 3. Intermediate models as means to introduce design decisions

Reducing the level of automation of transformations incur additional costs on the introduction of an intermediate level of models. Changes in models at a high-level of platform-independence may lead to changes in all intermediate models and their associated markings. If intermediate models affected by changes need to be modified or marked manually, propagation of changes may lead to high costs. In contrast, in fully automated transformation chains, changes are automatically propagated through transformation. Since the state-of-the-art still requires significant developer intervention along transformation chains, the propagation of changes contributes to a large portion of the costs incurred by introducing separate levels of models. These costs should ideally be contained by appropriate traceability mechanisms in MDA tools.

With the introduction of an intermediate level of models, it may be necessary to develop specific languages, metamodels, profiles or marking models for that level. This incurs some additional effort for the preparation activities. For the case of partially automated transformation, developers using the intermediate models in execution activities must learn how to use the specific metamodels, profiles, or marking models required at that level, which usually incurs training costs and increases the threshold for developers to apply the particular model-driven development process.

4 Concluding remarks

In MDA development, opportunities for reuse of transformations play an important role in deciding the organization of the execution phase in terms of levels of models and transformations. A single transformation from high-level models to implementations may be costly to develop and is rendered useless in the face of technology platform changes. Given that technology platforms are generally regarded as unstable, it is important to attempt to recognize (intermediate) stable abstract platforms that can be used for a large number of applications. This allows transformations to and from this intermediate abstract platform to be reused.

In the example we have presented, we have considered an intermediate level of models based on the EDOC CCA UML profile, which enables the modeling of distributed applications as recursive compositions of abstract components. Recently, similar modeling capabilities have been incorporated in UML 2.0, with the introduction of composite structures [10]. Consequently, UML 2.0 and the EDOC CCA Profile can be seen as alternatives for modeling distributed applications. The proliferation of different (incompatible) intermediate levels of models reduces the opportunities for large-scale reuse of intermediate models and transformations to and from intermediate models. This calls for the standardization of a small number of abstract platforms that are, to a great extent, application-domain-neutral and platform-independent.

A conclusive study with respect to the costs and benefits of introducing different levels of models requires empirical verification. Such a study should consider a multitude of application requirements, as well as the opportunities for reuse across different instances of model-driven development projects.

In the absence of such an empirical study, we have discussed, in general terms, the benefits and costs of introducing different levels of models and transformations. We believe this forms a basis to enable trade-off analysis between the different factors in the preparation phase of MDA development.

Evaluating these trade-offs at early stages of development remains nevertheless a challenging activity, since the benefits of the separation PIM/PSM must be considered on the long run, particularly due to the role of reuse of models and transformations. Important open issues are how to estimate the stability of concrete platforms, application domains and applications and how to define stable abstract platforms that should be standardized.

Acknowledgements

This work is part of the Freeband A-MUSE project. Freeband (http://www.freeband.nl) is sponsored by the Dutch government under contract BSIK 03025. This work is also partly supported by the European Commission within the MODA-TEL IST project (http://www.modatel.org).

References

1. Almeida, J.P.A., van Sinderen, M., Ferreira Pires, L., Quartel, D.: A systematic approach to platform-independent design based on the service concept. In: Proceedings 7th IEEE Intl. Enterprise Distributed Object Computing Conference (EDOC 2003), IEEE Computer Society, Los Alamitos, CA (Sept. 2003) 112–123
2. van Deursen, A., Klint, P., and Visser, J.: Domain-Specific Languages: An Annotated Bibliography. In: ACM SIGPLAN Notices 35(6), (June 2000) 26–36
3. Almeida, J.P.A., Dijkman, R., van Sinderen, M., Ferreira Pires, L.: On the Notion of Abstract Platform in MDA Development. In: Proceedings 8th IEEE Intl. Enterprise Distributed Object Computing Conference (EDOC 2004), IEEE Computer Society, Los Alamitos, CA (to appear)
4. Gavras, A., Belaunde, M., Ferreira Pires, L., Almeida, J.P.A.: Towards an MDA-based development methodology for distributed applications. In: Proceedings of the 1st European Workshop on Model-Driven Architecture with Emphasis on Industrial Applications (MDA-IA 2004), CTIT Technical Report TR-CTIT-04-12, University of Twente, ISSN 1381 - 3625, Enschede, The Netherlands (March 2004) 43–51
5. Object Management Group: Common Object Request Broker Architecture: Core Specification, Version 3.0, formal/02-12-06 (2002)
6. Object Management Group: Getting Specs and Products, available at http://www.omg.org/gettingstarted/specsandprods.htm#GetProds
7. Object Management Group: IDL to Java Language Mapping, v1.2, formal/02-08-05 (2002).
8. Object Management Group: MDA-Guide, v1.0.1, omg/03-06-01 (June 2003)
9. Object Management Group: CORBA Component Model, Version 3.0, formal/02-06-65 (2002)
10. Object Management Group: UML 2.0 Superstructure, ptc/03-08-02 (2003)
11. Object Management Group: UML Profile for Enterprise Distributed Object Computing Specification, ptc/02-02-05 (2002)
12. Sun Microsystems, Inc.: JSR-000224 Java API for XML-Based RPC 2.0 (June 2003).
13. Sun Microsystems, Inc.: Java Web Services Developer Pack (Java WSDP), available at http://java.sun.com/webservices/jwsdp/index.jsp
14. World Wide Web Consortium: SOAP Version 1.2 Part 1: Messaging Framework, W3C Proposed Recommendation (2003), available at http://www.w3.org/TR/soap12-part1
15. World Wide Web Consortium: Web Services Description Language (WSDL) 1.1, W3C Note (2001), available at http://www.w3.org/TR/wsdl

A M3-Neutral Infrastructure for System Engineering

Olivier le Merdy (olivier.lemerdy@free.fr)

Sodius SAS (Nantes, www.sodius.com) and Ecole des Mines de Nantes (www.emn.fr)

Abstract

In this paper we report on some of the research activities at the Sodius Company in the domain of model-based system engineering. We start from the idea that even if Systems Engineering and Software Engineering, it is possible to create bridges at the highest level of abstraction and thus create correspondence at lower levels. The main message of this paper is that it is possible to consider software engineering and system engineering as two similarly organized areas, based on different metametamodels (M3-level). Consequently building bridges between these spaces at the M3-level seems to offer some significant advantages that will be discussed in the paper. We illustrate the space of system engineering with the well established CORE set of standards.

1 Introduction

Model engineering (or MDE for Model Driven Engineering) is being considered as an important departure from traditional techniques in such areas as software engineering, system engineering and data engineering. In software engineering, the MDA™ approach proposed by OMG in November 2000 allows separation of platform dependent from platform independent aspects in software construction and maintenance. More generally MDE is proposing to use models to capture specific aspects of a system under construction or maintenance, not only the business and platform aspects.

In the system engineering domain, a similar organization has been used for the last twenty years, mainly based on the TRW standard. However the overall organization was more implicit than explicit.

This paper describes one ongoing project at the Sodius Company in Nantes. The goal is to define a generic experimental advanced model management platform for system engineering. The idea is to consider that we have similarly organized technical spaces (MDA, CORE, Step/Express, Grammarware, XML, DBMS, XML, etc.). For each of these we have an implicit or explicit so-called M3-level. The MOF notation for MDA or the EBNF notation for grammarware play this role of defining, with different precision, the representation system for the entire technical space. In addition to this general M3-level organization, each space offers, at the M2-level, a rich set of specific domain specific languages (DSLs). These DSLs may be called grammars, metamodels, ontologies, DTDs, XML schemas, etc. Since these DSLs are used to capture specific aspects of systems, their relations or combinations is presently an important research concern. Transformation of programs written in various DSLs is one current very active research activity.

In this paper we propose the idea that it should be possible to establish generic coordination between different technical spaces by making explicit the M3-level properties and providing domain-independent transformation facilities at this level. This would be more efficient than providing ad-hoc, case by case transformation between various DSLs belonging to the same or different technical spaces.

This paper is thus organized as follows. In section 2 we introduce some general considerations on the three layer conjecture. Section 3 presents the domain of system engineering and the CORE set of standard. In Section 4, we show how the idea of defining bridges between these spaces at the M3-level may bring a lot of significant economies and other advantages. Finally we conclude by summarizing the project goals and sketching possible extension paths.

2 The 3-Layer Conjecture

In this section we recall the main characteristics of the three layer conjecture and we introduce one important technical space, namely the software engineering (MDA).

2.1 The OMG MDA Space

Each technical space is organized on a metametamodel (explicit or implicit) and a collection of metamodels. For the OMG/MDA the MOF and the collection of standard metamodels and UML profiles play this role.

In November 2000 the OMG proposed a new approach to interoperability named MDA™ (Model-Driven Architecture) [8]. MDA is one example of a much broader approach known as Model Driven Engineering encompassing many popular research trends like generative programming, domain specific languages, model-integrated computing, model management and much more.

The basic assumption in MDE is the consideration of models as first class entities. A model is an artifact that conforms to a metamodel and that represents a given aspect of a system. These relations of conformance and representation are central to model engineering [1]. A model is composed of model elements and conforms to a unique metamodel. This metamodel describes the various kinds of contained model elements and the way they are arranged, related and constrained. A language intended to define metamodels and models is called a metametamodel.

The OMG/MDA proposes the MOF (Meta Object Facility) as such a language. The Eclipse metametamodel is part of EMF and is compatible with MOF 2.0. This language has the power of UML class diagrams complemented by the OCL assertion and navigation language.

2.2 Technical spaces

There are other representation systems that may also offer, outside the MDA strict boundaries, similar model engineering facilities. We call them technical spaces [7]. They are often based on a three level organization like the metametamodel, metamodel and model of the MDA. One example is grammarware [7] with EBNF, grammars and programs but we could also consider XML documents, Semantic Web, DBMS, ontology engineering, etc. A Java program may be considered as a model conforming to the Java grammar. As a consequence we may consider strict (MDA)-models, i.e. MOF-based like a UML model but also more general models like a source Java program, an XML document, a relational DBMS schema, etc.

The main role of the M3-level is to define the representation system for underlying levels. The MOF for example is based on some kind of non-directed graphs where nodes are model elements and links are associations. The notion of association end plays an important role in this representation system. Within the grammarware space we have the specific representation of abstract syntax trees while within the XML document space we have also trees, but with very different set of constraints.

Associated to the basic representation system, there is a need to offer a navigation language. For MDA the language that plays this role is OCL, based on the specific nature of MDA models and metamodels. OCL for example know how to handle association ends. For the XML document space, the corresponding notation is XPath that takes into account the specific nature of XML trees. As a matter of fact OCL is more than a navigation language and also serves as an assertion language and even as a side-effect fee programming language for making requests on models and metamodels.

At the M3-level when the representation system and corresponding navigation and assertion notations are defined, there are also several other domain-independent facilities that need to be provided. In MDA for example generic conversion bridges and protocols are defined for communication with other technical spaces:

- XMI (XML Model Interchange) for bridging with the XML space
- JMI (Java Model Interchange) for bridging with the Java space
- CMI (Corba Model Interchange) for bridging with the Corba space

Obviously these facilities may evolve and provide more capabilities to the MDA technical space. We may even see many other domain-independent possibilities being available at the M3-level like general repositories for storing and retrieving any kind of model or metamodel, with different access modes and protocol (streamed, by element navigation, event-based, transaction based, with versioning, etc.).

3 System engineering

The system engineering technical space will be illustrated here by the CORE set of standards.

We provide in this section a metametamodel of this space and describe some specific DSLs by metamodels based on this CORE M3-level facility.

First assumption is that Systems Engineering gets very specific challenges in comparison to Software Engineering.

The role of the Laws of World: Systems are ruled by laws of Physics and Sociology. The influence of the System on its own context has to be taken into account.

The multiplicity of the disciplines and cultures: Systems involve lots of different actors who can have different interpretations of the same notions (e.g. Interface, Function).

The stake of the design vs integration: It is nearly impossible to test Systems at implementation level, for various physical, social or political reasons. Systems have to be validated at design level, before implementation.

The management at the Life Cycle level: The system desing shall take into account the evolution and the future ruptures and transitions within the life cycle.

Assuming these fundamental differences in terms of challenges, M2 level languages are also completely different. However, it is possible to identify for each of these sets of languages some common properties allowing to specify a compliant meta-meta-model. The comparison between M3-level language of Systems Engineering and Software Engineering shows similarities and thus bridgeability.

It is thus possible to define mapping rules between meta-meta-models in order to make metamodels transformation automatic.

The idea of metamodel agnostic systems has been accepted. We suggest here the idea that metametamodel agnostic systems are not much more difficult to handle and that they could bring significant advantages.

Furthermore we are presently convinced that the technological level has reached the point where it should be feasible to build a common open model engineering platform capable of handling artifacts based on different meta-meta-models.

3.1 CORE meta-meta-model (M3)

See Appendix A for a UML diagram of CORE meta-meta-model

CORE is based on the entity-relation-attribute approach and thus provides a number of meta-meta-model elements:

- The *Schema* is the enclosing element of CORE meta-meta-model. A *Schema* instance represents the meta-model itself.
- The *ModelElement* entity represents the basic element of a given CORE Schema. It is an abstract supertype containing common fields of all meta-model elements, like “name” or “creator”.
- A *Facility* instance represents a group of *Class* instances. A given Class instance can be owned by multiple *Facility* instances.
- The *AttributedElement* entity is an abstract supertype representing the ability to own *Attributes* (see thereafter).
- A *Class* instance represents a given concept in a meta-model.
- A *Relation* instance represents a link between two *Class* instances. Each *Relation* instance has a complement, which is the reverse *Relation*.
- An *Attribute* instance represents a property of a given *AttributedElement* instance.
- A *PossibleValue* instance represents a certain value that can be taken to given *Attribute* instance.
- A *Target* instance comes with a *Relation* instance and gives every *Class* instance reachable through this relation from a given *Class* instance.

3.2 CORE meta-model (M2)

The basic CORE Schema is based on the meta-model TRW and provides a broad set of elements usable in modeling systems. This Schema can be further enriched by adding, modifying or deleting elements – classes, possible values, relations… – specific to a given domain. Such an enriched Schema can then be considered as a DSL and as a specific meta-model.

For instance, specific metamodels exist for C4ISR (***C**ontrol **C**ommand **C**ommunication **C**omputer **I**ntelligence **S**urveillance **R**econnaissance*) and

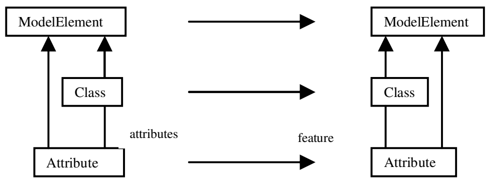
DODAF (***D****epartment* ***o****f* ***D****efense* ***A****rchitecture* ***F****ramework*).
As a DSL, a specific CORE meta-model can own a large number of elements spread between "essential" – elements common to every meta-model and undeletable – and "non-essential" ones. Essential elements cover classes necessary to any meta-model, such as the "System" whose instance would represent the real system which is modeled.

4 Bridging spaces

We describe here how the previous infrastructure may be used to define generic bridging facilities between these spaces.

4.1 M3 to M3 mapping

A *Schema* instance represents the meta-model itself and thus can be mapped in UML by a *Model* instance. Indeed, we should keep in mind that a meta-model can be considered as a model expressed in a meta-model that would be the meta-meta-model.
There is a correspondence between the notions of CORE *ModelElement* and UML *ModelElement*. Similarly, there is a correspondence respectively between notions of CORE *Attribute* and UML *Attribute* and between notions of CORE *Class* and UML *Class*.

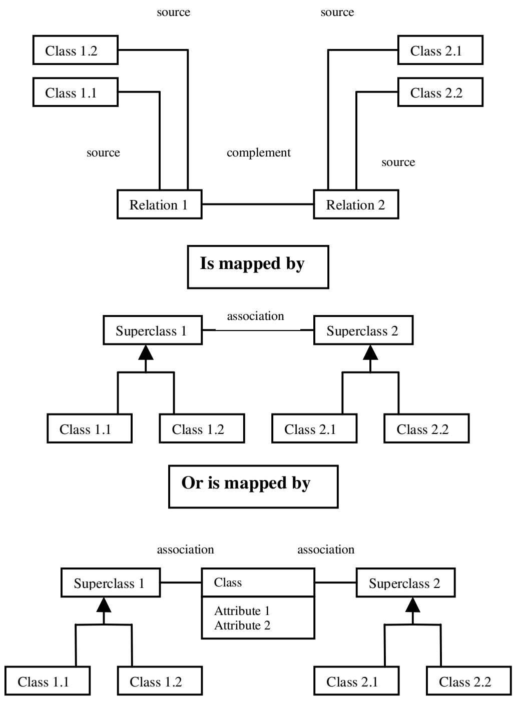


4.1.1. Schema of direct correspondences

Some of the links between and fields of these elements get their equivalent in UML representation:

- CORE *Class* "parent" link becomes a UML *Generalization*.
- CORE *Attribute* "initialValue" field becomes a UML *Expression* linked to the corresponding UML *Attribute* through the "initialValue" link.
- CORE *ModelElement* "abstract" field data is stored in the equivalent UML *ModelElement* "isAbstract" field.
- CORE *ModelElement* "schema" link which links each *ModelElement* instance to the top-level *Schema* is mapped by a "namespace" link between the corresponding UML *ModelElement* and the top-level UML *Model*.

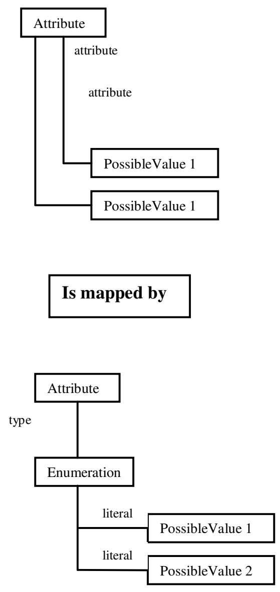
A CORE *Facility* can be mapped with a UML *Package*. UML *Class*es corresponding to this *Facility*'s CORE *Class*es are nested in this Facility through a UML *Dependency*.
Mapping a *Relation* involves to take into account the CORE *Relation* itself and its complement. Each of this relation is mapped by a super-class of all *Class*es sources of this relation, and another super-class of all *Class*es source of the complement. The link between super-classes and UML *Class*es is done through a UML *Generalization*.
Depending on whether the couple relation-complement owns *Attribute*s or not, the mapping is a direct UML *Association* between the two super-classes or an intermediary UML *Class* owning the UML *Attribute*s



4.1.2. Schema of Relations mapping

Properties of CORE *AttributedElement* are transferred to corresponding UML *Class*es and attributed *Relation*s. The relation "owner-attributes" is mapped by a UML *Association* "owner-feature".

CORE *PossibleValue*s are mapped with UML *EnumerationLiteral*s. These literals are attached to an *Enumeration* typing the *Attribute*.

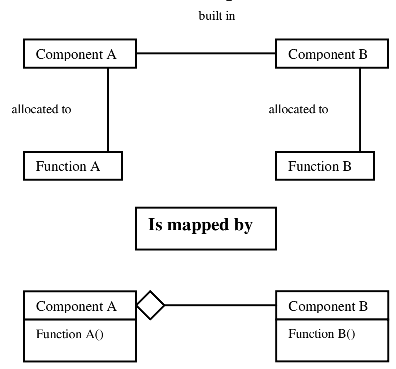


4.1.3. Schema of Possible Values mapping

CORE elements fields that do not get their equivalent in UML are stored in UML *Tagged-Value* attached to these CORE elements equivalent in UML. For example, CORE *ModelElement* "alias" field will be stored in a UML *Tagged-Value* named "alias" tagged to the corresponding UML *ModelElement*.

4.2 Applications of M3 to M3 mapping

Mapping CORE M3 Infrastructure with MOF M3 corresponding level allows a broad field of applications. The main purpose is the capacity of automatic meta-model translation by allowing definition of translation rules from one meta-meta-model to the other. This means significant economies in terms of time, making M2-level manual mapping useless. This also means significant gains in terms of quality of the resulting meta-model, thanks to automatic translation.
This allows then to work with automatically generated M2-level meta-models and, possibly, automatically generated M2-level mapping between both meta-models, with corresponding M1-level transformation tools.
Example of this M2 level transformation would be automatic transformation of CORE *Components* and attached CORE *Functions* through a link of "allocated to" in stereotyped UML *Classes* with attached UML *Operations*.



4.2.1. Schema of M2-level mapping for Component and Functions

As seen is the precedent section, working at M3-level allows to write clear and simple transformation rules thanks to the high level of abstraction and the fewer types of elements.

4.3 Benefits

Discussed mapping and applications offer a number of benefits:

- Seamless system to software process-communication
- Increase traceability and reliability.
- Direct interface model and code generation, since the interface definition belongs to the system level.

5 Conclusions

We have presented here some work in the application of MDE ideas to the domain of system engineering. MDA is probably now the most advanced and visible technical space of MDE in software engiennerin, with practical tools like Eclipse EMF being defined and becoming widely available. We believe it is possible to conciliate the best of both worlds (software engineering and system engineering) by a clear and regular framework based on the idea of technical spaces. Building generic bridges at the representation level (i.e. the M3-level) seems a very promising engineering practice. We have provided some illustrations in support of this hypothesis. There is still much work to be done in this area. However if the general framework is shown feasible in these areas of system and software engineering, it may probably also be applied to many other areas as well.

Acknowledgements

This work has been benefited from the constant support and encouragements of Jean-Philippe Lerat, head of the Sodius Company and of other staff of the company.

About the Authors

Olivier le Merdy is a student at the Ecole des Mines de Nantes. He has been working for several months at the Sodius Company.

References

[1] Sodius. Available from www.sodius.com

[2] OMG/MOF: Meta Object Facility (MOF) Specification. OMG Document AD/97-08-14, September 1997. Available from www.omg.org

[3] OMG/RFP/QVT: MOF 2.0 Query/Views/Transformations RFP, OMG document ad/2002-04-10. Available from www.omg.org

[4] OMG/XMI: XML Model Interchange (XMI) OMG Document AD/98-10-05, October 1998. Available from www.omg.org

[5] Bézivin, J.: In search of a Basic Principle for Model Driven Engineering, Novatica/Upgrade, Vol. V, N°2, (April 2004), pp. 21-24, http://www.upgrade-cepis.org/issues/2004/2/upgrade-vol-V-2.html

[6] Booch G., Brown A., Iyengar S., Rumbaugh J., Selic B.: The IBM MDA Manifesto The MDA Journal, May 2004, http://www.bptrends.com/publicationfiles/05-04%20COL%20IBM%20Manifesto%20-%20Frankel%20-3.pdf

[7] Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine, 2002.

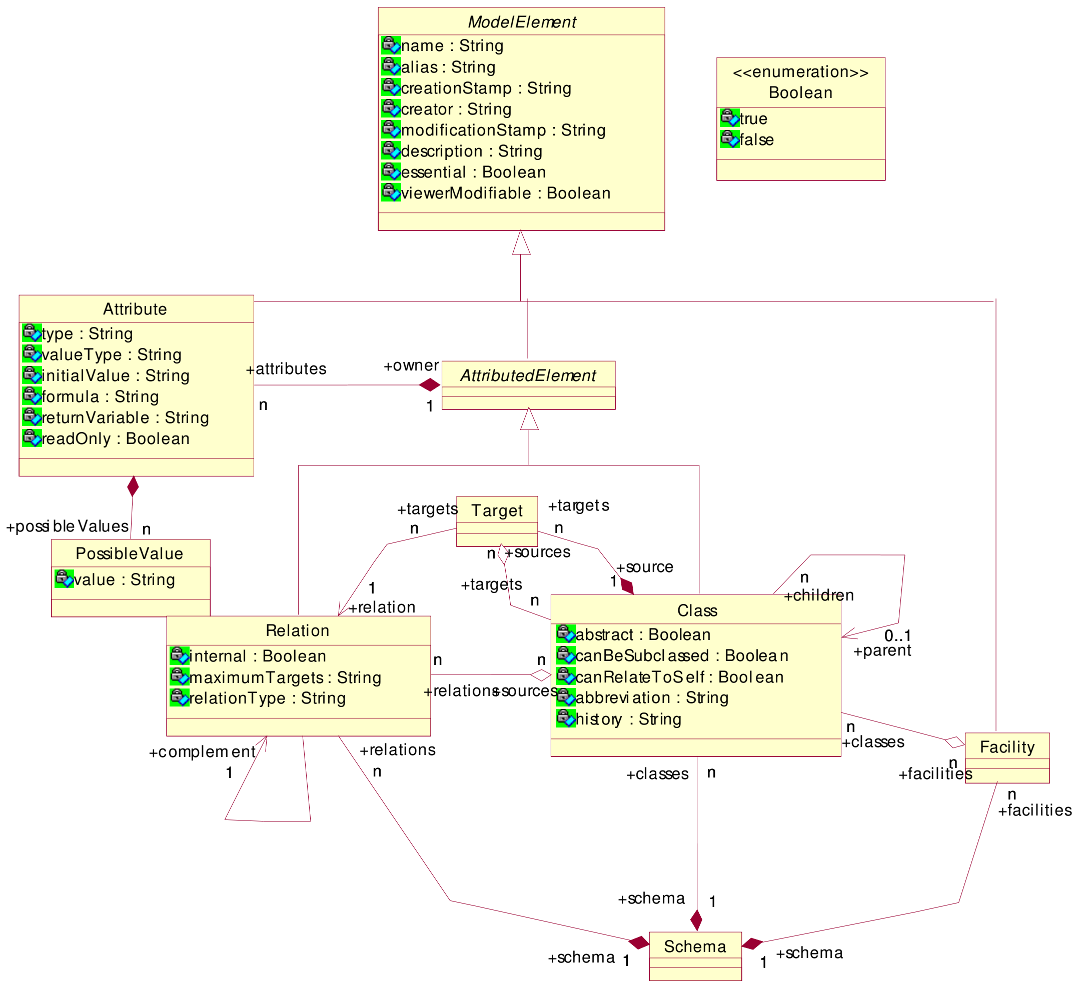
[8] Soley, R. & the OMG staff: MDA, Model-Driven Architecture, (November 2000), http://www.omg.org/mda/presentations.htm

[9] Vitech Corporation, founder of CORE tools. Available from www.vtcorp.com

[10] SysML. Available from www.sysml.org

[11] AP233. Available from http://step.jpl.nasa.gov/AP233/

Appendix A: UML Diagram of Core meta-meta-model



A formal MDA approach for mobile health systems

Val Jones, Arend Rensink, Theo Ruys, Ed Brinksma and Aart van Halteren.

Department of Electrical Engineering, Mathematics and Computer Science
PO Box 217, 7500 AE Enschede, The Netherlands
v.m.jones@utwente.nl

Abstract. M-health systems are safety critical systems intended for use by the public and are therefore characterized by especially strict requirements relating to safety, security, correctness, reliability, adaptability and user friendliness. This position paper proposes a methodology which realizes the MDA approach by utilizing formal methods to support verification, validation and transformation. The objective is to investigate the use of MDA enriched by formal methods to define a generic, evolvable architecture for m-health services which facilitates the rapid development and deployment of high quality adaptable m-health services.

1 Introduction

Currently available m-health systems range from simple alarm functions through patient monitoring functions to complete disease management systems. These systems tend to be closed, proprietary systems targeted at a single health condition or physiological measurement. Our vision is of an open and generic m-health service platform which can support an unlimited and evolving range of m-health devices and services including applications requiring high speed high bandwidth transmission and sophisticated analysis and interpretation of time-oriented clinical data [1], [2]. Such an m-health platform should support any combination of functionality sets allowing services to be customized to the needs of the individual at a certain point in time. It should also be accept on-the-fly upgrades to existing applications as well as completely new services. The service platform must therefore be (hardware and software) platform independent, flexible and adaptable.

The approach proposed here is a realisation of the MDA approach using formal methods to provide a sound foundation for the rapid development of mobile health systems. Formal methods are applied to support validation (by prototyping, model checking and formally based testing) and model transformation. The resulting methodology is expected to yield a robust software engineering approach for the development of mobile health services and applications.

The concept arises out of work undertaken in European projects including two FP5 IST Take Up Actions, MobiHealth (IST-2001-36006) and XMOTION (IST-2001-36059), which were completed in 2004. The research also draws on work at the University of Twente on model checking and on automatic test generation, implementation and execution. In the MobiHealth project a prototype health BAN (Body Area Network) was developed and trialled in various clinical settings. Many research issues arising from the experience gained are investigated in various new projects including the Dutch FREEBAND projects A-MUSE and AWARENESS and European initiatives MOSAIC (FP6-IST-2003-2 004341) and the Ambient Intelligence_at_Work initiative of the IST New Working Environments Unit. This paper discusses one of the lines of research arising, relating to software engineering methodologies. The approach proposed targets the rigorous development of a generic architecture for evolvable mobile health systems.

2 The m-health vision

2.1 Body Area Networks for healthcare

Body Area Networks [3], [4], [5] combined with wireless communications give a technology platform for realising the m-health vision. We define a *BAN* as a network of wearable devices which communicate amongst themselves (intra-BAN communication) and which may also communicate externally with a remote location (extra-BAN communication). A BAN consists of a mobile base unit or *MBU* (a central processor and gateway performing computation and external communication functions) and a set of devices. The MBU could be a PDA or a smart phone.

Specialising this concept, an *m-health* BAN is defined as a network of wearable medical devices which communicate amongst themselves (eg via Bluetooth) and externally (eg. via GPRS or UMTS) with a remote healthcare location such as a hospital system, a medical call centre or a doctor's mobile system. Examples of medical devices which may be incorporated into a BAN are sensors (e.g. electrodes for measuring ECG, EMG or EEG) and actuators (for example controlling implanted drug delivery systems or pacemakers). There may be any number of different specialisations of the health BAN. A specialisation can be thought of as an extension of the generic health BAN by equipping it with a certain (set of) device set(s) and the associated software. An example would be a BAN for insulin dependent diabetes patients. The diabetes BAN could include two devices: a blood glucose monitor (sensor) controlling an implanted insulin pump (actuator). The diabetes management application could include of a set of distributed functions running locally on the BAN or remotely, or a mixture of the two. The distributed nature of the execution should be hidden from the user. Several specialisations of a health BAN have been trialled during the MobiHealth project [6], [7], [8], [9], [10].

2.2 Special requirements of m-health systems

Mobile healthcare systems for patients are safety critical systems intended for (possibly unsupervised) use by the public. These systems are therefore characterized by strict requirements relating to *safety, security, correctness, reliability* and *user friendliness*. In addition, the prospect of large scale deployment of m-health systems in the community brings requirements for *scalability*, *run-time adaptation* (eg. in response to changing network conditions) and *dynamic evolvability*. Finally, m-health systems should be based on a *generic architecture*. We need robust methodologies to support the development of such safety critical systems. Here we focus on *correctness*, *evolvability* and *genericity* properties.

3 The approach

The objective is to contribute to the rigorous development of a software architecture which is able to support a variety of future BAN-based m-health services. The intention is to apply OMG's Model Driven Architecture™ (MDA) [11], [12], [13], augmented by formally-based software engineering methodologies and tools, to the m-health application domain. MDA is selected because it addresses the complete development life cycle and promises portability, cross-platform interoperability, and platform independence. In particular it is selected to support *genericity* and *evolvability* of the architecture and *domain specific modelling*. In our application of MDA to m-health we emphasise the need for formality and make explicit the activities of verification and validation. MDA is thus enhanced with formal methods in order to support the critical *correctness* requirements of health systems. Formal methods will be used to support verification (by model checking) and validation (by model-based testing) of critical properties, and to test equivalence between models and implementations. Model checking enables verification of logical consistency and correctness properties of a specification and detection of a variety of errors and undesirable characteristics such as deadlocks and race conditions. Together with formal testing, model checking can give a high degree of confidence in the correctness of the design and implementation (ie of PIM, PSM and code).

3.1 Combining MDA and formal methods

Figure 1 depicts a concept space for instantiation of the MDA approach, showing

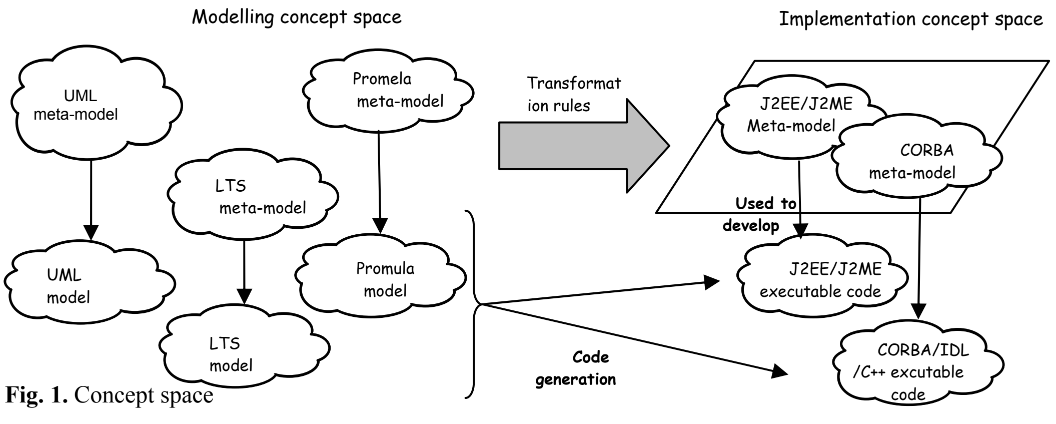


Fig. 1. Concept space

some candidate formalisms and implementation environments, and the role of meta-models and model transformation in deriving implementations.

The MDA approach is applied by developing a Platform Independent Model (PIM) and transforming it to one or more Platform Specific Models (PSMs) targeted at specific implementation environments. Applications are derived from the PSMs for those specific platforms. Model transformation refers to meta-models (models of the source and target languages/environments).

Complete proofs of correctness are demonstrably not feasible for realistic sized systems; however, we propose to use formal methods within the MDA framework to establish a high quality software production process which can give high levels of confidence in the correctness of the designed system. Formal validation techniques used include early prototyping (model execution by simulation); model verification; and model-based testing of implementations. The guidelines of [14] will be followed so that the formal verification is performed in a controlled and reproducible way.

Modelling is performed using executable formal or semi-formal languages (e.g., UML, OCL, *me too*). Verification approaches include model checking [15] with tool support (e.g. SPIN [16], [17]); for validation we use model-based testing (automatic test generation and execution [18] using tools such as TORX [19]) and rapid prototyping (e.g. the *me too* approach [20]). Possible implementation approaches include the transformation approaches of [21], [22], [23] and model transformation [24].

3.2 Some anticipated challenges

Although promising a usable software development process targeting interoperability, reusability and portability, MDA raises some interesting challenges, including:

1. How to represent the dynamic aspects of systems?
2. How to address what we may call the “Lossy transformation” problem; when the expressive power of the source language exceeds that of the target?
3. How to establish preservation of semantic properties - a problem made more intractable where the source or target language of a transformation lacks an explicit formal semantics?
4. How far can we go with auto generation of implementations from models?

3.3 Some proposed solutions

We will consider alternative formalisms to represent behaviour (e.g. process calculus and models based on generalised transition systems) in order to address problem 1 above. As well as UML we consider other more formally defined languages (including but not confined to the UML related OCL) in order to detect problem 2 and to address problem 3. (Adding alternative formalisms to the MDA repertoire implies development of meta-models, transformation definitions and (possibly) additional tools.) Problem 4 refers to the point that automatic generation of complete applications remains an unreachable goal. Generally parts of an implementation must be hand crafted. We propose to investigate how model transformation using formal methods can be applied where possible and then augmented by judicious use of principled software engineering techniques for development and validation of the remainder. A practical and scalable example which can form part of the solution is verification of the implementation by application of test suites automatically generated from the (platform independent) model.

Figure 2 shows one possible instantiation of our approach. An m-health application is modelled in UML, yielding a PIM (Platform Independent Model). Critical properties derived from the requirements are expressed formally (eg. as assertions). The UML model is transformed into a PROMELA model. The resulting PROMELA model together with the properties are input to the SPIN model checker, which verifies that these properties are met by the PROMELA model. So a degree of formal verification is achieved by model checking applied to the Promela version of the PIM. In this example, the target is a Java implementation. Applying model transformation again, a Java PSM is generated from the PROMELA PIM and Java code is derived from the Java PSM. A test suite is automatically generated from the PROMELA PIM using the test generation and execution tool TORX. The test suite is applied not to the model but to the Java implementation, providing formal validation by checking behavioural equivalence between model and implementation.

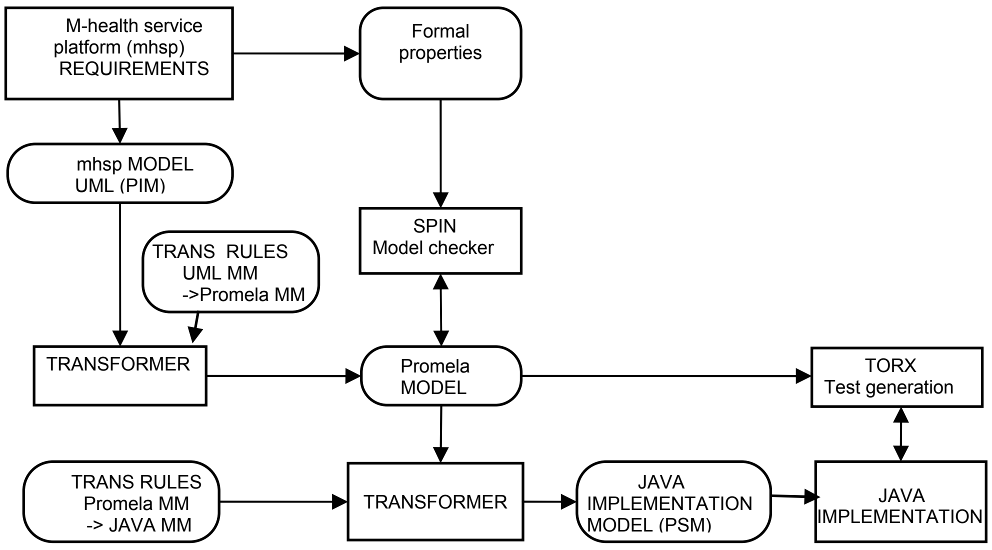

M-health service platform (mhsp) REQUIREMENTS
Formal properties
mhsp MODEL UML (PIM)
SPIN Model checker
TRANS RULES UML MM ->Promela MM
TRANSFORMER
Promela MODEL
TORX Test generation
TRANS RULES Promela MM -> JAVA MM
TRANSFORMER
JAVA IMPLEMENTATION MODEL (PSM)
JAVA IMPLEMENTATION

Fig. 2. One instantiation of the approach

We postulate a “Transformer”: a generic model transformation tool which accepts a set of transformation rules mapping language A to language B, and a model in language A, and automatically produces a model in language B which is behaviourally equivalent to the source model. Since as yet we have no such “omnipotent transformer” guaranteeing correctness preservation, we still need formal validation of the implementation by model-based testing.

Other possible instantiations of the approach will result, for example, from use of different modeling formalisms (eg. *me too* plus process calculus), or because different implementation platforms (eg Symbian) or languages (eg C#, SQL) are targeted, or by substitution of different validation methods (eg prototyping and/or simulation in place of model checking).

4 Discussion

The scientific focus of the proposed research lies in the investigation and advancement of software engineering methodologies for the development of domain specific services. This is achieved by testing theoretical developments from software engineering and formal methods against a real and complex engineering problem from the m-health domain and by instantiating the MDA approach for that domain. It is hoped that the research will increase understanding of the following issues:

- What are the real engineering challenges encountered by developers of distributed m-health services?
- How can we best model, validate and implement a software infrastructure that can be deployed in a distributed m-health service environment?
- What properties must a generic m-health architecture have in order to persist and support m-health product families (synchronic variation) and evolution of m-health products and services (diachronic variation)?
- How far can a fusion of the software engineering approaches of MDA and formal methods address these engineering challenges?
- Where are the boundaries between domain specificity and genericity (of models, model transformations and solutions)?

By exercising the chosen methods and tools on realistic m-health applications, we expect to derive a domain specific architecture for m-health services and a formally-based instantiation of the MDA approach. Hence the expected outputs include:

- A high level architecture for m-health services
- An MDA-oriented methodology for design and development of m-health services
- A proof of concept in the form of one or more applications of the methodology through to implementation. This will include models, meta-models, model transformations and prototype implementations.

The concept is in an early stage of development. Feedback from the MDA community is welcomed. It is hoped that through the proposed research we can make a contribution to MDA activities (eg via QVT [25]). Some of the QVT proposals are amenable to formalisation. It has been noted (eg. [26], [27]) that the theory of graph transformation appears to be especially suitable for the purposes of model transformation. If model transformation is defined on a formal footing, one can also expect to carry over formal verification results from one model to another.

References

1. Shahar, Y., and Musen, M.A. (1993). RÉSUMÉ: A temporal-abstraction system for patient monitoring. *Computers and Biomedical Research* **26**, 255–273. Reprinted in van Bemmel,

J.H., and McRay, A.T. (eds) (1994), *Yearbook of Medical Informatics 1994*, pp. 443–461, Stuttgart: F.K. Schattauer and The International Medical Informatics Association.

2. Shahar, Y., and Musen, M.A. (1996). Knowledge-based temporal abstraction in clinical domains. *Artificial Intelligence in Medicine* **8** (3), 267–298.

3. Zimmerman, T.G., 1999, 'Wireless networked devices: A new paradigm for computing and communication', *IBM Systems Journal*, Vol. 38, No 4.

4. Van Dam, K, S. Pitchers and M. Barnard, 'Body Area Networks: Towards a Wearable Future', Proc. WWRF kick off meeting, Munich, Germany, 6-7 March 2001; http://www.wireless-world-research.org/.

5. Schmidt, R., 2001, *Patients emit an aura of data*, Fraunhofer-Gesellschaft, www.fraunhofer.de/english/press/md/md2001/md11-2001_t1.html

6. Jones, V. M., Bults, R. A. G., Konstantas, D., Vierhout, P. A. M., 2001a, Healthcare PANs: Personal Area Networks for trauma care and home care, *Proceedings Fourth International Symposium on Wireless Personal Multimedia Communications* (WPMC), Sept. 9-12, 2001, Aalborg, Denmark, http://wpmc01.org/, ISBN 87-988568-0-4

7. Dimitri Konstantas, Val Jones, Richard Bults, Rainer Herzog, *MobiHealth – innovative 2.5 / 3G mobile services and applications for healthcare*, 11th IST Mobile Summit 2002, Thessaloniki, May 2002.

8. Dimitri Konstantas, Val Jones, Richard Bults and Rainer Herzog, "MobiHealth - Wireless mobile services and applications for healthcare*", International Conference On Telemedicine - Integration of Health Telematics into Medical Practice*, Sept. 22nd-25th, 2002, Regensburg, Germany.

9. Widya, A. van Halteren, V. Jones, R. Bults, D. Konstantas, P. Vierhout, J. Peuscher, 2003. Telematic Requirements for a Mobile and Wireless Healthcare System derived from Enterprise Models. *Proceedings IEEE ConTel 2003: 7th International Conference on Telecommunications, June 11-13, 2003*, Zagreb, Croatia.

10. Nikolay Dokovsky, Aart van Halteren, Ing Widya, BANip: Enabling remote healthcare monitoring with Body Area Networks, International Workshop on scientiFic engIneering of Distributed Java applIcations, November 27-28, 2003, Luxembourg, LUXEMBOURG.

11. Object Management Group, MDA website, http://www.omg.org/mda/

12..MDA Guide Version 1.0.1, © 2003, OMG, omg/2003-06-01, http://www.omg.org/docs/omg/03-06-01.pdf

13. Anneke Kleppe, Jos Warmer, Wim Bast, (2003) *MDA Explained: The Model Driven Architecture™: Practice and Promise*, Addison Wesley Professional.

14. Theo C. Ruys and Ed Brinksma, Managing the Verification Trajectory, *Software Tools for Technology Transfer* (STTT), 4:2, Feb. 2003, pp.246-259.

15. Theo C. Ruys, (2001), *Towards Effective Model Checking*, PhD Thesis, University of Twente, Enschede, The Netherlands,March 2001.

16. G.J. Holzmann,(1991) Design and Validation of Computer Protocols, Prentice Hall, New Jersey, 1991, ISBN 0-13-539925-4.

17. G.J. Holzmann, (2003) The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, ISBN 0-321-22862-6.

18. E. Brinksma, (1999). Formal methods for conformance testing: Theory can be practical! In N. Halbwachs and D. Peled, editors, *Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*, pages 44-46, Trento, July 1999. Springer.

19. J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8-12, 1999. EuroStar Conferences, Galway, Ireland.

20. Alexander H and Jones V (1990). *Software Design and Prototyping using* me too. London: Prentice Hall International. ISBN 0-13-820259-1.

21. Correctness Preserving Transformations for the Early Phases of Software Development; *T. Bolognesi, D. De Frutos, R. Langerak, D. Latella.I,IN* Bolognesi T, van de Lagemaat J and Vissers C.A. (ed), *LOTOSphere: Software Development with LOTOS*, pp. 348-368, Kluwer Academic Publishers, 1995.

22. Jones V (1995). Realization of CCR in C, In Bolognesi T, van de Lagemaat J and Vissers C.A. (ed), *LOTOSphere: Software Development with LOTOS*, pp. 348-368, Kluwer Academic Publishers, 1995.

23. Jones VM (1997) *Engineering an implementation of the OSI CCR Protocol using the information systems engineering techniques of formal specification and program transformation*. University of Twente, Centre for Telematics and Information Technology Technical Report series no. 97-19. ISSN 1381-3625.

24. Bézivin, J., Dupé, G., Jouault, F., Pitette, G., Rougui, J.E.: *First experiments with the ATL model transformation language: Transforming XSLT into XQuery*. OOPSLA 2003 Workshop, Anaheim, California, October 27, 2003

25. OMG MOF 2.0 Query / Views / Transformations Request for Proposals. URL: http://www.omg.org/cgi-bin/doc?ad/2002-4-10

26. Sabine Kuske, Martin Gogolla, Ralf Kollmann, Hans-Jörg Kreowski, An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In Butler et al., *Integrated Formal Methods, Third International Conference*, LNCS 2335, Springer 2002, pp. 11-28.

27. T. Mens. Conditional graph rewriting as a domain-independent formalism for software evolution. In Nagl et al, editors. *Applications of Graph Transformations with IndustrialRelevance,* volume 1779 of Lecture Notes in Computer Science. Springer-Verlag, 2000, pages 127–143.

Composition rules for PIM reuse

Salim Bouzitouna[1] and Marie-Pierre Gervais[1, 2]
[1]Laboratoire d'Informatique de Paris 6
8 rue du Capitaine Scott, F-75015 Paris
{Salim.Bouzitouna, Marie-Pierre.Gervais}@lip6.fr
[2]Université Paris X

Abstract: *In order to reduce the cost of the evolution of companies' applications, this evolution should be led in a systematic way by reusing existing applications. In MDA approach, this should be done by the reuse of PIM and PSM of the concerned applications. Indeed, the reuse of models exploits those that already exist and which have been checked and maintained. It aims to construct new applications by composing, extending or modifying existing distributed applications. To this end, we propose a new initiative of distributed applications' construction by reusing models in MDA approach. Our initiative is based on two principal points: the expression of the reuse of PIM and the automatic generation of glue binding their corresponding PSM from this expression. In this paper we focus on the first point which is the expression of the reuse in terms of composition, extension and modification of PIM.*

1. Introduction:

To make the migration of company's applications towards new platforms easier, MDA approach [OMG 03] recommends a well-delimited separation between business aspects and implementation details aspects of an application. This separation is expressed via two models: PIM (*Platform Independent Model*) which specifies business aspects of a distributed application and PSM (*Platform-Specific Model*) which specifies implementation details on a specific platform. However, we can observe that the merge and reorganization of companies requires the evolution of their applications. For instance, in the case of fusion of two companies, this evolution can be expressed in terms of composition of applications. It can be also expressed in terms of extension or modification if the functionalities of existing applications are respectively extended or modified. In order to reduce the cost of these evolutions the reuse of existing applications is essential. In MDA approach, this reuse consists in reusing PIM and PSM of the existing applications.

Many approaches are interested in the problem of reuse. If we consider known levels such as MDA PIM, MDA PSM or code, none of the current approaches deal with the reuse in these levels. The majority of the approaches provide reusability in terms of code and not of abstract models. Examples of such approaches are *Subject Oriented Programming* [Harrison 93], *Aspect Oriented Programming* [Kiczales 97] or *Component Oriented Programming* [OMG 99] [Sun 99]. Only few approaches provide reusability of abstract models, similar to PIM, such as the *Subject Oriented Design* [Clarke 01]. Moreover their means carry out direct changes on the reused models. This does not guarantee a good traceability of the evolution of the reused models.

We present a new initiative based on the reuse of models for the construction of new distributed applications in MDA approach. ***Our initiative is based on two principal points: the first one is the expression of the reuse of PIM in terms of composition, extension, and modification while the second one concerns the automatic generation of the glue from this expression.*** This glue binds PSM corresponding to the reused PIM. As it depends on the platforms considered for PSM, it could be considered as the component that will be used for the assembly of these PSM. Figure 1 shows the idea of composition of two applications.

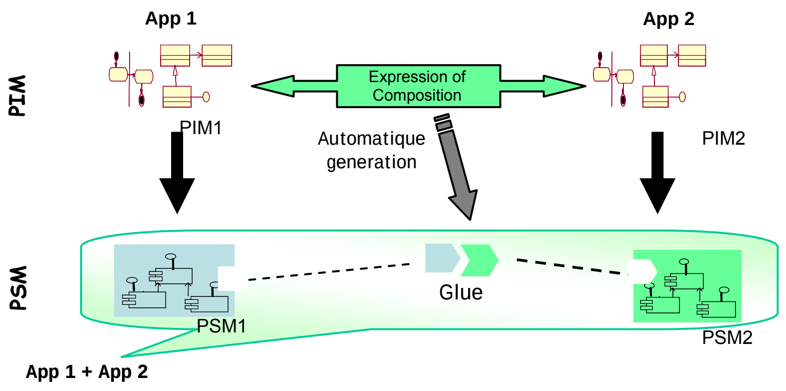


Figure 1. Our approach for the composition of MDA applications

Through this initiative, we propose a solution which deals with the reuse of MDA applications on all levels. It allows the reuse of models that already exist and which have been checked and maintained. At PIM level, the expression of the reuse of models is only used to describe how they are composed, extended or modified, but does not change them. This allows a suitable stability of reusable PIM by keeping good traceability of their evolution - which is the basic principle of MDA approach - since it supposes that the PIM of a given application remains stable. At PSM level, the glue allows the corresponding PSM to be kept unchanged, which therefore makes it possible to exploit the codes corresponding to these PSM in new applications with no change. Moreover, it allows to use the same PSM to build several new applications according to the intentions' of reuse expressions of PIM.

In this paper we focus on the first point of our initiative which is the expression of the reuse of PIM. For that, we study the different types of reuse of PIM. From these, we then define a set of rules for reuse expressions for composition as well as for extensibility of PIM. The last section concludes the article and presents some future works.

2. Integration of the reuse of PIM in MDA approach

2. 1 Expressions of PIM

PIM considered in MDA approach are expressed in a well-defined precise modeling language. This describes the structural aspect as well as the behavioral aspect of the application. The OMG recommends within the context of MDA approach the use of UML language [OMG97]. In this article, we are particularly interested in UML class diagram and UML collaboration diagram. These diagrams are very appropriate for expressing the structure and behavior of an application respectively. Using these two diagrams, we propose to describe an application independently of any platform. The class diagram represents the set of entities interacting in a given application, as well as the relations between them. It also expresses the progress of different operations defined by the entities used in the collaboration diagram. As UML recommends the gathering of these diagrams in packages according to application’s functionalities they describe, we consider that applications are packaged.

2.2 Reuse of PIM

Generally, the reuse of a software unit can be expressed by several intentions, illustrated in figure 2:

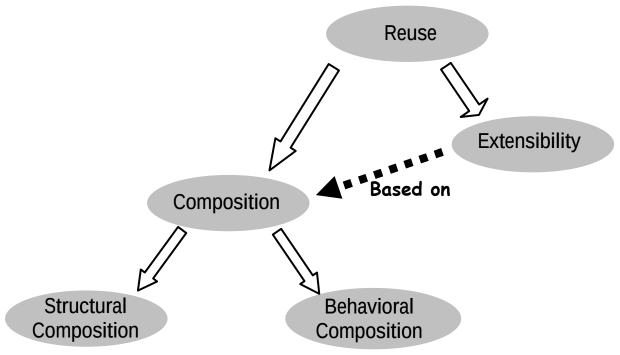


Figure 2. Different intentions of unit's reuse.

A first aspect of units' reuse is composition. It expresses the way in which this unit is assembled with others in order to form a new application. We consider two types of compositions: *structural composition* and *behavioral composition.* The structural composition aims at modifying elements of the units, while *the behavioral composition* aims at the expression of interactions between the various operations of the units. In our context the units correspond to PIM.

Applying structural composition at PIM level consists in focusing on UML class diagrams. The composition consists of merging different elements belonging to these models, such as classes and attributes. This merge consists in putting these elements together. However in order to avoid redundant elements, the elements which correspond to the same entity (*the classes' elements for example*) or the same property (*the attributes' elements for example*) will be represented by only one element among them.

We also consider modification as a form of structural composition. Basically, it consists in defining all the changes to be brought on a PIM, in another separate model. Then, it is a matter of replacing elements of the first model by those defined in the second one.

The behavioral composition is related to UML collaboration diagrams which correspond to the various PIM. It describes the interactions between the operations defined in the classes of these models. This composition consists, for example, in combining a set of operations belonging to different models by coordinating them in a given order.

The second aspect of units' reuse is extensibility. This consists in adding new functionalities to units**.** Most of reuse approaches recommend adding a new component such as *Subject Oriented Programming* [Harrison 93], *Aspect Oriented Programming* [Kiczales 97]. Their idea consists in placing all functionalities to be added in a new unit, and then composing it with the original units. Similarly to extend PIM functionalities, we propose to specify the new functionalities in a separate model and then compose them with the original model. This approach has many advantages. It will allow to keep a good traceability of the evolution of PIM. Furthermore, it allows to apply several extensions to same the PIM, which do not depend on others. We thus note that the composition of models also encompasses extensibility.

To express structural as well as behavioral composition, we define a set of rules. As these rules, applied on PIM, are abstractly defined, we call them *patterns of composition*. Thanks to these patterns, a designer can model the application he wants to build, modify or extend. However, contrary to the major trend, we do not advocate the elaboration of new PIM. Actually, many approaches, such as *Subject Oriented Design* [Clarke01], propose to apply rules on existing models in order to obtain new PIMs that replace current ones. In this way, latest changes are carried out on the current models. This does not guarantee a good traceability of the evolution of the reusable models. This compromises the basic principle of MDA approach which supposes that the PIM of a given application remains stable. To face these disadvantages, we propose to keep PIM unchanged when they are reused. Indeed, our rules do not apply to the PIM source model for building a new PIM. They are only used to express the composition between existing original models. The resulting model is composed of PIM original models and the newly defined composition rules. Figure 3 compares our step with those of other approaches.

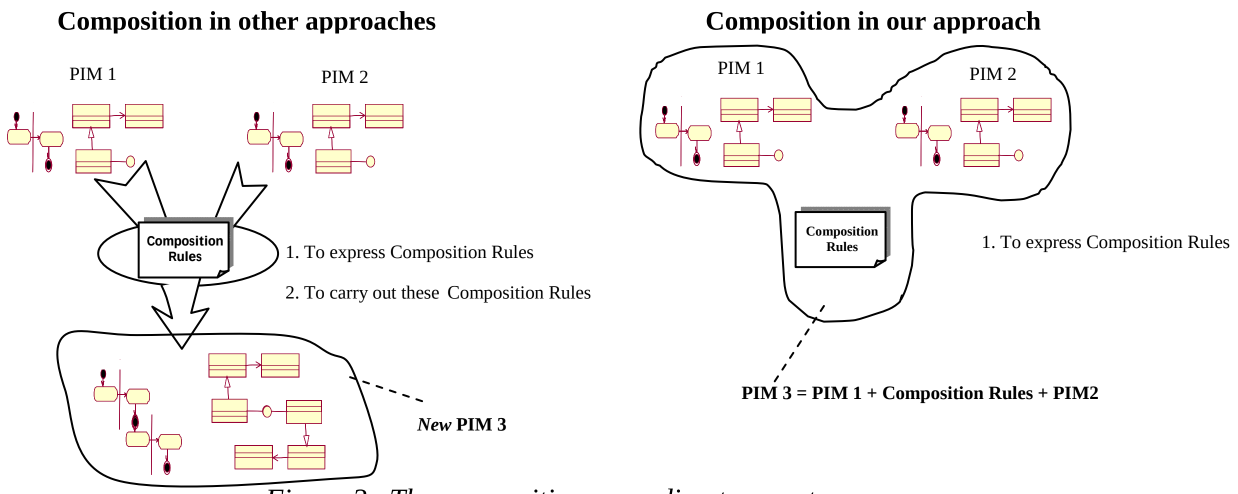

Composition in other approaches
PIM 1
PIM 2
Composition Rules
1. To express Composition Rules
2. To carry out these Composition Rules
New PIM 3
Composition in our approach
PIM 1
PIM 2
Composition Rules
1. To express Composition Rules
PIM 3 = PIM 1 + Composition Rules + PIM2

Figure 3. The composition according to our step vs the composition in the other approaches

Our rules of composition are defined as being composition patterns. This approach enables their later implementation by using any language that allows parsing models. This is proposed by many model transformation languages. To this end, we consider in the near future the use of MOF QVT [OMG 02] suggested by OMG. The choice of such a language allows compliance with OMG standards.

The set of the mentioned rules are presented in the next section.

3. Rules for PIM composition

To identify different compositions between PIM, we studied application construction approaches aiming at conceptual model's reuse as well as and those aiming at the code reuse such as [Clarke 01] [Van 99] [IBM 03a] [IBM 03b] [AspectJ 03]. We examined more particularly the means and techniques which they offer to make the composition of their component units. This enabled us to define a set of composition rules which allow to specify many types of composition of PIM.

For structural composition, these rules allow to identify more precisely, in models to be composed, different packages to be integrated, as well as elements that specify the same concept and which thus must be combined. For behavioral composition, these rules specify combination of operations defined in models to be composed. This combination consists in running all these operations when one of them is activated. However, control structures can be defined to modify the behavior of this run. We classified the rules which we defined in the three following categories.

3.1. Correspondence rules

Correspondence rules establish relation between elements (*packages, classes, operations, attributes)* of the models which will be later composed. These elements must be of the same type, and specify the same concept, but each element belongs to its own model. Correspondence rules do not specify how these elements can be combined. This is carried out by other rules which are defined in the second category.

Contrary to the *Subject Oriented Design* [Clarke 01], or *Subject Oriented Programming* [Kiczales 97], all correspondences must be expressed explicitly through correspondence rules. Elements having the same name in different models are not necessarily in correspondence. This avoids implicit compositions which are not wanted by the designer.

The following rule has been defined for expressing the correspondence between several packages:

- *CorrespondPackages [package1, package2…]*

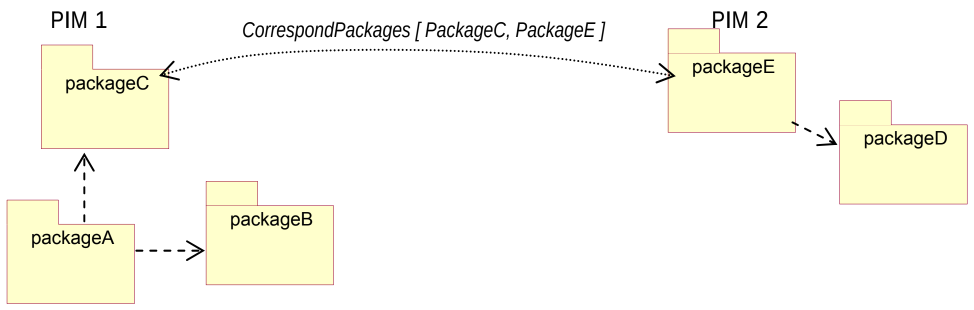


Figure4. Expression of correspondence between two packages

Figure 4 shows an expression of correspondence between two packages, each one belonging to separate model.

The expression of correspondence between packages is insufficient to express the composition between two models. We also need to specify the correspondence between their elements. This correspondence can be related to their sub-packages. In this case, it will be expressed with the same *CorrespondPackages* rule. On the other hand, it may be related to the classes of the elements. For this case, we define the following rule to express such correspondence:

- *CorrespondClasses [package1.Class1, package2.Class2…]*

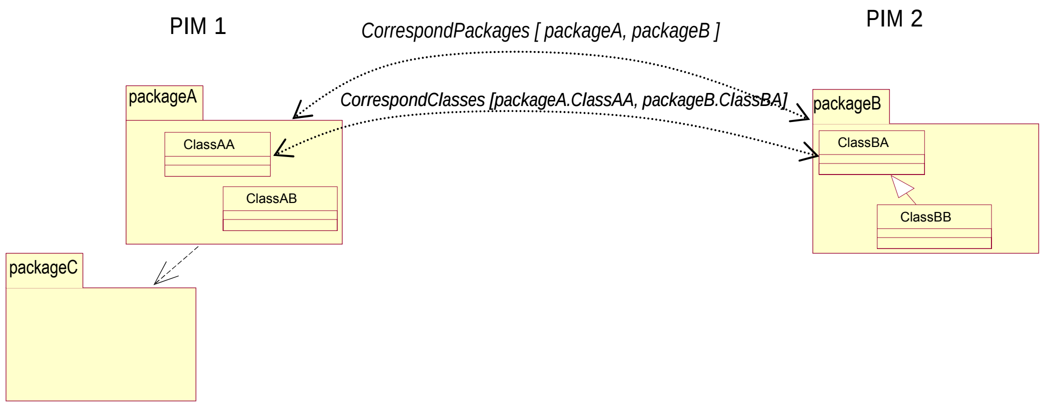

PIM 1
CorrespondPackages [packageA, packageB]
PIM 2
packageA
CorrespondClasses [packageA.ClassAA, packageB.ClassBA]
packageB
ClassAA
ClassBA
ClassAB
ClassBB
packageC
Figure 5. Expression of correspondence between two classes

Figure 5 shows the expression of correspondence between two classes: *ClassAA* defined in *packageA* and *ClassBA* defined in *PackageB.* These classes represent a priori the same entity. Note that this correspondence can be specified only if the correspondence between packages in which these classes are defined is also specified.

We can also express correspondences between attributes and operations defined in classes which have already been put in correspondence. Correspondence between attributes means that they represent the same property. Likewise, correspondence between operations of classes means that they aim at the same processing but they may perform it differently. For expressing these two types of correspondences, we propose the following rules.

Correspondence rule between the attributes:

- *CorrespondAttributes [package1.Class1.Att1, package2.Class2.Att2…]*

Correspondence Rule between the operations:

- *CorrespondOperations [package1.Class1.Op1, package2.Class2.Op2…]*

3.2. Combination rules

Combination rules are used to express the way in which composition is carried out between a set of elements (packages, classes, operations). These elements should be put beforehand in correspondence. Although a correspondence between a set of elements means that these elements represent the same concept, each one must define its proper sub-elements to specify this concept, according to its application. Thus, the composition of elements put in correspondence consists in unifying their sub-elements.

If there is a correspondence between two sub-elements, only one among them will have to be kept in their union. This is indicated by an expression of combination rules unifying elements which contain them. This indication is defined by a priority associated with each parameter of a combination rule. However, if new combination rules are defined between these sub-elements, they will cancel the priority defined between elements which contain them.

In addition, we regard the composition of a set of operations as being the execution of one or more operations in a given order. The operations to be executed as well as their order are defined using control structures which are specified in the combination rules of operations. These control structures correspond to conditional processing such as *if then, switch,* or iterative processing ones such as *for, while.*
To express a combination between many packages, the following rule is defined:

- *JoinPackages [package1, package2…]*

This rule expresses the union of classes (sub-elements) defined in each package *package1, package2…* In this union, classes which are in correspondence are represented by only one class, which is defined in the package with the greatest priority. This priority is assigned to each parameter of this rule, and corresponds to its order of appearance. Thus, classes defined in *package1* have more priority than those defined in *Package2* and so on.

We can also express combination between classes. They must be put in correspondence beforehand. To express this combination we define the following rule:

- *JoinClasses [package1.Class1, package2.Class2…]*

If a combination rule is expressed between *package1* and *package2,* a priority is assigned between their elements and thus between *Class1 and Class2.* By defining the combination rule above, the priority between these two classes are redefined. Like in a *JoinPackages* rule, the order of appearance of *JoinClasses* rule parameters defines their priorities. This defines the priority between sub-elements of classes placed in these parameters.

JoinClasses rule described above express the union of sub-elements in terms of operations and attributes defined in classes *package1.Class1 package2.Class2.* In this union attributes which are in correspondence are represented by only one attribute defined in *Class1.* Conversely, operations which are in correspondence are maintained while unifying their processing. This consists in executing all these operations when one of them is activated. The execution is carried out according to the order of priorities. Therefore, the execution of operations of *Class1* will precede the execution of that of Class2.

However, we can express the execution process of operations which are in correspondence differently from the one imposed by combination rules defined between their classes. This process may express the execution of some operations under certain conditions. It may also express the execution of one or more operations several times. To this end, we define a combination rule of operations. This rule introduces an execution process of these operations into a new operation which we call *ControlOperation.* It expresses the execution process of operations by using control structures such as *if then, switch, for* etc. Combination rule of operations is defined as follows:

- *JoinOperations[ControlOperation, package1.Class1.Op1, package2.Class2.Op2...]*

3.3. Replacement rules
Replacement rules are used to express updates of elements defined in a given model. These elements can be packages, classes, attributes or operations. An update of an element consists in replacing it by a new element of the same type, i.e. a class can be replaced only by one class, idem for operations and attributes. The definition of new elements instead of the updating of existing ones offers a good traceability of the evolution of the models.

Thus, we recommend to specify all updates of an existing model, in a separate model which we call substitute model. This one defines all new elements which will replace those defined in the original model. Therefore, it is also necessary to establish correspondences between the elements to be replaced in the original model and those of the substitute in model. This will allow the identification of the relation (*source element, substitute element*). Thus, for expressing replacements we define a set of rule which we present as follows:

- *OverridePackage [sourcePackage, updatePackage]*

This rule expresses a replacement of elements defined in *sourcePackage* by their correspondents defined in *updatePackage*. Elements defined in *updatePackage* which do not have correspondents in *sourcePackage* will be added in this one.

- *OverrideClass [package1.Class1, package2.Class2]*

This rule expresses that properties of *Class1* replace those which correspond to them in *Class2*. These properties are considered in terms of attributes and operations. Thus, if we want to replace an attribute or an operation of a given class, it is necessary to define a new class which specifies new attributes or new operations. This happens because in UML model, we cannot define an attribute or an operation apart from a class.

Generally, the rules defined in the three categories presented above can be combined. This makes possible to express the combination of two or several models while replacing some elements of the original models by elements of other models. To this end, it will be necessary to first use correspondence rules in order to define the relationship between elements that can be further combined or updated in models. Then, combinations or replacements between should be expressed by using combination or replacement rules.

4. Conclusion and future works
In this paper we have presented a solution to face the evolution of distributed applications in MDA approach. We propose in this solution the reuse of already established PIM and PSM of these applications. This solution is based on two main points: the expression of the reuse of PIM, and the generation of glue which binds their corresponding PSM. This solution is particularly useful for the reuse of existing MDA applications, in terms of composition and extensibility, without changes of their PIM and PSM.

This paper covers the first point of our solution which is the expression of the reuse of PIM. A few approaches found in the literature also propose the reuse of abstract models similar to PIM. However, the means they offer introduce direct changes on the reusable models. This compromises the basic principle of MDA approach which supposes that the PIM of a given application remains stable. Considering these observations, we have proposed a solution based on the expression of PIM reuse. To this end, we have defined three categories of composition rules: *correspondence rules*, *combination rules* and *replacement rules* which allow the expression of different intentions for reusing of PIM, considered in UML.

This article summarizes the first part of our proposal. We are currently working on its extension and improvement by considering the following parts:

- Refinement of the reuse rules we have defined. Different types of reuse in terms of composition, extension and modifications could be specified. For example, we aim at defining composite rules which combine those defined in the various categories (correspondence, combination and replacement). This will help the designer to express composition, extensibility and modifications of PIM.

- Identification of the relation between the expression of the PIM composition and its mapping on PSM i.e. the so-called glue. For this we are considering specific platforms such as CCM [OMG99] or EJB [Sun 99].

- Development of the glue generation tool. This tool consists of two parts: an analysis part which examines the set of input rules to identify the glue to be generated, and a generation part that effectively generates the identified glue. The choice of having two parts allows the generation of glues for different platforms.

References

[AspectJ 03] Eclipse:AspectJ Team, "The AspectJ^{TM} Programming Guide", http://eclipse.org/aspectj/.

[Clarke 01] S. Clarke, "Composition of Object-Oriented Software Design Models", PhD thesis, School of Computer Applications, Dublin City University, January 2001.

[Harrison 93] W. Harrison, H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", Proceedings of OOPSLA'93, ACM Press SIGPLAN, Washington, USA, pp. 411-428, October 1993.

[IBM 03a] IBM Research: Subject-oriented Programming, "Support for subject-oriented programming in C++ on IBM VisualAge for C++ v. 4", http://www-3.ibm.com/software/awdtools/vacpp/version4/.

[IBM 03b] IBM Research: Subject-oriented Programming, Group: "Hyper/J^{TM}: Multi-Dimensional Separation of Concerns for Java^{TM} ", http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm

[Kiczales 97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda C. Lopes, J. Loingtier, and J. Irwin. "Aspect-Oriented Programming". In Proc. of ECOOP, pp. 220–242, 1997.

[OMG 02] OMG, Request for Proposal MOF2.0 Query /Views /Transformations, ad/2002-04-10, www.omg.org, April, 2002.

[OMG 03] OMG, Model Driven Architecture Guide version 1.0, Document Number : omg/2003-05-01. May 2003.

[OMG 97] OMG "Unified Modeling Language Specification v1.1" TC. Document ad/97-11-03. OMG. 1997. http://www.omg.org

[OMG 99] OMG "CORBA Component Model Volume 1". TC Document ad/99-01-01 OMG 1999. http://www.omg.org

[Sun 99] Sun: EJB, "Entreprise JavaBeans", http://java.sun.com

[Van 99] G. Vanwormhoudt, "CROME : un cadre de programmation par objets structurés en contextes", PhD thesis, Laboratoire d'Informatique Fondamentale de Lille I, Lille, 1999

MDA-Driven Development of standard-compliant OSS components: the OSS/J Inventory Case-Study

Nektarios Georgalas[1]*, Manooch Azmoodeh**
BT Group, UK
Tony Clark***, Andy Evans***, Paul Sammut***, James Willans***
Xactium Ltd, UK

*georgan@acm.org

***manooch.azmoodeh@bt.com*

***[firstName.surname]@xactium.com

Abstract: *The telecommunications-oriented Operational Support Systems (OSS) industry have recognised the value of technology independent modelling of OSS solutions as a way to reduce cost, add agility, validate and verify solution designs against architectural guidelines of an enterprise and most importantly provide traceability in the design methodology process. The challenges faced by the OSS community is how MDA tools can deliver the promise of advanced meta-modelling, model definition and validation and model transformation for both OSS software components and integration logic in the larger OSS landscape. This paper describes how an advanced extensible meta-modelling tool is used to build an OSS component following best practice industry guidelines. Extended MOF, extended executable OCL and a powerful transformation language are used to capture the constraints in the meta-models as well as models followed by complete, 100% code generation from models. Furthermore, meta-models are also developed to capture graphical user interface elements in conjunction with the inventory data models, which are then automatically translated into code. This work is the precursor for defining extensive meta-models for a component-based OSS infrastructure based on industry best practice, for adding high degree of formality to model specifications and for enabling the verification of domain requirements by executing the models through model snapshot creation, way before system implementation takes place.*

Keywords: *OSS, OSS/J, NGOSS, TMF, Component, Contract, OMG MDA, TNA,TSA, meta-modelling executable OCL, Inventory System, IP VPN*

1 Introduction

Developing and operating Operational Support Systems (OSS) for telecommunications companies (telcos) is a very expensive process whose const continuously grows year on year. With the introduction of new products and services, telcos are constantly challenged to reduce the overall costs and improve business agility in terms of faster time-to-market for new services and products. It is recognised that the major proportion of overall costs is in integration and maintenance of OSS solutions. Currently, the OSS infrastructure of a typical telco comprises an order of O(1000) systems all with point-to-point interconnections and using diverse platforms and implementation technologies. The telcoms OSS industry has already established the basic principles for building and operating OSS through the TMF NGOSS programme **[NGOSS]** and the OSS through Java initiative **[OSSJ]**. In summary, the NGOSS applies a top-level approach through the specification of an OSS architecture where:

- Technology Neutral and Technology Specific Architectures are separated
- The more dynamic "business process" logic is separated from the more stable "component" logic

[1] Address: Adastral Park, Orion Building – Ground Floor pp13, Martlesham Heath, Ipswich, IP5 3RE, UK

- Components present their services through well defined "contracts"
- "Policies" are used to provide a flexible control of behaviour in an overall NGOSS system
- The infrastructure services such as naming, invocation, directories, transactions, security, persistence, etc are provided as a common deployment and runtime framework for common use by all OSS components and business processes over a service bus.

Complementary to NGOSS is the low-level approach of the OSS/J, which provides a set of standard Java-based interface specifications with a roadmap for producing APIs covering the entire landscape of the OSS space (Trouble Ticketing, QoS monitoring, Inventory, Billing, SLA management, etc.). These specifications provide a technology specific set of OSS functional capabilities and as such can be a basis for building an implementation view of an OSS using J2EE platform.

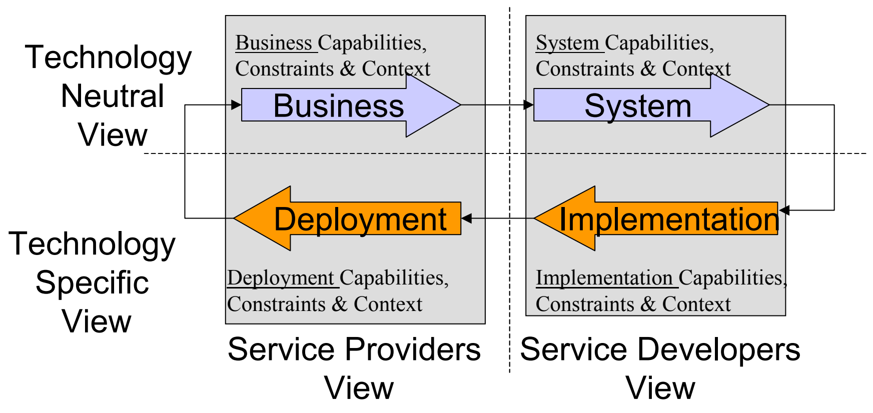

Technology Neutral View
Technology Specific View
Business Capabilities, Constraints & Context
Business
System Capabilities, Constraints & Context
System
Deployment
Implementation
Deployment Capabilities, Constraints & Context
Implementation Capabilities, Constraints & Context
Service Providers View
Service Developers View

Figure 1 - NGOSS lifecycle methodology

In addition to the above, NGOSS has defined a methodology for developing OSS solutions emphasising separation of concerns so that different actors in the overall design process are freed from polluting their models with details and aspects of other areas. Figure 1 shows four different lifecycle stages or views identified by the NGOSS methodology, namely, business, system, implementation and deployment view. The top row shows the logical views of the systems, which are ***technology neutral***. The business view captures business contracts, business processes, entities and interactions (using the eTOM **[eTOM]** and SID **[SID]** standards) without reference to how they are realised using automated computer systems. The System view provides the computational interactions among automated components, processes and policies. The bottom row shows the physical view of system, which are intrinsically ***technology specific***, where on the right, the implementation view contains the hardware and software to construct the system and on the left the deployment view captures the instance level operating systems and active monitoring of the system.

One of the main goals of the NGOSS lifecycle is to provide traceability from business requirements to systems descriptions to implementation details and finally to deployed systems, thus traversing from the top-level, technology neutral NGOSS specifications to the low-level, Java-based APIs and J2EE architectural principles of

OSS/J. The MDA technology **[MDA]** is identified as the key enabler for providing such automated traceability in an NGOSS environment. The main goal of our research is to use MDA standards and tools to define meta-models and transformation rules around the lifecycle so that various models and views in the lifecycle can be verified for correctness and completeness as well as auto-generating these models and systems. First research results presented in **[Georgalas 2004]** focused upon the applicability of an MDA enabled OSS architecture in a telcoms environment and a generic technology independent NGOSS component specification.

In this paper, we describe how an advanced meta-modelling toolkit, the XMF from XACTIUM **[XACTIUM]** in particular, can be practically used to specify and automatically generate a complete system implementation of a single OSS component. As an example, the component at issue is based on the OSS/J Inventory API specification and can manage models as well as instances of products, services and resources within a telco environment. The paper, also, shows the use of a constraint language, which is a version of OCL extended with imperative constructs making it a powerful formalism for representing complete behavioural specifications at the modelling level. Furthermore, meta-model transformations that automatically derive the component implementation are demonstrated and specified in XMap, a transformation rule language embedded in XMF with a powerful pattern matching capability.

The remainder of the paper is structured as follows. Section 2 will introduce the rationale for choosing the OSS/J Inventory API, present an inventory domain-specific language meta-model with an example inventory PIM that instantiates the meta-model and describe a mapping of the inventory language onto a Java and tool meta-model in order to drive the automatic generation of an inventory application tool. Section 3 will discuss a number of lessons learned from this study. Section 4 finally will complete the paper with concluding remarks on the presented work and a description of further research plans.

2 A Case-Study: the OSS/J Inventory

The merits of MDA have been formally recognised by the TMF as it is signified, on one hand, by the recently announced strategic alliance between TMF and OMG and, on the other, by the similarities encountered between MDA and the NGOSS methodology **[Strassner 2004]**. This leaves little doubt that MDA will play a significant role in the telcoms industry and in particular in the development process of telcoms OSS. Nevertheless, however much appraised at a strategic level, there is no evidence for the use of MDA in practice to develop OSS solutions[2]. With intent on investigating the full power of MDA in the context of OSS solution design, we embarked upon a small scale case-study aiming to generate a fully functional OSS component implementation driven solely by technology neutral model specifications.

The case-study was based upon OSS component APIs specified in Java and J2EE by OSS/J. OSS/J have issued a document with standard J2EE architectural patterns and design guidelines all OSS component specifications **[Gauthier 2001]** should comply

[2] The authors have not found any case-studies reported in the relevant literature. Furthermore, even within the TMF's Catalyst programme, where TMF member companies collaborate to build demonstrators that apply the TMF standards in practice, no project has been setup as of yet with clear focus on the use of MDA to develop OSS solutions.

with. In order to test conformance to the defined specifications and guidelines in practice, OSS/J additionally produce reference component implementations and technology compatibility kits. The case-study was specifically driven by the OSS/J Inventory component API **[Gauthier 2004]** and set as its goal to prove the ability to automatically conduct such compliance tests by means of MDA. This end acquired more value by the fact that this particular API specification lacks, as of yet, a reference implementation and compatibility kit that would permit its practical validation. Given short project budget and timescale limitations, the study made careful assumptions, where necessary, in order to simplify the API's complexity without compromising its results.

The exercise targeted to a twofold outcome, as shown in Figure 2:

- **Automatic generation of PSMs conformant to the eTOM SID standard**: The OSS/J Inventory specification document includes a UML class diagram of an inventory meta-model and some textual, i.e. informal, description of its semantics. The meta-model defines the types of information/content the inventory will manage, such as products, services and resources. These types stem from a bigger model, namely, the Core Business Entities **[Reilley 2004]**, that OSS/J have defined in line with entities and interfaces encountered in eTOM SID for use by the OSS/J component APIs. In the case-study, we will be capturing the meta-model and some of its semantics in an MDA environment and instantiate it with example PIMs. Based on transformation rules, technology-specific representations for entities of an inventory PIM will be automatically generated. These representations collectively form a PSM. The PSM entities in this case will actually be Java classes or EJBs (entity beans) representing, one for one, the inventory PIM entities. While the case-study generated technology specific outputs in Java and EJB, the paper will focus only on the Java ones.

- **Automatic generation of a system implementation conforming to standard OSS/J architectural patterns and design guidelines:** While the PSM entities, i.e. Java classes or entity beans, bear the structure and deliver the behaviour of inventory entities as described in the original inventory PIMs, end-users should not interact directly with these entities. Rather, entities should be accessed through a single interface that exposes a simple set of management methods and hides their complexity. This is a standard OSS/J design guideline, which conforms to the *façade* design pattern and influences the architectural design of OSS/J components. In order to comply with the OSS/J guideline, the case-study aims at implementing an application tool that allows users to manage the inventory content through a simple GUI. Example users of such a tool may be front-desk operators who respond to customer calls and access the inventory to setup a new or change the state of an existing product/service instance. The case-study uses MDA to automatically generate the tool and associated GUI in Java and J2EE (session bean) in order to deliver the required OSS/J pattern and design guideline. Again, this paper only concentrates on the Java outputs.

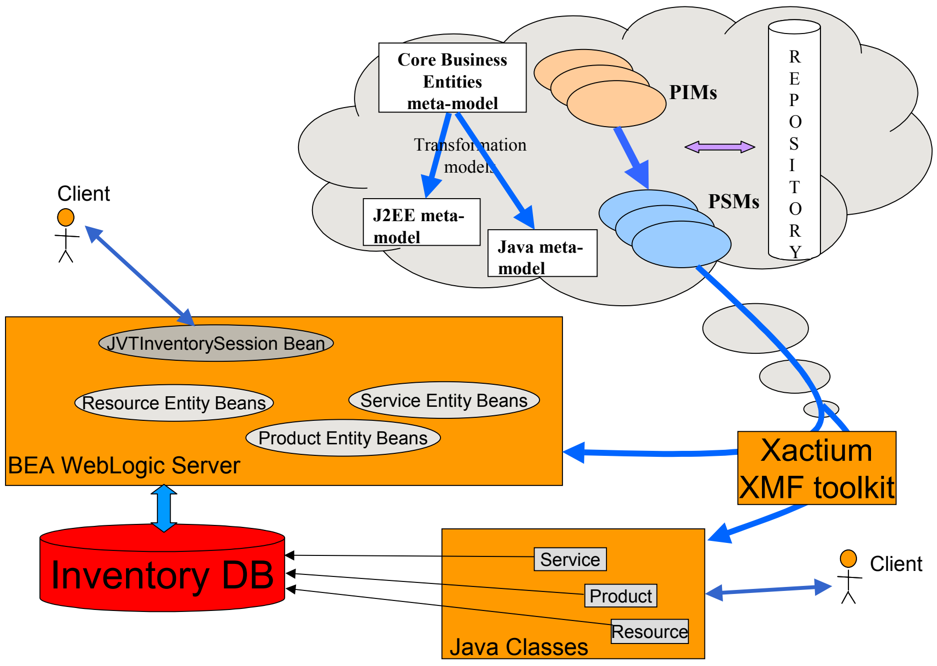


Figure 2 – The OSS/J Inventory case-study

Before embarking on the study, a brief evaluation of available MDA tools was carried out, such as iUML, Arcstyler, OptimalJ, Objecteering as well as XMF. It was found that for this particular application area, XMF offered advanced meta-modelling capabilities for expressing semantic aspects of models and definition of tools for any arbitrary language expressed using the MOF standard.

The XMF toolkit **[Clark 2004]** is a generic meta-programming environment that aims to support a wide variety of MDA development scenarios. To achieve this, XMF provides a variety of rich meta-modelling languages including: a package of OO meta-modelling concepts called EMOF, an executable version of OCL called XOCL, and a mapping language called XMap. Each of these languages has a well-defined executable semantics that is run on the XMF virtual machine.

Figure 3 shows how XMF was used to support the OSS/J Inventory MDA scenario. Firstly, a platform-independent domain specific language for inventories was defined by extending the EMOF meta-model. XOCL was used to specify meta-model constraints so that models written in the inventory language can be checked for correctness. That is, by means of XOCL, the meta-model semantics can be formally captured and automatically enforced, in contrast to the informal, textual description of the semantics presented in the OSS/J Inventory API specification document. Next, mapping rules written in XMap were constructed to transform the inventory meta-model into meta-models of two target platform specific languages: EJB and Java. This enables any model written in the inventory language to be translated into models that corresponded to programs written in EJB and Java. The former generates plain EJB code, which can then be manually deployed onto the BEA Weblogic application server. The latter is more sophisticated in that it generates a fully deployed Java tool for instantiating the generated models and for checking constraints and running operations on the models. The aim is to show that the source language was rich enough to be translated into sophisticated domain specific applications.

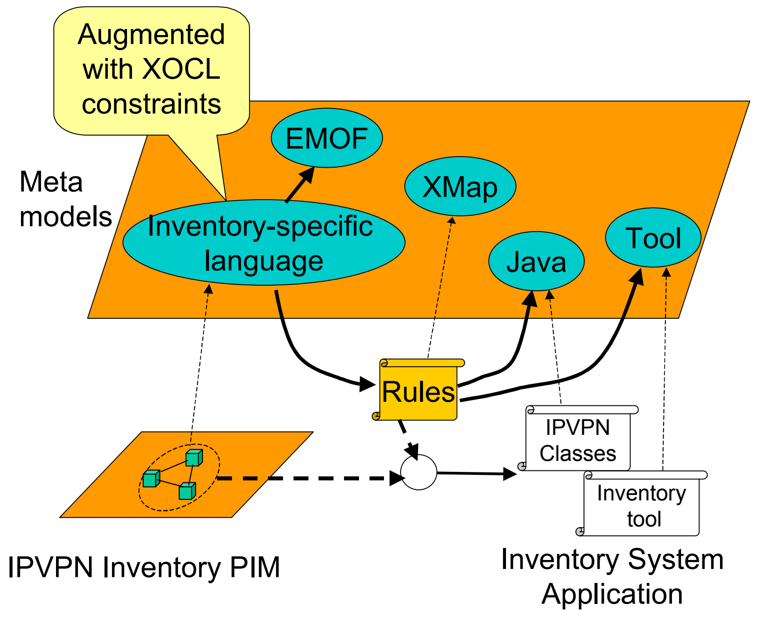

Augmented with XOCL constraints
EMOF
XMap
Meta models
Inventory-specific language
Java
Tool
Rules
IPVPN Classes
Inventory tool
IPVPN Inventory PIM
Inventory System Application

Figure 3 – Using XMF to deliver the Inventory tool

2.1 Domain-specific language

Figure 4 shows the inventory domain specific language meta-model. As mentioned earlier, it includes concepts from the OSS/J Core Business Entities, which are a subset of TMF's SID. The inventory language consists of the following constructs:

- *Entity*, that represents any type of information included in the inventory. According to the specification, three types of inventory content are defined, namely, *Product*, *Service* and *Resource*, which extend type *Entity.*
- *EntitySpecification*, that represents configurations of *Entities*, i.e. constraints, such as range of values or preconfigured setting on features of the *Entity.* Again, the API specification defines three subtypes of *EntitySpecification*, namely, *ProductSpecification, ServiceSpecification* and *ResourceSpecification*, each representing specifications for *Service*, *Product* and *Resource, respectively.*
- *EntityAttribute*, that represents relationships between *Entity* types.

To represent this inventory domain specific language, a meta-model is constructed with classes that specialise classes of the XMF embedded EMOF package, each of which EMOF classes has a well-defined executable semantics. More specifically:

- *Entity* specialises the class *EMOF::Class*, hence it can be instantiated and contain attributes, operations and constraints.
- *EntitySpecification* inherits from *EMOF::Constraint*. It can, therefore, be owned by an *Entity* and contain an evaluate-able XOCL expression. In the Inventory API specification document, *EntitySpecification* is represented as a UML class, which has a simple semantics, and thereby great modelling incapacity to express in full potential the concept semantics as an *Entity* configuration constraint. Therefore, by modelling *EntitySpecification* as a pure constraint, rich expressive power is conveyed to the concept enabling it to represent complex *Entity* configurations.

- *EntityAttribute* specialises the class *EMOF::Attribute* and is used to associate different *Entity* types.

A number of constraints (well-formedness rules) apply to the inventory language. These are expressed in OCL. As an example, the following OCL constraint states that if an *Entity* specialises another *Entity* it must be of the same type as the parent entity. That is, entity *IPStream_S* of Figure 5, for instance, can inherit from *IPStream*, as both are of type *Service*, but cannot inherit from *IPVPN* that is of type *Product.* Here, *of*() is an XOCL operation that returns the meta-class of the entity (i.e. the class that the entity is an instance of).

```
context Entity
  @Constraint SameParentType
    parents->select(p | p.isKindOf(Entities::Entity))->forAll(p |
        p.of() = self.of())
  end
```

Another noteworthy constraint formally delivering an important semantic property of the inventory meta-model, as per the OSS/J Inventory API specification document, involves the association of an *Entity* type with the correct type of *EntitySpecification.* In other words, classes of type *Service*, for instance, can only have specifications of type *ServiceSpecification* and not of type *ProductSpecification* or *ResourceSpecification*. Checking this and other similar constraints on a model that instantiates the inventory language meta-model can quickly and automatically validate the model for semantic correctness. The XOCL for the constraint follows.

```
context Entity
  @Constraint CorrectSpecs
    self.constraints->forAll(c |
     let ctype = c.of()
     in @Case ctype of
        [ IML::Entities::ServiceSpec ] do
          self.isKindOf(IML::Entities::Service)
        end
        [ IML::Entities::ProductSpec ] do
          self.isKindOf(IML::Entities::Product)
        end
        [ IML::Entities::ResourceSpec ] do
          self.isKindOf(IML::Entities::Resource)
        end
      end
   end)
```

Once the inventory language has been defined it is possible to create models that instantiate the language meta-model. An important question at this point is how this model can be visualised. One approach supported by XMF is to create a model of its diagrammatical syntax, which is then used to create a language specific diagram editor for the language. This has the advantage of being able to support very rich diagram types, but requires further modelling work.

A much simpler approach is to make use of a mechanism known as a *metaPackage*. Meta-packages allow a package to be represented as an instance of another package (its meta-package). Because XMF understands that the *metaPackage* represents a package of language definitions, it can provide appropriate stereotypes in the model package. Note that *metaPackages* represent a stronger variant of profiles **[UML 2003]** because the stereotyped elements are real instances of meta-model elements (as opposed to being virtual instances). This way, NGOSS architectural guidelines, patterns and standards can be captured in a rigorous manner so that designers are capable of continuously validating their models against NGOSS artefacts.

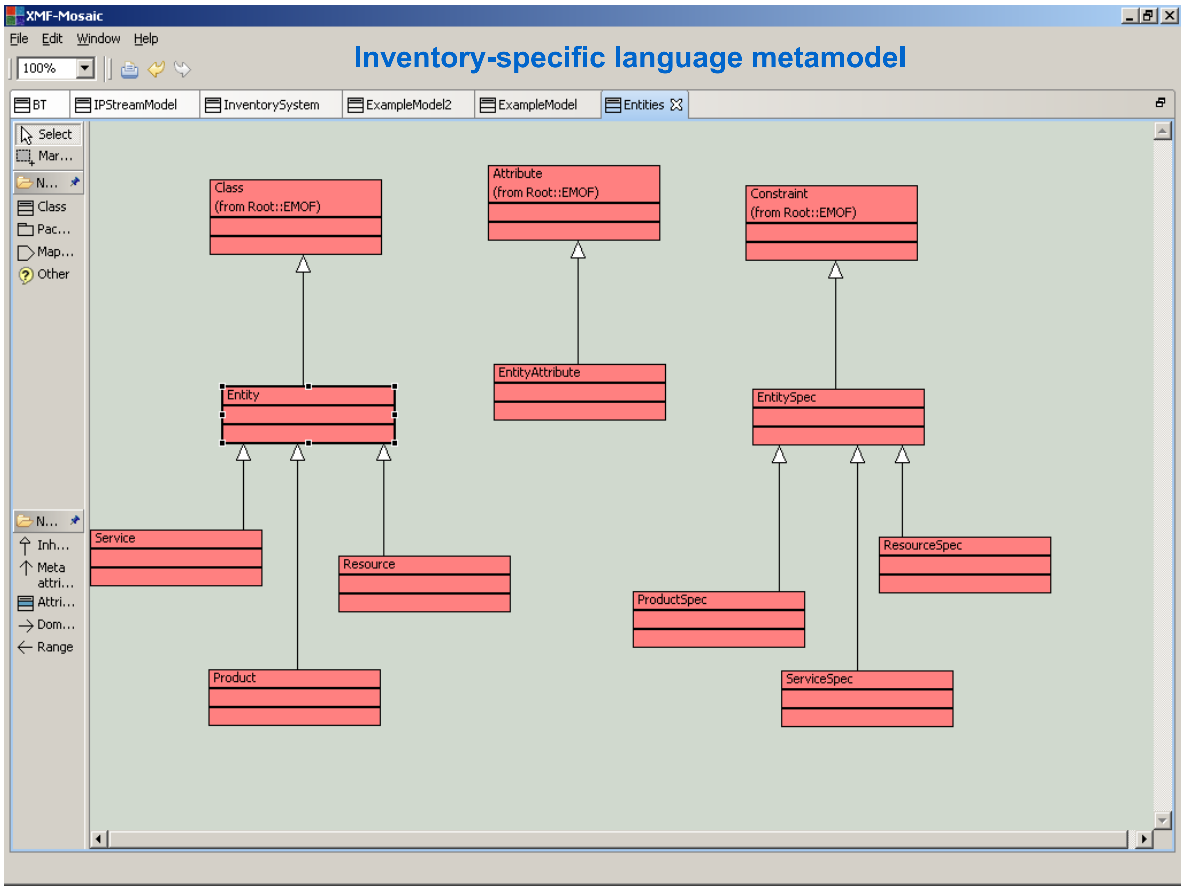


Figure 4 – Inventory-specific language

In Figure 5 a model is presented that is an instance of the inventory meta-model (its meta-package). It is based on an IP Virtual Private Network (IPVPN) product provided by BT and, in favour of simplicity, it only illustrates a subset of entities comprising the product[3]. The example IPVPN product, *inter alia*, would require a broadband link service between the connected customer ends. Hence, the model in Figure 5 shows *IPVPN* containing (*containedServices* attribute) many *IPStream* entities, a BT ADSL service that comes in different offerings for home and for office premises represented by *IPStream_S* and *IPStream_Office*, respectively. *IPStream_S* is further subclassed by *IPStream_S500*, *IPStream_S1000* and *IPStream_S2000*, entities differentiating on the downstream bandwidth of the link that is, respectively, 500, 1000 and 2000 kbps. Individual features of the latter entities are defined in the accompanying *ServiceSpec* constraints, namely, *S500Spec*, *S1000Spec* and *S2000Spec*. Similarly, features of the *IPVPN* product and the *IPStream_S* service are specified in the *IPVPNSpec* and *IPStream_SSpec* specification constraints.

[3] In reality, the IPVPN product at issue could come in different versions packaged with additional features to the broadband link, such as, equipment for the customer premises and/or frontdesk support.

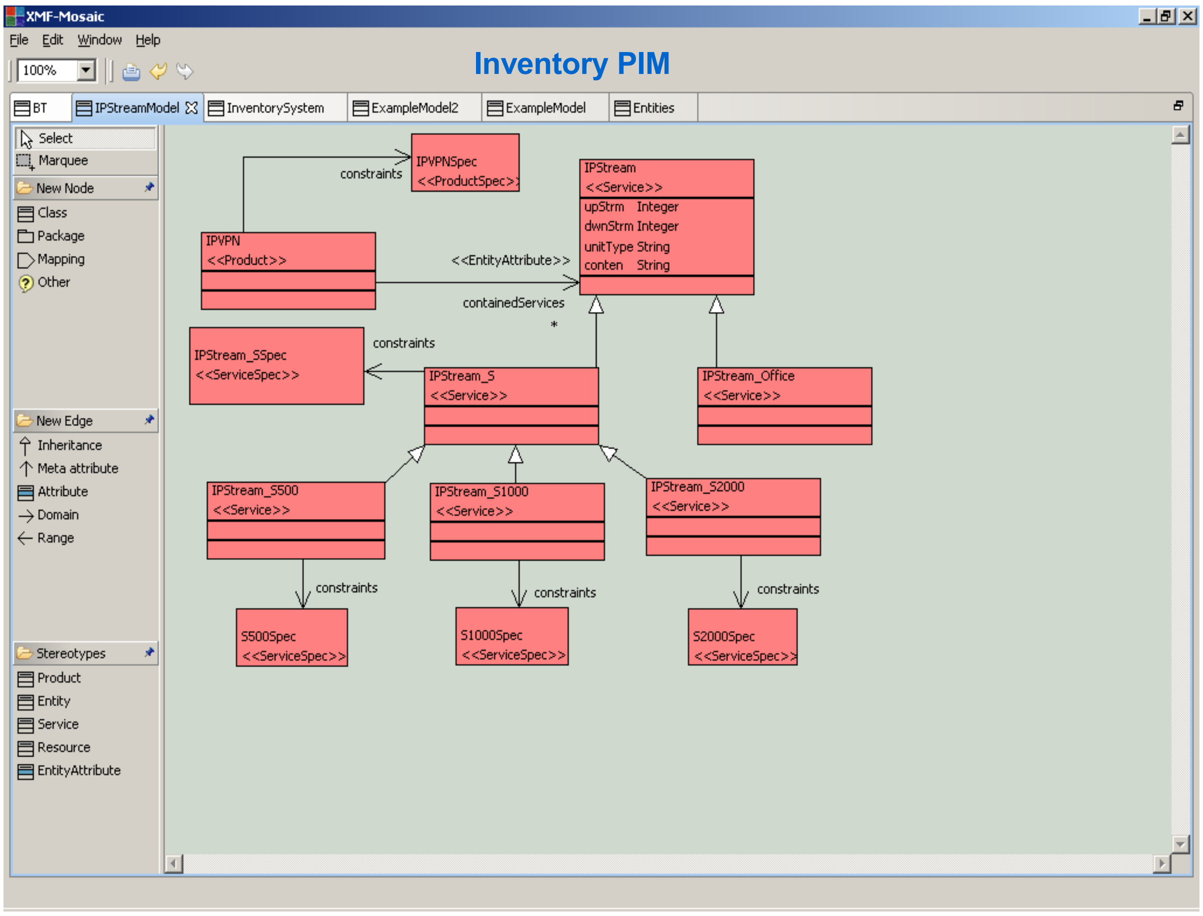


Figure 5 – Inventory PIM

All above model entities have as their types meta-classes defined in the inventory language meta-model of Figure 4. Hence, all entities in the model diagram of Figure 5 are shown as stereotyped classes constituting instances of the inventory domain specific meta-classes, for example, *IPStream_S2000* is an instance of meta-class *Service*. This way a PIM has been designed for the inventory using modelling constructs and semantics customary to the specific domain of interest.

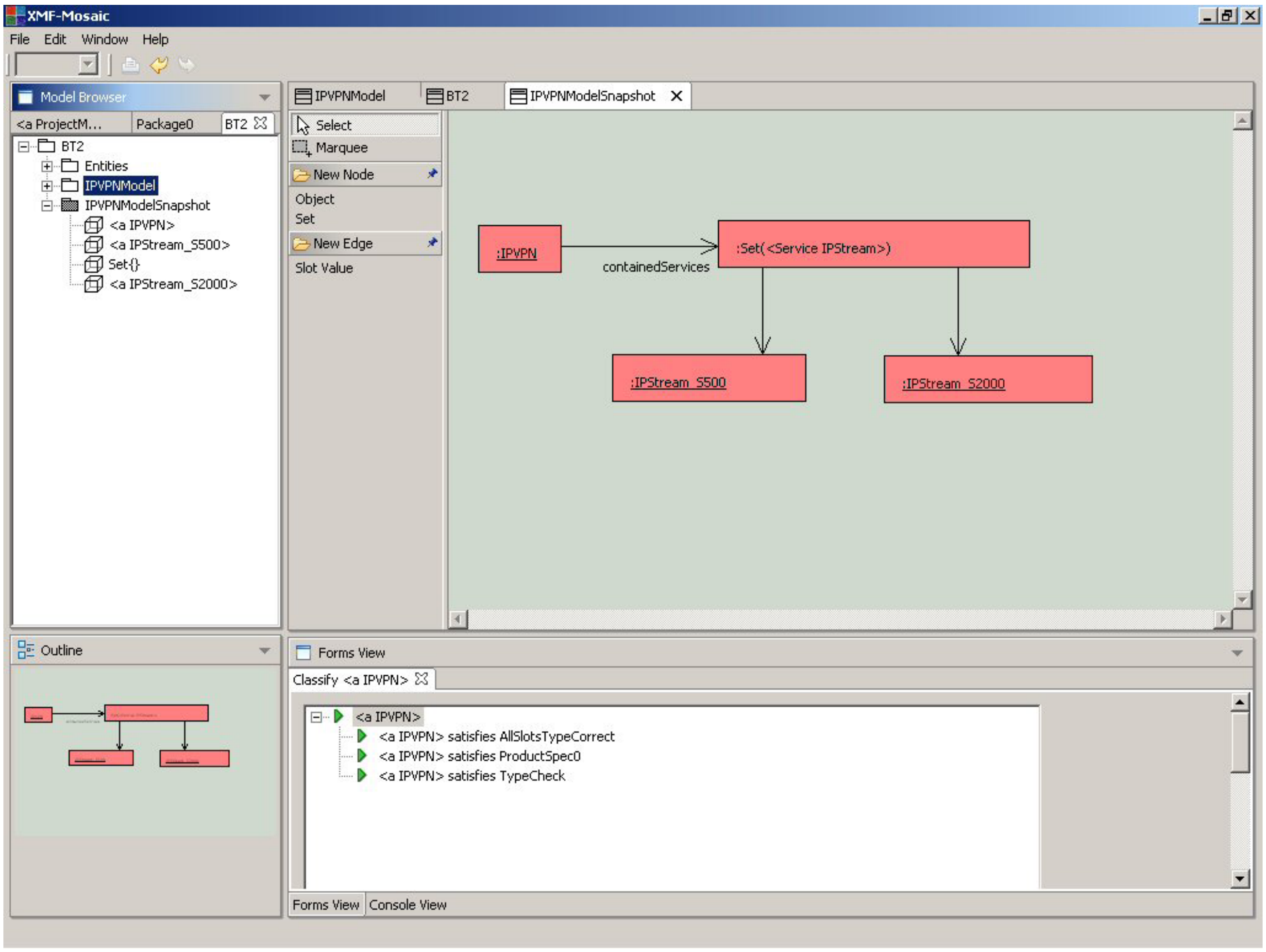
Because all model entities of Figure 5 are instances of inventory meta-classes that specialise *Entity*, which, in turn, extends class *EMOF::Class*, they inherit the ability to have constraints, attributes and operations (and their associated specialisations, namely, *Specifications* and *EntityAttribute*). As an example, the *IPStream_S2000* is associated with *S2000Spec*, which has the following OCL body:

```
self.upStream = 250 and self.downStream = 2000 and self.unitType = "kbps"
```

In addition, XOCL can be used to write operations on the PIM model. XOCL extends OCL with a small number of action primitives, thus turning it into a programming language at the modelling level. As an example, the following operation creates an instance of an *IPStream* and adds it as a *containedServices* attribute to an *IPVPN*:

```
context IPVPN
  @Operation addIPStream(up,dwn,unit,con)
    self.containedServices :=
      self.containedService->including(IPStream(up,dwn,unit,con))
  end
```

Finally, because the entities in the model are themselves instantiable, it is possible to create an instance of the *IPStreamModel* and check that the instance satisfies the constraints that are defined in the PIM model. This is a further level of instantiation that is possible because of the *metaPackage* relationship between the inventory PIM model and the inventory language meta-model. Such a "snapshot" mechanism allows the validity of the model to be established early in the development process without the need to generate a prototype. In many respects it is more powerful than prototyping because it allows the construction and checking of counter-scenarios, that is behaviour that the system should not exhibit at runtime. This gives the designer confidence that the system eventually generated will function in the required manner. An example snapshot is shown in Figure 6.



```
<a IPVPN>
    <a IPVPN> satisfies AllSlotsTypeCorrect
    <a IPVPN> satisfies ProductSpec0
    <a IPVPN> satisfies TypeCheck
```

Figure 6 - A snapshot of the IPVPN model

2.2 Transformations of PIMs to PSMs

Using XMap, two mappings were defined from the inventory language. The first was to generate EJBs, whilst the second focused on the generation of Java and a Java class tool. We concentrate on the second one here.

The model of Figure 7 shows the mappings that were used to generate Java. Rather than mapping directly from the inventory language meta-model, a more generic approach was taken in which the mapping was defined from EMOF classes. Because the inventory language extends the EMOF meta-model, they therefore also apply to inventory models (and any other language specialisations defined in the future).

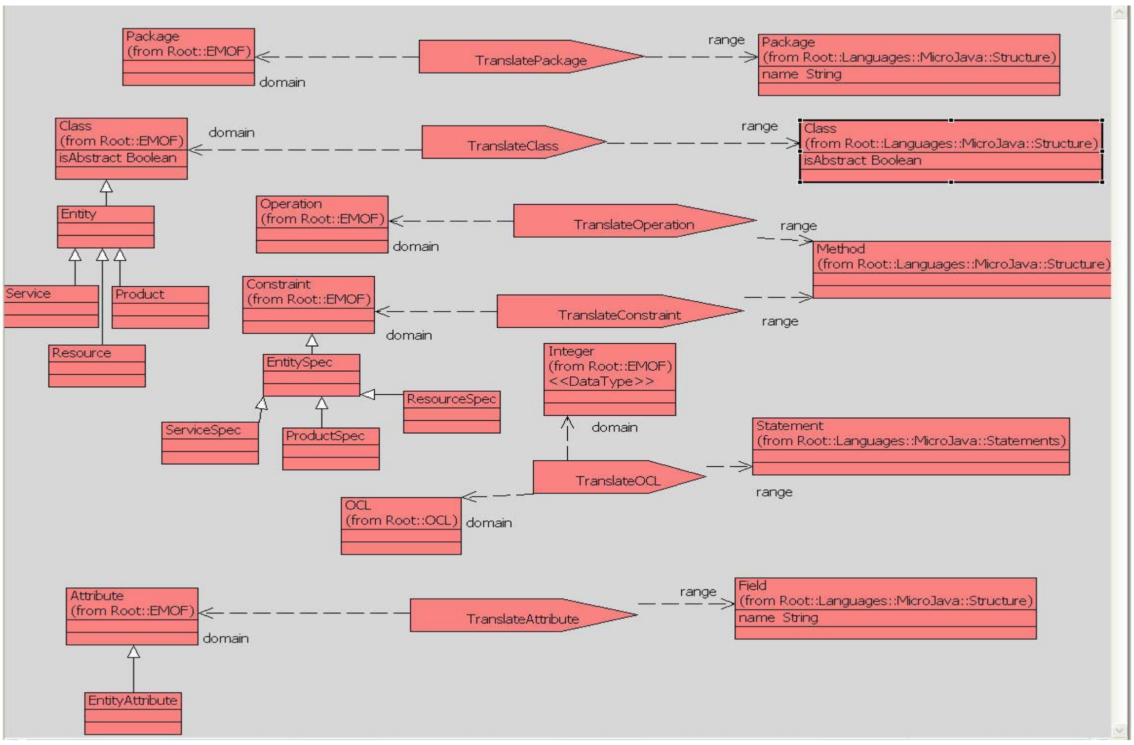

Package (from Root::EMOF)
domain
TranslatePackage
range
Package (from Root::Languages::MicroJava::Structure) name String
Class (from Root::EMOF) isAbstract Boolean
domain
TranslateClass
range
Class (from Root::Languages::MicroJava::Structure) isAbstract Boolean
Entity
Operation (from Root::EMOF)
TranslateOperation
range
domain
Method (from Root::Languages::MicroJava::Structure)
Service
Product
Constraint (from Root::EMOF)
TranslateConstraint
domain
range
Resource
EntitySpec
Integer (from Root::EMOF) <<DataType>>
ResourceSpec
ServiceSpec
ProductSpec
domain
Statement (from Root::Languages::MicroJava::Statements)
TranslateOCL
range
OCL (from Root::OCL)
domain
Attribute (from Root::EMOF)
TranslateAttribute
range
Field (from Root::Languages::MicroJava::Structure) name String
domain
EntityAttribute

Figure 7 – Mapping of Inventory language to Java

Every element in the EMOF package has a mapping to a corresponding element in the Java meta-model. In XMap, mappings are represented by an arrow from source objects (the domain) to target objects (the range), and contain pattern matches between their values. An example of simple pattern match is described by the following XMap code:

```
context TranslateClass
    @Clause Class2Class
       EMOF::Class[name = N, attributes = A]
       do
       MicroJava::Structure::Class[name = N, features = F]
       where
         F = A->collect(a | TranslateAttribute()(a))
    end
```

Here, a Class is mapped to a Java Class, where the name of the Java Class matches the name of the Class and the attributes of the Class are mapped to fields belonging to the Java Class. Because the bodies of EMOF operations are also mapped, the mapping results in generating an executable Java program that precisely implements the behaviour of the PIM. This Java code constitutes the PSM representation of the entities in the inventory PIM.

2.3 Tool Generation

Whilst the above mapping generates a standalone Java program corresponding to an inventory model, it would more useful to users of the language if the model it represents could be interacted with via a user interface. To achieve this, a mapping was constructed from EMOF to a meta-model of a class tool interface for managing object models. The meta-model of the class tool interface is shown in Figure 8. A class tool provides an interface that supports a standard collection of operations on objects, such as saving and loading objects and checking constraints on objects. In addition, a class tool defines a number of managers on classes, which enable instances of classes to be created and then checked against their class's constraints or their operations run.

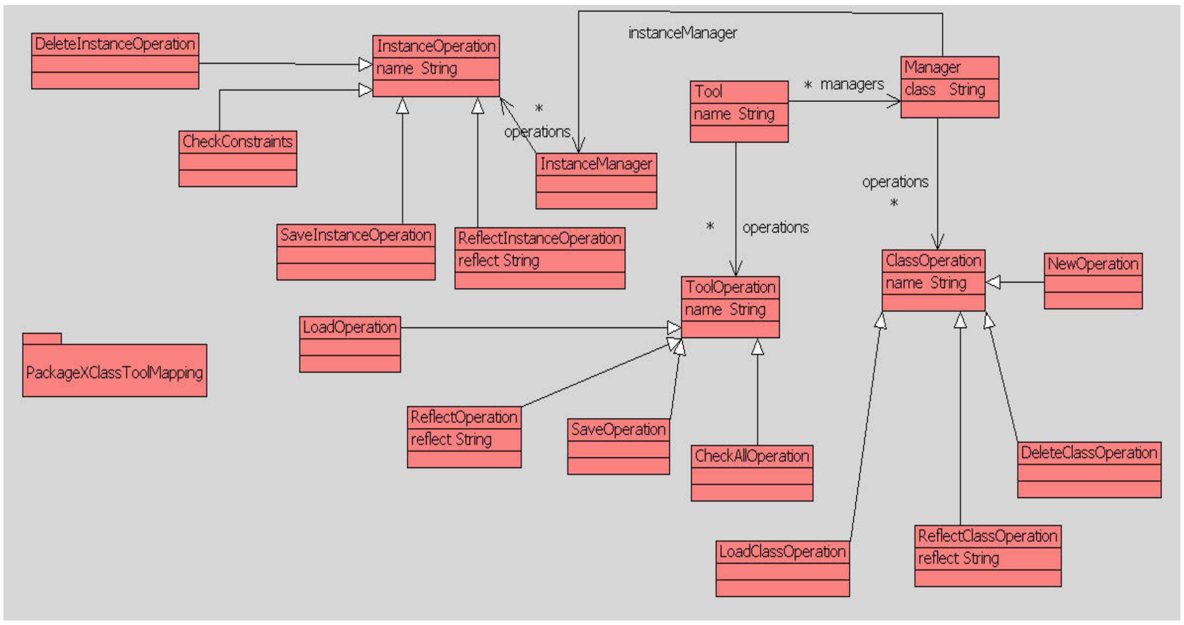


Figure 8 – Tool meta-model

For any EMOF model, a mapping can be defined to the class tool meta-model, which generates a tailored user interface for creating and manipulating instances of a meta-modelling language such as the inventory language. An overview of the mapping is shown in Figure 9. For each class in the source model, a user interface element is created which provides access to operations to create new instances of the class and to manage the operations and constraints provided by the class.

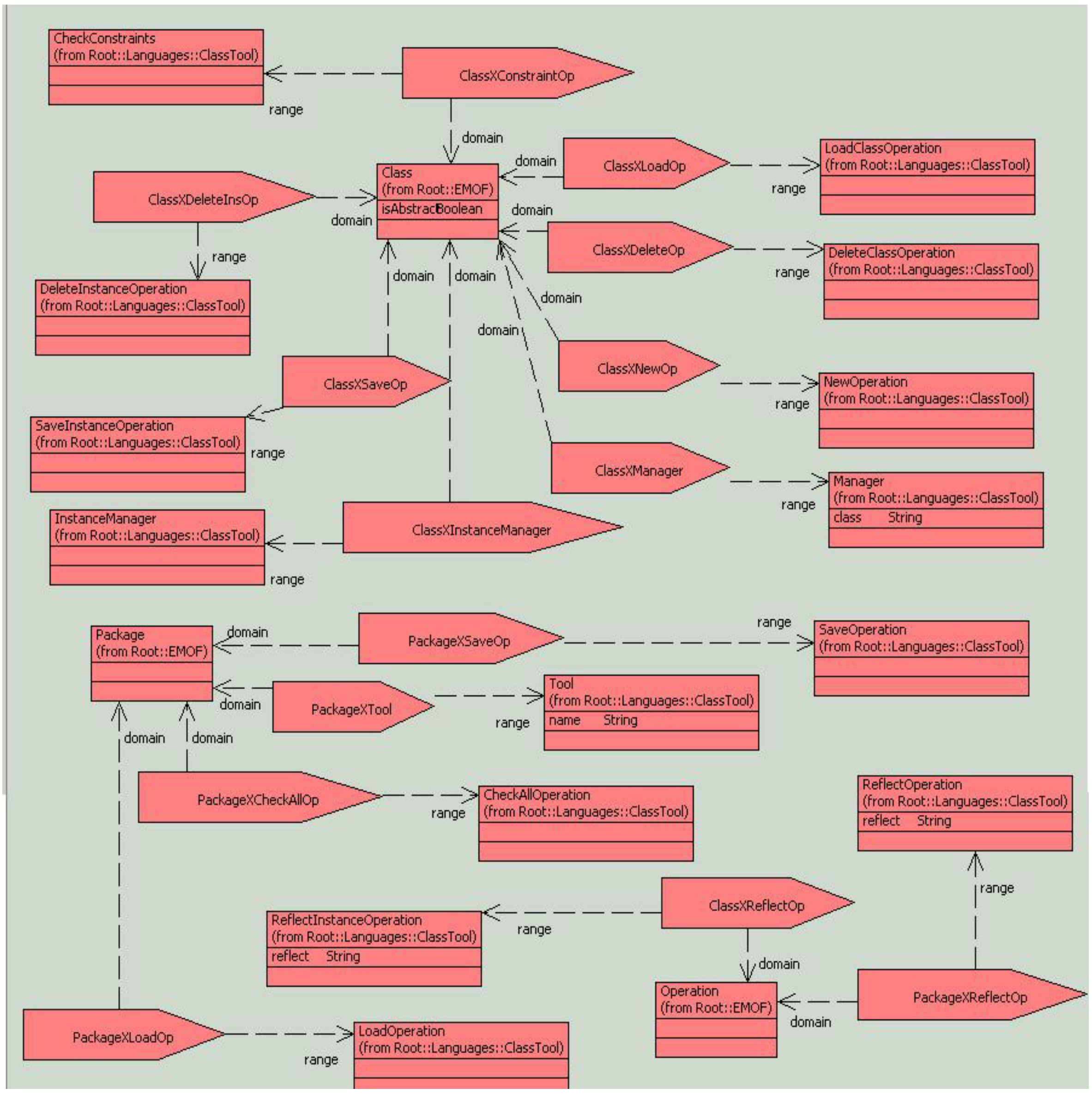
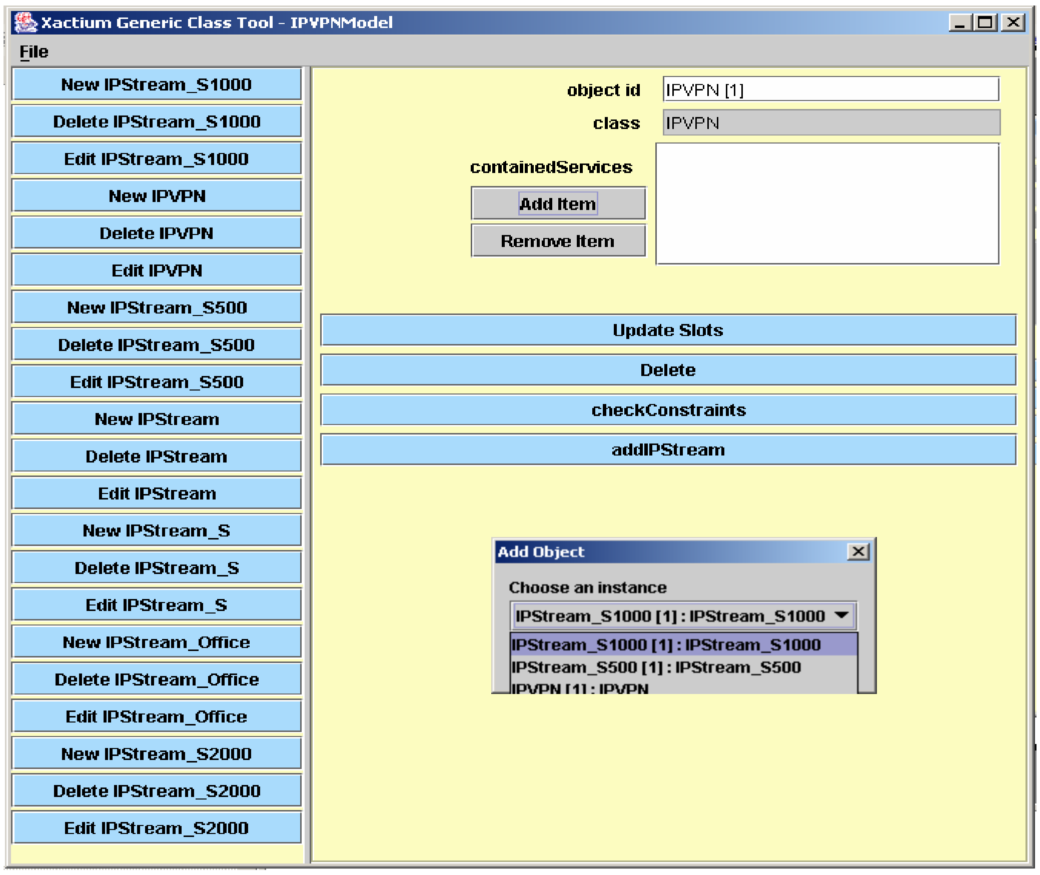


Figure 9 – Mapping of meta-modelling language to class tool meta-model

Applying this mapping to the IPVPN model shown in Figure 5 results in the generation of the class tool in Figure 10. Here, buttons have been generated for each of the entities in the model. These allow the user to create new instances, edit their slot values and delete instances. As the figure shows, a button for invoking the *addIPStream*() method defined earlier has also been added in the GUI executing functionality that implements in Java the method's behaviour described in the model with XOCL.



object id: IPVPN [1]

class: IPVPN

IPStream_S1000 [1] : IPStream_S1000

IPStream_S1000 [1] : IPStream_S1000
IPStream_S500 [1] : IPStream_S500
IPVPN [1] : IPVPN

Figure 10 – Inventory tool

3. Lessons learned

A number of very interesting lessons were learned during the conduct of the case-study:

- **Models can be validated against precise meta-models.** The use of MOF and its well-defined, rich semantics for the definition of language meta-models allows for the construction of precise, non error-prone design models. All these models will be checked for validity against rules and constraints captured in the meta-models leaving no room for mistakes and ambiguity. The case-study demonstrated this through the example inventory PIM of the IPVPN product in Figure 5, which when checked against the semantics of the inventory language meta-model it sussessfully passed the validity tests as both meta-model constraints, namely *SameParentType* and *CorrectSpecs* are satisfied. What is more interesting is that tools, like XMF, are already available to provide the necessary automation in support of this process. This becomes more important in a large industrial environment, where solution desigers and developers constantly exchange models involving implementation and integration of complex OSS solutions and a correct understanding of the designs in one go can save in costs (no bouncing for explanations), produce results faster and minimise the possibility of error.

- **Models can be executed.** Models include full specification of structure and behaviour. Given an interpreting environment, a designer can execute the models and test them against different scenarios. The snapshot mechanism and the interpreting environment of XMF facilitates just-in-time instantiation of models and running/simulating “what-if” situations. This eliminates the requirement of implementing a system prototype first before one can test the durability and robustness of a model. Executing the models is actually a form of rapid prototyping that takes place in the modelling space and very early in the development lifecycle. This capability is very useful in the context of OSS as with a little more effort spend in modelling early on, solutions and ideas can enter a fast-track testing stage before even a single line of code is put together.

- **Automatic generation of platform-specific implementations out of PIMs.** With a bit of more effort invested in the modelling phase, most part of a system’s implementation can be automatically generated. This is ideal for the development of tactical solutions since systems are rapidly produced out of PIMs. Moreover, systems can survive through paradigm shifts and technology changes because PIMs remain intact. This is expected to have great effect in the gradual migration of legacy OSS onto new platforms. Additionally, existing systems can evolve as requirements change since every new feature or change introduced in the technology neutral model can be automatically reflected in the implemented system after re-generating the code. In other words, the use of MDA achieves synchronicity between models and system implementations as it is demonstrated, for instance, by reflective changes in the inventory tool GUI as soon as a new operation, e.g. *addIPStream()*, was added in the PIM.

- **Domain-specific languages can be standardised**. With the rigorous definition of appropriate meta-models one can unambiguously specify architectural styles, design patterns and guidelines. This is especially important in the environment of a large enterprise, which needs to apply company-wide and standardise across the business a particular set of system development principles or requires to precisely define a catalogue of reusable system capabilities, without room for interpretation. With special regards to the current NGOSS meta-modelling we could go far beyond its informal description in UML diagrams and specification documents, by capturing its full semantics using XOCL and by completely generating platform specifc models using XMap. This allows the architectural guidelines expressed in the NGOSS Technology Neutral Architecture be specified and enforced by automated tools, like XMF.

- **Specifications and standards can be verified for correctness very early in the lifecycle**. Ambiguities are removed. For instance, in the absence of a reference implementation and compatibility kit for the OSS/J Inventory, using MDA we could achieve fast validation of the specification both by executing the models to check model and meta-model constraint satisfaction and by generating an executable system in a technology of choice that would completely conform to the semantics captured in meta-model and PIM.

- **Standards and specifications can obtain full tool support from the very start of their textual definition.** This is due to the use of rigorous meta-modelling techniques with fully-defined and executable semantics based on OMG standards, such as MOF and OCL. Tools, such as XMF, which are based on these OMG standards, can be easily extended and customised to support new standards and specifications, such as NGOSS and OSS/J. The study, in particular, demonstrated

XMF's support of an important architectural OSS/J design guideline, the *façade* pattern, through the rigorous definition of the inventory language, the generic class tool meta-model and the eventual automatic generation of a front-end system that provides access to and management of inventory entity instances. Furthermore, the complete generation of executable systems out of meta-models and PIMs can rapidly provide prototype technology-specific reference implementations for practical tests of standards on criteria such as performance and scalability.

- **The richer the definition of the platform independent language (including semantics) the richer the mapping can be to platform specific modelling languages**. In particular, it is possible to generate 100% of the code necessary to support the execution of the translated model. In our case-study the inventory domain specific language is fully executable, hence 100% code generation was achievable.

- **MDA has the power to integrate many different types of languages and technologies**. The case-study, for instance, showed clearly the integration of a domain specific language (inventory meta-model) with a platform specific language (Java meta-model) and a user interface tool (class tool meta-model).

- **MDA tools are currently maturing towards constituting a viable and robust solutions used to capture all the complete structural and behavioural aspects of a system in a model**. Despite the small scale of the presented case-study, there was clear evidence that MDA supporting tools are viable, robust and worth be tested to a more extreme extent of an industrial scale.

4 Conclusions and future work

In this paper, we described how XMF, an advanced meta-modelling tool, can be used to develop, verify and generate models, code and GUI interface of an inventory system based on OSS/J standards. We demonstrated that a ***complete*** system description covering structural and behavioural aspects of the system can be captured in an executable model using extended meta-modelling and constraint languages, which are based on OMG standards.

We have demonstrated the power of the XMF meta-modelling tool producing an OSS component completely based on well-defined precise and accurate models. This has raised our confidence in the maturity of the MDA technology in a rich and complex OSS environment. Of course, OSS is more than merely a single component. Often it is made of diverse set of components based on varying platforms interconnected through complex integration hubs, business process, workflow and policy engines. Thus, the ongoing work aims at using MDA tools to provide fully automated support in all stages of an OSS methodology lifecycle.

Initially we intend to extend the model definitions to cover a few OSS/J components (such as trouble ticketing, QoS monitoring and service activation) and aim to capture integration logic of these components in an MDA tool. The integration logic will be captured at business and system view models of the lifecycle together with mappings to implementation and deployment views. An important aspect of the overall methodology is to encourage more reuse of the modelling artefacts in the entire lifecycle and hence means are required to store model elements in meta-data repositories and enable designers and architects to discover suitable model fragments for use in their designs. This necessitates a method of expressing requirements for OSS components and business processes in a precise language before searching meta-data repositories.

In an OSS environment, code generation is not necessarily a prime driver for adopting MDA. This is due to the fact that the OSS industry is moving towards a plug and play architecture based on available COTS components as a way to reduce the cost of in-house development and reducing vendor lock-in risks. Hence, the greater emphasis is on development of rigorous architectural guidelines and frameworks capturing an enterprise's computing policies together with automated tool support so that the designs of OSS solutions (integration and OSS components, business process definitions and policies governing the behaviour of components and processes) can be validated and verified. This work thus is the first step towards building meta-models that capture the NGOSS lifecycle views, including various forms of components, contracts, process definitions, policies and their inter-relationships.

In conjunction with COTS components, it is recognised that operators tend to customise as much as 80% of COTS software resulting in high costs and use of proprietary tools. As part of the COTS modelling exercise, we intend to capture the customisation of COTS components in high-level models, where any modifications can be done at the model level and then using suitable transformations to apply the necessary changes on the COTS-specific development environment.

References

[Ashford 2004] Ashford C., "OSS through Java as an Implementation of NGOSS", White Paper, April 2004, http://www.ossj.org/learning/docs/wp_technologycomparison1.0.pdf

[Clark 2004] Clark T., Evans A., Sammut P., Willans J., "Applied Metamodelling", book to be published

[eTOM] TeleManagement Forum – enhanced Telecom Operations Map (eTOM), http://www.tmforum.org/browse.asp?catID=1647

[Gauthier 2001] Gauthier P., "OSS/J through Java J2EE Design Guidelines", OSS/J Architecture Board, October 2001, http://www.ossj.org/downloads/design_guidelines.shtml

[Gauthier 2004] Gauthier P., "OSS Inventory API – Overview (Part 1)", Public Draft version 0.9, OSS through Java Initiative, April 2004

[Georgalas 2004] Georgalas N, Azmoodeh M, "Using MDA in Technology-independent Specifications of NGOSS Architectures", First European Workshop on MDA (MDA-IA 2004), Enschede, The Netherlands, March 2004

[MDA] Model Driven Architecture, http://www.omg.org/mda

[NGOSS] TeleManagement Forum - New Generation Operations Systems and Software, http://www.tmforum.org/browse.asp?catID=1911

[OSSJ] OSS through Java Initiative, http://www.ossj.org

[Reilley 2004] Reilley J.P., Gauthier P., "Core Business Entities Concepts and Principles", 2004, http://www.ossj.org/downloads/cbe.shtml

[SID] TeleManagement Forum - SID Overview, http://www.tmforum.org/browse.asp?catID=2008

[Strassner 2002] Strassner J., Fleck J., Huang J., Faurer C., Richardson T., "TMF White Paper on NGOSS and MDA", TeleManagement Forum / Object Management Group, February 2004, http://www.tmforum.org/browse.asp?catID=1875&sNode=1875&Exp=Y&linkID=28972

[UML 2003] Object Management Group – The UML Specification, version 1.5 (final), March 2003, http://www.omg.org/docs/formal/03-03-09.pdf, pp 73-85

[XACTIUM] http://www.xactium.com

Enterprise Change Methodology with MDA

Tony Mallia
Principal Consultant

CIBER Inc. Federal
7900 Westpark Drive, Suite A515
McLean, VA 22102, USA
e-mail amallia@ciber.com

Abstract. This paper describes the practical application of MDA and UML tools in the development of large multi-system projects or system of systems involving multiple development organizations, platforms and tools. A change engineering architectural framework is described with its three view dimensions and how it relates to enterprise architecture. The roles of models at both the change management and methodology views and the separation and use of CIM, PIM and PSM are described in relation to the establishing of integration contracts during the life cycle process. Particular attention is focused on the political reality of multi-organizational development and the delegation of technical decisions. A focus on specifications in the methodology view covers the CIM models (both Ontology and Business Process) and how they transform into PIM Message Templates (Sometimes called a document model) and Component models. Then these PIM models are transformed into PSM component contracts. This paper does not cover PIM and PSM to executable code transformation which is widely covered by current papers. These concepts are illustrated in the implementation of a US Federal Health project which is in operation and in current work being implemented with an XML Schema Factory which shows current off the shelf tools performing transformations.

Introduction

In a large multi-system environment, selection of a single application development tool for all application development is unlikely due to the diversity of language and communications platform technologies and the preference and experience of the various development teams involved.

Successful techniques to produce a coherent implementation across the environment rely on delegation and de-coupling approaches such that the effort can be spread across the teams but that when the parts are assembled together there is high probability of successful integration. Not only will the development be successful but the organization can respond to changes in a routine way maximizing systems development agility.

Change Engineering Architectural Framework

A change engineering framework proposed here has three view dimensions: Perspectives, Focus and Transformation. As shown in figure 1, they provide a space in which to describe the degrees of Transformation:

1. Operational system
2. Change management system
3. Change methodology system
4. Change engineering

These transformation views are applied to the Perspective and Focus dimensions.

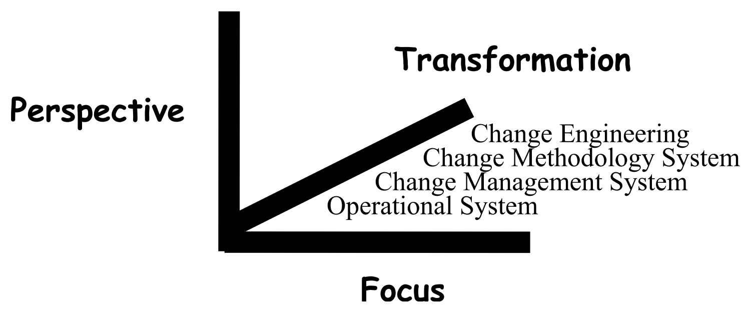


Fig. 1.

Transformation Views

In effect, the organization must look to a Change Management System to make the activities and procedures for change well understood and managed. The Change Management System produces the required Operational System which is used in the Enterprise in day to day activities and is equivalent to the Functioning Enterprise in the Zachman Framework. The development and maintenance of a Change Management System is by a Change Methodology System. MDA provides techniques and tooling to be used by a Change Methodology System to implement the Change Management System.

A system implemented at one transformation view is specified by the model in the higher view. Thus the Operational System is specified by the Change Management Model and the Change Management System is specified by the Change Methodology Model.

Perspective

A number of approaches have defined the perspectives which are targeted towards different players in the organization. A set of 4 perspectives have been found to work in the large multi-system environment. They are shown as the colors in Figure 2.

- The Business perspective defines the environment for the system and contains the manual and computer assisted activities of the operations and their degrees of transformation.
- The Enterprise System Perspective, sometimes called the superordinate system, is the enveloping harness which applies end to end integration around the application systems.
- The Application Systems Perspective, subordinate or subsystems, are the components either bought or built which provide the functionality.
- The Technology Perspective is the language or transport platform on which the application systems and enterprise integration run.

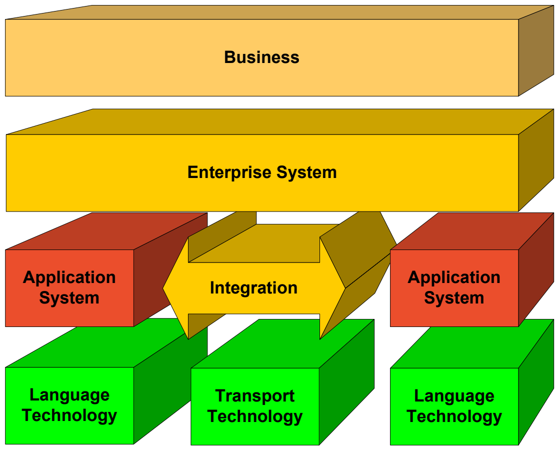


Fig. 2

While these perspectives are not exactly the same as the CIM, PIM, PSM and Platform views of MDA, they provide a better alignment with the organization of the enterprise and the responsibilities for managing large complex systems.

Focus

Focus has been derived from the Zachman Framework but is different in a fundamental way. Where the Zachman framework separates the specification of the system (to be) into the different focus categories, the Enterprise Architecture in this paper defines the focus to be actual instance parts of the system at the appropriate transformation view. The specification is in the model ("What" focus) of the higher transformation view. Thus the Models in the Change Management View describe the 6 focus categories in the Operations View but are not necessarily organized in these categories. The "How" of the Change Management View are the actual activities to produce the Models which describe the Operations View.

The "What" focus defines artifacts or work products which are produced or consumed by the system activities (the "How"). The Where, Who, When and Why define location, participants, schedule and reason for the activities. Thus the Enterprise Change System row describes the project plan and execution of the Enterprise System Change.

Transformation and Focus for Enterprise System Perspective

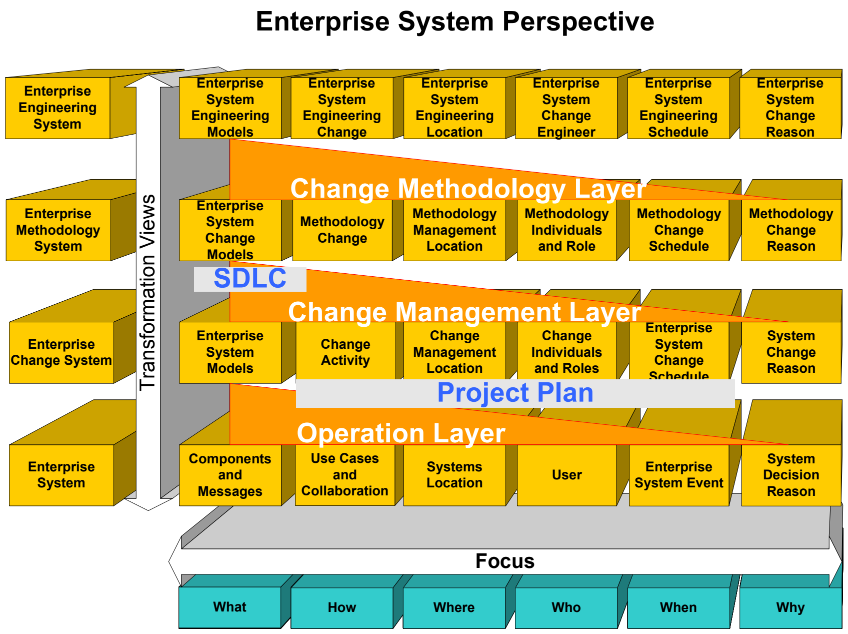

Enterprise System Perspective
Enterprise Engineering System
Enterprise System Engineering Models
Enterprise System Engineering Change
Enterprise System Engineering Location
Enterprise System Change Engineer
Enterprise System Engineering Schedule
Enterprise System Change Reason
Change Methodology Layer
Enterprise Methodology System
Enterprise System Change Models
Methodology Change
Methodology Management Location
Methodology Individuals and Role
Methodology Change Schedule
Methodology Change Reason
SDLC
Change Management Layer
Enterprise Change System
Enterprise System Models
Change Activity
Change Management Location
Change Individuals and Roles
Enterprise System Change Schedule
System Change Reason
Project Plan
Operation Layer
Enterprise System
Components and Messages
Use Cases and Collaboration
Systems Location
User
Enterprise System Event
System Decision Reason
Transformation Views
Focus
What
How
Where
Who
When
Why

Fig. 3.

Figure 3 extracts a horizontal slice through the Enterprise Architecture for the Enterprise System perspective layer showing the Transformation and Focus dimension Views. The Enterprise Change System is described by the Enterprise System Change Models commonly known as Software Development Life Cycle (SDLC) for the Enterprise System.

Execution of the Change Methodology System results in a definition and deployment of the Enterprise Change System and execution of the Enterprise Change Management System results in the Enterprise System Models and the implementation of the required Enterprise System.

Models of the SDLC can be defined in UML using techniques out of the Systems Engineering Models such as the UML SPEM Profile. The Change Management System is the actual instances and actions of the change process which results in the operational system. Thus a horizontal line of cells represents a system realization.

MDA and Change Management

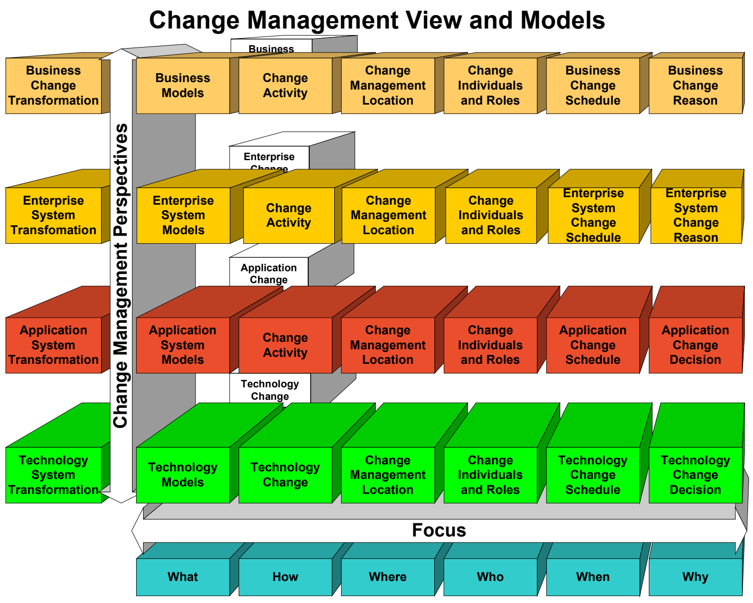

Change Management View and Models
Business Change
Business Change Transformation
Business Models
Change Activity
Change Management Location
Change Individuals and Roles
Business Change Schedule
Business Change Reason
Enterprise Change
Enterprise System Transfomation
Enterprise System Models
Change Activity
Change Management Location
Change Individuals and Roles
Enterprise System Change Schedule
Enterprise System Change Reason
Application Change
Application System Transformation
Application System Models
Change Activity
Change Management Location
Change Individuals and Roles
Application Change Schedule
Application Change Decision
Technology Change
Technology System Transformation
Technology Models
Technology Change
Change Management Location
Change Individuals and Roles
Technology Change Schedule
Technology Change Decision
Change Management Perspectives
Focus
What
How
Where
Who
When
Why

Fig. 4.

The roles of models at the change management system and the coherence of CIM, PIM and PSM allow the bridging between the perspectives. This allows coexistence between MDA and Component Based Architecture where the contracts between Application Systems must be taken to the PSM level to ensure interoperability without platform bridging.

In the Change Management View which is now a vertical slice of the Architecture Framework as shown in Figure 4, model instances of the systems at each perspective are so far disconnected. Coherence between the Business models, Enterprise System Models and Applications System Models would ensure that there is alignment between the Business and the systems implementation and between the Application Systems and the Enterprise Integration Contracts.

By providing the Enterprise System Contracts at both the PIM and PSM levels to the Application Systems developers, integration can be facilitated. Some of the PIM model instances from the Enterprise System can be reused in the

Application Systems development and some examples to show the models which are reusable.

Although the details of the organization are not discussed here, it is assumed that there is a central architecture group which is able to develop and govern the Enterprise System and its models. Distributed Applications Systems development groups would work with the Enterprise Systems development group to facilitate reuse of models and negotiate integration contracts.

In the same way, Technology Systems need to be selected and integrated in a coordinated fashion whether single technologies are selected or multiple technologies must be bridged. It has been found that it is not necessary to generate code in the Application Systems to provide a high degree of Application Systems independence from the Transport Technology rather a binding layer of the Enterprise System can provide interfaces to the Application System where the nature of the Transport Technology is transparent.

Using MDA in the Methodology Model

The MDA CIM, PIM and PSM model types are aligned to the Business, Enterprise and Application System perspectives as shown in Figure 5. The CIM maps to the Business perspective and the Models contain concepts which exist without a computer system. The PIM maps to both the Enterprise and Application System Models as does the PSM. Since the focus of this paper is the Enterprise System Methodology and the development of the integration contracts to allow successful collaboration between Application Systems, the role of the Enterprise System Models used in development will be explored.

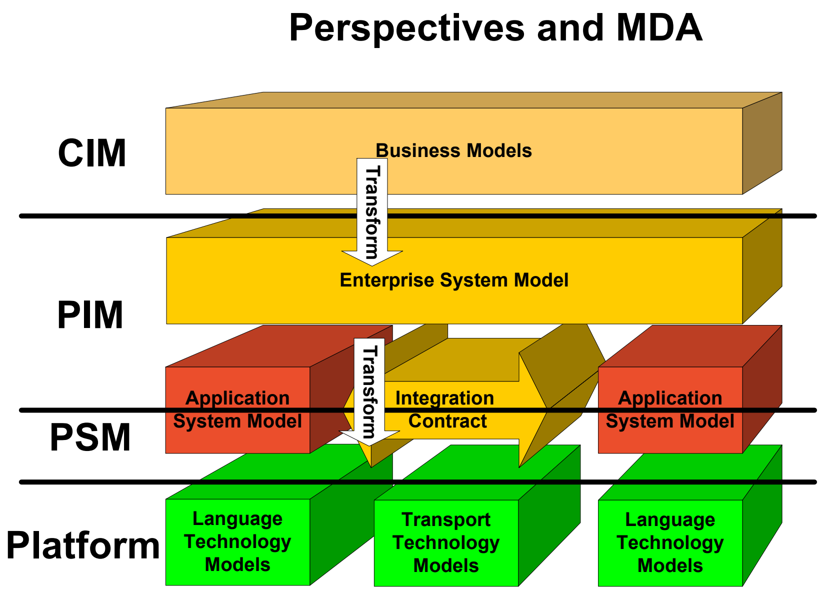


Fig. 5.

The CIM model type as defined in the Methodology System Model is separated into an Information View and a Behavioral View. Both can be defined in UML. Table 1 shows the models involved in the Enterprise System Methodology.

	Information View (What in Operations)	Behavioral View (How in Operations)
CIM Business	Business Domain Model (Ontology)	Business Process Model
PIM Enterprise	Message Template Model	Component Model and Collaborations
PSM Enterprise	Message Payload Schema (e.g. XML)	Component Interfaces and Methods

Table 1.

The CIM Information View Domain Model defines the concepts and relationships of the Business. In this sense it is a lower level ontology and can be governed by a middle level ontology as a UML Profile. An example of a CIM Domain Profile (also in the Methodology System Models) might contain Entity, Role, Act and Identity stereotypes. These stereotypes can be used to mark classes which can be use to transform the Domain model to the Component Model – an Act class might generate a “Process Component”. However no tool has been investigated which can do this but it might be possible with user defined transformation languages to achieve this.

Typically the Domain model will contain packages for Subject areas as well as Datatypes and Terminologies. The Domain Model requires careful construction because elements will find their way transformed to the PSM of the Integration Contract. The contents of the model are the concepts that exist in the business which are independent of the computers systems.

The Business Process model contains Activities and Object Flows representing the actions caused by business events. Some of these activities can be Enterprise Systems Use Cases where an actor is interacting with the external boundary to initiate or respond to an Enterprise System event. The effect of the system on the business environment can be modeled and a superficial message identification as an Object Flow can be defined. The more fine grained actions in the Use Case are added when showing the interaction of the actor to the system boundary and their linkage to the Component Model Collaborations.

At the Enterprise PIM level again the models are separated into an Information View Message Template model and a Behavioral View Component Model.

The Message Template model represents the payloads of messages being exchanged over the integration transport. A number of techniques have been tried to represent the structure and scope of the payload and the most effective has been found to be a UML class diagram showing a message root class associated with the first content class from the CIM Domain model and limiting the scope through the visibility of elements in the diagram. If the element is not visible then it will not be in the scope of the message.

The Component Model shows both the subsystems which will collaborate along with the collaborations which will realize the Use Cases on the system boundary. Since the Enterprise System is superordinate, it provides the behavioral roadmap for the subsystems interactions. Subsystems are considered as black boxes with external interfaces and behavior – their internal structure or behavior is hidden.

At the Enterprise PSM level the Information View Message Payload is defined in the transport platform's language such as an XML Schema and the Behavioral View is expressed in UML as the specific interfaces and methods which will be used such as Home and Remote Interfaces in the J2EE platform.

Model Transformations

Transformations discussed here in this example Methodology include Domain models and how they transform into PIM Message Templates (Sometimes called a document model) and how the Message Templates transform into the Message Schemas. This paper does not cover PIM and PSM to executable code transformation which is widely covered by current papers.

Business Domain to Message Template Transformation

The transformation from Business Domain to Message template is a selection and restriction process which is performed by hand in the UML tool. The restriction is that any concept or relationship introduced in the message template must have existed in the Domain model. No new concepts other than the type of message can be introduced in the Message template and all structures must be referenced in a Domain package.

Message Template Model to Message Payload

A number of ways of performing the transformation from the Message template to the Payload schema have been tried which include custom scripts to process the content of the model including the generation of CORBA IDL and dictionary descriptions to feed into message transformation bridges.

A commercial tool has been used to convert marked UML Enterprise Message Template and Domain models into XML schemas. This will be described more fully in the XML Schema factory example.

CIM Instance Example

These concepts are illustrated in the implementation of the US Federal Health Information Exchange project which is in operation. The project involves an integration server which exchanges health records between two US Federal Agencies. The records are normalized into standard structures controlled by Message Templates which are derived from the Business Domain model.

An example of a Domain package is shown in Figure 6. The package contains concepts about Person and their roles as Patient and Practitioner as required by the scope of the project.

The relationships show the semantic paths which are permitted.

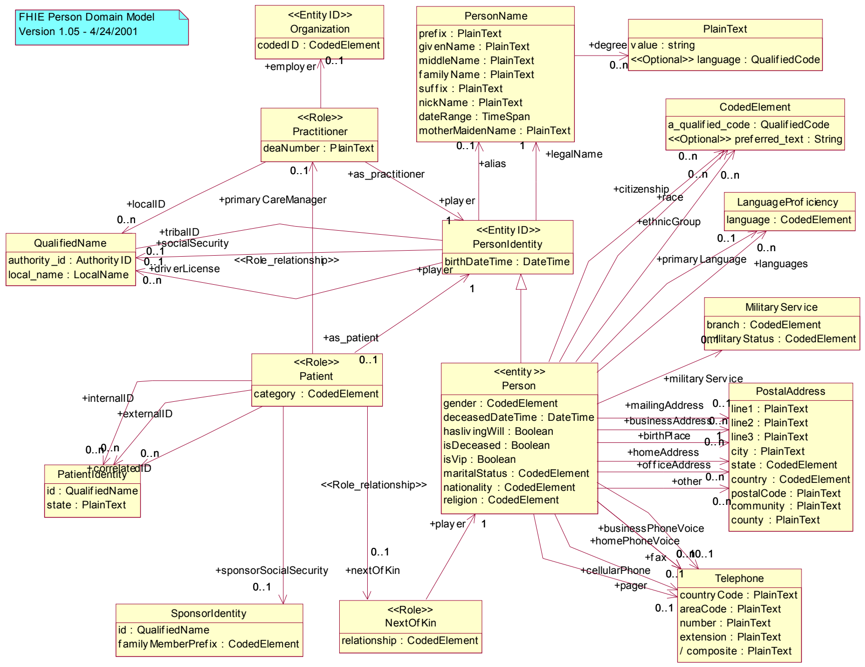


Fig. 6.

Message Template

The Message Template example in Figure 7 shows a fragment of the Patient Encounter message template where the diagram includes only the classes and relationships which are included in the message. The Message root is at the lower left of the figure and is associated with a single instance of PatientEncounter class. The Patient Encounter can have an appointment, admission, discharge and procedures. If you walk all the semantic paths from the message root you get all the semantic concepts which can be included in the message.

In this project the platform was Java and the transport uses serialized Java objects as graphs to convey the PatientEncounter message and the model is used to generate the well formed graph at run time. In this case the run-time bridge reads the model to understand how to construct the graph and transformation is therefore by interpretation.

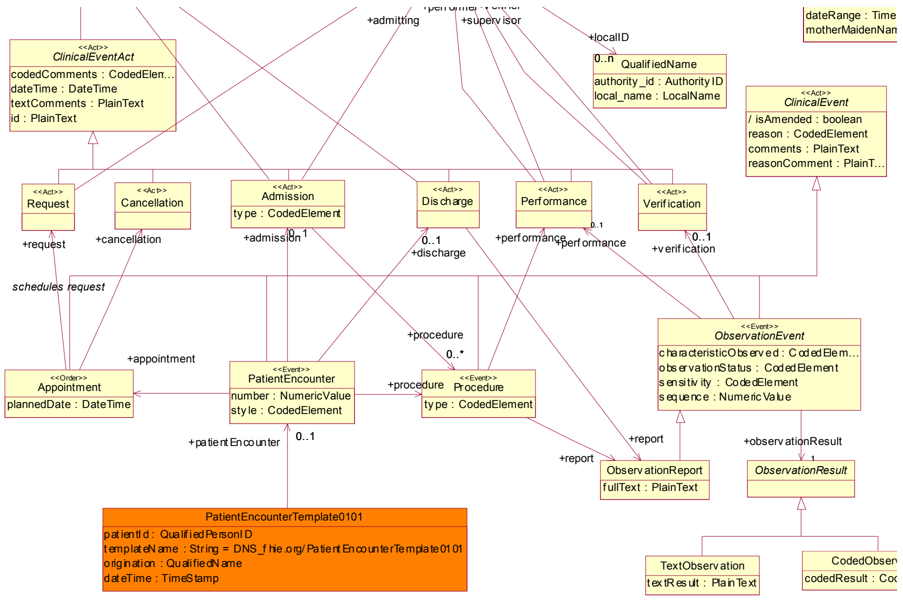


Fig. 7.

XML Schema Factory

The second example is current work being implemented with an XML Schema Factory which uses commercial off the shelf tools performing transformations. Figure 8 shows part of the life cycle with the UML editor on the left, the Transformer tool in the middle and the target middleware IDE on the right.

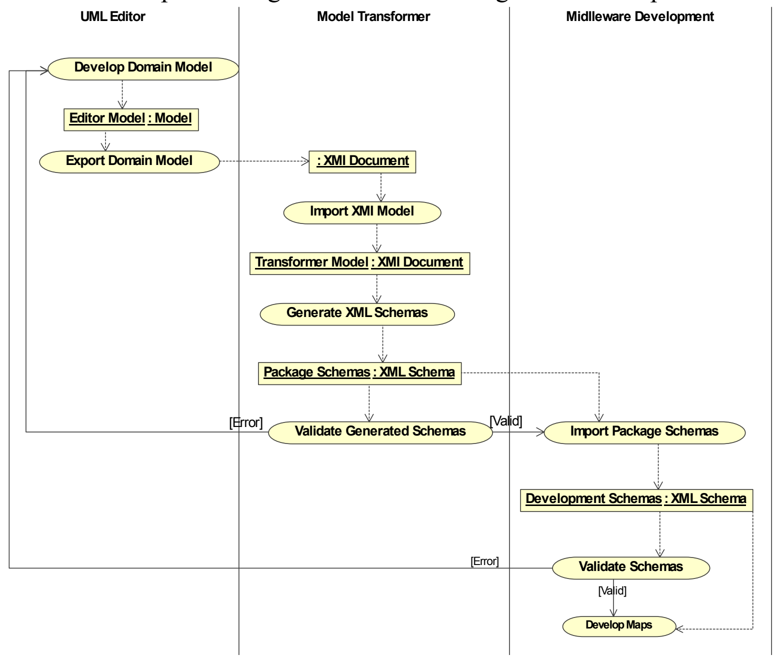


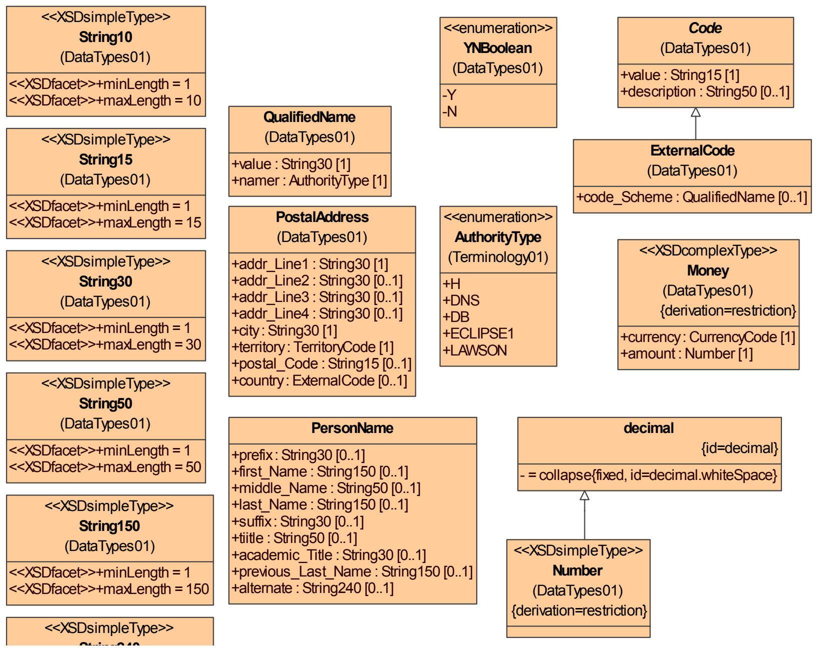
Fig. 8.

The UML editor exports the Domain and Message template models together as a single XMI document which is imported into the Transformer tool. The Transformer tool has preset defaults but can read the marked elements to condition the transformation.

The XML schemas corresponding to packages are generated along with all their external references, namespaces and include statements and can be validated.

They are then exported into the middleware development tool which can use them as the backbone schemas for mapping against other incoming or outgoing schemas and can generate sample documents. The total round trip time is less than a minute.

Figure 9 shows a fragment of the Domain model – in this case part of the Datatypes package and illustrates some simple problems such as defining predetermined string lengths and a Number Datatype which will be generated as a restriction derivation of the XML decimal. The marks appear as stereotypes on classes and attributes as well as a few tagged values.



<<XSDsimpleType>>
String10
(DataTypes01)
<<XSDfacet>>+minLength = 1
<<XSDfacet>>+maxLength = 10
<<XSDsimpleType>>
String15
(DataTypes01)
<<XSDfacet>>+minLength = 1
<<XSDfacet>>+maxLength = 15
<<XSDsimpleType>>
String30
(DataTypes01)
<<XSDfacet>>+minLength = 1
<<XSDfacet>>+maxLength = 30
<<XSDsimpleType>>
String50
(DataTypes01)
<<XSDfacet>>+minLength = 1
<<XSDfacet>>+maxLength = 50
<<XSDsimpleType>>
String150
(DataTypes01)
<<XSDfacet>>+minLength = 1
<<XSDfacet>>+maxLength = 150
<<XSDsimpleType>>
QualifiedName
(DataTypes01)
+value : String30 [1]
+namer : AuthorityType [1]
PostalAddress
(DataTypes01)
+addr_Line1 : String30 [1]
+addr_Line2 : String30 [0..1]
+addr_Line3 : String30 [0..1]
+addr_Line4 : String30 [0..1]
+city : String30 [1]
+territory : TerritoryCode [1]
+postal_Code : String15 [0..1]
+country : ExternalCode [0..1]
PersonName
+prefix : String30 [0..1]
+first_Name : String150 [0..1]
+middle_Name : String50 [0..1]
+last_Name : String150 [0..1]
+suffix : String30 [0..1]
+tiitle : String50 [0..1]
+academic_Title : String30 [0..1]
+previous_Last_Name : String150 [0..1]
+alternate : String240 [0..1]
<<enumeration>>
YNBoolean
(DataTypes01)
-Y
-N
<<enumeration>>
AuthorityType
(Terminology01)
+H
+DNS
+DB
+ECLIPSE1
+LAWSON
Code
(DataTypes01)
+value : String15 [1]
+description : String50 [0..1]
ExternalCode
(DataTypes01)
+code_Scheme : QualifiedName [0..1]
<<XSDcomplexType>>
Money
(DataTypes01)
{derivation=restriction}
+currency : CurrencyCode [1]
+amount : Number [1]
decimal
{id=decimal}
- = collapse{fixed, id=decimal.whiteSpace}
<<XSDsimpleType>>
Number
(DataTypes01)
{derivation=restriction}

Fig. 9.

The style of schemas produced have very high re-use of common elements and cannot determine exactly the scope for an individual message. The Message template diagram must be used in conjunction with the XML schemas for the Application System developer to understand the payload. Some investigation is continuing into the possibility of the tool developing XML schemas based on the Message Template diagram itself. However this will have to wait for standard diagram exchange in XMI to be implemented by the tool vendors.

The XML factory is going into its first production project and has already demonstrated the strength to build well formed XML schemas and apply the governance needed for successful Enterprise integration. Notice in the generated sample below that all the documentation from the domain model is carried into the XML schemas.

```
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- Class: PersonName  -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
          <xs:element name="personName" type="dt01:PersonName"/>
          <xs:complexType name="PersonName">
           <xs:sequence>
            <xs:element name="prefix" type="dt01:String30" minOccurs="0">
             <xs:annotation>
              <xs:documentation>Salutary introduction, such as Mr. or Herr
              </xs:documentation>
             </xs:annotation>
            </xs:element>
            <xs:element name="first_Name" type="dt01:String150" minOccurs="0">
             <xs:annotation>
              <xs:documentation>First name of the person
              </xs:documentation>
             </xs:annotation>
            </xs:element>
```

Conclusion

Implementation experience has shown the need to clearly define an Architectural Framework which must be aligned with the development process and organizational structure within an enterprise. A new Change Management Architectural Framework has been explored which allows the allocation of Systems Change to various teams and where MDA and transformation or alignment between models provides a coherence between views whether they are between the Business and the Implemented systems or between systems implemented with contracts in a Component Based Architecture.

This framework declares the relationship between the methodology system and the change system which it defines.

Enterprise MDA®
or
How Enterprise Systems Will Be Built

Oliver Sims

Abstract. Success for MDA in the longer term depends on its ability to bring compelling value to enterprise IT. Neither UML® nor executable UML are currently major players in enterprise IT. They risk becoming also-rans if their potential fails to be realized in the enterprise context, both in developing new applications and, arguably more important, in the integration and interoperability of existing systems. In addition, MDA must inter-work with other present and emerging standards, such as web services, business process management (BPM), and JCP (Java Community Process) initiatives. This paper first analyses the OMG's stated goal for MDA, a goal that is well beyond the oft-heard talk of model transformations and code generation. It then describes how a small number of disparate but important advances in the industry can be brought together in a truly synergistic way to achieve that goal. Lastly, a roadmap for reaching the MDA goal is suggested.

1 MDA - What's it all about?

MDA often appears on the surface to be all about transformations between four different kinds of model—a CIM, a PIM, a PSM,[1] and code (code being a model of the run-time). Often the code generated is thought of as being restricted to skeleton code, plus some glue code. And if this is all it is, then it would certainly not live up to the OMG's claims. What claims? Well, the home page of the OMG's MDA website (http://www.omg.org/mda/), under the heading "*How systems will be built*", presents what might be called the MDA vision:

> *MDA provides an open, vendor-neutral approach to the challenge of business and technology change. Based firmly on OMG's established standards, MDA aims to separate business or application logic from underlying platform technology. Platform-independent applications built using MDA and associated standards can be realized on a range of open and proprietary platforms, including CORBA, J2EE, .NET, and Web Services or other Web-based platforms. Fully-specified platform-independent models (including behavior) can enable intellectual property to move away from technology-specific code, helping to insulate business applications from technology evolution, and further enable interoperability. In addition, business applications, freed from technology specifics, will be more able to evolve at the different pace of business evolution.*"

Why will systems be built this way? Because achieving MDA's aims can radically reduce development, maintenance, and evolution time, provide for enhanced flexibility, and can bring the solution design much closer to the problem definition.

But how can MDA do this?

The clues lie in these key parts of the vision statement: "separate business from technology", "enable IP to move away from technology-specific code", and "fully-specified PIMs (including behavior)" on "a range of platforms". This means that a PIM that includes behavior specified using an action language could either be the basis for generation of 100% of code, or could be

[1] CIM: Computation-Independent Model (often referred to as a business or requirements model); PIM: Platform-Independent Model; PSM: Platform-Specific Model.

directly executed (or more accurately, interpreted).[2] In the enterprise system context, "platform-independent" means independent of platforms such as J2EE, CORBA, .NET, various databases, various GUIs, and so forth. Clearly achieving this aim would be of extreme value! Even a partial attainment of the goal could be highly attractive.

But what factors can enable the aims to be attained? MDA alone cannot do it all. MDA's scope is certainly larger than simply defining kinds of model and the relationships among them: it is based on the sound foundation of MOF™, such that languages other than UML can be used, and important inter-language capabilities can be managed automatically. However, MDA certainly does not include all the various aspects needed for scalable, flexible, service-oriented, and interoperable enterprise systems. So MDA must assume other factors that support its central vision. What are these factors?

To answer this, let's re-state the key parts of the MDA vision in terms of questions:

1. How do we separate the business logic from technology logic?
2. What is a good structure for generated code, and therefore of a PIM, such that the necessary 'ilities (scalability, flexibility, service-orientation, etc.) are delivered?
3. How do we ensure that the "business" in the business logic actually relates well to the business in the business?

Happily, there are good answers to these questions. Briefly, they are:

1. A **product line approach** can separate business logic from technology logic.
2. Effective **architecture**, preferably component-based, can be delivered to application developers through UML profiles and Domain Specific Languages (DSLs)[3].
3. An approach to **bridging the "Business/IT Divide**" that provides a good CIM-to-PIM transformation.

And even more happily, these three answers tend to be very synergistic. Indeed, some have argued that to be successful in addressing enterprise IT challenges with one, you have to address the other two as well. Of course, other factors such as process and organization must also be addressed; however, implementing the three approaches just listed tends to lead inexorably to others, so they're not forgotten. Meanwhile, let's consider the three approaches and how they contribute to MDA goals.

2 Implementing the Goal – MDA's Enterprise Companions

2.1 Separation – a Product Line Approach

MDA "aims to separate business or application logic from underlying platform technology." Business logic can be defined as any development artifact, or part of an artifact, that is unique to the business. For example, code that calculates a price, or a screen definition that defines how a sales order record should appear, or a data schema for customers, are all "business logic". On the other hand, code that sets up the necessary conditions for a web service to be invoked, or handles the window creation mechanism at the GUI, or manages an ACID transaction, is "technology logic."

[2] There is a valid argument that a PIM that is executable is actually a PSM that is specific to whatever engine handles the execution. However, in the context of enterprise systems, where "platform" generally means the commercially-available middleware, we prefer retaining the term "PIM" for a model that can be executed on two or more of those platforms.

[3] DSL stands for Domain-Specific Language. DSLs have been addresses in previous MDA Journals (see [5][10])

Ideally, the business application development team—whether working on new applications or integration projects, whether outsourced or in house—should be concerned only with business logic, everything else being consigned to a COTS (Commercial Off The Shelf) "platform," as shown at the top part of Figure 1. However, such a COTS platform does not exist today. Although much technology logic is, of course, handled by COTS products (middleware, operating systems, DBMSs, EAI managers, and so on), there is always a gap between the platform provided by a given set of COTS products and the business logic within any given development project ("platform gap" in Figure 1).

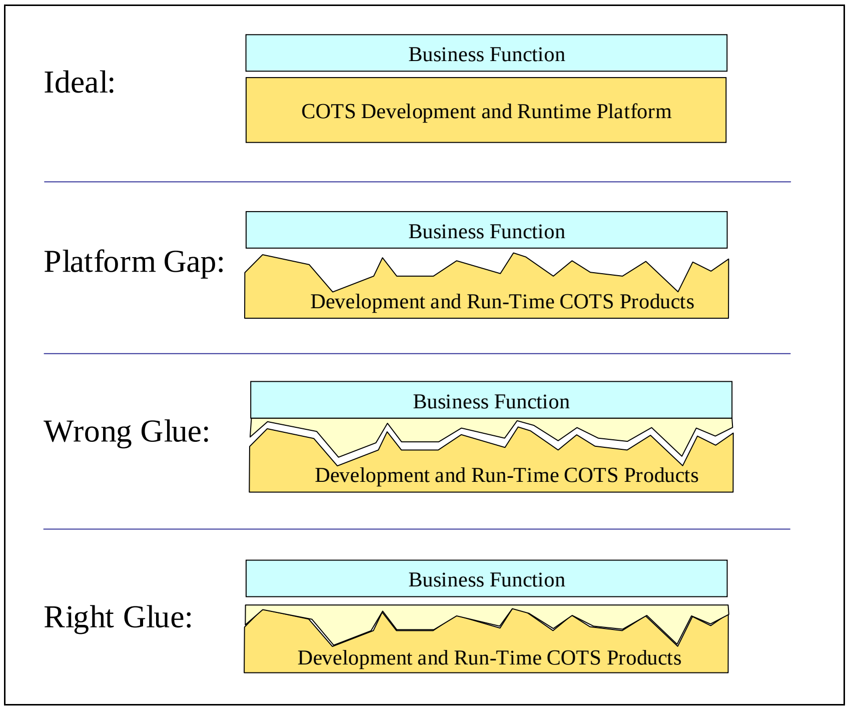

Ideal:
Business Function
COTS Development and Runtime Platform
Platform Gap:
Business Function
Development and Run-Time COTS Products
Wrong Glue:
Business Function
Development and Run-Time COTS Products
Right Glue:
Business Function
Development and Run-Time COTS Products

Figure 1

The gap comprises both answers to "how to" questions, and specific artifacts that deliver the answers pre-packaged for business developers. Examples include how to handle concurrency, transactions, transparency across different communications stacks, and pooling of various sorts (threads, DB connections)—as well as how to integrate the various COTS products. There are also how-to questions about the development environment (itself a fairly complex IT system) such as how to share models effectively, transform models, structure models and code, and define and collect useful metrics. Finally of course, there are the big issues: how to provide for flexibility, scalability, re-usability, and so forth.

A specific example of an artifact that helps "fill the gap" is a client-side proxy that enables a business developer to invoke another component either synchronously or asynchronously with respect to his or her thread of control, providing a call-back method/operation for an async reply.[4] This isolates the developer entirely from the nature of the underlying communications stack. Another example is a script that enables a model to be transformed into another (for example, from CIM to PIM).

The gap is often informally filled either by all business application developers learning a great deal about software technology, or by a few expert software technologists within a project team

[4] Experience suggests that most of the time invocations are made synchronously. But occasionally async invocation is needed—and when it's needed, it's really needed!

or shared across project teams. However, the filling—often called "glue"—is seldom captured and re-used by other projects. Hence the technology efforts are often duplicated, and often projects run late because the technical skills required to fill the gap are underestimated. In summary, filling the gap on a by-project basis is the wrong way to provide glue ("wrong glue" in Figure 1).

Platform vendors may argue that the gap is necessary, since they must provide for a very wide range of customer requirements. This is true as far as it goes; however, such platform vendors do not seem to have taken on board the fact many applications—or integration projects—are often technically incredibly similar to each other. This similarity is what makes product lines feasible. Technically similar systems are often said to conform to the same "architectural style" [1] or "approach" [2]. Sometimes the similarity derives from similarities in the business area being addressed, and sometimes from similarities in the technical approach to solving quite different business problems. In any case, systems built to the same architectural style have very similar glue requirements. The observation that many systems are technically similar is at the heart of the product line initiative. The product line concept [3] has been summarized in previous MDA Journals [4] [5], and there have been other proponents of the same approach, albeit under different names, including Herzog [2] and Hubert [1].[5]

Indeed, platform vendors could find it profitable to provide for the common architectural styles of enterprise systems ("right glue" in Figure 1). Currently, however, this glue must be provided by the IT organization; and providing it informally within each project is a huge waste of corporate resource. A product line approach enables the glue to be captured and used (or re-used if you prefer) for multiple projects.

Separation of business from technology logic can be done in two ways:

- Provide the glue as an addition to the COTS runtime, already deployed, and treated as an in-house addition to the run-time—for example, a logging service.
- Generate the glue code each time it is needed by a project, and deploy it with the application—for example, code to log to a common logging database.

In general, one should generate as little code as possible within a given project. The reason for this is that the more is generated, the more has to be tested. Imagine that much of the technology logic in a logging service were to be generated for each application. A change to that generated logic would require that all the applications that embed it will have to be re-tested and re-deployed. Some might say that the supposedly unchanged business logic does not have to be re-tested; others would be more cautious, and say that re-compilation or even re-link must be re-tested even though a large part of the source code has not changed (as far as anyone knows!).

In reality, both approaches will be needed. However, it is clearly much better, wherever possible, to produce glue once, and to deploy it once, thereby effectively creating a higher-level platform. Since "platform" normally refers to COTS products, I apply the term *virtual platform* to the combination of COTS products (middleware, GUI frameworks, DBMSs, application servers, etc.) and glue.

Separating application development tasks and artifacts from platform tasks and artifacts, however, can take a significant management effort. Standardizing on a given set of COTS products is something that many enterprises already do. However, it's much better also to rigorously separate as much of the "glue" as possible. This leads to an organizational structure

[5] The product line concept is not new: a good argument can be made that the billions of lines of mainframe COBOL code that ran the world's businesses in the last third of the last century were often produced in a similarly-structured environment.

whereby a "platform" (or "infrastructure") group has as its mission to provide as many transparencies as possible for a separate application development group. Making that organizational change is often not so easy. The higher-level virtual platform is created and maintained by a platform group, whose mission is to "delight" (as one of my clients put it) the application development group. Application development projects are run within the latter group. In this way, business change and evolution is separated from that of technology.

Now that we've got the business logic as fully separated as possible from technology "stuff," we can now consider how MDA applies to business logic by itself. But a "fully-specified PIM", with action language providing behavior, should in principle be executable. And executability requires more than just the business logic: it also requires a specification of the structure being executed—especially so if the target platform is a distributed system. So: how should the model be structured—and how should the generated code be structured?

2.2 Enterprise PIMs

It is relatively easy to answer the question of how code generated from an enterprise PIM[6] should be structured. It should, of course, be structured following best-practice modularization (e.g. high cohesion low coupling), where modules interact so as to deliver scalability and to minimize dependencies for flexibility. It must also conform to the virtual platform so that viable code can be generated. Best-practice modularization means taking a mature computer-based software engineering (CBSE) approach. By "mature", I mean the hard-headed software engineering approach as opposed to the "let's buy everything off the shelf and mix-n-match to magic a solution" approach.

Structure and Architectural Style

But how should the PIM as a whole be structured? Arguably the best way is to structure the PIM according to mature CBSE as well. Since each component encapsulates and realizes a specific business concept, it becomes fairly clear where the business logic fits. The UML2 component provides an ideal model element for this approach, since it "addresses the area of component-based development and component-based system structuring, where a component is modeled throughout the development life cycle and successively refined into deployment and run-time." [6]

In a product line approach, the question of technical structure is handled by the platform group—or by a separate architecture group (possibly a third IT organizational element). Such structural design is usually called an "architecture", and is the expression of an architectural style. Of course, there are other forms of architecture; for example, of particular use is a "business" architecture that provides business patterns—for example, the pattern for design of a Contract, perhaps as suggested in [7].

Structure—including allowable interactions between components—should be defined by the platform group, and delivered to the application development group in the form of a tested UML profile. The profile also defines how a PIM is packaged, so that, for example, a design-time can be taken as a whole from a repository and plugged into (re-used in) a PIM. Indeed, there is no reason why, as soon as the component is identified, it should not become "executable" within the development environment, so that it can be queried, for example, as to its development status, and also be able to run whatever simple test cases may be available: for example, create an instance which has a unique key. This approach to "living components" is well described in [1]. It is

[6] I say "PIM" singular: but of course a large system may be designed within several models, each at the same level of PIM-ness.

particularly useful in its ability to generate appropriate test data on an ongoing basis as function is added to the component.

The PIM's Profile

In defining a UML profile for PIMs, the architecture group first has to understand the particular architectural style being addressed. Then a model of this architectural style is created. Such a model (for example, [8]) might define:

- Distribution tiers
- The kinds of component in each tier and allowable interactions across and within tiers
- The kinds of classes that realize components (such as a "focus" class—a UML-defined stereotype)
- Patterns for such things as component granularity, scalability, and dependency management
- The kinds of binding (tight, loose, etc.) used for different model elements
- The required structure of the PIM itself, including namespaces (for example, a component could be required to be a separate namespace)

Second, and always assuming that the business developers are using a modeling tool that can make use of UML profiles to guide developers when they build their PIMs, the architects create a UML profile from the architectural style model. (Actually, with the better profiling tools, building the model also builds the profile). They then test the profile, before shipping it to what might be called the "development-time platform". In this way, the developers have architecture delivered to them through their everyday tools (just as detailed procedures defined by a development process might be similarly delivered).

A sample fragment of a profile for distributed enterprise component-based systems is shown in Figure 2, where relationships in red would be specified in OCL and are shown here for convenience only. The profile makes use of the new Component concept in UML2, which obviates the need for the more complex profiles that were required to model enterprise components with UML 1. The figure is addressing component granularity, and defines four levels, each characterized by a different kind of component. An Application Component is an application delivered as a component (although seldom as a single artifact). It is realized by a collaboration of "Business Components", which is the core concept presented by Herzum [2]. A Business component encapsulates a business concept such as "Customer" or "Order Manager" across the distribution tiers, and is realized by a collaboration of Distributed Components. A Distributed Component is an abstraction of the kind of component provided by EJB, COM+, and CCM. Two subtypes of Distributed Component (not shown in the figure) allow for implementation using either a programming language (e.g. EJB, COM+)—the Algorithmic DC, or using only a declarative script of some sort (e.g. BPM definitions)—"the Declarative DC". Finally, a Service Component is a collaboration of distributed components that realizes a closely-related set of services provided by a given distribution domain, for example, the "logical server" domain, or the "user interaction" domain. A full discussion of discussion of distribution domains and tiers is beyond the scope of this paper, but can be found in Sims [8].

Figure 3 shows a fragment of a model using this profile. The figure shows a Service Component in the "logical server" domain. The model uses standard UML stereotypes (focus and auxiliary) for the classes that constitute the realization of the Order component. The implementation of the profile enforces scalability through such things as restricting an ACID transaction to occur within a single invocation of the logical server domain from other domains such as user interaction or business process management (BPM) domains.

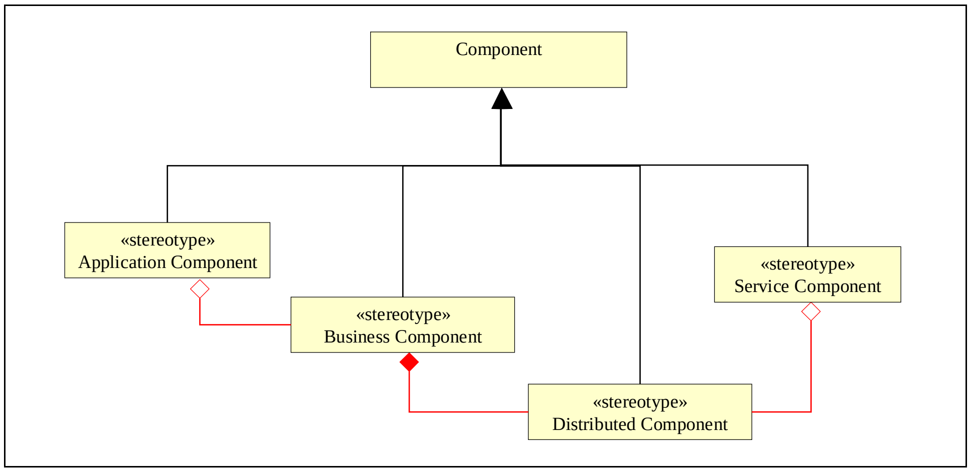


Figure 2

A modeling tool that enforces a profile can ensure that business logic developers conform to the defined architectural style. The "ilities" (scalability, maintainability, re-usability, accessibility as services, flexibility, configurability, manageability. etc.) can be, to a large extent, enforced through constraints specified within the profile, by the virtual platform, and by code generated from models that are created via the profile.

A profile of this kind can also provide for "wrapper" components that provide a service through their interfaces, and internally access legacy applications. Where there is an enterprise interoperability bus that defines a specific "real time" (as opposed to batch) interface technology and design (such as a particular usage of WSDL), the profile can be used to define wrappers from which the appropriate WSDL interfaces can be generated, perhaps wrapping some EAI adapters or BPM scripts.

Behavior – the Action Language

To be computationally complete, a PIM must include algorithmic behavior. This can be achieved through use of the UML Action Semantics (see [9]). Using an existing 3GL would require that the language be subsetted. In practice, this means that the subset must be documented, taught, and maintained—probably a much larger job than using an existing action language. Having said that, it must be pointed out that today, although profile building and action languages are available on the market in modeling tools, I am not aware of any tool that combines both.

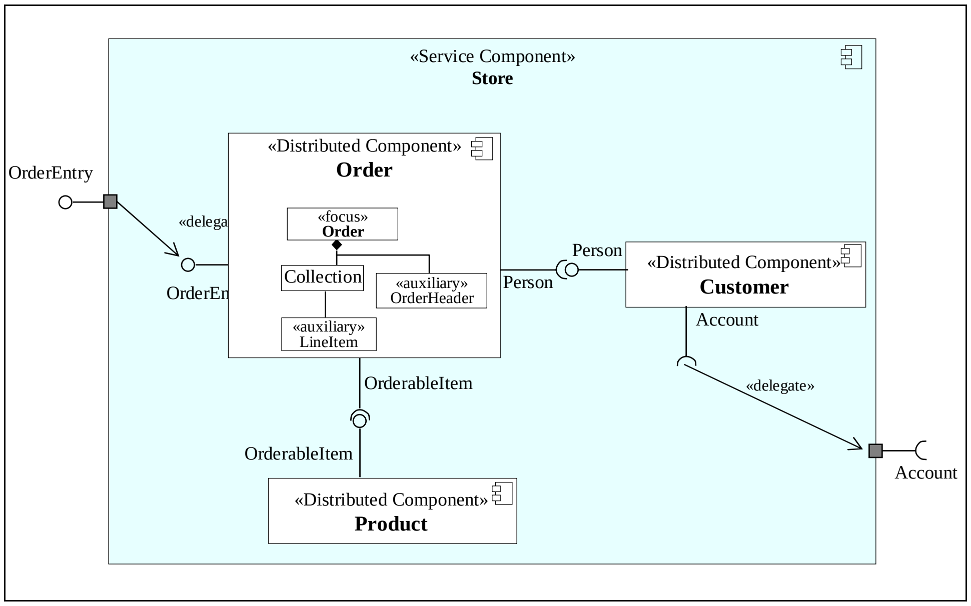
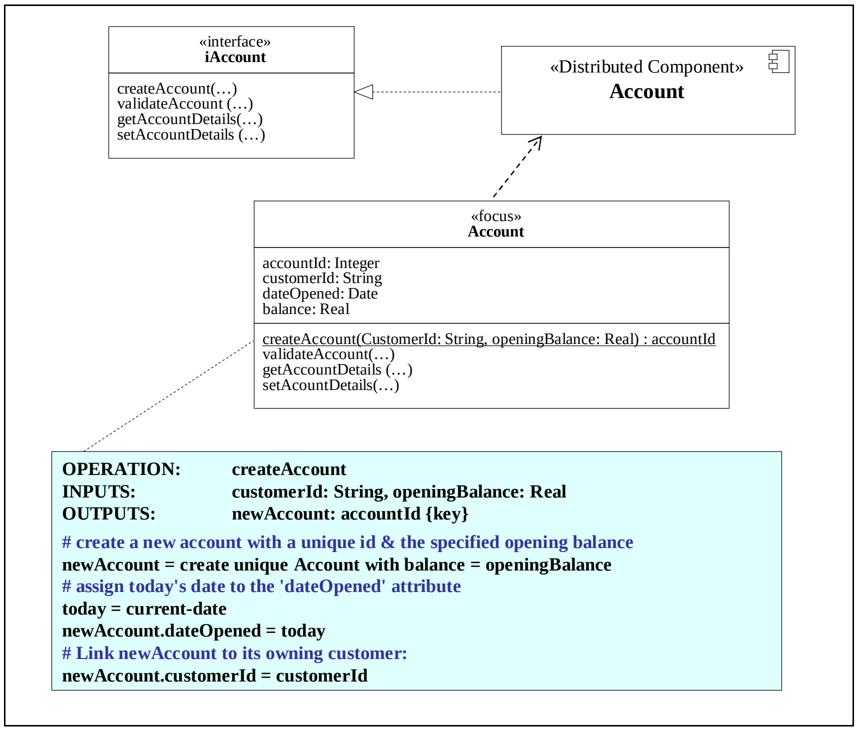


Figure 3

Figure 4 illustrates use of the action language.[7] The figure shows an Account component that provides an interface, and that has a realization—the focus class “Account”. Some design detail that would be present in a real PIM has been omitted or compressed, so please do not take this as a fragment of a real working PIM. The behavior of one operation—createAccount()—is shown, although a tool would not normally present the action language as a UML comment as the figure suggests.[8]

[7] UML defines the abstract syntax for Action Semantics, but does not define a specific notation (that is, a concrete syntax). The concrete syntax shown in the figure is Kennedy Carter’s implementation of Action Semantics. The content is a modification of a sample taken, with their kind permission, from Kennedy Carter’s tutorial on Executable UML (xUML—see http://www.kc.com/MDA/xuml.html). However, any errors or omissions in the figure I claim as my own.

[8] See [9] for an example of action language used to fully define a state machine.



```
OPERATION:      createAccount
INPUTS:         customerId: String, openingBalance: Real
OUTPUTS:        newAccount: accountId {key}
# create a new account with a unique id & the specified opening balance
newAccount = create unique Account with balance = openingBalance
# assign today's date to the 'dateOpened' attribute
today = current-date
newAccount.dateOpened = today
# Link newAccount to its owning customer:
newAccount.customerId = customerId
```

Figure 4

2.3 Business/IT Bridge

"Perhaps the greatest difficulty associated with software development is the enormous *semantic gap* that exists between domain-specific concepts … and standard programming technologies used to implement them." [10] This has often been termed the "Business/IT Divide", and has often seemed particularly intractable.

However, the CIM-PIM-PSM trichotomy strongly suggests that MDA provides for CIM-to-PIM generation as well as PIM-to-PSM-to-code. This means removing the business/IT divide.

A CIM is often known as a business model, or as a requirements model. There are many ways of interpreting what such a model is. To some, it is a model of the enterprise. To others, it is an idealized model of an application. Since the context for MDA is that of IT systems, I interpret a CIM to be a computation-independent model of that part of the business that is to be addressed by an IT system. While parts of a CIM might be simulated (using, for example, the "naked objects" [17] approach), it cannot, even in principle, be automatically transformed or interpreted such that it can be directly deployed into an operational system.

I also believe that the work of building a CIM stops when there are no more questions to be asked about the business in order to build the IT system. This includes low-level business processes (procedures or algorithms) about, for example, exactly how a price is calculated, or how stock is to be allocated against a sales order across perhaps several warehouses with different delivery schedules and shipping routes.

Of course, this does not imply that a CIM must be complete before work on the PIM starts: they can be gainfully and quite happily be overlapped.

So the problem is this: how can we develop a valid CIM that is also structured such that it can be straightforwardly transformed into a skeleton PIM, where the skeleton provides the structure for, and some of the content of, a valid IT system?

An answer is provided by the recent EU Combine project [13], which suggested that a business model can be seen in terms of four categories. The first two are:

- Processes that require human intervention (these are candidates for implementation as workflow using declarative distributed components that are typically implemented using a COTS workflow product)
- Processes that do not require human intervention, but where the business requires that a record be made, for future consultation, of intermediate states. (these are candidates for BPM approaches using declarative distributed components that are typically implemented using a BPM or EAI COTS product)

But what about the core business systems that BPM, EAI, and Workflow processes call upon? This is where the third and fourth categories come into play. The Combine project developed an approach, based mainly on Taylor's concept of business engineering [11] and also on my own early experience with the GUI end of CBSE [12], called "business element analysis".[9] Full exposition of business element analysis is beyond the scope of this paper; however, the basic idea is to produce process and information models as usual, then, using a set of defined heuristics, separate the model into two categories of "business element" as follows:[10]

- Processes that do not require human intervention and where the business is not interested in keeping (for future consultation) a record of any intermediate states. These processes—which can often go down to the procedure level—can often be grouped by the resource they primarily operate upon—for example, create order, amend order, delete order, and query order(s). Each such group is a "process business element",[11] and is often the responsibility of a single organizational unit. (A process business element a candidate for implementation as a process business component.)
- The "important" entity resources[12] (for example, Sales Order, Customer, Addresses) that the business needs to record for use by processes. Each such important resource is typically a group of the resources in an information model (for example, Customer has several kinds of address, various codings such as "major customer", customer number, and so forth.) Each such group is an entity business element (and is a candidate for implementation as an entity business component).

Business element analysis provides a view of the business that is much less cluttered than many others, since low-level but essential business detail is hidden within each business element. But it does something more important from the MDA point of view. Given a component-oriented architectural style of the kind mentioned previously, it becomes clear that business elements can be nicely—and automatically—mapped into business components, where the type of the business

[9] A paper on the Combine approach to business modeling, which included a section on business elements, was presented at the EDOC 2003 conference. [14]

[10] For those familiar with the approach, it is often useful to start with the obvious business elements and derive process and information models, and further business element models, as you go.)

[11] I appreciate the term "element" being used for a group of things is oxymoronic; however, that's what the term is at present anyway.

[12] The word "important" is being used in a special sense here. A description of the identification process for "important" resources is presented in [14] and is beyond the scope of this paper. Briefly, however, in an ERP system (for example) Customer and Order are "important" whereas "Address Line" or "Quantity Ordered" are not. Suffice to say that business people have no problem with the concept. They will say for example, "Our business deals with customers, suppliers, orders, pricing engines, etc." They realize of course that such resources have attributes such as address line or actually composed people, But they do not say: "Our business handles address lines, quantities ordered, etc." A more formal version of this reality is outlined in [14]. Finally, it should be mentioned that Combine defined two quite different kinds of resource: "artifact" (information or entity) resources, and "actor" resources. For the purposes of this paper I have ignored the latter.

element becomes the type of the UML2 component. Indeed, if not outlawed as just too heretical, the same model element can flow from CIM right through to code! Now that's traceability!

Figure 5 illustrates part of a CIM, with business elements represented by stereotypes of the UML2 component (the "BE" in the stereotypes signifies "business element"). Figure 6 shows a PIM that could have been generated from the CIM fragment in Figure 5. The "BC" in the stereotypes refers to the business component concept mentioned previously.

Of course, the PIM is initially very skeletal, and must be greatly refined to approach the kind of fully-specified PIM discussed previously. For example, each business component is refined into however many of the architecturally-defined distribution tiers are necessary to properly express the business concept in the system, each tier being realized by one or a few distributed components. Service components are also defined for each distribution domain.

The key point is, however, that it is quite possible to generate a PIM from a CIM such that there is isomorphic traceability. Thus there is a straight-line process from CIM through to code, which Hubert [1] calls "component metamorphosis." Note, for example, the similarity between Figure 5 and Figure 3!

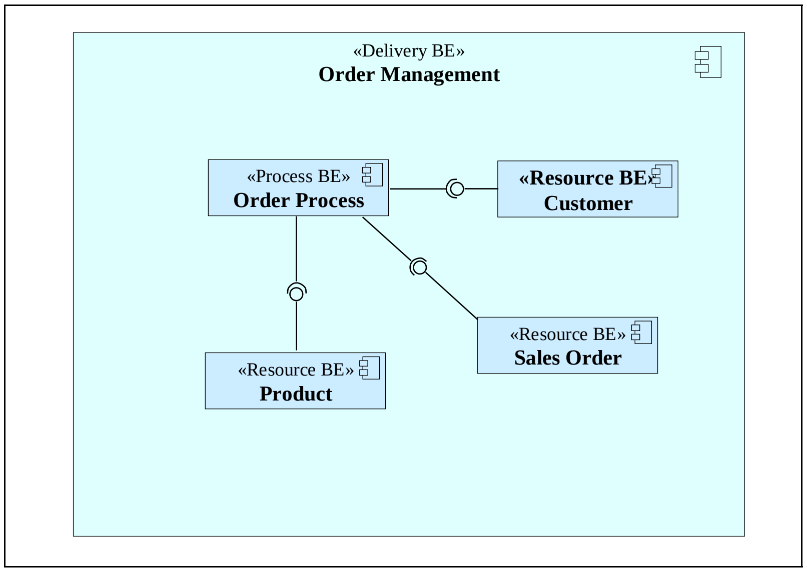


Figure 5

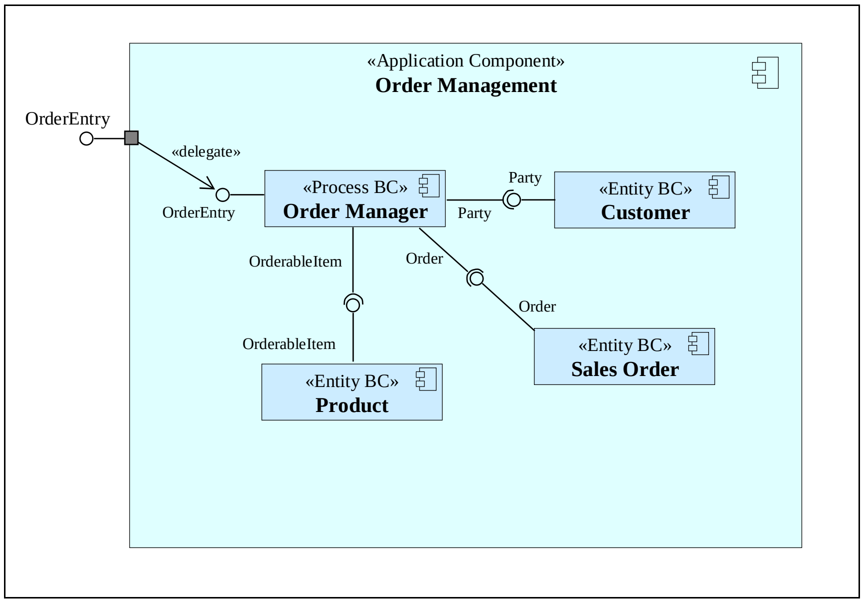


Figure 6

2.4 Domain Specific Languages (DSLs)

DSLs are an old idea currently being given a whole new set of rather attractive clothes. Described in previous MDA Journals [10] [15], they are currently being pursued by both IBM and Microsoft. There is a real sense in which a UML profile is a DSL. However, the kind of DSLs being foreseen will have a look and feel quite different than today's typical UML tools, even when profiles are applied. This is partly because an enterprise system needs other design tools than UML alone, and partly because a DSL is likely to be presented to the user as one of a family of tools, all hosted in a more general tool such as Eclipse.

I have recently been working with a client on DSLs for a particular domain within the finance industry. The domain is a small part of an enterprise system. Very surprisingly, it seems that it may be possible to start work now on a product that, in two year's time, may provide for end users to define their own applications with only a little help from their IT people. This is an extremely attractive proposition. It is made (we think) possible first through the constrained nature of the domain, second by the ideas presented in this paper, and third by the imminent emergence of mainstream DSL tools.

MDA and MDA-like approaches + Product Line + Architecture + DSLs → The Future!

3 Getting there – a roadmap

In this paper, I have tried to show that the MDA vision, in enterprise systems, can only be fully achieved through a combination of product line, architectural style thinking, and solving the business/IT divide. This doesn't mean that things like process engineering, software engineering, testing, deployment, project management, and so on are not also affected. However, the driving forces are the organizational and technical directions of the product line approach, and the structural and simplifying directions of applied architectural styles and CIM-PIM linkage, with the MDA approach tying them together synergistically.

Although Microsoft is progressing along a slightly technical different road than MDA, its essential goals for enterprise systems seem to be similar.

So how do we get there? In effect, we’re looking at a transition from where we are now to a much-improved development environment. And although all the capabilities needed are not yet integrated into commercially available tools, there is certainly sufficient support available now for early adopters to start the journey. In particular, we can construct UML profiles for enterprise systems today, and generate at least skeleton code.

The question is, how does an IT organization make the transition to MDA?

Luckily, transition processes are not new. Guttman and Matthews [16] describe a particularly good one. The key is non-intrusion into current and planned development projects, and certainly to avoid big bang approaches. Thus the idea is to start small, developing initial capability in the context of a few (one to three) real projects. These projects might be termed “pilot projects” because, although real development projects, they are the vehicle to pilot the transition.

But who does the extra work (for extra work there will inevitably be)?

Another key part of the transition is to fund a group that has, or is in the process of gaining, technical knowledge of the MDA approach, and who share the MDA vision. People in this group, skilled architects, designers, modelers, and software technology engineers, will devote somewhat more than 50% of their time to working in the pilot projects as project members, helping to produce project deliverables. The other part of their time is spent growing the virtual platform, based on their project experience and on project priorities. This will include capturing the architectural style in a UML profile, applying that profile through tools, and so forth.

In effect, this group is the genesis of a separate Infrastructure/Architecture/Process unit (what we called the “platform group” previously) within the IT organization. It is funded to provide and evolve a high-productivity environment for business application developers. Working within projects, and developing re-usable “glue” based on project priorities, should prevent this group becoming an ivory tower; especially when the results of their efforts are deemed null-and-void unless they are delivered through tools, and unless business application developers—their customers—like the results. In other words, the IT organization must evolve to one where the platform group provides high-quality and immediately useful services and “products” to their customers, who are the business application developers and their managers.

The transition process moves forward from the initial pilot projects through several defined stages (with go/no-go points built in), ending with phased roll-out of MDA capability to the whole of the IT development organization. On the way, the platform group will probably evolve into two groups: an “infrastructure” group responsible for provisioning and maintaining the virtual platform, and an “architecture” group responsible for designing the virtual platform. Process may be handled by the architecture group or by a separate group within the platform area.

Experience suggests that the major impediments to success in such a transition are funding problems and difficulties in making the required organizational changes. Creating a product line environment that majors on the MDA vision requires a fixed focus on making life easier for the business application developer. After all, dealing with the awesome intricacies of the reality of business is challenging enough. Dealing at the same time with forty-eleven services thoughtfully provided by middleware vendors is a truly Herculean task. We must change to an environment that does not require each business application developer to be a Hercules.

4 Summary

We asked how the claims made for MDA can be substantiated in the context of enterprise distributed systems. One answer lies in the synergistic combination of three major approaches: product line, component-based enterprise architecture, and the business element approach to resolving the business/IT divide. Product line organization separates technology logic from business logic; clean business logic is structured, and key 'ilities provided for, by a component-based enterprise architecture; the enterprise architecture defines a structure into which the CIM business elements can be isomorphically mapped. MDA provides the catalyst. The result is the prospect of fully specified enterprise PIMs.

Figure 7 illustrates this vision. Within the platform group, the infrastructure group delivers the development and run-time environments, complete with generators and transformation tools, as well as the various COTS products. In the figure, the major tools used by business application developers are represented by the "Modeling and Development Tools" box. Profiles that the architecture group provides are used to define PIMs and to configure the various tools. Solid arrows show the main data flow through the development lifecycle where the "data" consists of artifacts such as the CIM, the PIM, and so on. Dotted arrows show dependencies; for example the modeling and development tools depend on the enterprise SOA profile.

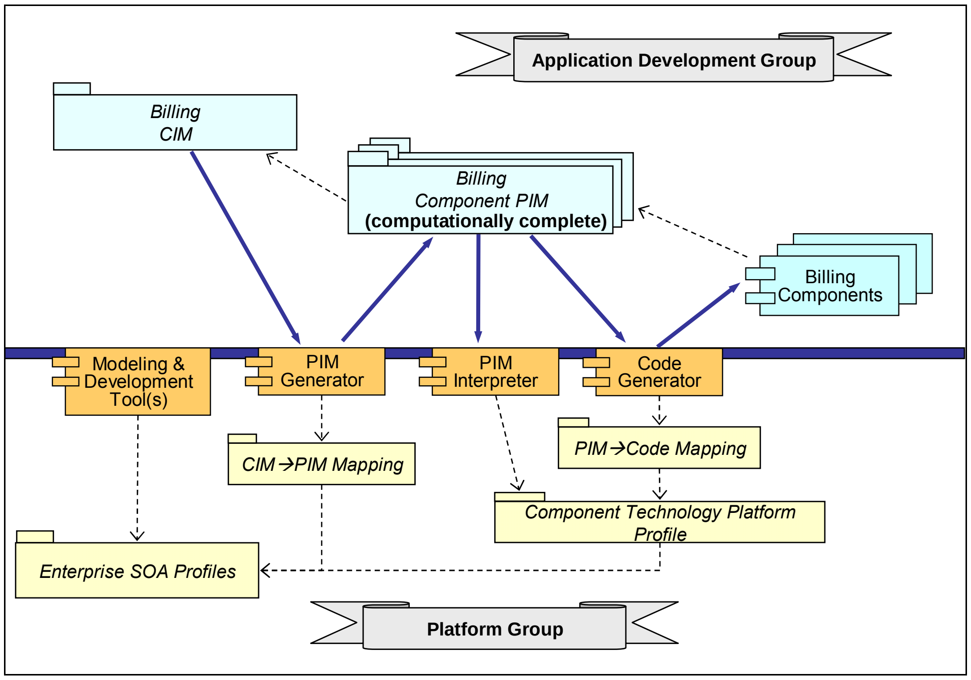

Application Development Group
Billing CIM
Billing Component PIM (computationally complete)
Billing Components
Modeling & Development Tool(s)
PIM Generator
PIM Interpreter
Code Generator
CIM→PIM Mapping
PIM→Code Mapping
Component Technology Platform Profile
Enterprise SOA Profiles
Platform Group

Figure 7

Achieving such synergy is not easy. However, a viable way ahead is available for those who choose to take it. Already today there are many organizations taking the first steps in MDA, and there are many vendors who provide various aspects of the MDA vision. Indeed, everything needed to achieve the vision is available today—just not in the same place! This isn't about being an early adopter of MDA—it's too late for that. It's about being an early adopter of the longer-term aims of MDA—with which we started this paper.

The end point of this vision is that, perhaps late in this decade, there will be a generation of application developers, whether in-house or outsourced, who never write programs in today's languages. Instead, they will design fully specified PIMs of their business logic and structure, PIMs that can either be directly executed by an architecture-aware UML virtual machine, or used as the basis for automatically generated code.

MDA has put a stake in the ground. The stake is a signpost to a most desirable future. Both IBM and Microsoft have announced their intention of getting there, albeit by different technical routes. Their chosen tools centre on IBM's Eclipse with EMF (Eclipse Modeling Framework), and Microsoft's Visual Studio with the extensions mentioned by Steve Cook [15]. Steve says that

Microsoft will not be using precisely the same MDA technologies (MOF, UML, etc), but will use variations of them. Both include the concept of DSLs, whether UML-based or not.

However the future unrolls, it is the MDA vision, as opposed to its current technologies, that points a way out of the current morass of cottage industry approaches to IT, to the broad sunlit uplands of truly effective, productive, agile, and flexible system development. Today code is king. Tomorrow design will be king. And the process of bringing innovative solutions to business challenges will become progressively simpler as more and more of the underlying software technology that today's business developers battle with becomes increasingly buried in the platform—which is the true domain of software technology experts.

5 References

[1] R. Hubert, *Convergent Architecture*, Wiley 2002.

[2] P. Herzum & O. Sims, *Business Component Factory*, Wiley 2000.

[3] P. Clements & L. Northrop, *Software Product Lines*, Addison-Wesley 2002.

[4] D. Frankel, *The MDA Marketing Message and the MDA Reality*, MDA Journal, March 2004.

[5] J. Bettin, *Model-Driven Software Development*, MDA Journal April 2004.

[6] OMG Document ptc/04-05-02 *UML 2-0 Superstructure Specification* (www.omg.org).

[7] H. Kilov, *Business Specifications*, Prentice Hall 1999.

[8] O. Sims, *A Component Model*, Cutter Executive Report Vol 5 No. 5, May 2002.

[9] S. Mellor, *Agile MDA*, MDA Journal June 2004'

[10] G. Booch et al., *An MDA Manifesto*, MDA Journal May 2004.

[11] D. Taylor, *Business Engineering with Object Technology*, Wiley 1995.

[12] O. Sims, *Business Objects*, Wiley 1994 (now out of print but available in pdf form at http://www.simsassociates.co.uk/books.htm).

[13] The COMBINE Project – See http://www.opengroup.org/combine/overview.htm. Details of the project results are not yet publicly available.

[14] S. Tyndale-Biscoe, et al., *Business Modelling for Component Systems with UML*, paper presented at the EDOC 2002 Conference, Lausanne.

[15] S. Cook, *Domain-Specific Modeling and Model Driven Architecture*, MDA Journal, January 2004.

[16] M. Guttman & J. Matthews, *Migrating to Enterprise Component Computing*, Cutter Executive Reports, 1998/9.

[17] R. Pawson & R. Matthews, *Naked Objects*, Wiley 2002.

Relating MDA and inter-enterprise collaboration management

Lea Kutvonen
Department of Computer Science, University of Helsinki
Lea.Kutvonen@cs.Helsinki.FI

Abstract

The goal of MDA (Model Driven Architecture) approach is to provide tool chains that support generation of application implementations, and interoperability of applications by ensuring that communication models can be shared by different components of distributed applications.

This paper discusses the relationship of MDA tools and components with the inter-enterprise collaboration management that has become crucial for the success of enterprises. The MDA components and tools are seen as localized, intra-enterprise elements, with structural requirements on shared abstract computing platform. That platform is expected to enable inter-enterprise business processes to be run, using the MDA provided components as participants. Essentially, the MDA tools are visioned as factories taking service descriptions and generating implementations, metainformation for local management services, and metainformation used for inter-enterprise collaboration establishment and management.

This relationship between MDA and inter-enterprise collaboration middleware induces needs for shared model and pattern repositories, and ontologies supporting queries from them. Furthermore, this relationship between MDA and process-oriented systems reserve MDA techniques on the (ODP) engineering level solutions, while (ODP) enterprise level descriptions are used as metainformation for collaboration middleware.

1 Introduction

The OMG MDA (Model Driven Architecture) [1, 2, 6] aims for tools and solutions that rise expressiveness of programming tools and provide interoperability of software components across platforms. The MDA approach uses a unified system model by taking the full application network and capturing it into a single (or composed) model, PIM (platform independent model). This model is then transformed (stepwise using several refinements and modification) into an (partial) implementation. Parts of the unifying model may be transformed using different set of transformation rules, giving a solution for a heterogeneous platform.

Looking at the emerging ICT support for inter-enterprise collaboration, the first necessary step is to the development of enterprise systems, and intra-enterprise business processes within them. For this work, MDA brings a welcome and necessary improvement. The three modeling layers – CIM, PIM and PSM – allow process-aware software components to be developed and interoperate because they are developed using the shared CIM model that represents enterprise operational needs.

However, the second step in enterprise system evolution is the adjustment to various business networks. New generation ERP systems, distributed or collaborative workflow systems, and inter-enterprise business process management systems require modeling of a "global" collaboration model within which partners have specific roles to be fulfilled by their ICT system services.

The inter-enterprise arena is not directly addressed by MDA. Still, MDA components (component used to refer to produced software components, MDA tools, transformation rules etc like in [4]) bring a significant element to the overall collaboration architecture.

The business networks can be established and managed in various ways, namely by integration, unification via shared model, or by federation. Integrated solutions are what we have seen in EAI and B2Bi solutions [13, 13]. Unified solutions trust on shared metalevel model for coordination and interoperability, like in MDA. In an inter-enterprise setting, MDA tools can be used directly, but only if the network of enterprises and their collaborative business processes can be designed together and participants are willing to replace their internal process components with new ones or are able to map new processes on top of existing services. Federated solutions require separate facilities to exist to provide an environment, a breeding environment [5], to find appropriate process models, negotiate of their use, and agree on participation on the established network together with terms and conditions of the operation.

Even in the case of federated solutions, and cases where the global business process is not used for generating executable elements, but for monitoring conformity, the actual service components need to be created with some tools. Here, MDA tools are very applicable, as the metainformation required by both facilities are of the same type.

Relevant points of design include the platform models assumed. In the following, the shared abstract computing platform for inter-enterprise business process management is briefly commented, and its effects on the structure of MDA components is discussed. Special attention needs to be placed for communication, and agreement on communication contents and context; the ODP viewpoints [8] provide a method for describing what platform elements and contractual elements need to be involved.

For the inter-Enterprise processes, choreographs between independent services are relevant. Therefore, the question arises on how MDA supports production of service implementations taking both the platform requirements and the service description (signature, behavior, NFA features) into consideration.

2 Idea(l) of shared abstract computing platform

The overall architecture model discussed here is the one used in web-Pilarcos project [12, 11, 9, 10]. In the model, a federation contract is formed to define the collaboration processes and roles between enterprise services. The B2B middleware carries responsibilities of running the partner discovery, contract management and behavior monitoring protocols. The service components are independent from each other and only required to provide the service denoted in terms of external behavior and information exchange. The autonomy of service providers is emphasized; the internal implementation or deployment aspects are strongly encapsulated.

In environments where enterprise applications become members of dynamically established inter-enterprise business networks, the following metainformation services are needed:

- identification of the intended network structure, involving the topology of the network for responsibility distribution and collaborative business process models;
- discovery of potential partners for the roles in the network;
- static verification of interoperability between communicating partners; and
- contract management (establishment, monitoring, exception management, termination).

The key element in the infrastructure is contract and contract management facilities. The business network contract (federation contract) captures the business process models involved and maps the roles presented in such a way that each participant has one and only one combined role in the network. The roles are populated using discovery service for suggestions, and by assuring the selected service offers present an interoperable network.

The essential part of the role requirement is that of provided set of services and required set of services from peers. Implementation requirements of the service may call for requirements on service from the local platform; all "side effects" of processing towards peers should be visible in the service. Some integration requirements may however be present: requirements for binding object support and requirement for the use of integrated repositories need to be set as specific service properties.

Figure 1 illustrates how these services can be seen as potentially external infrastructure services between enterprises. The requirement for each enterprise is to support interfaces for corresponding metainformation exchange protocols.

The model repositories (type repository and business process model repository) are to support static verification steps during the network population phase and during any repopulation events later in the network lifetime [17, 14]. Therefore, the MDA components need to be present in the operational infrastructure services. As the contract is phrased in platform independent terms, all participants need to be able to reflect their own solutions relationship to the abstraction. The repositories need to provide an open, incrementable set of transformation both horizontally and vertically. The existence of horizontal transformations (PIM-PIM, or PSM-PSM transformations) a) requires an underlying (implicit or explicitly stored) unifying model to exist and b) indicates an interoperability relationship to exist. The existence of vertical transformations a) support traversal of the relationship tree for analysis purposes and b) support code generation and dissemination of best practices knowledge.

The MDA transformation rules and transformation filters need to be stored into service, information representation, process model and NFA definition ontologies for runtime use. Verification of relationships is resource consuming task, and thus needs to be performed separately.

The network of relationships is built by a set of designers, filter programmers, ontology creators etc. New kind of infrastructure requires an enhanced set of new "professions" as described for example in [7]. In addition, standardization efforts providing standard collaborative processes (like RosettaNet PIPs [3] or proprietary supply chains processes) gain from a shared publication method online and thus easier adoption cycle.

For business network establishment, each enterprise provide metainformation via traders about the use of those ser-

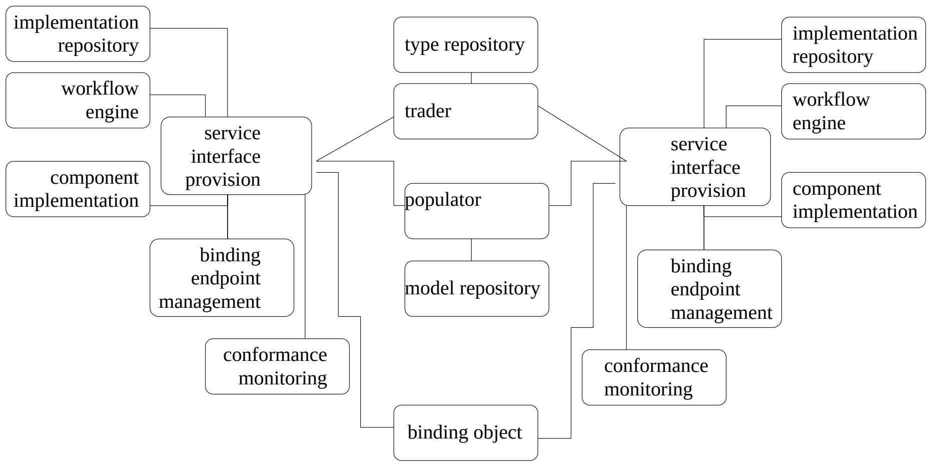


Figure 1. Architecture

vices they provide. Traders are supported by type repositories for resolution on whether two interfaces are alike, replaceable by each other, or not compatible. The populator fills in a business network with interoperable services.

For the runtime communication, the essential element in the architecture is that of distributed, open binding object. The object is constructed according to a binding contract, which declares the selected distribution transparencies, transaction choreographs, QoS agreements, and endpoint characteristics.

For the runtime verification of model conformant business process enactment, monitoring services are needed. Sensors can become standard elements of the alternative binding architectures. For the development of pervasive monitoring services, the language concepts for the monitored phenomenon need to be agreed on. This means ontologies of various aspects, like NFA features, dependent on each business process application domain.

3 Deriving requirements on PIMs from the business network environment

Within the architecture, three modeling points are of specific interest: the business network models, the external behavior models of service interfaces, and the model for the service realization within the enterprise. Here, the term realization is selected in favor of implementation, as the realization will often span a group of applications, data repositories etc.

The MDA processes to be used here would produce service realizations, starting from a set of models and producing appropriate implementation code (frameworks), and metainformation to be published in the B2B middleware repositories. The code should not include binding management, interoperability tests with peers, partner selection logic, or other elements provided by the B2B middleware. Instead, only the application logic should be present and conform to the external behavior model of its service interface type. Implementation must be parameterizable by NFA alternatives and other contract values.

The MDA road map from OMG describes MDA process with three model layers, CIM, PIM and PSM, roughly related to ODP viewpoints. CIM (computation independent model) relates to the enterprise viewpoint, PIM (platform independent model) to computational viewpoint, and PSM (platform specific model) to engineering model. The process is started from top, generating PIM models from the CIM models, with the advise of some transformation rules. Likewise, more detailed patterns advise the generation of PSM models from PIM models.

In the inter-organizational setting, the CIM model of focus describes the enterprise service internal logic that is externally visible through the provided service interface. Thus, in the MDA tool, a new CIM model needs to be verified against a published external service type. The set of enterprise business processes is more or less consistent and preplanned for efficient use of computing solutions. Modeling the processes and analyzing the processes as a set (BPA, BPR, etc) is an important goal in itself, especially combined with the view of inter-enterprise processes.

This CIM model can be further refined to PIMs. At the PIM level, also other models should appear in the enterprise model repository, namely those processes that support aspects of computing platform properties (security, trust management, QoS management, authorization, enterprise policies, service and binding factory management). These models should be prepared in such a way that aspects of behavior that can be negotiated within the inter-enterprise network can be configured either by selecting a suitable service component or by setting configuration attribute values.

For any enterprise service to be generated starting from a CIM model, a derivative PIM should be produced using selected CIM patterns. In addition, a set of PIMs that represent required computing platform properties should be joined with that business logic PIM for analysis. For code generation, the PIMs of computing platform properties should be dealt with as platform definition, giving the target concepts to be used by the implementation.

The binding elements should be provided as separate service elements. The production of these elements should go through the same kind of production process as the services within collaborative business processes.

Information or documents exchanged in the business processes are not described in all modeling techniques. However, modeling of information is an essential aspect that should indeed be modeled explicitly, and as a separate modeling issue. So, in addition to PIM models, there should be separate PII models (presentation independent information models) that can be mapped down to various representation formats. Transformations between representations of the same models could be placed as a responsibility of bindings.

When an enterprise service becomes deployed, it needs to be made available in the network. This is done by exporting appropriate service offers. For the proper establishment of dynamic business network contracts, the service offers need to capture metainformation that describes the service from several points of view, capturing the service description from ODP enterprise viewpoint and ODP computational viewpoint in respect of the actual service, and from ODP engineering viewpoint and ODP information viewpoint in respect to bindings.

4 Producing new enterprise services

To make the relationship of inter-enterprise collaboration management and MDA process more concrete, an enterprise service production process is briefly sketched. Figure 2 illustrates the process and the flow of model information in the architecture.

For the service elements two sources of model information is needed: type repository and realization models. The type repository is used for retrieving an existing service interface definition with behavioral, syntactic and NFA related information. Naturally, the MDA process can start by definition of new service interface type or subtype, and its publication to the type repository; in the publication phase, relationships to other existing models can be stated (or generated) and verified. The realization models should be available as a repository as well; most likely the repository is embedded into MDA development environment and thus may be vendor specific although free exchange of models would be ideal.

The service types can be located either directly browsing the type repository, or by browsing the business network models first. When an appropriate business network model is found, one of the roles can be chosen and service types associated to it can be picked up. The business network models can be stored for example as enhanced, abstract BPEL4WS [16] descriptions, or in a home-brew notation for ODP enterprise language. Service descriptions can be stored for example in enhanced WSDL [18].

The service types can then be organized as interfaces, and several implementation models can be selected to create the overall PIM for the service logic. Into this basic framework, several aspects PIMs can be intertwined to include for example non-functional property management (security, QoS, trust, policy-based protection of service abuse). The resulting network of communicating objects/components/subservices/workflow has to be analyzed for its viability, and code generated. The platform model and aspect models need to be available to describe the target environment onto which code is intended to run. Part of the platform model form facilities for binding management, which has to become a standardized, abstract PIM.

When the implementation has been generated and deployed, metainformation has to be provided: service offers exported to traders describing all service interface properties, binding requirements, range of nonfunctional properties that can be adapted to, etc.

5 Conclusion

This paper tries out some initial ideas on how MDA tools could be used for production of enterprise services that are autonomous but interoperable within a collaboration environment. The environment outline is that of web-Pilarcos project.

The exercise shows that the MDA process is applicable when the tools used are able to take several input models and produce several different kind of output: code and metainformation for the runtime environment.

Essential for the produced applications is that they use the abstract services of the collaborative operational environment, especially the binding facilities. Other parts of the computing platform are fairly much isolated.

The federation contract structures that are focal in the web-Pilarcos architecture capture requirements from all ODP viewpoint models. Consequently, the MDA stepwise process running from CIM to PIM and to PSM must pick up requirements from the contract structures at each step. Likewise, requirements for information contents and presentation should follow the same method.

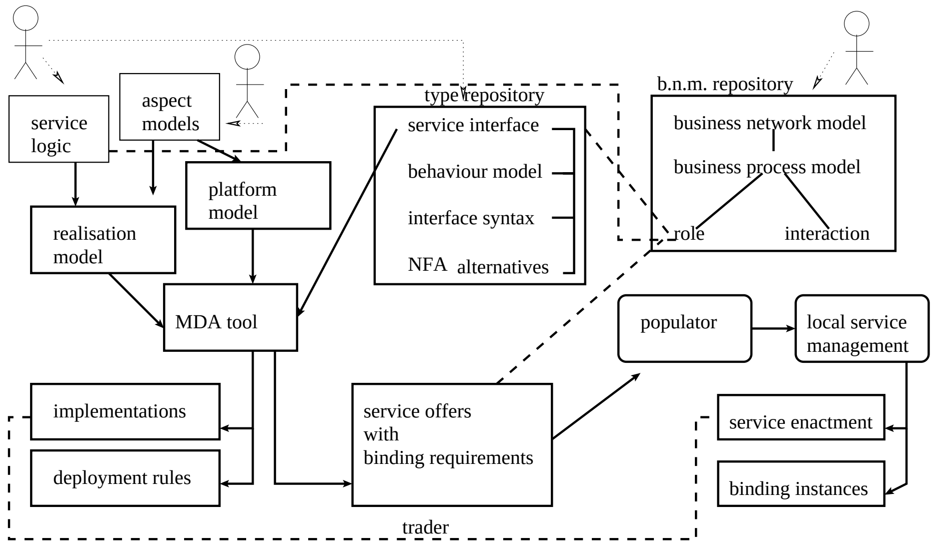


Figure 2. Flow of models.

Using the collaborative business network models as a source of requirements for the enterprise applications cause the need of identifying some commonly accepted property, policy, and behaviour alternatives. The ontologies for these should be presented within the type and model repositories of the collaboration infrastructure; this provides a method for disseminating standard ontologies. It is not plausible to develop a unified ontology for all services, but instead, it is probable that certain ontologies can have a business network model or a few business domain as their scope.

6 Acknowledgment

This workshop paper stems from work performed in the web-Pilarcos project at the Department of Computer Science at the University of Helsinki. In web-Pilarcos, active partners have been VTT, Elisa and SysOpen. The web-Pilarcos< project is a member in national ELO program (E-Business Logistics) [15].

References

[1] *Model Driven Architecture (MDA).* ormsc/01-07-01.

[2] *MDA Guide Version 1.01.*, 2003. omg/2003-06-01.

[3] Rosettanet implementation framework: Core specification v02.00.00, 2004. http://www.rosettanet.org/.

[4] J. Bezivin, S. Gerard, P.-A. Muller, and L. Rioux. Mda components: Challenges and opportunities. In *Metamodelling for MDA*, University of York, 2003.

[5] L. M. Camarnha-Matos. Infrastructure for virtual organizations – where we are. In *Proceedings of ETFA'03 - 9th international conference on Emerging Technologies and Factory Automation*, Lisboa, Portual, Sept. 2003.

[6] D. S. Frankel. *Model Driven Architecture - Applying MDA to Enterprise Computing.* OMG Press, 2003.

[7] A. Gavras, M. Belaunde, L. F. Pires, and J. P. A. Almeira. Towards an mda-based development methodology for distributed applications. In *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, 2004.

[8] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processing. Part 3: Architecture*, 1996. IS10746-3.

[9] L. Kutvonen. Controlling dynamic ecommunities: Developing federated interoperability infrastructure. In *INTEREST 2004 workshop*.

[10] L. Kutvonen. Using business network models in web-pilarcos. In *EMOI 2004 workshop*.

[11] L. Kutvonen. Automated management of interorganisational applciations. In *EDOC2002*, 2002.

[12] L. Kutvonen. B2b middleware for managing process-aware ecommunities. In *submitted manuscript*, 2004.

[13] D. S. Linthicum. *B2B Application Integration - eBusiness-Enable Your Enterprise.* 2001.

[14] T. Ruokolainen. Component interoperability. Master's thesis, University of Helsinki, Department of Computer Science. In Finnish. Manuscript to be accepted in April 2004.

[15] TEKES. *ELO program*, 2003. http://www.tekes.fi /programs/elo.

[16] S. Thatte. Business process execution language for web services, version 1.0. Technical report, July 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[17] M. Vähäaho. Arkkitehtuurikuvauksia hyödyntävä meklaus. Master's thesis, Department of Computer Science, University of Helsinki, Dec. 2002. C-2003-NN.

[18] WSDL Specification. Technical report, 2004.

MDA and Real-Time Java: The HIDOORS Project

Jean-Noël Meunier (meunier@aonix.fr)
Frank Lippert (lippert@aonix.de)
Ravi Jadhav (jadhav@aonix.com)
Nigel Harding (nharding@aonix.co.uk)

June 2004, Aonix Europe, Partridge House, Newtown Road, Henley-on-Thames, OXON RG9 1HG

Keywords:
UML, RT modelling, MDA, Profile, Automatic code generation, Java, Safety critical applications, RMA

Abstract:
Embedded Systems very often consist of a number of concurrent tasks, which have to be executed in a given time frame. Special tools are needed to analyse the schedulabity and to detect the overrun of given time targets ("Worst-Case-Execution-Time-Analysis"). Standard Java lacks some of the special features, such as deterministic garbage collection, needed for Embedded Systems.

The HIDOORS (High Integrity Distributed Object-Oriented Real-time System) project is focused on the development of a RT-Java environment and is funded by the European commission. This includes a UML based modeling environment and MDA technology, which facilitate the transformation into high integrity real time systems in Java.

Aonix developed a special RT-Java profile for this project, which is based on the OMG's SPT Profile ("Schedulability, Performance and Time") for embedded systems. This profile is the central part of HIDOORS and is used by tools such as the modeling tool, the model checker, the WCET Analyser and model transformation to the Java environment.

This paper will give a brief overview of the HIDOORS project, discuss how UML 2.0 Profiles are used to describe the required aspects of embedded systems and explain the model transformation process with an example.

1. Introduction

HIDOORS (High Integrity Distributed Object-Oriented Real-time Systems, http://www.hidoors.org) is a 30-month project consisting of European companies and research institutions and is partially funded by the European Commission (IST 2001-32329). The main goal of the project is to bring Java to applications that are hard real-time, embedded, distributed and safety critical. Additionally, the project aims to include all technologies and tools related to the development of a hard real-time application, real-time modelling, real-time analysis and proof of correctness. The project can be considered as being divided into two parts. Firstly, it relates to real-time Java Platform with an aim to solve problems such as deterministic garbage collection, real-time network support, fast RMI (Remote Method Invocation) as a means to communicate between components in a distributed environment. Secondly, it relates to real-time modeling with an intention to consider the question: how can critical and embedded real-time applications be modeled? This document focuses on the latter part and tries to at least partially answer this question.

To model an application, whatever the domain (real-time or not), it seems impossible to ignore the UML notation since it is now a well-known and recognized standard from the OMG group [1]. One of the main advantages of UML is that it is a generic notation that can address almost any domain (real-time, business, web applications, etc.). But designers often see this as an important drawback because the notation appears to be too general and too ambiguous to be used easily and efficiently for a particular domain. Fortunately, UML provides general extension mechanisms by means of stereotypes, tagged values and constraints to adapt UML to a specific domain. This is part of the UML profile definition. A UML profile describes the context of use of UML for a given domain and is defined by a subset of UML and some UML extensions. Profiles help to reduce ambiguity, reduce the complexity of models and to enrich their semantics. Models are then easier to specify, read, and process (profiles enable better automatic code generation and better model validation). That is why in most domains, a UML profile needs to be defined and used.

For the real-time domain, a profile already exists; it is the "UML Profile for Schedulability, Performance and Time" [2] (hereafter referred to as SPT) and has been adopted by the OMG group. This profile provides the basic constructs for real-time modeling. The feedback from the HIDOORS project related to this SPT profile is that:

1) The profile is too general as it covers all real-time problems both soft and hard.
2) The profile mainly defines the fundamental concepts, in other words the syntax, but it does not provide any indications concerning ways to use them, just like a dictionary of language that gives the definition of words but without any indication about how to build sentences by using these words.
3) Some concepts such as the communication means between tasks (see the next section) are missing in the profile.

For all these reasons, the HIDOORS project introduces a new profile named "HIDOORS profile", compliant with OMG's SPT and which takes into account the HIDOORS feedback and addresses distributed, critical and embedded applications.

Section 2 deals with the HIDOORS profile. It presents the objectives of the profile and the two views that the profile aims to address: the Rate Monotonic Analysis view and the task / inter-task communication view. To make the paper clearer, an example related to the communication pattern is presented. Section 3 covers the automatic code generation (a Model Driven Architecture approach) that takes as input a real-time UML model and generates as output real-time Java source code. More particularly, it shows how the code generation takes into account the HIDOORS profile concepts and maps them into Java source code.

2. The HIDOORS UML profile

The HIDOORS profile [3] aims to fulfil the following goals:

- to be compliant with the standard OMG's SPT profile
- to provide concepts that enable the specification of a RMA (Rate Monotonic Analysis) view of the model.
- to provide concepts that enable the specification of a task view (including inter-task communication) for the model.
- to provide a high level representation of asynchronous communication channels by introducing new patterns (for definition or more details on patterns, see [4] for general patterns and [5] for patterns related to real-time systems).
- to provide concepts that enable the specification of distribution concepts

Currently, concepts related to distribution have not yet been studied by the HIDOORS profile.

2.1. Rate Monotonic Analysis

There are two reasons for choosing the RMA view capability. First is the schedulability model, which is part of SPT profile and is mainly based on RMA. Secondly, for the HIDOORS project, one of the project validation applications is checked against RMA techniques.

The question related to the schedulability analysis is whether tasks can be executed such that all deadlines are met. RMA is often applied on the source code of a real-time system but performing such timing analysis at the model level enables the detection of potential specification errors earlier in the development process. If the rate monotonic analysis performed on the model concludes that the system is not schedulable, it is not worth continuing so long as the problem is not solved. However, the contrary is not true, that is if the rate monotonic analysis concludes that the system is schedulable, it does not mean that the final system will be schedulable. Still, performing this timing analysis at the model level can prevent many errors and has many benefits.

For a single processor/multiple threads system, the compliance of a model to rate monotonic analysis relies on describing the system from a concurrency point of view and as a set of scheduling jobs. Each job is composed of one trigger and one response. This description is called a real-time situation [6]. A *trigger* is principally described by an occurrence pattern (e.g. periodicity or statistical distribution) and figures as events of typical real-time modelling. A *response* is a set of sequential actions, which are principally described by their duration and the resources they need to access. They can be nested as sub-actions of an action, similar to the nested statements of a source code. A *resource* is any logical or physical item necessary to perform the action. Actually, it is not mandatory for designers to describe all resources used by the system. From the RMA point of view, the only relevant items are the shared resources that are resources, which are potentially used by several concurrent actions.

As a consequence, the goal of the HIDOORS profile is to define a set of elements and a set of rules that will allow the UML model to show scheduling jobs and consumed resources (see figure 1):

- A real-time system is a set of scheduling jobs.
- A scheduling job is made of one trigger and one response.
- A response is a set of actions.
- A trigger contains event timing information and is associated with one or more actions.
- An action contains duration information and is associated with zero or more resources. An action can also be made of sub-actions.

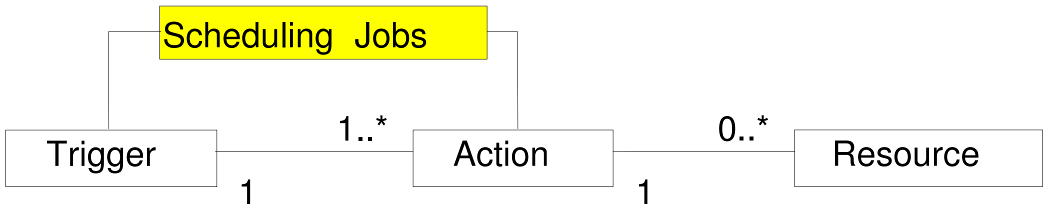


Figure 1. Scheduling jobs, triggers, actions and resources

The HIDOORS profile defines a set of elements based on the basic concepts of SPT: triggers are messages stereotyped as <<SATrigger>>, actions are messages stereotyped as <<SAAction>>, resources are objects stereotyped as <<SAResource>> (see figure 2). Thus, the HIDOORS profile gives more assistance to designers by providing these elements for re-use in models and defining which diagrams can be used for that purpose.

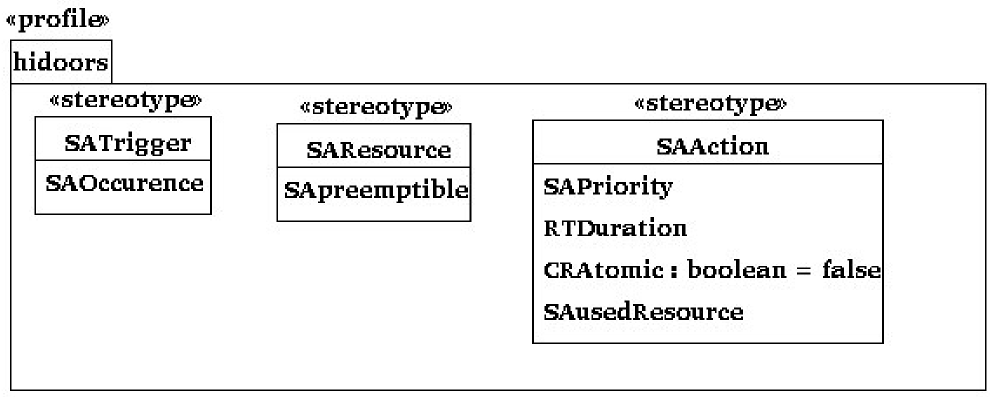


Figure 2. HIDOORS profile excerpt – triggers, actions and resources

2.2. Task view and inter-task communication

Another goal of the HIDOORS profile is to increase the level of abstraction of models thereby simplifying the effort of designers particularly when specifying asynchronous communication between tasks. New stereotypes, model elements and rules have been defined for this purpose. Three communication patterns are taken into account in accordance with the ARINC 653 standard in Avionics [7]:

- Buffer: messages are transmitted via queues with predefined capacity in FIFO order. This provides a communication channel with a "First In First Out" type of service (refer to ARINC 653 standard for more details).

- Blackboard: A message is put in a board and is either read by the receiver or overwritten by the next written message. There is no queuing of messages, but a message may be lost. This provides a communication channel with a "Last Message Only" type of service (refer to ARINC 653 standard for more details).

- Event: the event represents a simple synchronization channel (refer to ARINC 653 standard for more details) that can be used to notify another task that something happens. It works like a flag.

In the following, only the buffer communication pattern is presented, as the other communication patterns work in a similar manner.

The stereotype <<HIBuffer>> is an association between two classes representing both concurrent units (stereotyped <<HIConcurrent>>), conceptually using an instance of the class ARINCBuffer (see figure 3). It is important to understand that this template class instance is completely hidden, that it is implicit information that does not need to be specified in the model by designers and that will never appear in the generated Java source code. However, this information is useful for the automatic code generation to produce the correct source code corresponding to the communication pattern specification (see next section).

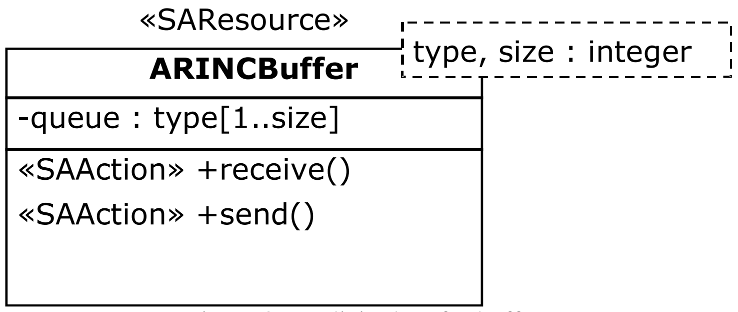


Figure 3. Implicit class for buffers

The *type* parameter of the ARINCBuffer template class must correspond to the type of the messages. This parameter can be set as a UML association class or as an association name. The *size* parameter should correspond to the maximum number of messages allowed simultaneously in the FIFO buffer. This value can be set from system specification or from simulation. The default value is infinite. This parameter can be set as a HIBufferSize UML tagged value, or within the association name (multiplicity). Figures 4 and 5 give an example of the use of this communication pattern. In the static view (figure 4), a task ("Sender") sends messages to another task ("Receiver"). The buffer size is set to 512. The message exchanged between the two tasks is of type "Message" (association class). In the dynamic view (figure 5), for a period the "Sender" sends two messages, however, the "Receiver" gets only one during that period (which means that the period of "Receiver" will have to be half that of "Sender" otherwise messages could be lost).

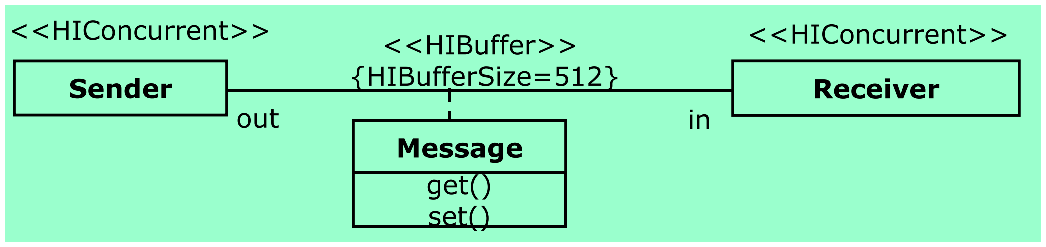


Figure 4. Example of buffer specification - static view

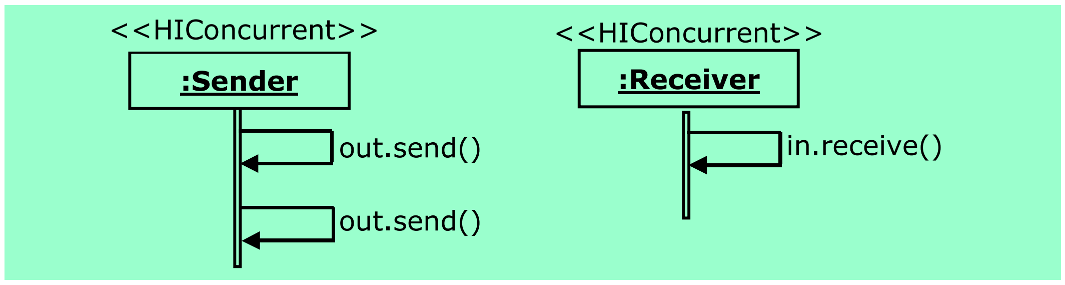


Figure 5. Example of buffer specification - dynamic view

Figure 6 gives an excerpt of the HIDOORS profile related to the three communication patterns: buffers, blackboards and events.

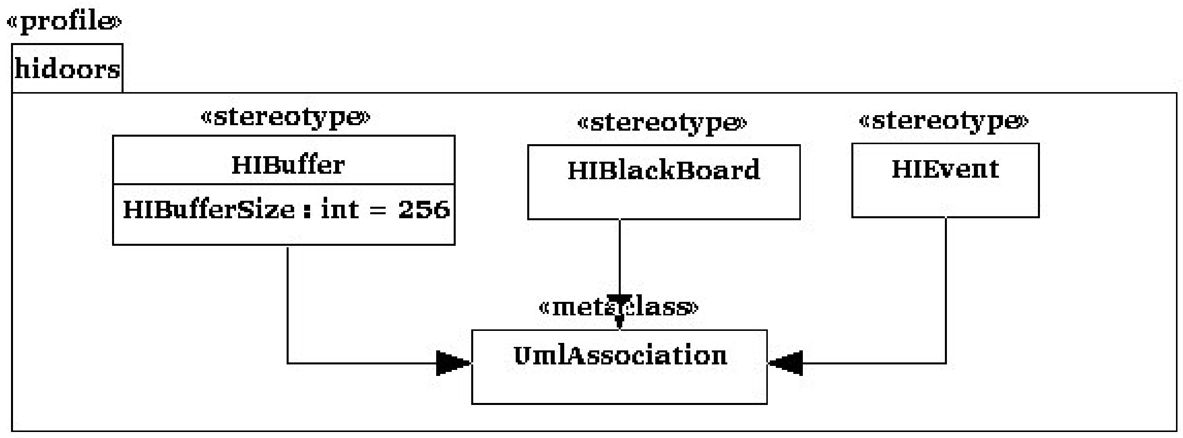


Figure 6. HIDOORS profile excerpt – the communication patterns

3. Automatic code generation

The Model Driven Architecture (MDA) approach is recommended by the OMG who recognised the need to improve software quality and to reduce development costs (see [8] for more details). The OMG place emphasis on model transformation and particularly the mapping of a Platform Independent Model (PIM) into a Platform Specific Model (PSM). The role of the automatic code generation is crucial to such an approach. In the HIDOORS project the automatic code generation consists of transforming the real-time model into real-time Java source code. The main work in the HIDOORS project is to add rules relating real-time behaviour to the general UML modeling approach (and also to translate UML concepts into Java concepts). The added value is then on the generation of source code from the HIDOORS profile constructs. More particularly, the role of automatic code generation is to break down the high level modelling and to make explicit constructs, which are implicit in the model. Figure 7 shows how the buffer communication pattern is handled. The abstract model of the example in figure 4 is mapped into a low level model taking into account the implicit information concerning the ARINCBuffer (see figure 3). Figures 8 and 9 give the resulting Java source code.

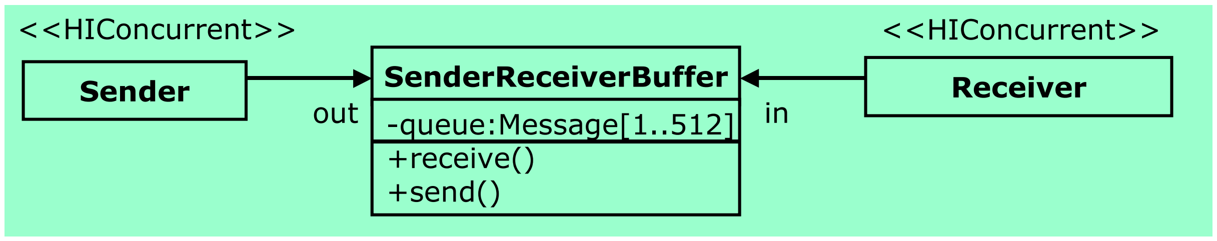


Figure 7. Part of model transformation - the example of the buffer communication pattern

```
public class Sender {

  // -------------------------------------------------------------
  // instance attributes
  // -------------------------------------------------------------
  private SenderReceiverBuffer out;

  //#ACD# M(UDAT::UID_65c15e75-0000067a-3ee5acf3-000626c6-00000004)
  //user defined code to be added here ...

  //#end ACD#
  ...
}

public class Receiver {

  // -------------------------------------------------------------
  // instance attributes
  // -------------------------------------------------------------
  private SenderReceiverBuffer in;

  //#ACD# M(UDAT::UID_65c15e75-0000067a-3ee5acfa-000aca45-0000000b)
  //user defined code to be added here ...

  //#end ACD#
  ...
}
```

Figure 8. Java source code for the Sender and Receiver class

```
public class SenderReceiverBuffer {

  // -------------------------------------------------------------
  // instance attributes
  // -------------------------------------------------------------
  /**
   * The buffer array holding the messages.
   */
  private Data[] queue = null;

  // -------------------------------------------------------------
  // methods
  // -------------------------------------------------------------
  /**
   * Obtains the next message from the message FIFO queue.
   */
  public void receive() {
    ...
  }

  /**
   * Puts a message at the last position in the message FIFO queue.
   */
  public void send() {
    ...
  }
```

Figure 9. Java source code for the SenderReceiverBuffer class

4. Conclusion

UML has a standard way to extend its semantics by stereotypes, constraints and tagged values. A collection of these is called a 'profile'. With the help of profiles, UML can be adapted to application domains for which standard UML is not specific enough.

This paper shows the development and application of a UML profile suitable for real-time modelling. The profile is largely compliant with standards and at the same time meets the specific needs of the HIDOORS project. To meet these requirements it uses a subset of the SPT real-time profile developed by the OMG and communication patterns from the ARINC 653 standard.

From an implementation point of view, the profile is implemented in a modelling tool through a profile editor as specified in the UML 2.0 standard. Also implemented is a Java code generator that makes use of the real-time profile by evaluating extensibility items applied to model elements. The modelling tool and the code generator StP (Software through Pictures), like the other tools for the HIDOORS project, are integrated into the Eclipse framework [9].

The HIDOORS profile and its corresponding automatic code generation are currently both used and tested by one of the three real-time applications aiming to validate the HIDOORS project.

One topic not covered by this paper is the analysis and validation of real-time models using a Worst Case Execution Time (WCET) tool . This work is performed by a HIDOORS partner.

5. Bibliography

[1] "OMG Unified Modelling Language Specification", Version 1.5, OMG group, March 2003, (http://www.omg.org/cgi-bin/doc?formal/03-03-01)

[2] "UML Profile for Schedulability, Performance and Time", Proposed Available Specification, OMG group, April 2003, (http://www.omg.org/cgi-bin/doc?ptc/2003-03-02)

[3] "A UML Profile for High Integrity Distributed Object-Oriented Real-time Systems (HIDOORS)", internal document, HIDOORS project, January 2003

[4] Gamma E., Heml R., Johnson R., Vlissides J., "Design Patterns", Addison-Wesley, 1995

[5] Douglass B. P., "Real-Time Design Patterns", Addison-Wesley, 2002

[6] Klein M. H., Ralya T., Pollak B., Obenza R., Gonzalez Harbour M., "A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems", Kluwer Academic Publishers, 1993

[7] ARINC specifications 653: http://www.arinc.com

[8] Model Driven Architecture (MDA) resources: http://www.omg.org/mda/

[9] The Eclipse platform website: http://www.eclipse.org

Middleware Unaware Software Development and Interoperability using MDA

Nelly Bencomo, Gordon Blair

Computing Department, Lancaster University,
Bailrigg, Lancaster, LA1 4YR, UK
nelly@acm.org, gordon@comp.lancs.ac.uk

Abstract. The main interest of this position paper is how to separate the development of distributed applications from specific middleware technologies. Unfortunately, the large number of middleware technologies conspires against this purpose – the development and maintenance of distributed systems have become coupled to the constant evolution and changes of middleware technologies. Authors back the idea that the Model Driven Architecture (MDA) gives the basis to tackle this problem. Our proposed research focuses on interoperability among applications relying on different platforms and the necessity that transformations and mapping of concepts between different PSMs should be address according the context given by the Domain Problem.

1. Introduction

The goal of middleware is to provide an integration means for diverse computing platforms. Middleware makes the development of distributed applications much easier, providing the abstractions to cope with distribution and its coordination. Currently, we have different middleware technologies such as CORBA, Java/RMI, EJB, Jini, Web Services (XML/SOAP) and .Net. All share the purpose of given the infrastructure for distributed application development by providing abstraction over the complexity and heterogeneity of the underlying distributed environment with its multitude of network technologies, machine architectures, operating systems and programming languages [6]. Unfortunately, the proliferation of middleware technologies has brought a new difficulty to distributed software development – the constant change and evolution of middleware technologies. Software Engineers are then challenged both in the area of development of new and scalable middleware systems, where open and adaptable platforms should offer richer functionality and services, and in the area of application development, where developers have to worry about constructing distributed applications that are able to evolve with the underlying middleware technologies. Our interest focuses on the latter challenge, how to manage the differences among Middleware Technologies when developing software applications, where the main idea is to study how software development can be carried out unaware of middleware concerns. The Model Driven Architecture (MDA) has been proposed as a good solution for this research problem. MDA applies the basic principle of separation of concerns by separating the specification of the system functionality from its specification on a specific platform. The former is defined as a Platform Independent Architecture (PIM), the latter as Platform Specific Model (PSM). The mapping from PIM to PSMs is performed using transformation rules. Interoperability among applications relying on different platforms can be realized by tools that not only generate PSMs, but the bridges between them. We support the idea that bridges need to focus on the context of specific Domain Problems. Our proposed research focuses on the identification of pertinent Domain Problems and the consequent definition of mappings and transformation between the abstractions of different Middleware Technology Models.

2. Middleware technologies à la carte

CORBA, Java/RMI, EJB, Jini, Web Services (XML/SOAP) and .Net. address, in general, the same problems but with different approaches. For example, the Common Object Request Broker Architecture (CORBA) is the Object Management Group's specification for achieving interoperability between distributed computing nodes. Their objective was to define an architecture that would allow heterogeneous environments to communicate at the object level regardless of who designed the two endpoints of the distributive application. A cornerstone of CORBA is its support for multiple programming languages like C, C++, Java, COBOL, Smalltalk, and Python. The CORBA standard includes mappings from IDL for each supported programming language [5]. Currently Web Services present another alternative distributed computing infrastructure; an alternative that is being strongly promoted (commercially and from the point of view of research) as preferable to the use of distributed object middleware such as Java RMI or CORBA. This new distributed computing solution exploits the openness of specific Internet technologies to address many of the interoperability issues of CORBA and other former solutions.

For years it was assumed that a clear winner would emerge and stabilize this state of flux, but the time has come to admit openly: The string of emerging contenders will never end! And, despite the advantages (sometimes real, sometimes imagined) of the latest middleware platform, migration is almost always expensive and disruptive [9]. On the other hand, companies have to preserve their software investments as the middleware landscape changes underlying it.

3. MDA: a solution

The first step in MDA is to construct a model with a high level of abstraction that is independent of any middleware technology, obtaining the Platform Independent Model (PIM). Within a PIM, the system is modeled from the viewpoint of how it best supports the business [2]. Whether the system is going to be implemented using CORBA, Java/RMI or Web Services technologies plays no role in a PIM. In the next step, the PIM is transformed into one or more Platform Specific Models (PSMs). In our specific case, the PIM is mapped (transformed) to one or more Middleware Technologies Models via OMG Standard Mappings. This transformation might be made by a MDA tool that applies a standard mapping to generate a PSM from the PIM. Depending on the tool, code production will be partially automatic, partially hand-written. Finally, each PSM is mapped (transformed) to code. Because a PSM fits its technology rather closely, this transformation is relatively straightforward [2].

4. Interoperability: mapping between the concepts

One important aspect to take into account is interoperability of applications that use different middleware technologies. This is achieved in MDA using bridges between PSMs. It is necessary to transform concepts from one platform into concepts used in another platform. The results of these transformations will be used to construct the bridges between the PSMs. If we are able to transform one PIM into two PSMs, all the information we need to bridge the gap between the two PSMs is available [2]. For each element of one PSM we know from which element in the PIM it has been transformed. From the PIM element we know what the corresponding element is in every PSM. We can therefore deduce how elements from one PSM relate to element in other PSMs. So, we have all the information we need to generate a bridge between every pair of PSMs.

Interoperability among applications relying on different platforms can be realized by tools that not only generate PSMs, but the bridges between them. The idea of the OMG is that transformation definitions should be in the public domain, perhaps even standardized and tunable to the individual needs of its users [2].

5. Proposed Research

We support the idea that bridges need to focus on the context of specific Domain Problems instead of using proposed generic PSM-to-PSM transformations [3]. We think that a standard bridge between CORBA and Web Service applications, for example, is difficult, if not impossible to develop. Bridges should address different Domain Problems, for example, Banking, E-commerce, Telecommunications, etc. PSMs must model the target platforms with sufficient precision. The use and definition of general-purpose concepts might lead to unsuccessful results due to their latent complexity. We are investigating a formal definition of Domain Problems to consequently start defining mappings and transformation between the abstractions of different Middleware Technologies.

References

1. Gray N.A.B.: Comparison of Web Services, Java-RMI, and CORBA service implementations, Fifth Australasian Workshop on Software and System Architectures, Melbourne, Australia, April, 2004
2. Kleppe A., Warmer J., Bast W.: MDA Explained The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003
3. Kovse J.:Generic Model-to-Model Transformations in MDA: Why and How?, Workshop Generative Techniques in the Context od Model Driven Architecture, OOPSLA 2002
4. Mellor S., Scott K., Uhl A., Weise D.: MDA Distilled Principles of Model-Driven Architectures, Addison-Wesley, 2004
5. Vinoski S.: It's just a mapping problem, IEE Internet Computing, pp. 88-90, May/June 2003
6. http://dsonline.computer.org/middleware/
7. http://www-106.ibm.com/developerworks/webservices/library/ws-arc3/
8. http://www.xs4all.nl/~irmen/comp/CORBA_vs_SOAP.html
9. http://www-106.ibm.com/developerworks/rational/library/403.html

Practical Model Driven Development process

Xabier Larrucea, Ana Belen García Díez, Jason Xabier Mansell

European Software Institute
Xabier.Larrucea@esi.es, anabelen.garcia@esi.es, jason.mansell@esi.es

Abstract. Nowadays many organizations are adopting MDA to describe their systems. This fact forces organizations to transform their software development process into a Model-Driven Development process. This paper proposes a software development methodology focused on MDA, and describes both the MDD process as well as the main process workflow. The UML Profile SPEM is used to describe the process. In this paper we present a MDD process and a set of System Family Engineer concepts to adapt the MDD process according to user and functional requirements. This methodology has been developed in a European IST project (MASTER project IST-2001-34600)

Introduction

Many organizations have already realized that the UML usage is becoming more and more important to define their systems. In fact this modelling language is a core concept within the MDA (Model Driven Architecture [11]) standard defined by the OMG (Object Management Group). When these organizations put into practice the MDA philosophy, they need to adopt a Model-Driven development process and the appropriated tools to support it. This paper is focused in the MDD process.

Nowadays many software development processes (SDP) like RUP (Rational Unified Process) are being applied in the industry. However these processes are not taking into account MDA concepts and they must be fit into this context. Others SDP like XP (eXtreme Programming), are also being applied. This SDP is an agile method and therefore the design phase is code-oriented whereas MDA is model-oriented. In [4] Stephen J. Mellor et al. combine the notion of “agile” and “model” and other work related with processes has already been published, such as [8] and [7].

This paper presents a methodology developed in the MASTER project, a European IST project (IST-2001-34600). In this paper we present a MDD process and a set of System Family Engineer concepts to adapt the MDD process according to user and functional requirements. The process is described in SPEM [12] (Software Process Engineer Metamodel) notation. This paper completes the work presented in [6].

This paper is structured in three main sections; Section 2 provides an overview of the MDD process; Section 3 outlines the adaptative process. Finally section 4 concludes the paper with future research action lines.

The MDD process

Many software development processes are considered as heavyweight processes. Moreover processes like RUP (Rational Unified Process) could be adaptable to Model Driven Architecture. For example, in [5] Chris Raistrick et al. have demonstrated how MDA could be applied ("Using MDA in a typical project"). However their process is a heavyweight process, it's focused in eXecutable UML(xUML) formalism and they do not take account the architectural layers. Our main process could be also considered as a heavyweight process but with some differences. Our process is based on the different architectural layers defined to describe and model a domain [3]. These layers are well-defined through the different metamodels definition.

In this section the MDD process is defined outlining the different phases with a brief description. Each phase contains a set of activities that are deeply explained in MASTER project deliverables [2]. The phases and the activities are tightly related with PIM layers definition. The basis of the architectural layers are already described in others works [10].

Figure 1 and Figure 2 provide an overall picture of the methodology proposed. Figure 1 provides an overview of the phases of the methodology whereas Figure 2 provides a more detailed overview of the MDD process workflow, describing the work products required and derived in each phase of the methodology.

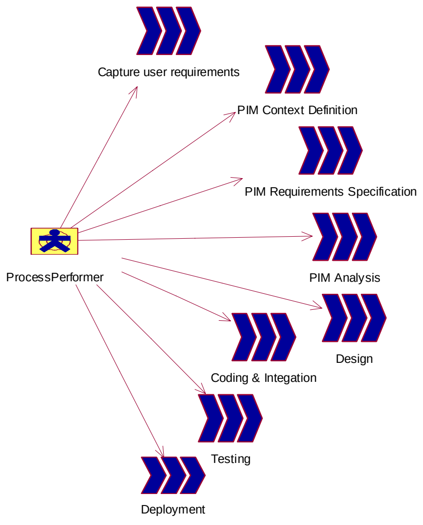


Figure 1 : Phases overview

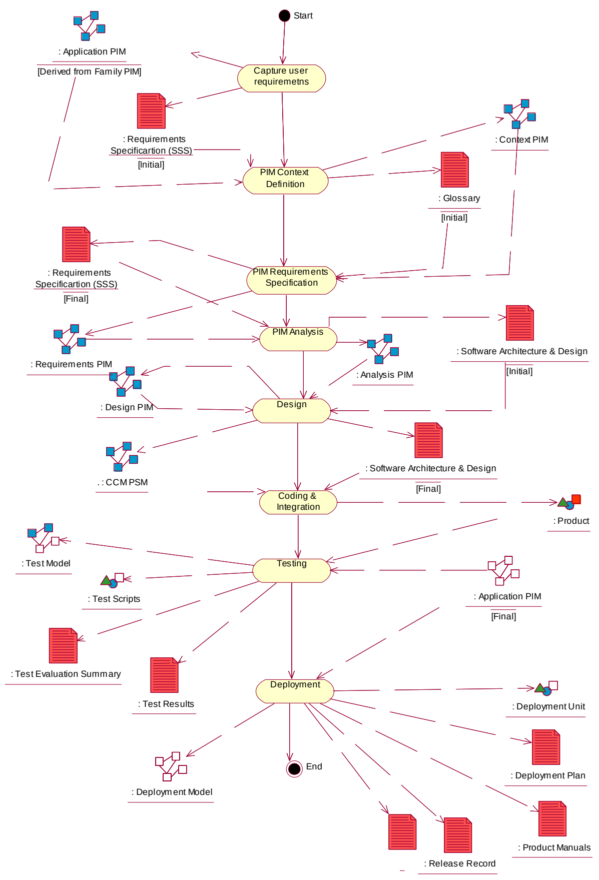


Figure 2: MDD process workflow

Figure 1 provides and overview of the phases that make up the methodology proposed. The phases are**:**

- **Capture User Requirements:** The objective of this phase is to elicit, agree and document the customer requirements that the software system needs to fulfill. This includes establishing a common understanding with the customer on functional and non-functional requirements. This phase includes the following activities: formalize the customer requirements in an Application Model and derive an initial Application PIM and an initial functional requirements specification from the common infrastructure of reusable assets.

- **PIM Context Definition:** The objective of this phase is to clearly define the scope of the software system to be developed. The result is an unambiguous definition of the system, its objectives, and scope following a black-box approach. Main activities are:
    - Establish the system goals and business principles.
    - Describe the external actors that interact with the system.
    - Identify the high-level services offered by the system and their key behaviour.
    - Define the business events, and exchanged business objects.
- **PIM Requirements Specification:** The objective of this phase is to build a model of customer requirements clear and complete and to have a unique requirements description that all subsequent models will use. In order to model the system functional and non-functional requirements, the main activities of this phase are:
    - Refine the PIM Context
    - Identify services, events and business objects produced and consumed by the system and the actors interacting with the system
    - Specify capabilities (use cases), forces (non-functional requirements), and atomic requirements
    - Identify and model the relationships between functional and non-functional requirements.
- **PIM Analysis:** The objective of this phase is to model the internal view of the system without any technological consideration and maintaining the separation of concerns between functional and non-functional aspects. The main activities of this phase are:
    - Describe the system functionalities: the objects (with classes, attributes, packages, etc.), the functions (with operations), the system boundary (with interfaces), the behaviour (with sequence diagrams), etc.
    - Describe the system QoS aspects (refine the classes) and their application to the functional elements of the model.
    - Maintain traceability with the Requirements PIM.
- **Design:** The objective of this phase is to model the detailed structure and behaviour of the solution (software application) that fulfils the system functional and non-functional requirements. This implies making decisions on how the system will be implemented and which architectural style, patterns, standards and platforms will be used. Following an MDA approach, the design is performed in two steps:

    - Specify and design a platform-independent solution (how) for all the requirements (what). The PIM will be defined with different elements depending on the architectural style selected for the solution, e.g., for a Components Design PIM the solution is expressed in terms of software components (component, interface, port, connector).
    - Specify and design the platform-specific solution by refining the platform-independent solution. The PSM is intended to be automatically derived from the PIM through transformation engines. The PSM contains models specific of the platform (e.g., CCM, EJB, .NET) and is detail and complete enough to allow the codification and deployment of the solution

- **Coding & Integration:** The objective of this phase is to develop and verify the software code that implements the software design fulfilling the software requirements. This phase includes activities such as: develop the components and classes (according to the models used as inputs), define the organization of the code, execute unit tests, and integrate components and subsystems. Following a MDA approach, the code is intended to be automatically produced from the PSM through transformation engines.
- **Testing:** The objective of this phase is to demonstrate that the final software system satisfies its requirements. This phase includes activities such as: plan tests, prepare test model, test cases and test scripts, execute tests, correct defects and document testing results. Test models are traceable to PIM models (specially to PIM Requirements) and, following an MDA approach, test models will be refined from the PIM and test cases and test scripts will be automatically produced from the test model through transformation engines.
- **Deployment:** The objective of this phase is to ensure a successful transition of the developed system to the final users (including resources, environment, schedule planning and execution). This phase includes activities such as: create a deployment plan (dates of installation, resources, etc.), create a deployment model (derived from the PSM Deployment model and adapted to the specific execution environment of the customer), create the product manuals, maintain records of the product that is being delivered to the client, and provide the installation of the product in the client premises

In this Model Driven Development process a set of roles are also described. Moreover each phase is described through a workflow diagram in SPEM notation. The purpose of this paper is not to give a deep and exhaustive description of the elements of the entire process. However these elements are described in the MASTER deliverables [1] and [2].

Adaptative Process

In the previous section a MDD process is described. This process is shown as standard software process (SSP). Many organizations adapt their SSP to their specific needs and requirements to provide software development plans. These plans have a *set of items that have common aspects and predicted variabilities* [9] (a process family). Therefore System Family Engineering concepts can be applied in this domain. The MDD process could be adapted to user requirements establishing relationships between application models (the MDD process and functional model).

Figure 3 provides an overview of the system familiy engineering process, in which based on a detailed analysis of a domain, a set of decisions can be defined which identify univocally any product of a domain. These set of decisions are captured in a decision model which captures the variability of a domain. Once this variability is solved by using the user requirements, an application model is produced. This application model captures user requirements and is used in order to inititate the derivation process by transformations in which the specific requirements are introduced within the derivation process and the application variabilities are resolved. As a result of the derivation process, in which all variabilities within a domain are solved, the application assets for a specific customer are produced.

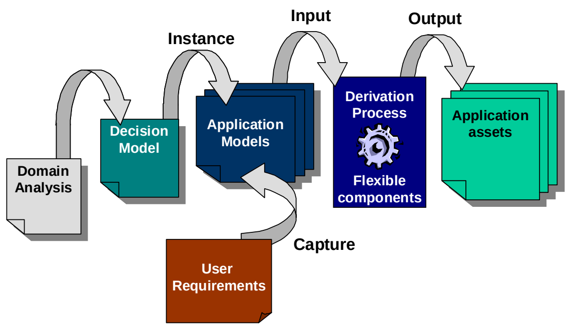


Figure 3: System Family Engineering overview

The main purpose of this paper is not describe how the MDD is produced step by step but how the MDD process described in the previous section is tailored with user needs and how some activities are removed or added depending on the requirements (derivation process). This customization process is defined through variability management, described in Figure 3. Within ESI a tool suite called V-Manage is used to define and to implement the variability of the MDD process.

Figure 4 provides an overview of how Model Driven Engineering and System Family Engineering have been used to produce a MDD adapted process.

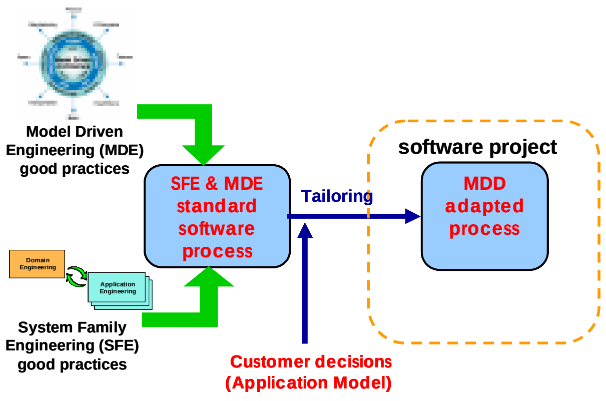

Model Driven Engineering (MDE) good practices
SFE & MDE standard software process
Tailoring
software project
MDD adapted process
Domain Engineering
Application Engineering
System Family Engineering (SFE) good practices
Customer decisions (Application Model)

Figure 4: MDD adapted process

Conclusions and future work

This work has been developed in a European IST project called MASTER project (IST-2001-34600) and it has also been applied in the context of Air Traffic Management jointly with Thales ATM.

In this paper a Model Driven Development process has been described. Some parts of this methodology like roles and work products description have been omitted to limit the size of the paper. Moreover SFE has been applied to take into account user requirements to customize the general process. However to complete the overall process an appropriate tool suite must to be provided. Actually, it would be suitable to have an IDE (Integrated Development Environment) supporting the domain analysis phase throughout the deployment phase. Many tool vendors have MDD compliant tools but do not provide support for the overall process or do not provide features such as non-functional aspects (Rational XDE Modeller) related with behavioral features.

In the context of methodology an emergent initiative related with MDA, agile modelling (AM) [13] is growing. However tool vendors must improve their tools to be able to execute models. Methodologies related with this "agile" area will be the focus of our future work.

Reference:

[1] Deliverable D3.1 "*Enriched PIM with project management information*". MASTER project: IST 34600. (http://modeldrivenarchitecture.esi.es/mda_publicDocuments.htm#D3.1)

[2] Deliverable D3.2 "*Process model to engineer and manage the MDA approach*". MASTER project: IST 34600. (http://modeldrivenarchitecture.esi.es/mda_publicDocuments.htm#D3.2)

[3] Deliverable D2.1 "*PIMs Definition and Description to model a domain*". MASTER project: IST 34600. (http://modeldrivenarchitecture.esi.es/mda_publicDocuments.htm#D2.1)

[4] *MDA Distilled. Principles of Model-Driven Architecture*. Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise. Addison-Wesley. Series Editors. Object Technology Series

[5] *Model Driven Architecture with Executable UML*. Chris Raistrick, Paul Francis, john Wright, Colin Carter, Ian Wilkie. Cambridge

[6] *Process Engineering and Project Management for the Model Driven Approach*. Ana Belen Garcia Diez, Xabier Larrucea. First European Workshop Model-Driven Architecture with Emphasis on Industrial Applications , Enschede, the Netherlands , March 17-18 2004

[7] *Application of MDA for the development of the DATOS Billing and Customer Care System (Case study on the use of MDA for the development of a larger J2EE System)*. Jorg Guther, Chris Steenbergen. First European Workshop Model-Driven Architecture with Emphasis on Industrial Applications , Enschede, the Netherlands , March 17-18 2004

[8] *Towards an MDA-based development methodology for distributed applications*. Anastasius Gavras, Mariano Belaunde, Luis Ferreira Pires, Joao Paulo A. Almeida. First European Workshop Model-Driven Architecture with Emphasis on Industrial Applications , Enschede, the Netherlands , March 17-18 2004

[9] *Software Product-line Engineering. A family based software development process*. David M.Weiss,Chi Tau Rober Lai. Addison-Wesley

[10] *PIM Definition and Description*. Daniel Exertier, Benoit Langlois, Xavier Leroux. First European Workshop Model-Driven Architecture with Emphasis on Industrial Applications , Enschede, the Netherlands , March 17-18 2004

[11] *MDA Guide v1.0.1*. Object Management Group, omg/03-06-01, June 2003

[12] *Software Process Engineering Metamodel v1.0 (SPEM)*, Object Management Group, formal/02-11-14 November 2002

[13] *Agile modeling* http://www.agilemodeling.com

New Roles in Model-Driven Development

Jan Øyvind Aagedal and Ida Solheim

SINTEF Information and Communication Technology, Forskningsvn 1, N-0314 Oslo, Norway
{jan.aagedal|ida.solheim}@sintef.no

Abstract. In this paper we outline a set of roles that are needed in model-driven development (MDD). The set of roles are based on state-of-the-art component-based methodologies, and we add new roles to accommodate the new activities of meta-modelling, transformation specification and method engineering. Finally, we list a set of tools to support the proposed roles.

1 Introduction

Model-driven development (MDD) has been advocated by academia and industry for many years. Today, most of the popular and widely used software engineering (SE) methodologies use models as the primary tool to develop software, and can thus claim to follow a model-driven approach (e.g., [1, 2]). This trend has increased as a consequence of the Model Driven Architecture initiative (MDA®) [3] launched by the Object Management Group (OMG). During its relatively short lifetime, MDA has gained a lot of attention by SE researchers, practitioners, tool vendors and others. MDA promises an integrated framework for model-driven software development. Since the Unified Modeling Language (UML™), the Meta Object Facility (MOF™) and the Common Warehouse Metamodel (CWM™) are in the core of the MDA, the *models* are the core artefacts of an MDA-based development process. An important part of the MDA vision is to equip developers with fully integrated tools to support the development of system models as well as executable code. These tools should provide synchronization of code and models, cope with different model views and abstraction levels, and provide utilities for model transformation and code generation.

General adoption of such advanced tools implies a new practise in systems development. In addition to the activities and responsibilities defined in current model-based methodologies, someone must be responsible for 1) specifying the meta-models of the chosen PIM and PSM levels, 2) defining appropriate transformations between the PIMs and the PSMs, and 3) checking the consistency between models, both on different levels of abstraction and between viewpoints on the same level of abstraction. The new responsibilities call for new roles to be included in an MDA process. In the following, we outline the responsibilities of these new roles and specify their contributions to the systems development process.

2 Additional MDD Roles

2.1 Background

From the MDA Guide [3], one gets the strong impression that OMG's current vision of model-driven architecture is still open for some interpretation. Indeed, in relevant forums, much effort is used to discuss the meaning of central concepts such as "platform", "independent", "transformation" and "architecture". Despite these sometimes philosophical discussions, practical tools and techniques are emerging which assist software developers moving towards the MDD vision. When these tools are introduced into an organisation, one soon discovers that they assume new ways of working that requires new skills in the organisation. These skills build upon, but are not similar to, existing skills that one needs in traditional model-based development. The MDD community assumes that the investments an organisation has to make in order to get these new skills are outweighed by the returns in software productivity, maintenance and flexibility.

In the MDA Guide and elsewhere, it is explicitly stated that the OMG will not propose any standard methodology or process; it will only provide standardised building blocks for making domain- or organisation-specific methodologies. In [4], the assumption is that MDA will fit with most state-of-the-art methodologies, including agile software development, extreme programming and more heavy-weight processes like the Rational Unified Process. In the EU project MODA-TEL [5], efforts have been made to define a MDD methodology, the results of which are summarised in [6]. This is a general methodology that spans the identified phases of project management, preliminary preparation, detailed preparation, infrastructure setup, and project execution. However, in this paper we focus on the additional skills that are needed in an MDD project, and that may for instance be positioned in the methodology outlined in [6].

2.2 The meta team

We have grouped a number of skills into what we call "the meta team". These skills are needed to define modelling languages, domain and platform concepts, and to customise tools. In the following we detail these skills. Note that we use the term "platform" to denote any coherent and agreed-upon set of concepts, not limited to a computing platform such as J2EE or .Net. A PIM is independent of the concepts in the platform, whereas a PSM is dependent on them. Note also that when we refer to "the PIM level" or "the PSM level", we do not indicate that there is only one such level. Indeed, we appreciate the recursive structure of "PIMness"; a PSM may be a PIM with respect to another platform.

2.2.1 The domain expert

Domain experts are necessary irrespective of how the software is developed. The domain expert is a person with detailed understanding of the application domain and who is able to abstract and categorise the required concepts and their relationships in the domain. In MDD, the domain expert should also to be able to capture this knowledge in a domain model that can be used as a baseline for the PIM meta-model. In [7], this skill is referred to as ontological meta-modelling since it focuses on the meaning of things instead of the form, which linguistic meta-modelling does.

2.2.2 The platform expert

This expertise is also necessary irrespective of software development techniques and processes. Detailed knowledge about the platforms is needed in order to produce quality software that utilises the features of the platforms. In MDD, platform experts need to be able to specify the essential platform properties in a platform model that can be used as a baseline for the PSM meta-model. Again, this is an ontological meta-model, with the subject matter being the platform.

2.2.3 The language engineer

The language engineer creates customised modelling languages suited for a purpose. This may be to identify a UML subset or to design a new domain-specific language. In any case, in MDD, the language engineer needs to use a meta-meta-modelling framework, such as the MOF from the OMG or the Ecore in Eclipse, to define the language(s) in a uniform manner if the concepts in each language are to be related in a transformation process. The language engineer performs linguistic meta-modelling, creating languages that are able to express the concepts from the platform model(s) and the domain model(s). Thus, the language engineer creates the PIM and PSM meta-models. In addition, the language engineer may create mapping languages, i.e., languages used to annotate PIMs so that they can be the source of transformations to PSMs. The language engineer needs to have expert knowledge in language design to define the abstract syntax of the languages, and needs knowledge in semiotics to create the concrete syntax of the languages, especially if they are diagrammatic. If a language is related to, or a subset of, another language (such as the UML), the language engineer also needs intimate knowledge of that language definition.

Note that we assume existing modelling languages can be used without the involvement of a language engineer. This is especially important for special-purpose modelling languages that are designed to support different kinds of model analysis. For instance, a real-time modelling language may support schedulability analysis for an organisation without the involvement of a language engineer, unless this modelling language should be tailored to specific needs.

2.2.4 The transformation specifier

From the crucial role of model transformations in MDD, it follows that the skills of the transformation specifier are extremely vital for an MDD organisation. It is the responsibility of the transformation specifier to define the relationships between PIMs and PSMs. This can be done at the model level or at the meta-model level by relating the PIM meta-model to the PSM meta-model. In any case, the transformation specifier needs to know both source and target of the transformation, and needs to know the transformation language (e.g., the language which is emerging from the QVT-Merge proposal [8]). In addition to creating the transformation, the transformation specifier also defines what should be recorded from the transformation. These records are essential to support traceability and round-trip engineering. The transformation specifier is the one to bridge the worlds of the domain expert and platform expert, and must as such understand both worlds in sufficient depth to be able to relate the concepts. It is absolutely essential that the transformation utilises the features of the platform, which may be hard to obtain without intimate knowledge of the platform. Therefore, in many cases one person will play the roles of both the transformation specifier and the platform expert.

A transformation may not only be to take one PIM and turn that into a PSM. In many cases, several models are weaved together on the PIM level and then turned into a PSM. The ability to weave together models requires insight into the different domains of the models so that consistency criteria can be defined. In the terms of IEEE 1471 [9] that is used in the MDA Guide, model weaving may be regarded as view integration. This can be done at the meta-model level by defining consistency criteria between the different meta-models, or, in IEEE 1471 terms, between the different viewpoints. It remains to be seen whether the result of the QVT process is suitable to also address the issue of model weaving and viewpoint consistency. However, the transformation specifier needs to handle this issue irrespective of whether standardised mechanisms exist.

2.2.5 The method engineer

The final skill needed in the "meta group" is that of the method engineer. The responsibility of the method engineer is to identify and orchestrate the activities needed in the MDD software development project. The method engineer needs to identify the modelling artefacts that should be produced during the project, and relate them with appropriate transformations. Furthermore, the method engineer should customise the tools to support the individual tasks in the software development. Finally, the method engineer should organise the activities into a process and possibly customise a process support tool to support the enactment of the process. In [10], the authors define the notion of MDA Component as a collection of know-how about the individual tasks in a MDD process. Using this term, the responsibility of the method engineer is to identify and organise the MDA Components available in an organisation.

2.3 The project team

The project team does the application development. They base their work on the foundations of the meta team, and applies the MDD tools and techniques in each project. Their skills do not differ substantially from a regular development team; they need to use state-of-the-art tools to solve complex problems. In the following we briefly outline the skills that are pertinent to MDD.

2.3.1 The application designer

We group all aspects of application construction under this role. Requirements capture, architectural design, detailed design, coding and testing are all activities performed by the application designer. The difference in an MDD setting is that the designer should use the modelling languages provided by the language engineer when performing their activities. Moreover, the application designer should use the transformations provided by the transformation specifier instead of performing the transformations manually as in the traditional approaches. The application designer needs to understand the transformations that are used during application construction so that the consequences of different design choices are known. The use of (semi-) automatic transformations also assumes that the application designer uses one or more marking languages to mark the PIMs to become transformable.

2.3.2 The system analyser

Again, this role is part of traditional application development. System analysis may include analysis of the system's real-time behaviour, scaleability, maintainability, etc. The distinguishing feature in MDD is that the models are the primary artifacts, not the code, so system analysis can be done at the model level instead of at the system level. This means that the system analyser needs to be able to instrument the models in order to get them analysable.

2.3.3 The system tester

The system tester is the final role that requires additional skills in an MDD approach. As opposed to the traditional approaches, the models can also be tested in an MDD approach since model transformation steps are made explicit and can be verified. Note the difference between testing the models and generating tests that can be used for testing the system. Model-based test generation is already part of state-of-the-art approaches, whereas model testing is still largely unexplored. Model simulation is a technique to support model testing. Some modelling languages have accompanying simulation tools, but this is not the case for the UML. The skills needed for model testing is largely those needed for testing in general. The difference is that one tests the models, and for this the system tester needs to interpret the testing results in terms of modelling concepts instead of as system concepts. However, most of this should be supported by tools and most good traditional testers should be able to become good model testers.

3 MDD Tools

To support the activities outlined in the previous section, a number of useful tools can be identified. Below we list and briefly characterise the tools we have identified.

- Model editor. The obvious tool in a MDD approach is a model editor that supports creation and manipulation of models. The model editor should not care whether the models are application models or meta-models, a model is a model as far as the model editor is concerned. However, the model editor should perform conformance checks so that the modeller only can produce models that are according to the relevant meta-model. Preferably, a model editor should be able to be customised to support any modelling language that the language engineer produces.
- Model repository. The models need to be stored and managed; this is the responsibility of the model repository. The model repository should support management of models in any modelling language according to the meta-meta-modelling approach that is chosen, in addition to traditional repository services such as persistence and browsing.
- Model transformers. The model transformations should be encoded in a tool so that the transformations can be as automatic as possible. Note that total automation is in many cases not achievable or desired, human intervention is often needed in each case to decide some of the issues that the transformation addresses.
- Model analysis tools. Many kinds of analysis can be performed, and existing analysis tools can in most cases be used, perhaps tailored to deal with the chosen modelling approach.
- Model simulator. Finally, a model simulator is useful for certain tests. Many modelling languages already have simulator tools that can be used, but for UML this is still not available.

4 Conclusions

In this paper we have identified and discussed the different skills needed in MDD. We have also outlined some necessary tools needed to support the roles that an MDD approach prescribes.

If a manager in a software developing organisation reads this list, some concerns may arise, especially with respect to the meta team. Most software developing organisations look at method engineering as unproductive work that preferably someone else should do for them, and they should be able to pick up an appropriate methodology from a book or a course. In MDD, this is in general not the case since one of the basic ideas is to have specific tools and techniques (hereunder languages and transformations) for each application domain. However, most software developing organisations are not alone in their problem domain, and one can foresee standardised (de facto or more formal) techniques that are useful for many kinds of software development organisations.

Large organisations, however, may want to use a proprietary language to protect their investments from being directly transferable to competitors. Such organisations may want to have all roles in the meta team filled by internal resources.

Acknowledgements. The work reported in this paper is carried out in the context of MODELWARE, an EU IP-project in FP62003/IST/2.3.2.3.

References

1. Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Object Technology Series, ed. G. Booch, I. Jacobsen, and J. Rumbaugh. 1999: Addison-Wesley. 463.
2. Atkinson, C., et al., *Component-based Product Line Engineering with UML*. Component Software Series, ed. C. Szyperski. 2002: Addison-Wesley. 506.
3. Miller, J. and J. Mukerji, eds. *MDA Guide Version 1.0.1*. 2003, Object Management Group: Needham.
4. Kleppe, A., J. Warmer, and W. Bast, *MDA Explained*. Object Technology Series, ed. G. Booch, I. Jacobson, and J. Rumbaugh. 2003: Addison-Wesley. 170.
5. MODATEL, www.modatel.org.
6. Gavras, A., et al. *Towards an MDA-based development methodology*. in *First European Workshop on Software Architecture (EWSA 2004)*. 2004. St Andrews, Scotland: Springer Verlag.
7. Atkinson, C. and T. Kühne, *Model-Driven Development: A Metamodeling Foundation*. IEEE Software, 2003. **20**(5): p. 36-41.
8. QVT-Merge Group, *Revised submission for MOF 2.0 Query/Views/Transformations RFP*. 2004, Object Management Group.
9. IEEE, *Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems*. 2000. p. 23.
10. Bézivin, J., et al. *MDA Components: Challenges and Opportunities*. in *First International Workshop on Metamodelling for MDA*. 2003. York, UK.

Memops: Data modeling and automatic code generation in multiple languages

Rasmus H. Fogh[1], Wayne Boucher[1], Wim F. Vranken[2], Anne Pajon[2], Tim J. Stevens[1], T.N. Bhat[3], John Westbrook[4], John M.C. Ionides[2] and Ernest D. Laue[1]

[1]Department of Biochemistry, University of Cambridge,
80 Tennis Court Road, Cambridge, CB2 1GA, UK
{r.h.fogh, wb104, tjs23, e.d.laue}@bioc.cam.ac.uk
[2]MSD group, EMBL-EBI, European Bioinformatics Institute,
Wellcome Trust Genome Campus, Hinxton, Cambridge, CB10 1SD, UK
{wim, pajon, jmci}@ebi.ac.uk
[3]Biotechnology Division (831), NIST,
100 Bureau Drive, Stop 8310, Gaithersburg, MD 20899-8314, USA
bhat@nist.gov
[4]Department of Chemistry and Chemical Biology, Rutgers University, Rutgers,
State University of New Jersey, 610 Taylor Road, Piscataway, NJ 08854-8087, USA
jwest@rcsb.rutgers.edu

Abstract The Memops framework is a tool for data modelling and the fully automatic generation of subroutine libraries for data access in multiple computer languages. The data model is entered in a UML subset similar to XMI. Code is generated automatically for several languages, with Python and Java being supported so far, and C/C++ and Perl support planned. The product includes an object-oriented data interaction API and its implementation, complete with data validation and checking and a notifier facility. Data storage in either XML files or relational databases is integrated in the data access subroutines. XML and database schemas and documentation is also generated from the UML model.

To achieve long-term maintainability across different platforms, Memops uses a single platform-independent model directly as the basis for code generation. Platform-specific information, which cannot be completely dispensed with, is entered in the UML model as a series of tagged values. As an example, model-specific, language-specific code is kept in the model as code snippets. These amount to less that 1 per cent of the final generated code. The approach is successful because Memops is targeted to a limited field - data modelling and data access. Memops is currently used for a data model in the structural biology field with 300 classes. A Python API (250 000 lines), and a number of applications based on it have been released.

1 Introduction

1.1 Project Goals

Memops is a product of the CCPN project [1], which was funded by the BBSRC to create a data exchange standard for the field of macromolecular NMR spectroscopy. Such a standard should allow a conforming application to modify data in a plug-and-play manner, with all modifications being kept for eventual database deposition. As might be expected in a developing scientific field, the situation facing CCPN was characterised by a substantial agreement on the kinds of data that needed to be stored, a great variety of potential uses and algorithms for exploiting the data, and the expectation of significant future changes for both. Organisationally, existing software in the field was developed by a large number of poorly resourced academic groups, each making its own choices with respect to platforms, programming languages, and data representation and storage. The resulting programs tended to be closely attuned to the needs of local users, but to have severe problems with respect to interoperability and long-term maintenance because of the lack of coordination and resources. With the rise of structural biology and high-throughput methods, however, there was an increasing need for automation, for joining different analysis programs together into software pipelines, and for large-scale harvesting and deposition of data.

1.2 MDA and Autogeneration

Model-Driven Architecture and automatic code generation seemed the only way of achieving a data exchange standard capable of being adopted and used in the field. In the absence of a mechanism for enforcing compliance, a standard could only hope to be adopted if it allowed programmers to continue working with their favourite platform. To make the changeover attractive the model must come with enough functionality in its subroutine libraries to actually make it easier to develop applications with the Memops libraries than without them. With MDA the underlying model could be precisely specified to serve as a standard, and at the same time implementations could be provided for a variety of programming languages and storage platforms. As a corollary, something very close to fully automatic code generation is indispensable to allow supporting highly functional subroutine libraries across multiple platforms with a realistic expenditure of resources. Not finding a suitable application at the start of the project, we decided to develop Memops to meet the twin requirements of simultaneous multi-platform support and 100% automatic code generation.

2 The Data Model

A data model is a description of the data for a particular subject area, how they are defined and organized, and how they relate to one another. In Memops, the data model serves as the specification for all generated code, in keeping with the Memops strategy of providing a data access layer rather than a complete application

2.1 Model organization - packages

A Memops data model is represented as a platform-independent model in UML. Memops uses a UML subset very similar to the XMI subset used for metamodel definition, with some additional tagged values. The model is generated with a standard UML editing program.

The model is subdivided in packages, which ideally should represent separate domains of knowledge and be loosely coupled to other packages. Packages serve to organize both the model description, the generated subroutine libraries, and the storage of the actual data. The purpose of this organization is to allow an application (or a data modeler) to work on part of a multidisciplinary project without having to consider either code or data for packages that are not relevant in the context. This also facilitates the production of integrated data standards for large areas of knowledge, since widely separated domains can have full control over their own packages, while sharing packages for domains that are in common.

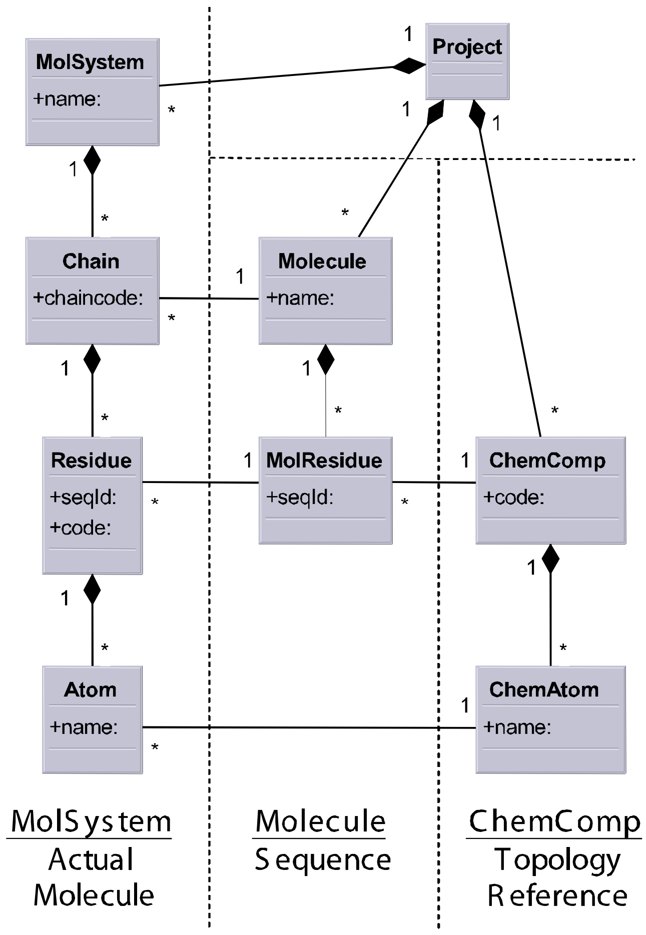


Fig. 1 A simplified part of the CCP macromolecular Data Model. Only composition ('parent') links, attributes making up the class key, and some of the more important links are shown. Dotted lines separate different model packages.

2.2 Model Organization - Relationships between Classes

There are some constraints on the allowed models to permit simple and efficient API implementations (see Fig. 1 for an illustration). All classes must have a composition association to another class, known as the 'parent' class (not to be confused with inheritance). The 'parent' links connect all data objects into a tree with a single root object. This has the dual purpose of providing a clear navigation path between any pair of objects, and of specifying a containment hierarchy for

XML storage. There is a further requirement that any class must have a set of attributes (or links) that uniquely identifies each object relative to sister objects with the same parent. If no natural key is present, an integer 'serial' must be provided. Combined with the tree of 'parent' links this provides a unique, persistent, composite identifier for each object without relying on absolute URLs or locally generated random integers, either of which may change with time. These identifiers are used to specify inter-file links between objects for XML storage.

2.3 Methods and Constraints

Class methods are mostly implicit in the model, as the methods needed for data access (see section 4.5) can be generated fully automatically once the data type and cardinalities of an attribute are known. Methods are specified explicitly if their behavior differs from the standard, or if it is desired to provide additional functionality. A case in point is derived attributes and links. These are specified to behave like normal attributes as far as the interface is concerned, but are calculated on-the-fly rather than stored; here the necessary derivation functions must be specified. When specifying a method (or a constraint) code snippets are added for the supported languages (currently Python and Java). For the future it is considered to enter code snippets in OCL, and to provide automatic translation to the supported languages [2].

Constraints may be entered on attributes, links, classes and data types, in the same way as for methods. These constraints are then evaluated either before modifying . ed/data or in a validity checking step, and serve to prevent illegal data from being entered.

3 Automatic Code Generation

As illustrated in Figure 2, subroutines for data interaction (APIs), data storage, and documentation are all generated automatically from the abstract data model. Autogeneration guarantees that all of the generated documents are synchronized, greatly simplifying the maintenance of the project. For API implementations, I/O routines, and even documentation, over 99% of the final code (or documentation) can be generated fully automatically from the data model itself. The remaining 1% is added to the model in the form of tagged values with code snippets or documentation strings, or written to a separate file as backward-compatibility I/O code. As a result there is no post-generation editing, and the generated code is ready for use immediately after generation.

3.1 The Generation Process

The automatic code generation is a two-stage process. In the first stage the information describing the model is extracted from the UML modeling tool (ObjectDomain [3]), transformed into a set of Python objects in memory, and then written to a set of files. In the second stage these files are read to recreate a set of in-memory Python objects, which then form the basis for the various generation scripts. This approach decouples the generation process from the UML modeling tool, and allows the substitution of other tools at the price of changing only a single module of the generation software.

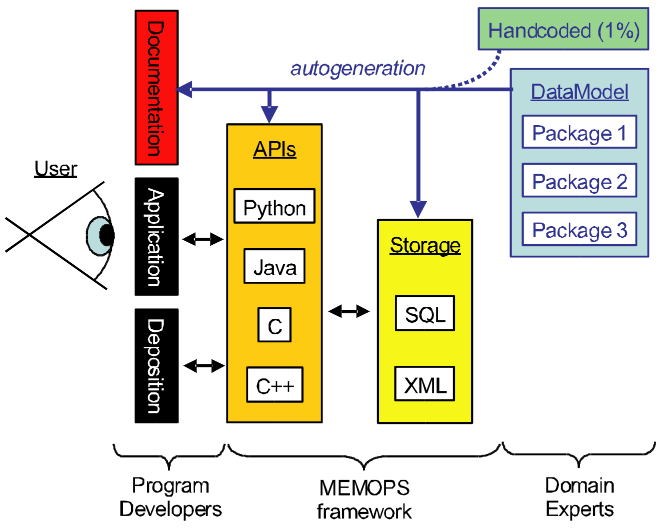


Fig. 2 Implementation of Memops code generation. Users interact with applications or deposition tools as before, while software developers use the APIs to interact with the underlying data. The actual data model is written by domain experts in a separate process with limited programming input. APIs and their implementations, storage format descriptions, I/O routines and documentation are all generated automatically from the UML data model, to the extent of over 99%. The APIs will remain stable over time even when the underlying data formats or data model change, thus insulating application programs from future changes.

3.2 Generated Libraries

Generated libraries include Python and Java API implementations, XML and SQL schemas, subroutines and mappings for I/O, and documentation. Most of these are essentially one-to-one mappings of the model. A class in the model will correspond to a Java or Python class, an XML element, or an SQL table. The same name, or an automatic derivation of it, is used throughout, to avoid the need for special mapping files. Given the nature of the platforms a one-to-one mapping is not, however, enough. XML requires extra elements for some attributes and links, relational databases require extra tables for many-to-many associations *etc.*, but in each case the extra code follows directly from the nature of the model without requiring (or allowing) extra input. There is of course an infinite number of ways of making *e.g.* Python API implementations or XML schemas that correspond to a given data model. The goal of MEMOPS is in each case to derive one useful implementation in a simple and fully automatic way, rather than to make the process customizable by the application programmer or data model developer.

4 The API implementation

The use of APIs (rather than data formats or models) as the invariant target for application programmers' efforts has a number of advantages for software integration and interoperability. APIs can be designed to be less tied to the precise detail of the underlying model than e.g. a parser would be, as they represent a higher level of abstraction. This allows the API to protect applications that use it from having to modify their code even as the data model changes. Additions to the model are especially easy to handle, since the addition of new functions to an API does not interfere with the existing ones. Changes in names, or in which data are stored and which are calculated on the fly are also relatively unproblematic, and it will frequently be possible for the API to accommodate even more fundamental changes in the structure of a data model.

4.1 General Architecture

For an application programmer the impact of using the Data Model is determined mainly by the APIs. The quality and ease of use of the API implementations is therefore extremely important. Memops API implementations are optimized for querying, for maintaining consistency in the presence of continuously changing data, and for supporting multiple projects with multiple users using different approaches and techniques. Automatic code generation in itself reduces the potential for bugs and guarantees a consistent style across the entire body of code. In addition, the APIs have been designed to include a wide range of functionality. Comprehensive validity checking is incorporated in all operations that modify data, to ensure that the data remain in a consistent and legal state. Data loading is done automatically, and the API keeps track of which data packages are modifiable, or have been modified and thus require saving.

The Memops APIs were designed as interfaces not to a specific XML file, but to a single, consistent representation of the data in a project. The prototype use case in structural biology research, where applications should be able to work directly off the generated API, accessing all relevant data, leaving the project accessible to any other conformant program, with information carried along towards an eventual deposition of the data. The emphasis on consistency checking, on persistent identifiers, and the decision not to use URL-based link mechanisms, arise from these considerations.

4.2 Notifiers

A notification facility is built into the API, to facilitate the building of graphical user interfaces (GUIs). The notifier registers a function to be called, with the relevant object as a parameter, when a given method is executed or when a given type of object is created, modified, or deleted. This can be a great simplification for GUI coding. By registering a notifier for *e.g.* creation and deletion of e.g. Molecule objects, a GUI could keep a list of all current molecules without having to change the code actually handling the molecule objects.

4.3 Storage management

The current API interacts with data stored in a mixture of XML files and local or remote databases. The price for this flexibility is that data must be loaded essentially one file at a time, which would be appropriate for situations where each project is accessed mainly by one person at a time. Data storage is by package, and each package may be stored in an XML file or database, locally or remotely. The Implementation package, which is loaded first, contains the storage locations for all other data. These are then loaded automatically by the API when the data they contain are needed. The API keeps track of which packages have been loaded and which have been modified (and should therefore be saved). Packages can also be marked as read-only, which will prevent attempts to modify the data they contain.

An alternative API implementation (currently in alpha test) provides concurrency, security and fine-grained control for simultaneous, multi-user access, transaction control, and roll-back, but this implementation depends on all the data being kept in a single database.

4.4 Derived Attributes

'Derived' attributes and roles follow the same syntax as real attributes and roles, but are in practice a convenient way of executing function calls. In a data model for person data, for instance, one could store each person separately, with links from children to their parents. A derived attribute 'mothersMaidenName' could then return the appropriate value without making it necessary to store the mother's maiden name in the model. If the model is changed so that an attribute is no longer stored explicitly, a derived attribute that mimics it can be added to avoid breaking existing code. Derived attributes and roles are especially useful since it is recommended that models be fully normalized, so that each piece of information is stored in only one place. If a piece of data is of interest in several places, derived attributes can make it available in all of them without duplication of the stored information.

4.5 Example - the Python API

The Python API consists of a Python class for each class in the model. Each class comes with a creation method (an __init__ in Python parlance), a delete method, and a checkValid method. Attributes and roles can be accessed and set using the normal Python 'object.attribute=value' syntax, but the code is organized using the Python 'properties' mechanism, so that these accesses are intercepted and passed to the relevant 'set' and 'get' methods. Access methods are generated from the model depending on the cardinality of the attribute/role. A single attribute, *e.g.* 'name', will give rise to methods 'getName' and 'setName', as will a single role. Multiple attributes will have two additional methods, so that you have *e.g.* getKeywords, setKeywords, addKeyword, and removeKeyword methods. Multiple roles will have a further three, *e.g.* findFirstAtom, findAllAtoms, and pickAtom; these methods select one or more atoms, either by filtering on their attribute and role values (the two 'find' methods) or by index (the 'pick' method).

Data are organized for fast retrieval rather than fast modification. Associations are stored at both ends, so that an employer knows his employees and an employee his employer, as it were. The API makes sure that the two ends of associations are kept consistent even if only one of them is explicitly modified, so that *employer.addEmployee(newEmployee)* and *newEmployee.setEmployer(employer)* will have the same effect. Validity checking code is built into all commands that modify attributes and roles, so that modifications that make the data illegal are prevented. Newly created objects are checked for validity after creation. The delete method works in a different way: If deleting object A makes object B invalid (*e.g.* because there was a mandatory link from B to A), object B will be deleted as well in a cascading delete.

5 Conclusions

5.1 Project Status

The Memops project has already matured sufficiently to prove that the approach works. The autogenerated Python API has been released, in the version based on XML data storage. It serves as the foundation for a couple of major scientific applications developed by CCPN, and is being interfaced with a number of other applications in the core area of CCPN, macromolecular NMR spectroscopy. The data model is being expanded into the area of (bio)chemical laboratory information management, and a Java API based on database storage is released in an alpha version. To illustrate the size of the project, the current model contains 318 classes, with 290 000 lines of code in the Python API implementation and 819 000 lines of HTML documentation.

5.2 Discussion

The decision to use a single platform-independent model as the basis for automatic code generation for several platforms has proved to work in practice, and has contributed greatly to the maintainability of projects using Memops. Of course it could be argued that the use of implementation-specific tagged values has confused the issue. The crucial factor, in our opinion, is that Memops is limited to generating data access layers, in a broad sense. This makes the problem sufficiently small and well-defined to allow the generation of efficient code from the platform-independent model with an efficiency of over 99%. It does not follow that a similar approach would be appropriate (or successful) in projects with a wider scope.

References

1. http://www.ccpn.ac.uk and references mentioned therein.
2. Akehurst D.H., and Patrascoiu, O.: Tooling Metamodels with Patterns and OCL. Proceedings of ‘Metamodelling for MDA’, York, UK, November 2003.
3. http://www.objectdomain.com

Transformations

Why IT veterans are sceptical about MDA

Graham Berrisford
Atos Origin (UK)
graham.berrisford@atosorigin.com

Abstract: This paper identifies problems with the MDA approach to specifying transformations, and barriers to MDA adoption such as the scale of the enterprise problem domain, the skills and knowledge required, and the distributed system problem.

The paper addresses question such as: What kind of detail is best suppressed from a model? What level of granularity is best for modelling business rules? How to capture all the essential business rules in a model? How to make UML more helpful to analysts looking to build a PIM or CIM? How to build enterprise-scale models?

The paper promotes the importance of understanding where and how persistent data is divided between discrete data stores and where units of work can be rolled back. It suggests models that are to be completable by business analysts yet also transformable by forward engineering must be event-oriented as well as object-oriented.

Introduction

An enterprise may own and maintain millions of lines of software code. We all know how difficult it is to read code and understand its purpose. Even the most extreme of extreme programmers agree that we need to maintain abstract specifications that are more concise than the code.

A never-ending fascination of our business lies in the immense variety of answers to three questions: What kind of abstract specification is best? How many levels of abstract specification do we need? How tightly do we maintain the abstract specifications in alignment with the code?

Many believe an abstract specification should take the form of a model. Readers of this paper will already have heard of the OMG's MDA (Model-Driven Architecture) and its three levels of model: CIM (Computation-Independent Model), PIM (Platform-Independent Model) and PSM (Platform-Specific Model). What these terms might mean is explored in the paper.

There is something to be said for an MDA scheme that is vaguely defined. Interest groups and vendors can use such a scheme as a springboard to invent new ways to be more efficient and effective in the analysis, design and construction of software systems. Even if they only reshape their existing ideas to fit the scheme, they are likely to clarify and elaborate those ideas.

There is value in the promoting and discussing MDA as a device to bring together interest groups and vendors from different realms. This encourages cross-fertilisation and new ways of thinking and working. It may help to improve UML and to reinvigorate efforts to define higher level or more universal programming languages. This last appears to be what many are focused on.

Nevertheless, the pedants amongst us hanker for more clarity in and wider agreement about the definitions of CIM and PIM. And the veterans amongst us are wary of people using MDA to recycle ideas that have not proved successful in the past. This paper sets out some reasons why some pedants and veterans are sceptical about MDA.

Two challenges facing MDA

Taking UML as a starting point, MDA is expected to meet the challenges of those wanting more abstraction and those wanting more detail. Let me quote the practical experience of a software engineer who religiously maintained an abstract specification of his software in the form UML diagrams.

"The [UML] models I produced were useful to me, but far too detailed to be helpful for talking about the design with someone else. I had to produce more simplified views for this, but I was proud of the fact that my models always represented the state of the code. Unfortunately, maintaining a very detailed model is really no easier than maintaining the code. My models tended to become rigid, even though I was working with UML diagrams. Finally, I have decided that working with too detailed a model is a trap. Basically this is the same trap that we were trying to avoid by working with UML in the first place. It's very important to work at the correct level of abstraction. I won't be going back to using forward code generation from a model."

This most earnest of software engineers concluded that higher-level or more abstract specifications are needed. So, MDA has to meet the challenge of analysts and designers who want languages that are abstract enough for their specification purposes.

At the same time MDA has to meet the challenge of analysts and designers who want languages that are semantically rich enough to specify all the necessary business rules. To code any process we need to know not just its inputs and outputs, but its preconditions and post conditions as well.

How can we build models that meet these apparently conflicting challenges (abstraction and comprehensive business rule specification) at once? Will MDA help?

On models

A model is an abstraction of the real world. A model can represent only tiny part of the real world, usually a part we want to monitor if not control. A comprehensive model has both structural and behavioural aspects; it features both persistent entities and transient events; it defines the business rules that are supposed to apply in the relevant part of the real world.

A business rule may be invariant (guaranteed to hold true at any time) or transient (guaranteed only immediately before or immediately after a discrete event). An invariant business rule may be declared as a property of a persistent entity. A transient business rule may be declared as a property of a transient event.

A running data processing system is itself model, it is an abstraction of the real world. So, a logical model of the real world model can be abstracted from the platform-specific definition of a data processing system. You may abstract logically discrete entities from physical database tables. You may abstract logically discrete events from physical database transactions or units of work.

On abstraction

Three abstraction tools can be used to raise the abstraction level of a model.

- Generalisation means creating a super type. This can save us from having to know about several subtypes. One might look to the OMG for a super type programming language, or a super type platform or operating system. (By the way, does a super-type platform help us to build a PIM? Or does it remove the PIM-PSM distinction?)
- Suppression of detail means delegating elaboration to another person or a machine. This can save analysts, designers and programmers from much tedious effort. E.g. we have long worked with platforms that automate the functions of data storage, indexing, sorting and transaction management. Suppressing the detail of transaction roll back is important in the conclusions of this paper.
- Composition means grouping details and hiding them behind the interface of a larger component. This enables people to manage large and complex systems. This has been a tenet of most if not all analysis and design methods since the 1970s.

These three devices are often combined and entangled in practical software engineering. Consider a class library whose classes offer general infrastructure operations that form or extend the platform and enable us to suppress detail from our applications. But it is useful to recognise them as three separable ideas.

On business rule specification

Business rules embrace business terms, facts, constraints and derivations. Terms are the names of things; they appear in software as the name of entities and their attributes, of events and their parameters. Facts are relationships between things; they appear in software as pointers, foreign keys and message-passing interactions. Constraints appear in software as data types, domain value ranges, relationship multiplicity constraints and validation tests. Derivations appear in software as the calculation of a data item value, the sub division of one data item, the concatenation of several data items.

Business rules of all these kinds appear in data structures and processes at every level of software engineering, from user interfaces to databases. They might be specified as preconditions and post conditions of processes at any level of software engineering, in conditions governing a step-to-step transition in a business process, and in conditions governing the commit/rollback of a database transaction.

Forward and reverse engineering transformations

A good way to explore the meaning and usefulness of the three levels of model in MDA is to consider possible CIM<->PIM<->PSM transformations, while recognising that these transformations may never be fully automated.

Automated transformations sell tools. Transformations that require human intervention are no less interesting, and they sell training courses. Of course we look for automated support wherever it is possible. But most of the transformations that I am interested in require some human intervention.

When people transform a model at one level into a model at a lower level or higher level, they often call this forward engineering or reverse engineering. Reverse engineering is a process of abstraction; it abstracts by suppression of detail, generalisation, or composition, or a combination of those three devices. Forward engineering is a process of elaboration; it elaborates by adding detail, specialising or decomposing, or a combination of those three.

For human beings, reverse engineering is infinitely easier than forward engineering. It is easier to remove detail than to add it. It is easier to generalise from than to create specialisations. It easier to group details into a composition than to detail the members of a group.

MDA brings the possibility of two reverse engineering transformations (PSM-to-PIM and PIM-to-CIM), and two forward engineering transformations (CIM-to-PIM and PIM-to-PSM). I will discuss all four, though intending to focus on transformation from the highest level of model, the CIM, to a PIM.

PSM-to-PIM reverse engineering

PSM-to-PIM transformations have been around for decades. Take a database schema, erase some of the DBMS or platform-specific details and you can express the database design as Bachman diagram. Erase some more detail and you have an entity-attribute-relationship model (aka data model).

This illustration shows that there are degrees of platform independence. Abstracting upwards from a PSM, there is not one level of PIM but many. And even at the first level of abstraction, we may choose to abstract in different directions, so there are potentially many branches as well as many levels of PIM.

Most of the people interested in MDA are interested in process structures more than data structures. (Indeed, some in the data management community are yet to be convinced the OMG or MDA are relevant to the issues of data management.)

Some are interested in abstraction by generalisation of coding languages. Where the PSM is a model of Java or C++ code, then the PIM could employ a more generic OOPL. An even higher level PIM could abstract between a generic OOPL and a procedural language like COBOL. But what MDA may deliver by way of a programming language is likely to be more complete rather than more abstract. Stephen Mellor has said:

“What UML calls a computationally complete "action language" will have at least the following features:

- complete separation of object memory access from functional computation. This allows you to re-organise data and control structure without restating the algorithms--critical for MDA
- data and control flow, as opposed to purely sequential logic. This [enables you to] distribute logic across multiple processors on a small scale (e.g. Between client and server, or into software and hardware)
- map functions and expansion regions that let you apply operations across all the elements of a collection in parallel. This … maximizes potential parallelism, again important for distribution, pipelining, and hardware/software co-design.

While not huge linguistic advances, these properties enable translation of complete executable models into any target. In my view, that is the key reason we build models of the more-than-a-picture kind."

I am less interested in turning UML into a more complete programming language than in building models that abstract by suppression of infrastructure detail. E.g. where the PSM defines all the details of transaction start, commit and rollback processes, then the PIM can be a model that is very much simpler because it includes only hooks for these platform functions.

Hmm … that last so-called PIM contained hooks for the transformation to a PSM. So it is not purely platform-independent, it posits the existence of a platform with transaction start, commit and roll back functions. I think this particular postulation is vital to the making of a CIM that can in practice be related to a PIM. I will return to this later.

Aside on true platform-independence

On the one hand, we want to build platform-independent models. On the other, we want those models to be transformable with minimal effort into software systems. The trouble is that software systems can be implemented in many ways and using many technologies. So, to ease forward engineering, people do in practice model with their chosen technology in mind.

A model that somebody claims to be a PIM may be more tied to a specific platform than the claimant recognises. When building a PIM for coding in C++, does the modeller ask: Would I draw this model the same way if we were to code in Java? And when building a PIM for coding in either C++ or Java, does the modeller ask: Would I draw this model the same way if we were to code in PL/SQL? Or VB.Net?

The meta model underlying UML looks, at its heart, to be a model of an object-oriented programming languages. If we want a truly universal modelling language, designed to model a truly platform-independent model, then the meta model might need some revision. I will come back to this later.

PIM-to-PSM forward engineering

It is possible to reverse the abstraction examples discussed above, and to employ a tool that will automate some of the forward engineering elaboration. This kind of PIM-to-PSM transformation dominates many people's view of MDA. So much so that one wag round here renamed MDA as MDCG (Model-Driven Code Generation). Allan Kennedy has, in an OMG discussion, defined MDA thus

"In a world where MDA is the dominant development paradigm, all that most developers will work with is an MDA development environment supporting executable UML as the new whizzy programming language supplemented by a number of commercially available model compilers for popular platforms.

Platform specialists and software architects will work with tools for building custom model compilers which might even be based on whatever emerges from the current QVT process. The need for the majority of developers to fill the 'gaps in their IT knowledge' will have been eliminated by the move to "platform-independent" UML as the abstraction level for specifying system behaviour."

Generation of lower-level code from a higher-level language is often presented as being an unquestionably good thing. The presenter may put down a challenge from the audience by reminding us of the luddites who wanted to continue coding in Assembler after COBOL was introduced.

But the resistance of those luddites faded in the 197Os. Several attempts were made in the 1980s and 1990s to move up from the level of COBOL. Several 'application generators' were sold on the basis that you could write in a general 'business rules' language, and from that generate either COBOL or C. Veterans bear the scars of these attempts.

(I ought to exclude here data model driven tools like Visual Data Flex that, when used with a data dictionary that captures the business rules, can be good for generating business data maintenance applications with relatively simple graphical user interfaces.)

Yes, we can forward engineer by specialisation. We could build a PIM using a generic programming language and/or assuming generic platform infrastructure, then transform this into Java or C++ or C or COBOL using platform-specific infrastructure. But it is far from obvious that the benefits outweigh the costs and risks.

Portability benefit? In practice, portability between programming languages or platforms is rarely a requirement you can clearly establish up front. Then, trying to anticipate the requirement can cost more than meeting the requirement if and when it arises. Transformation between closely related languages (e.g. dialects of SQL) costs little – probably less than efforts to anticipate the requirement. Transformation between very different languages (e.g. COBOL and Java) is, as far as I know, either impossible to anticipate or counter-productive because it denies the programmer opportunities to use the very features the language was designed to offer.

Productivity benefit? I recall an application generator salesman making a sale on the basis that the tool's 'business rules' language was up to 50 times more concise than COBOL. But actually, he compared the source code with the tool's *generated* COBOL. His business rule language was no more concise than COBOL; his generator simply wrote clumsy, long-winded code (be it COBOL or C).

It has previously turned out that the benefits promised by code generators (with the possible exception of data model driven tools when used for appropriate applications) were outweighed by the costs and risks listed below:

- forward engineering tools generate clumsy long-winded code that is less efficient, sometimes too inefficient to meet non-functional requirements.
- you become dependent on the vendor of a niche-market tool
- you become dependent on relatively scarce programmer resources, people who know the code generator’s specific language, and other product-specific features
- when things go wrong, you have to refer to the generated code anyway, meaning you remain dependent on programmers who understand that level also
- the code generator inserts invocations to infrastructure services when and where you don't want them

A colleague, while enthusiastically proposing we try a specific MDA tool/product, added the rider that “you need to be a Java, J2EE, struts, UML, MDA and product expert to properly leverage the product.” How do we find or train these people? Veterans will need a lot of convincing that mainstream projects should use a tool that requires designers to understand all that, the MOF (Meta Object Facility), and tool-specific patterns and transformations.

Veterans, listening to a presentation on MDA tools, are likely to worry about the potential costs and risks above.

A kind of forward engineering that yields real productivity benefits is based on suppression of detail. We don't want programmers having to model or write code that a general-purpose machine can do for them. So we look to automate forward engineering by getting a machine to elaborate, add detail, add generic infrastructure.

e.g. PIM-to-PSM transformers that add in the detail of platform-specific transaction management or database management functions enable us to limit our modelling effort to more business domain-specific concerns.

Having said that, IT veterans have been modelling and coding in ways that assume the support of transaction management and database management functions since about 1980. So marketing PIM-to-PSM transformation tools on the grounds that they add functions of this kind can raise something of a wry smile.

PIM-to-CIM reverse engineering

Two kinds of CIM have surfaced in OMG discussions. The first kind of CIM is a model of a business enterprise, a stand-alone CIM, independent of data processing and of potential software systems. A purely conceptual or domain model of this kind is interesting per se. It can be used to define some business rules. But forward engineering transformation is problematic. In my experience, the majority of software engineers do not find such models helpful when it comes to practical software projects. We do sometimes need to steer systems analysts away from paralysis by analysis and towards defining a CIM that is useful to software system designers.

The second kind of CIM is definitively related to one or more data processing systems. It can be transformed into software systems that consume input data and produce output data. Such a CIM may be thought of as a very abstract PIM. And given there are degrees of PIMness, there must surely be degrees of CIMness. As one person's PIM is another person's PSM, so one person's CIM may be another person's PIM.

It is always possible to abstract upwards or backwards. And again, reverse engineering is infinitely easier than forward engineering. Some people focus on abstraction by generalisation of variant forms into a common or shared form. But I propose abstraction by composition and suppression of detail will prove more profitable.

We can envisage abstracting one CIM from one PIM. This focuses the CIM on the domain and requirements of a single data processing system. We can then realistically ponder the forward engineering transformation from CIM to PIM. It might be interesting to explore this, but my concern here is to focus on the problem of the large enterprise with hundreds of distributed and loosely-coupled data processing systems.

We can construct one CIM by abstraction from many application-specific PIMs. We can use abstraction not only to reduce the number of business rule variations (by generalisation), but also to reduce the number of rules (by composition and suppression of detail). For example:

- If one application's PIM includes an EmailAddress (must include an @ sign) and another application's PIM includes TelephoneNumber (must be numeric), then we might define in a CIM a more <u>generic</u> ContactDetails item with a more generic data type.
- If one application's PIM includes an orderValue formula that calculates sales tax one way and another application's PIM includes orderValue formula that calculates sales tax another way, then we might define in a CIM a simpler orderValue calculation that <u>suppresses the detail</u> of tax calculation altogether.
- If a CustomerAddress has 3 lines in a regional application, has 5 lines in a global application, and has a structured set of attributes in another application (that uses name, town and postcode for other purposes), then we might define in a CIM a single <u>composite</u> CustomerAddress data item.

Still, we have to face two awkward questions about the enterprise-scale CIM.

First: How to resolve the million-rule problem? The large enterprise has hundreds of applications and a million business rules. We cannot maintain an enterprise CIM with that many rules. This has classically led enterprise ‘data architects’ (really, ‘data abstracters’) to define an abstract data structure containing a few generalised entities such as party, contract, place and event. They define for each entity a few attributes that appear in several applications. They may perhaps define a few business rules associated with those few attributes. Similarly, enterprise process architects have defined abstract business processes, each with a generalised sequence of business process steps such as register, authorise, process, deliver and close.

In practice, I haven’t found people making good use of such a highly abstract enterprise-scale CIM. How to separate the business rules that are somehow most essential or important from the impossibly vast multitude of necessary business rules? I don't see people successfully grappling with specifying business rules in an enterprise architecture (in the sense I mean enterprise, that is 'enterprise-scale', rather than simply 'business level') other by being highly selective, by focusing on only a tiny part of the enterprise problem domain.

Second: How to resolve the loose-coupling problem? The large enterprise works with many distributed and loosely-coupled systems in which different, perhaps conflicting, business rules apply. The enterprise's business processes have to work despite the fact that data in discrete data stores *will* be inconsistent. Surely an enterprise CIM (if it is to be useful for forward engineering into more than one PIM) must acknowledge that consistency cannot be guaranteed across all the discrete business data stores maintained by the enterprise?

Wherever the infrastructure does not exist to roll back a mistaken process across discrete data stores, then we have to design all manner of error handling and undo processing, and our models of the code have to incorporate all this design.

Where the infrastructure does exist, where we know a transaction can be automatically rolled back, we certainly don't want to model the error handling and roll back processes by hand. We can/should/must suppress the roll back details from our abstract models. Surely we can do this only by employing the corresponding abstract concept of a "unit of work" or "discrete event" in our models?

CIM-to-PIM forward engineering

I propose we cannot realistically envisage forward engineering from a purely conceptual CIM. We can however envisage forward engineering from a CIM that abstracts from data processing systems, and we can recognise this kind of CIM because it will:

- acknowledge the divisions between data in discrete loosely-coupled data stores
- define what units of work clients invoke or require on each distinct data store, with the preconditions and post conditions of each unit of work
- define what data must persist in each discrete data store for those units of work to be completable.

To put it another way: whatever paradigm you follow or platform you use, to build model that can be transformed into a data processing system, you must answer two requirements-oriented questions:

Q1) what units of work do clients invoke or require? A "unit of work" is a service. It is a process that acts on persistent data, or, if the necessary conditions are not met, it does nothing but return/output a failure message. A "client" could be a user, or a user interface, or an I/O program, or an actuator or sensor device.

Q2) what data must persist in a coherent data structure for those units of work to be completable? Every software system of note maintains some persistent data. The data structure could be anything from a handful of state variables representing the state of a few devices in a process control system, to millions of business database records representing the orders by customers for products.

In specifying the business rules of software systems, the persistent data structures and the units of work on them are fundamental. Whether your coding language is Java or PL/SQL, you will have to specify them.

Conclusions and remarks

How can MDA meet our two apparently contradictory challenges – to abstract from detail and to comprehensively specify business rules?

What kind of detail is best suppressed from a model?
A good way to keep a model simple is to postulate that a process can be rolled back automatically. This means we can ignore the design and specification of the backtracking needed when it is discovered that a processes' precondition has been violated. Indeed, it would be futile to model undo processing where we know our platform can automate the roll back of a process.

What level of granularity is best for modelling business rules?
A high-level abstract business process model can be recursively decomposed many times before we get to executable code. At which level of granularity should business analysts specify business rules, given we want these rules to be coded more or less directly from the specification?

The components and processes of a PSM must be defined down to the level of granularity dictated by our target programming language and platform.

The best level of granularity for a PIM or CIM is more open to debate. I propose we have to model the components and processes of a PIM with some minimal knowledge of the target platform's transaction management capability. We should know and declare two kinds of platform-related information in a PIM:

- the units of system composition - the discrete systems across which the chosen platform can automate roll back of a process - a discrete system has a discrete structural model and often maintains a discrete data store
- the units of work on each discrete system - the roll-backable services offered by each discrete system

People sometimes try to capture business rules by documenting the preconditions and post conditions of a use case. Often they get this wrong, because they specify for the whole use case what are rightly the preconditions and post conditions of one or more discrete back-end services. See FOOTNOTES below.

More controversially, I propose we may have to model the components and processes of a CIM with the same things in mind. We have to recognise discrete system boundaries. We have to model discrete events as well as discrete entities. We have to work on the assumption that discrete events can be automatically rolled back. At least, we have to do these things if we want the CIM to be readily transformable into a PIM. We cannot hope to do forward engineering from CIM to PIM if we cannot envisage the former as a reverse engineered abstraction of the latter.

How to capture all the essential business rules in a model?
An entity-oriented approach, defining a structural model, seems the natural way to analyse and specify invariant business rules. Some have proposed that a CIM should be defined using *only* a structural model. But if we want to model *all* the business rules, then this way lies the madness of defining every unit of work as an entity type and elaborating the model to include history of every attribute value over time.

So, an event-oriented approach, defining a behavioural model, seems the natural way to analyse and specify transient business rules. And to specify these business rules, we should define the preconditions and post conditions of processes at a specific level of granularity - the unit of work - the level where we assume roll back can and will be automated by the given platform.

(We can specify preconditions and post conditions for processes (say use cases) at a higher level of granularity than the unit of work. But most business rules belong at the unit of work level, since units of work act directly on stored business data, and one unit of work can be shared by several use cases. We can specify preconditions and post conditions for processes (say operations) at a lower level of granularity than the unit of work. In fact we can recast every condition and action within a unit of work as a precondition or post condition of a lower-level process. But let us not confuse the work of programmers, who have to work at the lowest level, with the work of analysts who must specify the business rules at the level users are conscious of.)

How to make UML more helpful to analysts looking to build a PIM or CIM in the ways indicated above?
Do we want a truly universal modelling language or method? Are we serious about building truly platform-independent models? Do we want to capture requirements in models? Do we want to help systems analysts document what matters?

I propose that units of system composition (discrete systems) and units of work (discrete events) should be first-class concepts in the UML meta model, not merely stereotypes of 'class' and 'operation'.

To define a forward engineerable CIM, we have to define the persistent data structures, the units of work on those data structures and the transient rules (preconditions and post conditions) of those units of work. Why?

- we have to capture what domain experts understand of how persistent data constrains processes and is changed by processes
- an enterprise's data is distributed, and it is vital to define which data stores the business requires to be consistent and which data stores need not be consistent
- business people should understand the effects of the units of work that they invoke from a system's user interface
- if we define invariant rules in a structural model, and postpone defining transient business rules to a lower level of design, then we are simply overlooking an important set of the business rules

- if we don't build a model on the assumption that unit of works can be automatically rolled back, then we are forced to model all manner of complexities, error handling and undo processing.

Its is difficult to teach event-oriented analysis and design techniques within the context of an OO methodology. UML does not include the unit of work. OO models contain operations at every level of granularity, and roll-backable operations are not marked out. People teach instead fuzzy concepts like the “responsibilities” of a class or component. For me, this is a limitation of the OO paradigm as currently taught. I want people to take both object and event-oriented views equally seriously - at least during the building of a CIM or a PIM.

I propose we teach people that event-oriented unit-of-work-level services are fundamental analysis and design artefacts, as important in an OO design as the entity-oriented components. We should teach that it is a good idea to identify the roll-backable units of work that clients invoke or require, and consider the effects of each unit of work on the entities in the persistent data structure. This analysis should reveal to OO designers the responsibilities of entity classes and the business rules that operations must apply to objects.

If the OMG truly wants UML to be a truly universal modelling language, then making units of work explicit in the UML meta model might help. This will make for a more complex meta model, since one unit of work may trigger operations on many entities, and the effect of one unit of work on one entity can involve more than one operation, but it will also make for a more universal meta model, one that embraces event orientation and object orientation as equals.

How to build enterprise-scale models?
We should not pretend an enterprise is a single coherent system, or is supported by a single coherent data processing system. We have to model a large enterprise as a set of discrete systems, with potentially conflicting business rules. Then we have to model each system model with much abstraction.

If we are to work at the highest possible level of abstraction, reduce the number of things to be modelled, produce the most concise specifications, then we must maximise the scope of the discrete systems we regard as units of composition, and maximise the size of the discrete events that we regard as units of work on those systems. How to maximise the size of discrete systems and discrete events is beyond the scope of this paper.

If we are to reduce the number of business rules to be modelled, then we have to further suppress detail somehow. Specifying rules using one or more derived data items is one way to hide the elementary input and/or stored data items. Some detail of a derivation rule can be suppressed by defining it using a derived data item calculated from lower-level data items. Some detail of a constraint rule can be suppressed by defining it in terms of a derived data item (e.g. and most fatuously, a "PreconditionsMet" boolean) that sums up the result of lower-level processes.

It is hard to think what abstraction devices to use beyond this, other than arbitrarily omitting business rules that we intuitively regard as unimportant, and using informal rather than formal syntax.

Finally: Is CIM-to-PIM-to-PSM a sensible basis for a software development methodology?
I fear MDA has confused in one scheme modelling the real world per se with modelling a data processing system (which is itself a model of the real world). The two are related, but nobody I know in the IT industry looks to define a PIM from a purely conceptual CIM. They define a PIM from a statement of data processing system requirements. And these requirements are better expressed in terms of use cases and input and output data structures, rather than in the form of a CIM. Inputs and outputs are an aspect of system theory that MDA seems, curiously, to have overlooked.

FOOTNOTES

On design by contract

I have no space here to discuss Bertrand Meyer's “Design by Contract”, but let me suggest that the opposite strategy of “Defensive Design” is better for multi-user database systems.

On why use cases are not enough

Extremist disciples of the distributed object paradigm and the relational database paradigm take a surprisingly similar view of their task. They both envisage building general-purpose components (be they distributed objects, web services, or databases) that sit there waiting. Waiting for clients to find them and make use of them. Waiting for requirements they can service. Waiting for their general services to be extended with additional features or specific variations.

It is a good thing in system design to anticipate future requirements a little and to generalise for future clients a little. But since we have to deliver systems to time and budget, our analysis and design method has to emphasise and prioritise the known requirements of current users. For this reason, most systems analysts nowadays define something akin to use cases during requirements analysis.

Use cases are far from object-oriented. They are procedural. They are also outward facing, user-task or HCI oriented. They define the flows of control that govern what users do at the user-system interface. Conventional use case definitions contain very little of what is needed to develop or generate code. We are likely to need also:

- the UI design, its appearance, fields and commands
- the I/O data structures (an XML schema-like grammar or regular expression notation might be useful for serial data flows)
- the state date of a user session (which may be stored on the client machine)
- any state transition constraints on the user session (state machine notations may be used here)
- the preconditions and post conditions of each *unit of work* that is invokable.

Use cases involve, or better, invoke, units of work. This isn't functional decomposition so much as client-server design. You can think of use cases and units of work as being arranged out-to-in rather than top-down.

A use case is a usage of a software system to facilitate a task in a business process - typically a one-person-one-place-one-time user-system dialogue or function – or a process to consume or produce a major data flow.

A unit of work is a process that is a success/commit unit, usually acting on a coherent data store. You might call this a "service use case". My company calls it a "business service". But I call it a "unit of work" here because this term implies roll-backableness, and that is essential to the concept I am promoting.

Other challenges facing MDA

There are many practical obstacles to successful forward engineering from one level of MDA to the next. Some are mentioned above. Other barriers to MDA (mostly suggested to me by Chris Britton) include:

- Existing systems: current tools generate UML from code - not much help really. Are there tools to reverse engineer PIMs and CIMs from legacy systems?
- System integration: how to build CIMs and PIMs for message brokers and adapters?
- Verification: how to verify models than have not yet been implemented?
- Abstraction from the distributed object paradigm: aren't user requirements essentially event-oriented rather than object-oriented? Aren't outline solution components (subsystems) really rather different from programming level components (DCOM objects, whatever)?
- Non-functional requirements: these constrain the results of a PIM-to-PSM transformation
- Primitive data types: how to define basic or generic constraints on data item values in a CIM or PIM?

On reuse

Use cases and units of work are not products of object-oriented analysis; they are products of event-oriented analysis. And after identifying what use cases and units of work are needed, an important task in detailed design is to optimise reuse.

Every discrete software system can be defined as a process hierarchy (though hierarchy here means a network rather than a strict hierarchy, since a lower-level process can be part of several higher ones). e.g. A use case may involve zero, one or more units of work, and a unit of work may be reused in several use cases.

You can also factor out common processes at one level. You might find two use cases share a common process, or two units of work share a common process. Sometimes, that common process is wanted on its own in another context, so it can be defined as discrete use case or unit of work. The lowest level common processes in units of work are operations on the lowest level encapsulated entities.

There is a formal event-oriented technique for defining reuse between units of work. In this technique, the unit of work is called a discrete "event" which has an “effect” on each of one or more “entities”. Two discrete events can share a common process, known as a "super event". The OO concept of a responsibility is akin to an effect, or more interestingly, to a super event.

In short: You identify events. You identify where two or more events have same preconditions and postconditions wrt an entity (that is, the several events appear at the same point in the entity's state machine and have the same effect). You name the shared effect as a super event. You analyse to see if the super event goes on from that entity (where the events’ access paths come together) to have a shared effect on one or more other entities, and if so, you adopt the super event name in specifying those other entities’ state machines.

I don't mean to promote this specific “super event” analysis and design technique. I am only wanting to indicate that event-oriented analysis and design has a respectable and successful history, since many OO designers are unaware of this history.

By the way, you may be able to generalise two similar units of work into one, but you have to define the two distinct requirements before you can know this is possible.

On practical experience of using UML as the programming level

“On some previous projects, I went full-scale for using Rose to produce detailed models and generate code from them. When I needed to extend the design, I would always return to the Rose model and make the change there, forward generate the code, and then fill in the details. I worked for several years this way and advocated for it strongly among my peers, although few took me up on the approach, at least in part because of the steepness of the learning curve for Rose with code generation in C++.

The models I produced were useful to me, but far too detailed to be helpful for talking about the design with someone else. I had to produce more simplified views for this, but I was proud of the fact that my models always represented the state of the code.

Unfortunately, maintaining a very detailed model is really no easier than maintaining the code. My models tended to become rigid, even though I was working with UML diagrams. Finally, I have decided that working with too detailed a model is a trap. Basically this is the same trap that we were trying to avoid by working with UML in the first place.

It's very important to work at the correct level of abstraction.

In my current project, we have taken the simple design and refactoring approach. We develop in three week iterations. We draw the designs we need for each iteration on a whiteboard. When we all understand them well enough, we code them. We refactor continuously, driven by code smells and design considerations. At the end of an iteration, we reverse engineer our code (using Rose) and produce simplified views that we use to inform our designs on the whiteboard in the next iteration.

This seems to have worked very well, and I feel our design is quite good. Partly, this represents that I have more experience as a designer.

However, I also see that this way of working keeps us focused on the right level of detail at the right time. In other words, a white board can be a more effective tool than Rational Rose for working out a high level design, and refactoring using the code can be a very effective tool for improving a design. The reverse engineering works well enough and the model is always up to date. We have a team of three that has been working for 15 months in this way, and the design and code have not become rigid. I won't be going back to using forward code generation from a model.”

What do we do with re-use in MDA?

Nathalie Moreno and Antonio Vallecillo

Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
`{vergara,av}@lcc.uma.es`

Abstract. MDA seems to be one of the most promising approaches for designing and developing software applications. It provides the right kinds of abstractions and mechanisms for improving the way applications are built nowadays: in MDA, software development becomes model transformation. MDA also seems to suggest a top-down development process, whereby PIMs are progressively transformed into PSMs until a final system implementation (PSM) is reached. However, there are situations in which a bottom-up approach is also required, e.g., when *re-use* is required. Here, re-use means for instance using pre-developed COTS components to build applications, or dealing with legacy systems. Moreover, many times we are not interested in the creation of new systems but in the maintenance or evolution of existing ones. How to deal with these issues within the context of MDA? How much benefit will MDA bring to those problems? In this paper we try to introduce the main problems involved in dealing with re-use in MDA, identify the major issues, and propose some ways to address them, particularly in the context of Component-based Software Development.

1 Introduction

The Model Driven Architecture (MDA) [16, 19] is an OMG initiative that provides an approach to system development based on models. It is model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification of systems. It provides an approach for specifying a system independently of the platform that will support it (Platform Independent Model, PIM); specifying platforms (Platform Models, PM); choosing one or more particular platforms for the system; and transforming the PIM into one (or more) Platform Specific Models (or PSM)—one for each particular platform.

Platform independence is the quality of a model to be independent of the features of a platform of any particular type. This aims at separating the business logic and rules from the technology and middleware platform(s) on which the system is implemented, protecting part of the organization investment in software development from changes in the fast-pace evolving software technologies.

Model transformation is the process of converting one model to another model of the same system. In MDA, software development becomes an iterative model transformation process: each step transforms one (or more) PIM of the system at one level into one (or more) PSM at the next level, until a final system implementation is reached. (Here, an implementation is just another PSM, which provides all the information needed to construct a system and to put it into operation.)

This process seems to imply a top-down development process, by which models at different levels of abstraction of the system are progressively transformed (merged and/or refined) until the implementation code is finally generated. However, there are situations in which a bottom-up approach is also required. For instance, how to use and integrate pre-developed COTS components into the application? How to deal with pieces of legacy code, or with legacy applications? Furthermore, many times we are not interested in the creation of new systems but in the maintenance or evolution of existing ones. How to deal with these *re-use* issues within the context of MDA? How much benefit will MDA bring to those problems? For MDA to become mainstream, the current re-use issue has to be properly addressed.

In this paper we try to introduce the problems involved in dealing with re-use within the MDA, identify the major issues, and propose some ways to address them in the particular context of Component-based Software Development (CBSD). More precisely, the structure of this document is as follows. After this introduction, Section 2 describes the major problems that we think that need to be addressed in order to deal with re-use within MDA. Then, Section 3 presents a solution based on a set of assumptions. The feasibility of such assumptions is discussed in Section 4, which analyzes how far we currently are to overcome the problems that each assumption faces. Finally, Section 5 draws some conclusions and outlines some future lines of research.

2 The problems

There are many different problems that need to be addressed in order to deal with re-use in the context of MDA. There are general problems of system modeling and of MDA, and specific problems of COTS components and legacy systems.

2.1 Problems related to the modeling of systems and MDA

The first kind of problems are general to all system modeling approaches, and in particular to MDA: What kind of information should the model of a software system contain? How do we express such information? (Not to speak about the methodology or approach to derive the model from the user's requirements.)

First, there seems to be no consensus about the information that comprises the model of a system, a component, or a service. In this paper we will suppose that this information contains three main parts: the *structure*, the *behavior*, and the *choreography* [20]. The first one describes the major classes or components types representing services in the system, their attributes, the signature of their operations, and the relationships between them. Usually, UML class or component diagrams capture such architectural information. The *behavior* specifies the precise behavior of every object or component, usually in terms of state machines, action semantics, or by the specification of the pre- and post-conditions of their operations (see [14] for a comprehensive discussion of the different approaches for behavior modeling). Finally, the *choreography* defines the valid sequences of messages and interactions that the different objects and components of the system may exchange. Notations like sequence and interaction diagrams, languages like BPEL4WS, or formal notations like Petri Nets or the π-calculus may describe such kind of information.

Most system architects and modelers currently use UML (class or component diagrams) for describing the structural parts of the system model. However, there is no consensus on the notation to use for modeling behavior and choreography. This is something that somehow needs to be resolved.

2.2 Problems related to COTS and legacy systems

The second set of problems is related to the COTS components or legacy systems that we need to integrate in our system. The kind of information that is available from them will allow us to check whether they match our requirements or not, as described by the system model. More precisely, this information should be able to allow us to:

(a) model the component or legacy system (e.g., by describing its structure, behavior, and choreography);
(b) check whether it matches the system requirements (this is also known as the *gap analysis* problem [8]);
(c) evaluate the changes and adaptation effort required to make it match the system requirements (i.e., evaluate the *distance* between the models of the "required" and the "actual" services, see e.g., [15]); and
(d) ideally, provide the specification of an adaptor that resolves these possible mismatches and differences (see e.g., [5, 6]).

Figure 1 shows these processes in a graphical way.

The problem is that both COTS components and legacy applications are usually back-box pieces of software for which there is no documentation or modeling information at all. Even worse, if a model of a component or legacy system exists, it may correspond to the original design but not to the actual piece of software. The current separation between the model of the system and its final implementation usually leads to situations in which changes and evolutions of the code do not reflect in the documentation—same as it happens in a building that gets refurbished but nobody cares to update the floor and electricity maps.

Some people propose the use of reverse engineering to obtain the information we require about legacy systems (basically, obtain their models from their code, whenever the code is available). Thus, a reverse transformation would convert the code of the legacy application into a fairly high-level model with a defined interface that can be used to perform all the previous tasks.

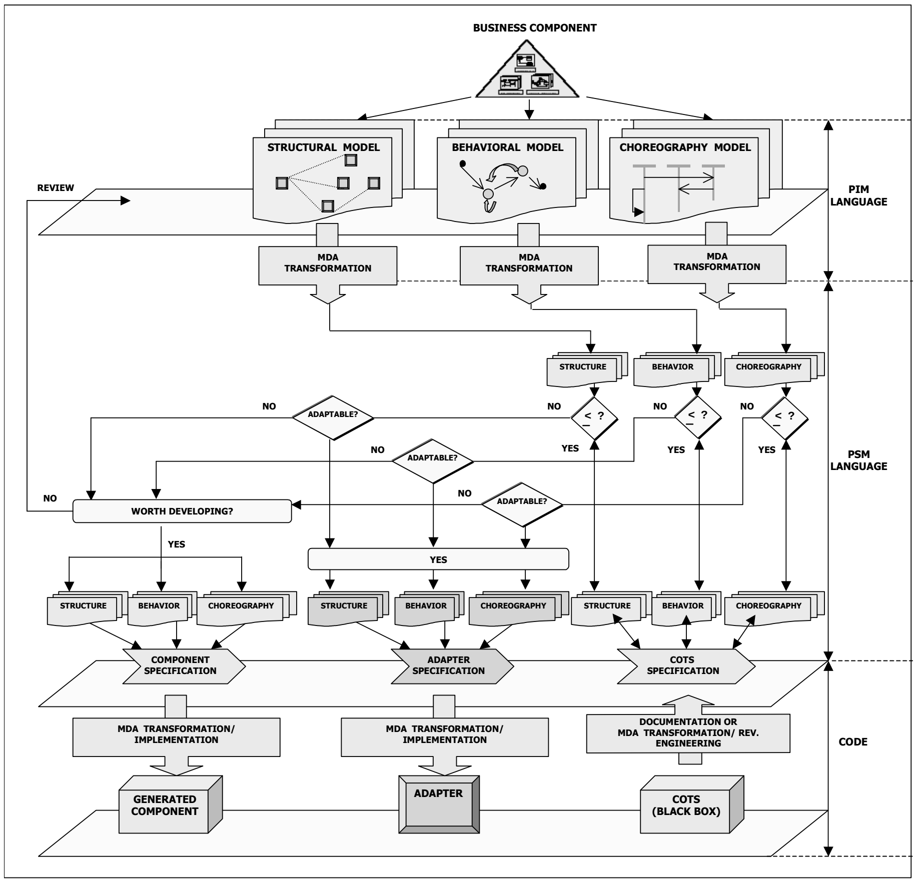


Fig. 1. Integrating COTS into the MDA chain

But the problem is that reverse engineering can only provide a model at the lowest possible level of abstraction. In fact, you can't reverse engineer an architecture of any value out of something that did not have an architecture to begin with. And even if the original system was created with a sound architecture, very often the original architecture tends to get eroded during the development process. So, what you usually get after reverse engineering is essentially just an execution model of the actual software in graphical form. At that point, most of the high level design decisions have been wiped out.

Our proposal is then to model just the interfaces to legacy systems and leave them as code—not to reverse engineer their contents. In this way we can deal with them as if they were COTS components, whose internals are unaccessible.

With regard to adaptation, an old rule of thumb claims that if more than 80% of the functionality of a component needs to be modified in order to be

Table 1. Examples of software elements and available notations for expressing their structural, behavioral, and choreography models.

Software element	Structure	Behavior	Choreography
Web Service	WSDL	RDF	BPEL4WS [9]
CORBA object	CORBA IDL	SDL [11]	Message Sequence Charts [10]
CORBA object	CORBA IDL	Larch-CORBA [12]	CORBA-Roles [7] or Petri-nets [2]
Java Class	Java	JML [13]	UML seq. diagrams
.NET assembly	C#	contracts [1]	BPEL4WS [9]

integrated into our system, it is faster (and cheaper) to develop it from scratch. In other words, if a legacy component can be wrapped and then successfully deployed with a "minor" repair/upgrade effort, then this is a reasonable approach. If any more than a "minor" effort is needed to make it match our high-level system requirements (as stated by the system PIM), then it is a strong candidate for forward engineering—of course using the existing legacy component as conceptual input.

Summarizing, the main problems related to the re-use of COTS components and legacy systems that we perceive are: the definition of the information (set of models) that needs to be provided/obtained for a piece of software in order to understand its functionality, and how to re-use it; the evaluation of the effort required to adapt it to match the new system's requirements; and the (semi)automatic generation of adapters that iron out the mismatches.

3 Assumptions for addressing these problems

This Section discusses how to address some of the issues mentioned above, making certain assumptions.

(1) First, we will suppose that we count with a model of the COTS component or legacy system that we need to re-use, with the information about its structure, behavior, and choreography. Table 1 shows some examples of software elements and the notations in which the information can be expressed. These models will constitute our target PSM.
(2) We will also suppose that the PIM of the application we are developing describes the system as a set of interacting parts, each one with the information about its structure, behavior, and choreography. (This information can be either individually modeled, or obtained for each element from the global PIM—by using projections, for example.)
(3) Third, we will assume that there are MDA transformations defined between the metamodels of the notations used in the PIM for describing the system structure, behavior and choreography, and those used in the PSM. For instance, MDA transformations from Message Sequence Charts to BPEL4WS.
(4) Fourth, we will suppose that associated to each notation for describing structure, behavior and choreography at the PSM level, there are a set of matchmaking operators that will implement the substitutability tests. These tests

are required to check whether the required business component (as specified in the PIM, and then "translated" into the PSM) can be safely *substituted* by the existing component or piece of legacy software. For simplicity, we will use the name notation ($\leq$) for referring to all these operators.

For example, at the structural level given two interfaces A and B, we shall say that $A \leq B$ if A can replace B, i.e., if A is a subtype of B using the common subtyping relations for interface signatures [21]. At the behavioural level, this operator can be defined to deal with the behavioral semantics of components, following the usual subtyping relations for pre-post conditions [22], for instance. Operator $\leq$ can also be defined for choreography models expressed using process algebras [6, 7, 17].

(5) An finally, we need to count on the existence of (semi) automated derivation of software adaptors (e.g., wrappers) that resolve the potential mismatches found by the substitutability tests.

Using all these assumptions, our approach is graphically depicted in Figure 1. As we can see, our starting point is the PIM of a business service or component. This PIM comes from the PIM of the global system, that we suppose composed of individual business services or components interacting together to achieve the system functionality. The PIM of each business service comprises (at least) three models with its structure, behavior, and choreography.

At the right hand side of the bottom of the Figure 1 we have the piece of software that we want to re-use, let it be a COTS component or a legacy application (for example, think of an external Web Service that offers the financial services we are interested in, or a COTS component that provides part of the functionality that our business requires). From its available information and/or code we need to extract its high-level models, that will the constitute the PSM of the software element. This PSM will be constructed using the information available from the COTS component or legacy application, and probably complemented with some information obtained using reverse engineering. The *Platform* in this case will be the one in which we express the information available about the element. Let us call P to that platform, and let M_s, M_b and M_c the models of the structure, behavior and choreography of the software element to be re-used, respectively.

Once we count with a PIM of the business service (our requirements) and the PSM of the available software in a platform P, we need to compare them, and check whether the PSM can serve as an implementation of the PIM in that platform. In order to implement such a comparison, both models need to be expressed in the same platform. Therefore, we will transform the three models of the PIM into three models in P, using MDA transformations. Let they be M'_s, M'_b and M'_c, respectively.

Once they are expressed in the same platform and in compatible languages, we can make use of the appropriate reemplazability operators and tools defined for those languages to check that the software element fulfils our requirements, i.e., $M_s \leq M'_s$, $M_b \leq M'_b$, and $M_c \leq M'_c$. If so, it is just a matter to use the PSM software element as a valid transformation from the PIM to that platform.

But in case the software element cannot fulfil our requirements (i.e. its PSM cannot safely replace the PSM obtained by transforming the PIM), we need to evaluate whether we can adapt it, and if so, how much is the effort involved in that adaptation. Some recent works are showing interesting results in this area [5, 6, 15]. The idea is, given the specifications of two software elements, obtain the specification of an adaptor that resolves its differences. If such an adaptor is feasible (and affordable!) we can use some MDA transformations to get its implementation from the three models of its PSM. Otherwise, it is better to forward-engineering the component, using MDA standard techniques from the original business component's PIM (left hand side of Figure 1).

Alternatively, the original PIM of the system might have to be revisited in case there is a strong requirement of using the software element, which does not allow us to develop it from scratch (e.g. in the cases of a financial service offered by an external provider such as VISA or AMEX, or of a Web Service that implements a typical service from Amazon or Adobe). In those cases, we must accommodate the software design and architecture of our system to the existing products, maybe using spiral development methods such as those described in [18].

4 Dealing with the assumptions

We have presented an approach to deal with COTS components and legacy code within the context of MDA, based on a set of assumptions. In this Section we will discuss how far we currently are from achieving these assumptions, and the work that needs to be carried out for making them become true. The assumptions were introduced in Section 3.

The first one had to do with obtaining the PSM of the piece of software to be re-used, and in particular the models of its structure, behavior and choreography. Examples of such models were presented in Table 1. Some of this information is not difficult to obtain, specially at the structure level: the signature of the interfaces of the software elements are commonly available (e.g. WSDL descriptions of Web Services, IDLs of CORBA and COM components, etc.). However, the situation at the other two levels is not so bright, and only for Web Services we think that it will resolve in a near future—this information is definitely required if re-use is to be achieved, and we perceive a clear support from software developers and vendors to re-use Web Services. Proposals for describing the choreography and behavioral semantics of Web Services are starting to be developed, and we expect to see them widely agreed soon. For the rest of the COTS components there are some small advances (see, e.g., the work by Bertrand Meyer on extracting contract information from .NET components [1]) but most of the required information will probably never be supplied [3], unless a real software marketplace for them does ever materialize.

Regarding legacy systems, the use of reverse engineering may be of great help, although it also presents strong limitations, as we previously discussed in Section 2. Basically, the result after reverse engineering is essentially just an execution model of the actual software in graphical form, without most of the high level design decisions and architecture.

The second assumption was about counting with a PIM of the individual business services that form part of the system, with information about their structure, behavior, and choreography. Again, there seems to be no major problems with the structure, but we see how the software engineering community currently struggles to deal with the modeling of behavior and choreography of business components and services (a quick look at the discussion happening at the MDA, Business Processes, and WS-Choreography mailing lists is very illustrative).

Although there is no agreed notation for modeling behavior (or even consensus on a common behavioral model), we expect UML 2.0 to bring some consensus here: even when UML 2.0 proposed behavioral and choreography models are far from being perfect, we expect the "U" of UML to do its job. However, this also strongly depends on the availability of tools to support the forthcoming UML 2.0 standard.

The third assumption relied on the availability of MDA transformations between the metamodels of the notations used in the PIM for describing the system structure, behavior and choreography, and those used in the PSM. We expect MOF/QVT to be of great help here. In fact, there are some proposals already available that provide transformations between different languages, such as UML (Class diagrams) to Java (interfaces), EDOC to BPEL4WS, etc. [4]. They are still at a fairly low level, but they are very promising when considered from the MOF/QVT perspective.

We also supposed, as fourth assumption, the existence of formal operations ($\leq$) and tools for checking the substitutability of two specifications. The situation is easy at the structure level, since this implies just common subtyping of interfaces. However, there is much work to be done at the behavior or choreography levels, for which only a limited set of operators and tools exist (basically, the works by Gary Leavens on Larch [13, 12], and the works by Carlos Canal et al. for choreography [6, 7]).

Finally, there is also plenty of work to do with regard to the (semi) automated derivation of software adaptors (e.g., wrappers) that resolve the potential mismatches found by the substitutability tests. There are some initial results only, but most of the problems seem to be unsolved yet: defining distances between specifications [15], deciding about the potential existence of a wrapper that resolves the mismatches, generating the wrappers at the different levels, etc.

5 Concluding Remarks

In this position paper we have discussed the issues associated to re-use within the context of MDA. We have also presented a proposal, based on a set of assumptions. The problem is that these assumptions are not feasible yet. However, their identification has helped us detect some areas of research that may help solve the problems associated to re-use, or at least alleviate it in some particular contexts, e.g., CBSD or Web Engineering.

The general problem of re-use is much more complex, though. We have over-simplified it by just concentrating on three aspects of the systems: structure, behavior and choreography. These models allow the specification and implementation of most of the "operationalizable" properties of systems: basically, its functionality and some QoS and security requirements. However, how to deal with the extra-functional requirements (e.g. robustness, stability, usability, demonstrability, maintainability,etc.)? Many of these requirements are more important than functionality when it comes to reuse or upgrade an existing system. As usual, we will leave them for future research.

Acknowledgements. This work has been partially supported by Spanish Project TIC2002- 04309-C02-02.

References

1. K. Arnout and B. Meyer. Finding implicit contracts in .NET components. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects (First International Symposium, FMCO 2002)*, number 2852 in Lecture Notes in Computer Science, pages 285–318, Leiden, The Netherlands, 2003. Springer-Verlag, Heildelberg. http://www.inf.ethz.ch/ meyer/publications/extraction/extraction.pdf.
2. R. Bastide, O. Sy, and P. Palanque. Formal specification and prototyping of CORBA systems. In *Proceedings of ECOOP'99*, number 1628 in Lecture Notes in Computer Science, pages 474–494, Lisbon, Portugal, 14–18 June 1999. Springer-Verlag, Heildelberg.
3. M. F. Bertoa, J. M. Troya, and A. Vallecillo. A survey on the quality information provided by software component vendors. In *Proc. of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2003)*, pages 25–30, Darmstadt, Germany, 21 July 2003.
4. J. Bézivin, S. Hammoudi, D. Lopes, and F. Jouault. An experiment in mapping web services to implementation platforms. Reserach Report 04.01, University of Nantes, Mar. 2004.
5. A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software, Special Issue on Automated Component-Based Software Engineering (in press)*, 2004.
6. A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web services choreographies. In *Proc. of the 1st International Workshop on Web Services and Formal Methods (WS-FM'04)*, volume 86 of *Electronic Notes in Theoretical Computer Science*, pages 1–20, Pisa, Italy, Sept. 2004. Elsevier.
7. C. Canal, L. Fuentes, E. Pimentel, J. M. Troya, and A. Vallecillo. Adding roles to CORBA objects. *IEEE Trans. Softw. Eng.*, 29(3):242–260, Mar. 2003.

8. J. Cheesman and J. Daniels. *UML Components. A simple process for specifying component-based software.* Addison-Wesley, Boston, 2000.

9. IBM. *Business Process Execution Language for Web services (BPEL4WS) 1.1.* IBM and Microsoft, May 2003. Available at http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

10. ITU-T. *Message Sequence Charts.* International Telecommunications Union, Geneva, Switzerland, 1994. ITU-T Recommendation Z.120.

11. ITU-T. *SDL: Specification and Description Language.* International Telecommunications Union, Geneva, Switzerland, 1994. ITU-T Recommendation Z.100.

12. G. T. Leavens. Larch-corba. http://www.cs.iastate.edu/ leavens/main.htmlLarchCORBA.

13. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999. JML web page: http://www.cs.iastate.edu/ leavens/JML.html.

14. A. McNeile and N. Simons. Methods of behaviour modelling, Apr. 2004. http://www.metamaxim.com/download/documents/Methods.pdf.

15. R. Mili, J. Desharnais, M. Frappier, and A. Mili. Semantic distance between specifications. *Theoretical Comput. Sci.*, 247:257–276, Sept. 2000.

16. J. Miller and J. Mukerji. *MDA Guide.* Object Management Group, Jan. 2003. OMG document ab/2003-05-01.

17. O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

18. B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(3):115–117, Mar. 2001.

19. OMG. *Model Driven Architecture. A Technical Perspective.* Object Management Group, Jan. 2001. OMG document ab/2001-01-01.

20. A. Vallecillo, J. Hernández, and J. M. Troya. New issues in object interoperability. In *Object-Oriented Technology: ECOOP 2000 Workshop Reader*, number 1964 in Lecture Notes in Computer Science, chapter 21, pages 256–269. Springer-Verlag, Heidelberg, 2000.

21. A. M. Zaremski and J. M. Wing. Signature matching: A tool for using software libraries. *ACM Trans. on Software Engineering and Methodology*, 4(2):146–170, Apr. 1995.

22. A. M. Zaremski and J. M. Wing. Specification matching of software components. *ACM Trans. on Software Engineering and Methodology*, 6(4):333–369, Oct. 1997.

Supporting Model Reusability with Pattern-based Composition Units

Andrey Nechypurenko
Siemens Corporate Technology
Munich, Germany
andrey.nechypurenko@siemens.com

Douglas C. Schmidt
Vanderbilt University,
Nashville, TN, USA
d.schmidt@vanderbilt.edu

Abstract

The growing complexity and criticality of distributed systems motivates software developers to raise the level of abstraction used to develop these systems. A promising approach for improving the quality and productivity of software development is to (1) assemble applications from higher-level building blocks that represent solution templates for certain application domains and (2) apply model-driven development techniques and tools to manipulate the building blocks and automate key tasks related to system specification, implementation, configuration, and deployment, rather than (re)writing the applications manually using third-generation programming languages. To simplify the manipulation of component building blocks, however, requires a well-formed set of rules and relationships. This paper contributes to the study of these topics by describing pattern inheritance relationships, showing how pattern inheritance can improve the reusability of models, and illustrating our approach with a concrete example.

Keywords

Model-Driven Development (MDD), patterns, inheritance.

1. Introduction

Emerging trends. The growth in the size and complexity of large-scale distributed systems is exceeding the ability of IT professionals and organizations to develop software for these systems with acceptable and affordable time and effort. To address this problem requires new technologies that enable developers to improve the productivity and quality of the software development process. A promising approach involves the combination of (1) *component middleware* [21], which provide mechanisms to configure and control key distributed computing aspects (such as connecting event sources to event sinks and managing transactional behavior) separately from the functional aspects of applications, with (2) *model driven development (MDD)* [1][22], which is a generative technology that helps reduce complexity by raising the level of abstraction at which software is developed.

The technical foundations of component middleware consist of various patterns and frameworks that have been covered extensively in earlier publications [4][5][10][14][17]. The technical foundations of MDD are less well codified, but the emerging consensus [11][22] is that the MDD paradigm involves (1) *metamodeling*, which define type systems that precisely express key characteristics and constraints associated with particular application domains, such as e-commerce, telecommunications, and automotive control, (2) *domain-specific languages*, which provide programming notations that formalize the process of specifying business logic and quality of service (QoS)-related requirements, and (3) *model transformations and code generators*, which help to automate and assure the consistency of software implementations using analysis information associated with functional and QoS specifications captured by models. Although there are various approaches [1][20] to realizing the MDD paradigm, MDD tools and techniques share a common goal of reducing complexity by raising the level of abstraction used to specify, implement, configure, and deploy software systems.

Unresolved problems.Despite improvements in third-generation programming languages (such as Java or C++) and run-time platforms (such as component middleware), the levels of abstraction at which application logic is typically integrated with the set of rules and behavior dictated by component models remains low relative to the (1) concepts and concerns within the application domains themselves and (2) advanced technologies available in the solution space described below:

- **Gap between domain and implementation abstraction levels**. A large gap exists in the levels of abstraction between (1) mainstream programming languages used by software engineers versus (2) the domain-specific terminology used by systems engineers to describe applications that are being built. The conventional solution is to apply a design process (such as object-oriented design or structured design) to map from the higher-level domain-specific abstractions to the much lower-level abstraction provided by mainstream third-generation programming languages. This mapping has historically been performed manually by conventional software development methodologies, such as RUP [25], which introduces various problems, ranging from simple implementation errors to missing customer requirements [22].
- **Gap between state-of-the-art and state-of-the-practice**. Another gap exists between the levels of abstraction and composition that represent (1) the state-of-the-art in software engineering R&D versus (2) the state-of-the-practice applied by most developers. In particular, mainstream third-generation languages do

not intuitively reflect the concepts used by cutting-edge software researchers and developers [9], who increasingly express their system architectures and designs using languages and tools that support higher level concerns, such as persistence, remoting, and synchronization.

Both these gaps can be narrowed by introducing intermediate abstraction layers, where the distance between problem domain abstractions and available solution domain abstractions is much smaller. As discussed [22], this approach motivates the development of generative MDD technologies that create families of domain-specific languages (DSLs). These DSLs can then be applied to express domain-specific problems more effectively and intuitively than general-purpose programming languages, thereby enhancing software productivity and quality.

Despite the promising benefits of MDD, other unresolved problems remain due to the fact that models of distributed systems can themselves be large and complex as applications grow in size and scope. In particular, it is hard to change and maintain models using conventional Model-Driven Architecture (MDA) techniques [1][20], which provide only a slightly higher level of abstraction and platform-independence than third-generation programming languages, such as C++ or Java.

Solution approach ⇨ Compose software systems from higher-level building blocks that are solution templates for certain problems. In previous work [2][3] we motivated the need for higher-level MDD abstractions that combine patterns, component middleware, and aspect-oriented software development (AOSD) techniques [13] to

- Resolve recurring distributed system development problems so they have fewer dependencies on platform-specific details, such as communication protocols, object models, and threading models, and
- Automate key system evolution tasks, such as implementing new customer requirements, refactoring certain parts of the system, and migrating to the newer versions (or versions from other vendors) of libraries and middleware used for development.

Our previous work, however, does not show how the pattern-based composition of different aspects and models could be implemented in component-based systems. This paper therefore explores another point in the solution space: illustrating a new design and problem decomposition approach that applies patterns for modeling different aspects of distributed systems to simplify model transformations and code generators for component-based systems. In particular, we investigate the relationships between patterns that can improve their substitutability and composability, thereby contributing to methodologies that can be applied to manipulate role-based solution templates as first class system composition units. We introduce the concept of *pattern feature inheritance relationships* and use a concrete example to illustrate the benefits gained from using the substitutability property of feature inheritance. It is our position that formalizing sets of composition and manipulation rules will enable greater automation of key modeling and code generation concerns that are hard to automate with conventional MDD technologies.

Paper organization. The remainder of this paper is organized as following: Section 2 describes how inheritance relationships between patterns help to support variability without degrading software symmetry [26][27]; Section 3 examines a concrete example that illustrates the applicability of concepts presented in Section 2 to solve the problems outlined in Section 1; Section 4 compares our approach with related work; and Section 5 presents concluding remarks and outlines future work.

2. Pattern Inheritance as a Key Mechanism to Encapsulate Variability and Improve Reusability

To manage software development effort and enhance software productivity and quality, the IT industry is continually trying to improve reusability and localize the impact of variability found in product families [8]. The paradigms developed over the past 3-4 decades range from functional decomposition to object-oriented decomposition and recently aspect-oriented decomposition [6][13]. Each paradigm prescribes a methodology for modularizing different dimensions of software systems. A theme that pervades all these software development paradigms is *patterns* [12][10][14], which are technology-independent, role-based descriptions of common ways of resolving key forces associated with recurring problems encountered when developing software.

Based on our experience developing and applying pattern-based [10][12][14] frameworks [4][5] and middleware platforms [16][17] for distributed systems over the past two decades, we believe that patterns are a valuable addition to the portfolio of higher-level system building blocks available to software developers. To enable patterns to become first-class citizens in MDD environments, it is necessary to define a set of composition rules and express relationships between patterns precisely. As discussed in [26], it is possible to substitute implementation artifacts that have inheritance relationships without affecting key properties of an entire system. This type of transformation can be treated as a *symmetry* [29], which is a special type of model transformation that preserves the key properties of a model. Examples of key model properties include *persistence*, which is the ability to read/write the state of an object to persistent storage and *remoting*, which is the ability to communicate with other system components over the network.

Coplien and Zhao [26] describe how object-oriented inheritance can also be treated as a symmetrical transformation because it preserves key behavioral aspects defined by base class. In turn, the concept of pattern feature inheritance introduced in this paper also preserves the key properties of the "base" pattern, so that substituting "derived" patterns provide variability without changing key system properties. This section describes how inheritance relationships between patterns help to support variability without degrading software symmetry. In particular, we treat transformation and inheritance as enabling mechanisms to simplify the substitution of certain system components without affecting other key system properties. These mechanisms therefore help make it easier to handle the types of variability typically encountered when developing MDD tools that support product-line architectures.

2.1 Handling Variability via Inheritance

Inheritance is a powerful mechanism for shielding certain parts of applications from side-effects caused by the need to customize certain functional aspects. To illustrate inheritance, consider the following classical Observer pattern [12] example shown in Figure 1. In this example, the **`Subject`** class is shielded from the variability introduced by different implementations of the **`Observer`** interface. The enabling mechanism in this case is *inheritance*, which in accordance to the Liskov Substitutability Principe (LSP) [7] allows **`Observable`** to work uniformly with all **`Observer`** subclasses, such as **`Notifier`** and **`Logger`**.

It would be nice to achieve the same level of substitutability with pattern-based building blocks. We therefore need to identify similar relationships between patterns. These relationships, in turn, should be used to facilitate the development of MDD tools that can automate pattern manipulation tasks and support the level of substitutability needed to address the challenges presented in Section 1.

2.2 Inheritance Relationships between Patterns

To explore the value of expressing inheritance relationships between patterns, we will examine the following set of patterns:

- **Observer** [12], which defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Reactor** [10], allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.
- **Interceptor** [10], which allows services to be added transparently to a framework and triggered automatically when certain events occur.

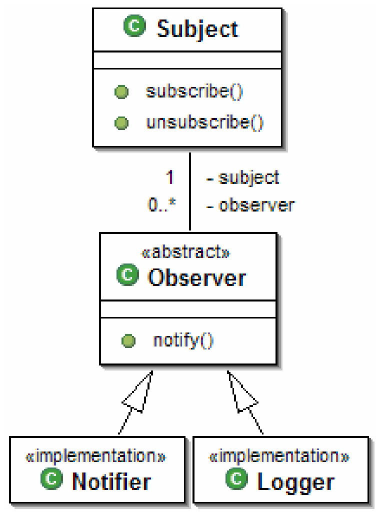


Figure 1. Observer Pattern Structure

There are common roles and responsibilities that cross-cut these patterns, e.g., there are certain events that can occur in a system, certain entities that need to be notified when such events occur, and certain ways these entities can express their desire to handle certain events by registering their interest. While this description is similar to the Observer pattern it does not mean that Reactor and Interceptor are simply different variants of Observer since each pattern has different forces and goals. Yet there are similarities that stem from the fact that these patterns share a higher-level relationship than just "different variants of Observer." We contend that this relationship can be represented by *feature inheritance*.

To explore feature inheritance relationships between patterns more concretely, consider again the Observer example presented in Section 2.1. The **`Notifier`** and **`Logger`** subclasses have different functionality and goals, i.e., notify users via a pop-up window and an output trace record, respectively. But they still conform to the "is a" relationship to the **`Observer`** base class. There are similar relationships for the Observer, Interceptor, and Reactor patterns, as shown in Figure 2.

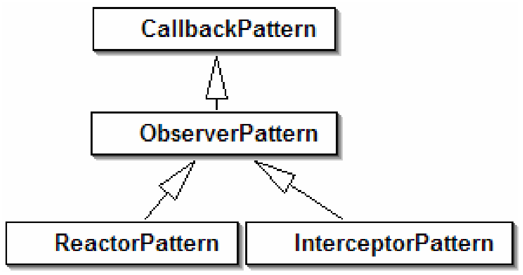


Figure 2. Relationships between Patterns

The pattern feature inheritance tree shown in Figure 2 defines the relationships between four patterns. At the root of the hierarchy is the **`Callback`** pattern, [30], which defines the basic mechanism (feature) used by all the rest related patterns – control inversion. In the **Observer** pattern, all registered **Observers** are called back by a registered **Subject**. Likewise, for the Reactor and Interceptor patterns the registered **Event Handlers** and **Interceptors** are called back, respectively, when the certain triggering conditions occur.

Despite the similarities between these four patterns, there are also some differences that bear mentioning because they motivate the **Observer** pattern as a basis for the set of related patterns and allow a cleaner connection between the patterns at Figure 2. In particular, a key difference between the Reactor and Interceptor patterns is the *event source*. The Reactor's event source is a demultiplexor, such as the **select()** or **WaitForMultipleObjects()** system calls, where-as the Interceptor's event source is incoming control flow, such as callback method invocation by CORBA Portable Interceptors [28] that are triggered during the remote invocation request/response flow. There is, however, no such role as *event source* in classical Observer pattern description – instead, that role is merged with the *subject* role. We therefore suggest the Observer pattern be extended, as shown in Figure 3.

The new **EventSource** role is responsible for monitoring possible condition changes and then initiating a notification propagation mechanism by triggering the **Subject** implementation, e.g., by invoking the **triggerUpdates()** method on the **Subject**. Introducing the **EventSource** role allows a cleaner separation of responsibilities for the Observer pattern. Moreover, compared with the previous approach shown in Figure 1, the **Subject** role is now only responsible for maintaining observers list and iterating over this list when notifications are propagated.

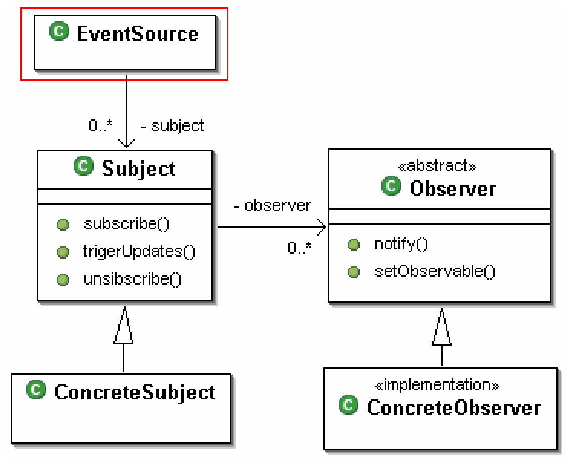


Figure 3. Extending the Observer Pattern with the Event Source Role

To illustrate different implementations of the **EventSource** role, the following cases could be considered:

- Different implementations of **EventSource**, e.g., example the family of different reactors, such as the **ACE_Select_Reactor**, **ACE_WFMO_Reactor**, and **ACE_Dev_Poll_Reactor** [5].
- A GUI event loop, which typically blocks on an OS demultiplexer, such as **select()** or **WaitForMultipleObjects()**, to detect incoming events (e.g., a mouse click) and then dispatch this event to the corresponding handlers (e.g., a button) , which in turn notifies observers about a change in state (e.g., button down).
- Hardware interrupt handlers can also be considered as event sources, which typically delegate event processing to observers in the OS kernel.

The Visitor pattern [12] could be also viewed as inheriting from Observer, where the event source is the traversing algorithm visiting various concrete nodes. For example, the Boost Graph Library (BGL) uses Observer pattern terminology (notify) for their generic visitor implementations of graph traversing algorithms [19].

We have identified other examples of inheritance relationships between patterns, as shown in Figure 4, which illustrates the set of patterns that solve similar problems using different methods.

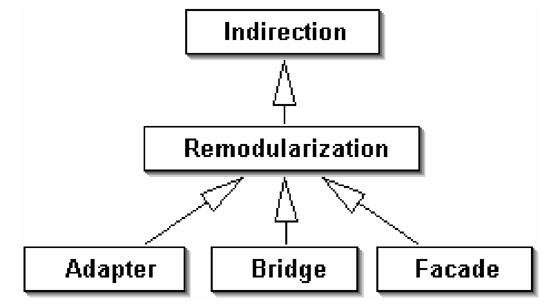


Figure 4. Example of Inheritance Relationships Between Patterns

Despite differences, the core mechanism used in these patterns is the indirection between two collaborating parties, which is why the **Indirection** pattern forms the root of this feature inheritance tree. The second level in the tree shows the **Remodularization** pattern, which enables collaboration between two objects even if a mismatch occurs between a provided interface and an interface expected by a collaborator. In turn, there are different circumstances and types of remodularization required in each concrete case, which is why the three other patterns in Figure 3 are specializations of the **Remodularization** pattern.

2.3 Applying Feature Inheritance in Practice

Section 2.2 shows how feature inheritance relationships between concerns can be presented in the form of patterns or other role-based definitions. Using this concept, we can provide a powerful mechanism to encapsulate variability at a higher level of abstraction than is possible with third-generation programming languages, such as C++ and Java. For example, we can encapsulate the impact of variability in the communication infrastructure (such as standard middleware or custom frameworks) on the rest of large-scale distributed systems.

The primary advantage of using feature inheritance in this way is to systematically introduce changes to a system using roles defined by certain role-based solution descriptions. For example, if a developer wants to add a Visitor pattern implementation to the code, a wizard provided by MDD tools could guide the user through the role mapping process to make sure that all roles defined by the Visitor pattern are mapped by the developer to the appropriate classes. The benefit of expressing feature inheritance relationships in this case is that after the mapping for base pattern role is done, subsequent substitutions of this pattern with concrete patterns can either be done automatically or semi-automatically (e.g., guided by wizards).

Figure 5 shows a high -level view of the complete modeling process described above.

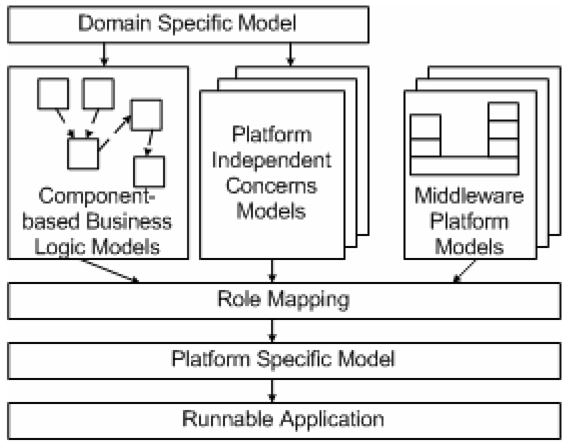

Domain Specific Model
Component-based Business Logic Models
Platform Independent Concerns Models
Middleware Platform Models
Role Mapping
Platform Specific Model
Runnable Application

Figure 5. Concern-based Modeling Process

This figure shows how domain-specific models are used as an input for various modeling tools. Next, the set of predefined role-based solutions can be introduced by means of a role mapping step. Finally, after completing the role mapping process, platform-specific models can be generated, followed by a runnable application.

3. Remote Button Example

This section presents a concrete example that further illustrates how the approach presented in Section 2 could be applied in practice.

3.1 Scenario

Consider a standalone application that is based on the refactored Observer pattern shown in Figure 3. This application has a simple GUI in the form of dialog box with a single button. Pressing this button causes the invocation of a method that implements application-specific functionality. As shown in Figure 6, the button plays the **`Subject`** role in the Observer pattern and the application-specific class plays the **`Observer`** role (with the application-specific processing implemented in the **`Observer`**'s **`notify()`** method), and the GUI event processing loop plays the **`EventSource`** role.

This figure represents the mapping between roles defined by Observer pattern (i.e., Subject, Observer and Event Source) and the application-specific classes (i.e., Button and GUI event loop implementation). As a result of feature inheritance, the Observer pattern can be replaced with derived patterns without breaking the key functional properties of this example system, i.e., "business class should be notified whenever the button is pressed." This example illustrates how pattern feature inheritance supports transformation without breaking symmetry.

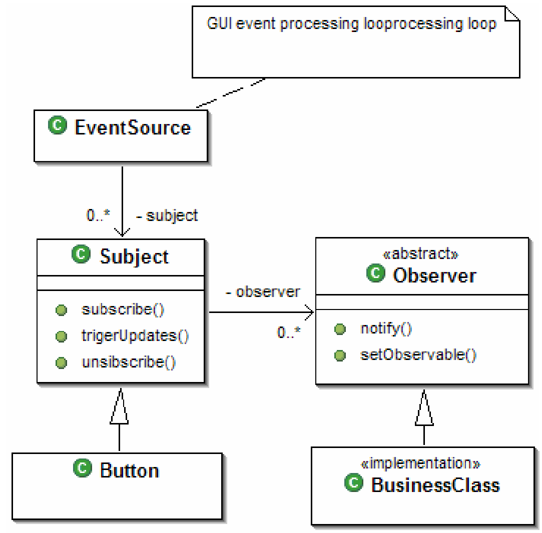


Figure 6. GUI Example Structure

3.2 Introducing the Remoting Aspect

The initial implementation of the GUI program shown in Section 3.1 was a standalone application. To work in a broader environment, assume that the scenario's requirements change so that it is necessary to split this application in two parts that communicate across a network. The first part (i.e., the GUI client) should be able to receive the push button event and then send this event over the network to the second part (i.e., the business server), which will then process this event the same way as in the initial scenario.

After this substitution, sample GUI application will be split into two parts that communicate with each other across a network. We thus introduce the *Remoting* aspect to the application, without changing key properties of the application, the i.e., **BusinessClass** will be notified when a button push event occurs.

We now analyze the impact of these changes on our initial application, in particular, on server-side of the new application. At one level, little has changed except for the event source, i.e., the source of the event notifications occurring in the system. In the standalone version, the event source was the GUI event loop that sent the mouse click notification to the standalone application. In the client/server configuration, conversely, the event source for the server-side will arrive from the network, i.e., the event source now is an OS demultiplexing, such as **select()** or **WaitForMultipleObjects()**.

Naturally, the **Reactor** pattern implementation is only part of the necessary interprocess communication (IPC) infrastructure. Introducing the Remoting aspect for larger application will therefore require more pattern implementations and associated aspects [17]. For the sake of clarity, however, this example assumes that the **Reactor** pattern implementation provides sufficient functionality to support our simple interprocess communication infrastructure.

3.3 Substituting Observer with Reactor

Based on the discussion in Section 2.2, if the **Reactor** pattern inherits from the **Observer** pattern, we can substitute our Observer-based implementation with a Reactor-based implementation *without* affecting the business components, i.e., **Button** and **BusinessClass** classes, which is written in terms of the **Observer** base class. The following list summarizes steps made as a result of the substitution mentioned above, focusing on the server-side modifications, which can be performed as follows:

1. Instead of running GUI event loop, the server needs to call the Reactor's **run_event_loop()** method, which will substitute the event source in the server application. Since this portion of the application is not part of the business logic and it will not require changes to application functionality, i.e., the implementation of Observer's **notify()** method by **BusinessClass** need not be changed.
2. The business logic implementation (i.e., the **Observer** role) contains registration logic (**subscribe()**) for events of interest. With the Reactor-based implementation the same step is required, i.e., event handlers should be registered with a reactor and need to pass an event mask that describes what types of events are of interest (e.g., the fact that there is data available in a socket). Once again, nothing should change in the application functionality.
3. The **Observer** (i.e., the event handler) will be notified by the reactor when there data is available in a socket registered with the reactor. After the reactor dispatches the handler, the handler can access the incoming data and perform the required processing steps.

Based on this analysis, it is clear that the processing steps for the original application functionality remain the same before and after adding the remoting aspect.

In a larger example, it may also be desirable to devise a solution that affects as little of the infrastructure software as possible. The approach described above does not provide this level of transparency due to differences in the APIs used for various tasks, such as accessing the event attributes, which in the case of *GUI events* come from GUI toolkit supplied data structures associated with the event and in the case of *network events* come from a socket. There are ways to further enhance the solution to minimize code perturbation, including:

- Using a patterns-oriented software library that is designed for composition and thus using uniform methods for accessing notification information. For example, the ACE [4][5] and TAO [15, 16] middleware platforms could be applied to our example application to minimize infrastructure rework.
- Remodularize the base code using aspect-oriented techniques. For example, [9] proposes an approach that uses the notion of *collaboration interfaces* for remodularization of interfaces that were not designed to interact with each other initially.

We believe that the second approach is more flexible and will concentrate our future research work in this direction.

4. Related Work

This section reviews work related to our approach.

Generative programming (GP) [23] is a type of program transformation concerned with designing and implementing software modules that can be combined to generate specialized and highly optimized systems fulfilling specific application requirements. The goals of GP are to (1) decrease the conceptual gap between program code and domain concepts (known as achieving high intentionality), (2) achieve high reusability and adaptability, (3) simplify managing many variants of a component, and (4) increase efficiency (both in space and execution time). GP is typically concentrates on single classes which could be parameterized to achieve the required functionality. Despite the powerful customization mechanisms, GP still remains at the level of abstraction supported by programming languages like C++, Java or C#. In contrast, our approach focused on higher level building blocks like design patterns which could be instantiated by using the approach which is similar to the way how templates are parameterized in GP. Role-based description of the solution could be treated as the kind of template which spans across multiple classes.

Aspect-oriented software development (AOSD) is a GP technology designed to more explicitly separate concerns in software development. AOSD techniques [13] make it possible to modularize crosscutting aspects of complex distributed systems. An aspect is a piece of code or any higher level construct, such as implementation artifacts captured in a MDA PSM, that describes a recurring property of a program that crosscuts the software application, i.e., aspects capture crosscutting concerns. In our approach, the role-based solution could represent either cross-cutting concern or concern which could be modularized using OO technique. In case of cross-cutting concern we will need to implement the special model transformation to distribute the particular functionality over the business code. This task is similar to the task typically performed by weavers in AOP.

Scope, Commonality, and Variability (SCV) analysis [24] is related work on domain engineering that focuses on identifying common and variable properties of an application domain. SCV uses this information to guide decisions about where and how to address possible variability and where the more "static" implementation strategies could be used. Our approach naturally supports SCV and provides the possibility to capture commonality and variability at the level which is much closer to the problem domain then it is possible using general purpose programming languages. In addition, pattern feature inheritance provides the powerful mechanism to deal with variability at the higher abstraction level end enables the substitutability of the pattern-based system building blocks similar to the substitutability at the class level provided by inheritance in object-oriented approach.

In [18] the authors describe the role based approach to forward and reverse-engineering in order to introduce or find pattern instances in existing code. This idea is very similar to what we suggesting in this paper. Our main contribution to the topic is the feature inheritance relationships between patterns which are required to allow better level of substitutability and composability at the model level.

5. Concluding Remarks

This paper presents the novel approach to pattern classification and composition by introducing the feature inheritance relationships between patterns. We also demonstrate how patterns can be used as higher-level building blocks to support the introduction of new aspects without affecting the main application logic. This approach is possible because of relationships between patterns that are analogous with inheritance in OO programming languages.

The work described in this paper provides the conceptual foundation for a certain type of model transformation that preserves key properties of applications being developed. This type of transformation can be treated as a symmetrical transformation and used to allow better substitutability of model parts defined as role-based solution templates. Our work also enables the automation of role-mapping process by MDD tools based on feature inheritance relationships between patterns. Pattern feature inheritance is an example of symmetrical transformation that is important for the next generation of modeling tools, which need to manipulate higher-level building blocks, such patterns or other role-based solutions.

The ultimate goal of our work is to create an Integrated Concern Manipulation Environment (ICME) [2][3], which is an MDD toolsuite that allows manipulation (i.e., adding, removing, and specializing) different aspects of large-scale distributed software systems using higher level building blocks (such as patterns and aspect-oriented techniques) to merge these blocks unobtrusively with the application logic implementations. To provide such ICME manipulation functionality we need to determine how to formalize pattern composition rules. In the pattern literature, *forces*, *benefits*, and *liabilities* are mentioned as key factors to make decisions about which pattern to use in which contexts and how to combine patterns together effectively. Our future work will analyze these descriptions in various patterns and devise MDD-based formalisms and tools that support automated and/or semi-automated analysis of pattern usage and composability. For example, MDD wizards can guide users through decision processes by asking questions and navigating through a graph of patterns to select suitable patterns.

References

[1] OMG: "*Model Driven Architecture (MDA)*" Document number ormsc/2001-07-01 Architecture Board ORMSC1, July 9, 2001.

[2] A. Nechypurenko, T. Lu, G. Deng, D. C. Schmidt, A. Gokhale. "*Applying MDA and Component Middleware to Large-scale Distributed Systems: A Case Study*," Proceedings of First European Workshop on Model Driven Architecture with Emphasis on Industrial Application, University of Twente, Enschede, The Netherlands, 2004.

[3] A. Nechypurenko, T. Lu, G. Deng, E. Turkay, D. C. Schmidt, A. Gokhale. "*Concern-based Composition and Reuse of Distributed Systems.*" Proceedings of the 8th International Conference on Software Reuse, 2004.

[4] D. C. Schmidt, S. D. Huston. *C++ Network Programming: Mastering Complexity Using ACE and Patterns*. Addison-Wesley Longman, 2003.

[5] D. C. Schmidt, S. D. Huston. *C++ Network Programming: Systematic Reuse with ACE and Frameworks*. Addison-Wesley Longman, 2003.

[6] P. Tarr, H. Ossher, W. Harrison,S.M. Sutton Jr., "*N Degrees of Separation: Multidimensional Separation of Concerns*," Proceedings of the 21st International Conference on Software Engineering, ACM, New York, 1999, pp. 107--119.

[7] B. Liskov, "*Data Abstraction and Hierarchy*". SIGPLAN Notices, 23,5, May 1988, p. 25.

[8] D. M. Weiss. "*Defining Families: The Commonality Analysis*," Proceedings of the 21st International Conference on Software Engineering, Los Angeles, 1999, pp. 671 - 672.

[9] M. Mezini and K. Ostermann. *"Integrating Independent Components with On-Demand Remodularization*," Proceed-

ings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOSPLA), Seattle, Washington, USA, November 4-8, 2002.

[10] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, "*Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*", Volume 2, Wiley & Sons, New York, 2000.

[11] J. Gray, J. Sztipanovits, T. Bapty Sandeep Neema, A. Gokhale, and D. C. Schmidt, "*Two-level Aspect Weaving to Support Evolution of Model-Based Software*," Aspect-Oriented Software Development, Edited by Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhan Clarke, Addison-Wesley, 2003.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "*Design Patterns: Elements of Reusable Object-Oriented Software*." Addison Wesley, 1995.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. "*Aspect-oriented programming*", Proceedings of ECOOP'97, Jyvaskyla, Finland, 1997.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "*Pattern-Oriented Software Architecture—A System of Patterns*", John Wiley and Sons, 1996

[15] D. C. Schmidt, D. L. Levine, and S. Mungee, "*The Design and Performance of Real-Time Object Request Brokers*" *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[16] D. C. Schmidt et. al, "*TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems*", IEEE Distributed Systems Online, vol. 3, no. 2, Feb. 2002.

[17] M. Völter, A. Schmid, E. Wolff. *Server Component Patterns: Component Infra-structures Illustrated with EJB*, Wiley and Sons, 2002.

[18] Gert Florijn, Marco Meijers, and Pieter van Winsen, "*Tool Support for Object-Oriented Patterns*" Proceedings of ECOOP'97, Jyvaskyla, Finland, 1997.

[19] J. G. Siek, L. Lee, A. Lumsdaine. "*Boost Graph Library, the User Guide and Reference Manual*". Addison Wesley.

[20] "eXecutable UML (xUML)," Kennedy Carter, http://www.kc.com.

[21] George T. Heineman and Bill T. Councill, "*Component-Based Software Engineering: Putting the Pieces Together*", Addison-Wesley, Reading, Massachusetts, 2001.

[22] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent, "*Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*", John Wiley & Sons, New York, 2004.

[23] Krzysztof Czarnecki, Ulrich Eisenecker. "*Generative Programming: Methods, Tools, and Applications*". Addison-Wesley Pub Co.

[24] J. Coplien, D. Hoffman, D. Weiss, "*Commonality and Variability in Software Engineering*", *IEEE Software*, November/December 1999, pp. 37-45.

[25] I. Jacobson, G. Booch, J. Rumbaugh. "*The Unified Software Development Process*". Addison-Wesley Professional, 1999.

[26] J. Coplien and L. Zhao. "Symmetry Breaking in Software Patterns," Springer Lecture Notes in Computer Science Series. , 2001.

[27] J. Coplien. "The Future of Language: Symmetry or Broken Symmetry?" Proceedings of VS Live 2001, San Francisco, California, January 2001.

[28] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith, "*Using Interceptors to Enhance CORBA,*" IEEE Computer, July 1999.

[29] J. Rosen, "Symmetry in Science: An Introduction to the General Theory," pp 9-10. New York: Springer-Verlag, 1995.

[30] Steve Berczuk, A Pattern for Separating Assembly and Processing, *Pattern Languages of Program Design*: *Volume 1*, Addison-Wesley, 1995.

Typing Relationships in MDA

Jim Steel, Jean-Marc Jézéquel

IRISA, Campus universitaire Beaulieu
35042 Rennes cedex France
{jsteel, jezequel}@irisa.fr

Abstract

As the OMG's Model-Driven Architecture matures from a field of research and specification into one of system engineering, it faces all of the challenges endemic to the practice. Among the foremost of these is the need to support re-use of its artifacts as they evolve. As systems begin to be built upon the basic ideas of models interrelated by model transformations, it will become increasingly important to have appropriate definitions for the typing relationships that can exist between models and metamodels, since it is these definitions that will determine the substitutability characteristics of these artifacts in model transformations. This paper seeks to enumerate a number of these relationships, to provide initial characterisations of them in terms of their significance to the goal of re-use in MDA.

1 Introduction

The Model-Driven Architecture[14] uses model transformations to describe (and probably to enforce) the relationships between models, as described by metamodels.

However, the metamodels (and by consequence the models) that are used in model-driven systems are diverse, they evolve, and they are frequently overlapping in their domains, even when they are not in their definitions. For all of these reasons, when we write model transformations, we want them to apply over a wide range of models. This feature, much studied in software engineering, is known as re-use.

The languages currently used to write model transformations, including but not limited to those proposed in [5, 1, 8] are dramatically diverse, as illustrated in [7, 4, 6], a situation that is unlikely to be completely remedied by the eventual arrival of an adopted specification for their definition[11]. All these approaches build on the notion of model element, seen as an instance of a specific class of a given meta-model MM1. A model M is made of model elements linked between themselves to form an arbitrary complex graph, conforming to the meta-model MM1. Such a model M can be provided as an input parameter to a transformation T.

In this paper we explore various cases of relationships between model elements, models and meta-model, in order to discuss under which conditions a transformation T can be safely applied to a model M. We start by presenting three main motivations for this work before entering into the details of model typing and conformance.

Clearly, the study of type systems is not a new one. In particular, much research has been conducted and validated within the functional languages community, and particularly the ML languages. Closer in heritage to MDA is the field of object-oriented systems. Some of the earliest work was by Liskov [9], in the form of the much-referenced substitutability principle. This was built upon formally by, notably, Cardelli & Wegner in [2], and further by Castagna in [3], as extensions of lambda calculus. Also relevant are the typing strategies that have been implemented in O-O languages such as Java, Eiffel, and dynamic languages such as Python and Ruby.

However, it is important to note that their important differences between the underlying data structures of object- and model-based systems. Most significant of these is the linking of fields/properties as opposites (represented in earlier versions of MOF as associations). This feature means that models form much more tightly-coupled graphs than objects, which could often be treated as atoms in isolation. In particular, this graph-ness has important implications for typing relationships, in that the the relationship between a model element and a class will generally involve the analysis of the types of the other model elements and classes in the respective graphs. It also introduces the need to deal with the inevitable circular dependencies that arise in evaluating type relationships across these graphs.

2 Motivation

We argue that the need for a flexible mechanism for re-use in model-driven engineering comes from the inevitable separation of the metamodels used to describe models. There are a number of reasons for this separation, a number of which are detailed here.

2.1 Physical vs Logical metamodel

For many reasons, it is not always possible to ensure that all models of the same notional metamodel are defined in terms of the same physical definition of that metamodel. For example, the physical metamodel may be in a serialized form, such as XMI, whereas a given transformation requires it in an object form, or vice versa. Ideally, these issues should not affect the ability of a transformation to apply equally to models whose metamodels are logically equivalent but physically distinct. This issue is also highlighted by the increasing application of models in the design and implementation of distributed systems, on such platforms as CORBA and web services.

2.2 Extension

There are number of mechanisms provided for extension of metamodels. In the 1.x versions of MOF[10], these included package import, extension and clustering. In MOF 2.0, there are additional mechanisms such as package merge and package combine. These relationships are established at the package level, and have varying implications for the corresponding relationships at the class level. While a full discussion of these relationships is not the domain of this paper, we will briefly analyse the package combine mechanism, since it is the most challenging form of metamodel extension in terms of re-use.

Package combine, as defined in the UML 2 Infrastructure submission, is a form of package merge, whereby all classes in the original package are copied into the new package. Any classes defined with the same names are "merged": the new set of features for the class is the union of the sets of each of the original classes. After application of the package merge, all relationships between the packages, and between the classes therein, are removed, so that the new package can be used independently of the original.

Strictly speaking, package combine is not a relationship, but an operation, but this does not diminish its usefulness in modelling. Moreover, its use should not prohibit a transformation defined in terms of the original metamodel from working with the extended one.

2.3 Evolution

Metamodels evolve over time, as do the models that they describe. In general, the problem of model evolution and versioning is very complicated, and is still the subject of active research.

However, simple changes such as the addition of an extra attribute to a class should not impact on the ability of a transformation defined in terms of the original version of the metamodel to work with the modified version. This should be possible regardless of whether the heritage of the metamodel, in terms of version history, has been properly preserved or not[12].

3 Relationships

We present here a number of relationships that can exist between classes and model elements, that might be used in typing models.

For the purposes of this section, we use definitions for model element, model, and metamodel as commonly understood in the OMG, and as mentioned briefly in the introduction. We characterise relationships using a number of properties, such as their normal modes of interaction (are they generally requested, or affirmed, and are they qualified with respect to a certain domain), and their relationships to one another in terms of supersets and subsets.

3.1 Instantiation

Model elements in model-driven systems are created from a class, be it directly or using a factory. In this way, the model element is given slots for each of the properties of the class and, typically, will delegate the semantics of an operation to a method attached to the method definition.

This relationship is the basic building block for the definition of other, more flexible typing relationships. Specifically, it defines the "provided type" that is used in comparison to the required type for the purposes of type-checking. Of itself, it offers little by way of flexibility, and addresses none of the issues raised in section 2.

This relationship is typically requested, rather than affirmed, and is always absolute, never qualified, since there can be only one correct response.

3.2 Reflection

Reflection is the process of asking a model element for a description of itself. More specifically, it involves learning what are the operations and properties provided/supported by the model element including, by extent, their types, and thus potentially extending over a large graph of types reachable from that of the original.

It should be noted that the type system in MDA, given by MOF, has no separation between between types and classes, and thus reflection provides a class. This includes details, such as the class name, that may not be relevant to the definition of reflection given here.

In theory, reflection is slightly more flexible than "instantiated by", since one may have multiple metamodels that equally describe the capabilities of the model element, through techniques such as collapsing subclasses. However, it is unable to handle substitutability problems such as instances of subclasses, or of structural subtypes.

This relationship, like "instantiated by", is typically requested, rather affirmed, and is usually absolute, although in theory could be qualified, such as by policies for collapsing subclasses. The "instantiated by" relationship is a subset of the reflection relationship.

3.3 Conformance By Inheritance

Inheritance, also known as generalization/specialization is an explicit relationship between classes dictating, among other things, that all features defined on the superclass will be available on instances of the subclass. In this relationship, a model element is conformant to a class iff its instantiated class is the same as the required class, or is an explicit subclass (either directly or transitively) of the required class.

This is the same relationship as is commonly seen in programming languages such as Java, and is the most common relationship presently used in model transformation languages. Moreover, it is the relationship used by OCL[13] and thus, by association, MOF and UML.

It has the advantage that it is more flexible than either instantiation or reflection, since it allows for instances of subclasses to be accepted, in addition to those of the specified class. Also, since it is explicitly defined, it is efficient to compute and well-suited to static evaluation.

Conformance relationships are affirmed, rather than queried, and as such are typically absolute rather than qualified. This relationship is a superset of instantiation, but not of reflection.

3.4 Structural Conformance

Structural conformance bears some similarity to reflection, in that it deals with the set of features (operations and properties) that are supported by the model element. In this way, a model element is structurally conformant to any class that is a subtype of its instantiating class, where the definition of subtype is based on that defined by Cardelli & Wegner in [2]. In fact, since Cardelli & Wegner's definition is based on objects, some small extensions need to be made to apply it to the realm of models, such as treatment of multiplicities. We present a summary of the definition.

```
A class A is a subtype of a class B iff:
    ∀ property P of B, ∃ property P′ of A, such that

        P′.name == P.name
        the type of P′ is a sub-
        type of the type of P (covariance)
        the multiplicity of P′ conforms to the mul-
        tiplicity of P
        P.isReadOnly == false im-
        plies P′.isReadOnly == false
    ∀ operation O of A, ∃ operation O′ of B, such that

        the return type of O′ is a sub-
        type of the return type of O (covariance)
        ∀ parameter R of O, ∃ parame-
        ter R′ of O′, such that

            the type of R is a sub-
            type of the type of R′ (contravari-
            ance)
            the multiplicity of R con-
            forms to the multiplicity of R′
```

Structural conformance of classes is characterized by covariance with the types of properties and the return types of operations, and contravariance with the types of parameters to operations.

The multiplicities, consisting of cardinality ranges, orderedness and uniqueness, of the properties, operations and parameters of the classes must also be considered. The simplest, but most restrictive, approach is to consider only exactly equal multiplicities as conformant. Alternatively, one can impose a hierarchy, whereby ordered collections are a subtype of unordered ones (but not vice versa), and cardinality ranges are given a priority order such as [0..*, 0..1, 1..1], where each range is conformant of any range that follows it. Such an approach would cover 90% of cases, although for full coverage, a more sophisticated heuristic such as partial orders would be needed to handle other ranges such as 1..*.

In a general purpose programming language, the failure to consider the behaviour of the operations would mean that structural conformance falls short of true substitutability. However, it is important to remember that MOF is not a general purpose programming language. In fact, it bears more resemblance to signature languages such as java interfaces, C++ templates, or CORBA interfaces. As such, any consideration of operation behaviour, such as would be required in terms of Liskov's substitutability principle[9], is out of scope.

Like inheritance-based conformance, this relationship is affirmed rather than qualified, and is generally not qualified. Structural conformance is a superset of direct instantiation, reflection, and conformance by subclassing. (That is, conformance by subclassing implies structural conformance; a useful axiom for implementation.)

Structural conformance offers considerable advantages over conformance by subclassing in terms of flexibility. In particular, with respect to the motivations presented in Section 2, it is much better able to handle the issues of evolution and extension, including package merge. Its disadvantage is that it is significantly more intensive to evaluate, and is less amenable to static evaluation.

4 Future Work And Conclusions

The contrast between inheritance-based and structural conformance for models is clearly one of efficiency versus flexibility. To evaluate these criterion in more detail, we now propose to prototype the different approaches and apply them to various examples of model transformation.

To this end, we have developed a MOF repository using the Ruby programming language[15]. Ruby is notable for its flexible approach to typing, which is often described as “duck-typing” (if it walks like a duck, and it quacks like a duck, then it must be a duck). The lack of any real type-checking in Ruby makes it well-suited for the evaluation of different typing strategies.

A possible subsequent avenue for exploration is that of typing strategies within models themselves. The relationship between the type of a property and the type of a model element that might fill it, for example, is very similar to the relationship between a transformation specification and those models that might be permitted as input. Moreover, a number of transformation languages, such as [5] and [1] are based largely on the population of functional or relational models (sometimes called traceability models) as the determining factor for creating, deleting, or modifying model elements. As such, any approach to structural type conformance in these languages would need support for structural conformance within these models.

References

[1] David H. Akehurst and Stuart Kent. A relational approach to defining transformations in a metamodel. In *UML 2002 - The Unified Modeling Language, 5th International Conference, Proceedings*, pages 243–258, 2002.

[2] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):211–221, 1985.

[3] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Birkhäuser, 1997.

[4] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, Anaheim, USA, October 2003.

[5] K. Duddy, A. Gerber, M.J. Lawley, K. Raymond, and J. Steel. Model transformation: A declarative, reusable patterns approach. In *Proc. 7th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2003*, pages 174–195, September 2003.

[6] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. A review of OMG MOF 2.0 query / views / transformations submissions and recommendations towards the final standard, August 2003. OMG Document: ad/03/08/02.

[7] A. Gerber, M.J. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The missing link of MDA. In *Proc. 1st International Conference on Graph Transformation, ICGT’02*, volume 2505 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[8] Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT: an extendible UML transformation framework. In *Proc. Automated Software Engineering, ASE'99, Florida*, October 1999.

[9] B. H. Liskov and S. N. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, April 1974.

[10] Object Management Group (OMG). Meta Object Facility (MOF) specification. OMG Document ad/97-08-14, September 1997.

[11] Object Management Group (OMG). MOF 2.0 Query/Views/Transformations RFP. OMG Document ad/2002-04-10, October 2002.

[12] Object Management Group (OMG). MOF 2.0 Versioning RFP. OMG Document ad/2002-06-23, June 2002.

[13] Object Management Group (OMG). The object constraint language (OCL), 2003. http://www.omg.org/docs/ptc/03-08-08.pdf.

[14] R. Soley and the OMG Staff. Model-Driven Architecture. OMG Document, November 2000.

[15] David Thomas and Andrew Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley Professional, 2000.

OMELET : Exploiting Meta-Models as Type Systems

Edward D. Willink

Thales Research and Technology (UK) Ltd
EdWillink@iee.org

Abstract. Meta-modelling is now well established for individual models. The MOF QVT proposal should support meta-model-based transformation between models. However, meta-model compatibility poses a major threat to the successful exploitation of transformation technology. We therefore introduce OMELET, a next generation 'make', that supports integration of diverse transformations and uses meta-models as a type system to ameliorate the threat and pave the way for automated composition of transformations.

1 Introduction

Activities such as the QVT proposal, XSLT schema support and the MDA have provided much needed impetus to model transformation. A model transformation supports the conversion of one (or more) input models into one (or more) output models, and each model is based on an associated meta-model as depicted in Fig. 1.

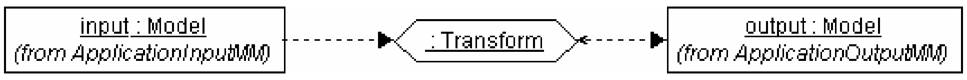


Fig. 1. Typical transformation invocation

In this paper we are interested in the problems that arise with multiple transformations, in particular the problem of meta-model compatibility between two transformations in a chain as depicted in Fig. 2.

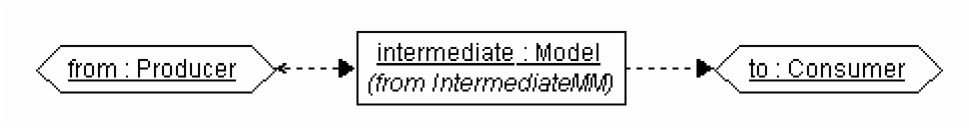


Fig. 2. Typical transformation interconnection

We seek to ensure that the `intermediate` model, produced by an instance of a `Producer` transform and consumed by an instance of a `Consumer` transform, is indeed based on the `IntermediateMM` meta-model.

It is convenient to say that our models are instances of our meta-models. However this is inaccurate; a meta-model is a package containing a variety of useful elements, some of which may be useful in a particular application. Bézivin [1] draws the distinction that a model is based on a meta-model. It is the elements in a model that instantiate elements of its meta-model and also comply with the associated constraints expressed in the meta-model.

We will briefly review the need for and hazards of multiple transformations, discuss some of the limitations of current technologies and suggest how the next generation of tools can address some of the problems.

2 Multiple Transformations

In [2] we introduced the Side Transformation Pattern as a technique to make model transformations modular and re-usable. This was achieved at the expense of changing a typical monolithic transformation involving two meta-models (input and output as in Fig. 1), into a composite transformation with four meta-models and three sub-transformations as shown in Fig. 3. The pattern therefore introduces two intermediate meta-models and four extra opportunities for incompatibility.

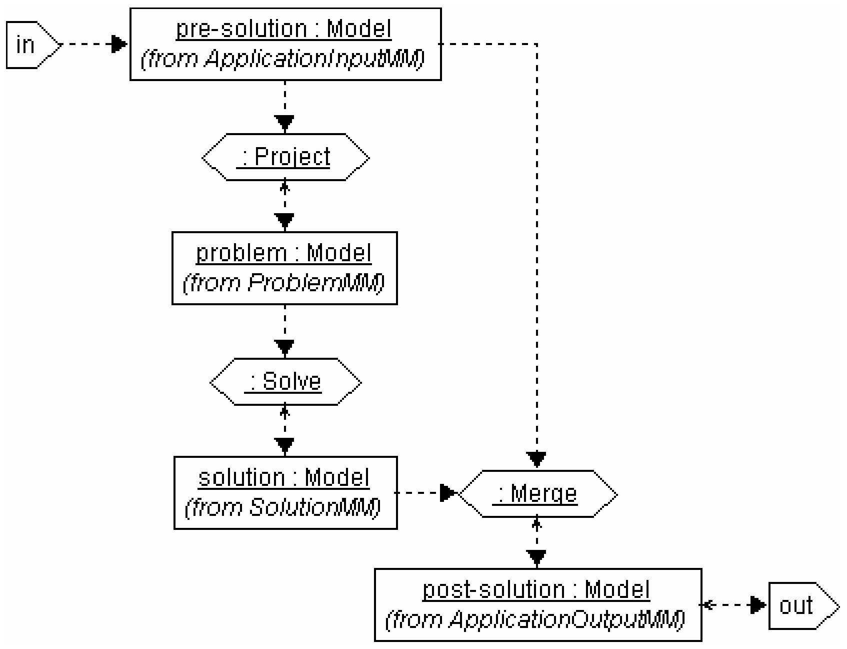


Fig. 3. Side Transformation Pattern

Increasing numbers of stages of transformation will be required as Model-Driven approaches are adopted with greater abstraction in a Platform Independent Model or in some Domain Specific Language in front of a PIM. These transformations will be more manageable if each stage resolves the concerns of a single form of abstraction. We may therefore expect the Model Driven Architecture to involve a chain of transformations to weave the various PIM, Platform Model and Mark Model concerns into a coherent Platform Specific Model. We can also expect the intervening stages in the chains to involve many distinct meta-models, or at least many distinct sub-sets of a smaller number of shared and often standard meta-models.

With many meta-models arising from transformation chains and further meta-models arising from using the Side Transformation Pattern to promote modularity and re-use, the integrity of these meta-models becomes critical to our endeavours. The problems of XMI dialects between early UML tools should act as a salutary warning.

3 Current Technology

Ensuring that models really are accurately based on their meta-models is difficult with current technology, and so there is rather too much reliance on the best endeavours of programmers and their intuition in choosing appropriate sub-sets of inconveniently large meta-models, such as UML. This provides ample opportunity for a joint development of

Producer and Consumer transformations to experience a rather troubled development. Problems are almost guaranteed when a more widespread attempt to re-use these pragmatic transformations is made.

The XML standard provides a good compromise between a human-accessible and a computer-accessible file representation. This makes it very appropriate for interchange between transformations where it is produced and consumed by computers, but needs to be intelligible by humans for at least debugging and sometimes manual interventions.

However, experienced XML users have discovered that XML conformance is a very weak discipline. It is all too easy for the conformant XML dialect of Producer and Consumer to differ, and as a consequence of the eXtensibility of XML, the difference in dialect is only detected after a number of intervening activities have conspired to make diagnosis difficult.

DTDs and now XSDs are therefore increasingly used to validate that the intervening files exhibit both semantic as well as syntactic consistency. This enables detection of errors in the Producer such as generation of spurious constructs and omission of mandatory constructs. However neither DTD nor XSD allow for more subtle validation of constraints on optional constructs. And of course no validation of the input can validate that the Consumer dialect is compatible.

XSLT provides its transformation capability within the XML Technology Space. Unfortunately the absence of comprehensive schema-aware support in current XSLT processors prevents diagnosis of seriously errant XPath expressions. This severely erodes the benefits that XSLT2 (or more readably, NiceXSL[4]) can offer.

Within the Modelling Technology Space, MOF-derived models provide for more accurate modelling in which OCL constraints capture subtle semantics. The lack of a direct model transformation capability should be addressed by the MOF QVT proposal. This should provide inherent rather than accidental compliance with the input and output MOF models and so introduce much needed discipline and efficiency to transformation programming.

When MOF models are converted to Java models to exploit the Program Technology Space, some inaccuracies in a Java-based Producer or Consumer can be avoided at compile time.

Until all transformations are defined in some language such as QVT that enforces model compliance, it is essential to perform as much model validation as possible in order to establish integrity for each intermediate model, and assist in diagnosis of inadequate transformations.

4 Tool Support

`make` and more recently `Ant` have established themselves as important parts of a programmer's tool kit. Both enable a number of programs to contribute to the solution of a larger problem. `make` also allows for some automated discovery of appropriate sequencing and invocation of those programs. However the composition of programs lacks discipline.

In `Ant`, the control flow (`depends`) defining the program sequencing is independent of the data flow (the task-specific input and output commands), so there is ample opportunity for typographic mistake and no inherent reason why the output of one program should be suitable as the input of another.

In `make`, the control flow is deduced from the file dependencies, so the control and data flow are consistent although sometimes surprising. The typical use of file name extensions to identify the data content of intermediate files encourages consistent usage, but there is still no inherent guarantee that the file extension correctly describes the content.

For transformations, we require the same ability to exploit a mix of custom and standard contributions, and we need to ensure that the usage of the transformations is valid. Meta-modelling provides the solution to these problems, since the appropriate meta-model provides a strict definition of the permissible type of each intermediate 'file' in the composition.

We may therefore look towards a next-generation `make` in which rules are defined by registering the capabilities of particular model transformations in terms of the acceptable input and satisfied output meta-models. Using a very simple `make`-like example; given a pair of transformation signatures (`name = input-model-name : input-meta-model -> output-model-name : output-meta-model`)

```
compile = c_file : c_MM -> o_file : o_MM
link = o_file : o_MM -> exe_file : exe_MM
```

and a request to produce a model based on the `exe_MM` from a model based on the `c_MM`, we can deduce a suitable transformation chain to comprise `compile` followed by `link`. We can augment the chain with validation of input, intermediate and output meta-model compliance.

Many practical transformations are only appropriate for a sub-set of the syntax or semantics of particular meta-models. For instance simplified support for UML state charts might exclude History States, and an executable profile must exclude facilities with ill-defined semantics. This inhibits arbitrary model-independent chaining of transformations, but if the transformation chain is deduced within the context of the models to be transformed, the actual meta-model sub-sets are known and sub-set transformations can be exploited reliably.

We therefore require transformations to accurately define the sub-set meta-models that the transformation supports. Since this information will not be automatically available for many transformation technologies, we must be able to assert this as part of a transformation declaration.

Determination of the sub-sets in use by particular models should be a relatively straightforward model analysis to be performed by the transformation tool.

We must allow the user to specify a transformation chain, explicitly when they need complete control, implicitly when automation is acceptable, partially when they need to exert some influence, and historically when they need to repeat a previous sequence.

The non-implicit specifications provide intermediate way-points in the transformation chain, between which a transformation chain must be established. The tool must enable the user to view the actual chain, understand why certain transformations are necessary, and more importantly understand why certain transformations are unsafe.

This is the goal of the Eclipse/OMELET project [3]. Upgrading the capabilities of `make` to adopt meta-models provides the opportunity to deduce powerful transformation compositions. Adopting the Java extensibility approaches underlying `Ant` provides the opportunity to integrate transformations arising from a wide range of differing technologies. Using meta-models allows the transformation intermediates to be validated and transformation chains deduced.

At the time of writing a preliminary OMELET release is available that demonstrates the ability to register and invoke a diversity of transformations and meta-models. A rather more useful release should be available by the time this paper is presented.

5 Acknowledgements

The author is grateful to Thales Research and Technology for permission to publish this paper, which is influenced by work done on the SPEAR and GSVF-2 projects.

6 Conclusions

We have shown how meta-models can introduce discipline to transformation chains and motivated the development of OMELET, a next generation `make`-style program that uses meta-models to impose a type system on transformations that are implemented in a diverse range of technologies.

7 References

1 Jean Bézivin, MDA : From Hype to Hope and Reality, Guest talk at UML'2003.

http://www.sciences.univ-nantes.fr/info/perso/permanents/bezivin/UML.2003/UML.SF.JB.GT.ppt

2 Edward D. Willink and Philip J. Harris, The Side Transformation Pattern - making transforms modular and re-usable, submitted to SETra-2004, October 2004.

http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/omelet-home/doc/SETra2004/SETra2004-Pattern.pdf

3 The Eclipse OMELET Project.

http://dev.eclipse.org/omelet

4 The SourceForge NiceXSL Project.

http://nicexsl.sourceforge.net

Coral: A Metamodel Kernel for Transformation Engines

Marcus Alanen and Ivan Porres

TUCS Turku Centre for Computer Science
Department of Computer Science,
Åbo Akademi University
Lemminkäisenkatu 14, FIN-20520 Turku, Finland
e-mail:{marcus.alanen,ivan.porres}@abo.fi

Abstract. A metamodel kernel is a program library or application framework that is used to manage models described in user-defined modelling languages. Metamodel kernels provide the basic functionality to create models, add, delete and update elements in an existing models and to store and retrieve models from a XMI document.
Coral is a metamodel kernel that is used to try in practice new research ideas in modelling technology. In this short paper, we describe Coral, our own implementation of a modelling tool and some discoveries related to modelling and metamodelling that we have found.
Keywords: Modelling Frameworks, Model Driven Engineering, Metamodelling, Modelling

1 Introduction

The advance of modelling techniques both in academia and industry has lead to the development of several commercial modelling tools. However, the research area of modelling tools is still relevant as solid frameworks are required to empower software developers to actually use the benefits of a model driven architecture. In this paper, we present the idea of a metamodel kernel and our work on Coral, a generic open source metamodel kernel.

A metamodel kernel is a program library or application framework that is used to manage models described in user-defined modelling languages. Metamodel kernels provide the basic functionality to create models, add, delete and update elements in an existing models and to store and retrieve models from a XMI document. Examples of metamodel kernels are the Eclipse EMF project and the Netbeans Metadata Repository.

All metamodel kernels are based on a specific metamodelling language. This metamodelling language defines the building blocks of all modelling languages support by the kernel. An example of a metamodelling language is the OMG Meta Object Facility (MOF) [4]. All the existing metamodel kernels implement at least a subset of MOF. However, MOF is not a metamodel kernel as such since it is not a software tool. Eclipse uses EMF as its metamodelling language that is comparable to EMOF, a simplified version of MOF. Coral uses its own Simple Metamodel Description (SMD) language. SMD is quite similar to EMOF but contains some extensions to deal with models described in multiple modelling languages.

A metamodel kernel can be used to implement model transformations as defined in the Model Driven Architecture (MDA) but, as such, the features provided by a kernel are too basic. In many cases, it is necessary to implement a transformation engine that can be used to execute model transformations defined in a high-level model transformation language. Examples of transformation engines are the implementation of rule-based transformations presented in [6] or the transformation tool for relational mappings presented in [1]. Although these tools implement different transformation languages, they share many common features related with basic model management that could be implemented by an independent kernel.

The discussion about model transformations is quite often centred on model transformation languages. However, we consider that is equally important to discuss the features and development of model transformation engines and metamodel kernels that support interesting model transformation languages. Coral is an attempt to seek practical and theoretical issues in these topics and provide a working solution for researchers.

In the rest of the paper, we present the more important features of Coral, and how it manages to create a flexible approach to querying and manipulating models that can be used to implement a transformation engine.

2 A Dynamic Metamodelling and Modelling Tool

The Coral kernel is based on few but important principles. The most fundamental is the notion of being metamodel-independent, i.e., metamodels and models can be created at runtime. In several other modelling tools, there is only one or a few static metamodels to choose from; in Coral, metamodels are first-class citizens. Large parts of Coral try to be as ignorant of the underlying metamodel as possible, and several interesting algorithms and problems arise from this.

Even though Coral can create any metamodel at runtime, there are still some predefined metamodel elements (*metaelements*) for primitive datatypes such as integers, strings and floating-point values.

A Coral modelling language is represented as a model in a language called the Simple Metamodel Description language. SMD can be seen as analogous to MOF. Whenever Coral needs the definition of a modelling language, the SMD model for this language is loaded dynamically and converted to a metamodel internally. Naturally this arrangement creates a chicken-and-egg problem in practice with respect to the SMD language in itself. This is circumvented by bootstrapping Coral with a hand-written SMD metamodel which is statically linked into Coral.

An interesting feature of the dynamic nature of the metamodelling layer is the concept of importing the contents of one metamodel into the namespace of another metamodel. This allows us to form hierarchies of metamodels. For example, a tool vendor uses its own namespace as the combination of UML 1.4 and XMI-DI 2.0. In Coral, this compatibility is achieved by creating the metamodels *separately* and then importing their contents into a third metamodel.

2.1 Mutually Independent Property Characteristics

In our opinion, the expressive power of metamodels does not come from the actual metaelements, but rather from the different characteristics of the interconnections between metaelements. An element's possible connections (*slots*) are described by its metaelement's *properties*. Two properties can be connected together to form a bidirectional meta-association.

In Coral, a property consists of several characteristics and describes the restrictions for each slot. Using a combination of characteristics several common constructs can be modelled, as well as more esoteric ones. It is important to notice that this part is static in Coral, i.e. users cannot change what characteristics are available. The various characteristics are:

- a *name* for convenience
- a *multiplicity* range $[l,u]$ defining how many connections to instances of the target the slot (instantiated property) should have to be well-formed. Common values are $[0..1]$ for an optional element, $[1..1]$ for exactly one, $[0..*]$ for any amount and $[1..*]$ for at least one element
- a *target*, telling what the type (metaelement) of every element in the slot must be
- a boolean *ordered* telling if the order of the elements in the slots is important and must be kept
- a boolean flag *bag* telling if the same element can occur several times in a slot
- a boolean flag *unserialisable* telling if corresponding slots should not be serialised when saving a model
- an optional *opposite*, giving the opposite property for bidirectional connections
- a link *type* enumeration value {association, composition } describing an ordinary connection or describing ownership, respectively.

The characteristics *unserialisable* and *anonymous* need more careful explanation. An unserialisable property means that the contents of the corresponding slots are not saved to an output stream. This is useful when elements in file A reference elements in file B, but without the elements in B having to know anything about file A. This occurs when creating models that resemble "plugins"; we are not sure what plugins are available and we do not want to change the main file every time something is added or removed. Instead, the available plugins are loaded at runtime and bidirectional connections are created, even though they are not serialised at both ends. Arguably the usefulness of the characteristic in this case is specific to the way current filesystems work using files as independent streams of bytes. A filesystem acting more like a database would not share the benefits from the unserialisable characteristic.

Anonymous properties provide fully bidirectional meta-associations between two metaelements, even though the meta-association was unidirectional at first. This is useful in cases where a language was not designed to be used together with another language. An example is a project management language (PML) keeping track of developers, bugs, timelines and several UML models. Since UML models do not know about PML, only unidirectional connections from PML to UML would be feasible, thus rendering any navigation from UML models to PML models impossible. But Coral automatically creates an anonymous property (with a private, nonconflicting name) at run-time from UML models to PML models, and thus it is indeed possible to navigate from any UML model to the corresponding PML model(s). The PML models can be saved in an XMI file separate from the UML file. Using the support from XMI for interconnecting model elements across files, the PML elements can still reference the UML elements.

Anonymous properties are necessarily also unserialised, since otherwise ordinary UML tools would not be able to read the model file with nonstandard slots. Anonymous properties provide an excellent way to combine models without changing the languages.

Most notably, the list is currently missing new characteristics from MOF 2.0, property *subsetting* and *derived unions*. These are important characteristics but have not been added to Coral yet. Otherwise, it is worth noting that the characteristics aim to be as mutually independent as possible. This has the benefit that very complex definitions can be modelled.

3 Metamodel Kernel API

These features of the Coral kernel can be accessed using a programming interface or API. The Coral kernel is implemented in C++, but we have created an interface for the Python programming language. It should not be difficult to support other programming languages such as Java, since the bindings for the specific programming languages are created using the SWIG tool.

Python is a highly dynamic expressive language which is easy to learn. Using Python, the interface to query models is very close to OCL [2], but with several methods added to also modify the model. Notably, model transformations can be written as Python programs with separate phases for preconditions, query and modification and postconditions. Support for transactions as well as checking of well-formed rules means that an illegal transformation can be rolled back, leaving the user with the original model. Examples of a rule-based model transformer can be found in [6].

Arbitrary scripts and well-formedness checks can be used to keep the design and evolution of a system within a predefined process or methodology. A success story is Dragos Truscan's work [7] on relations between data flow diagrams and object diagrams. It presents "an approach to combine both data-flow and object-oriented computing paradigms to model embedded systems." The work is fundamental for designing complex embedded systems since there is often a need to switch between the two paradigms. The design relies on an SA/RT metamodel for the data flow and the UML 1.4 metamodel for object and class diagrams. Python scripts are heavily used for the transformations between the domains.

In the future, using models also as the primary artefact for transformations using e.g. the upcoming OMG Query-View-Transform (QVT) [3] standard could be possible.

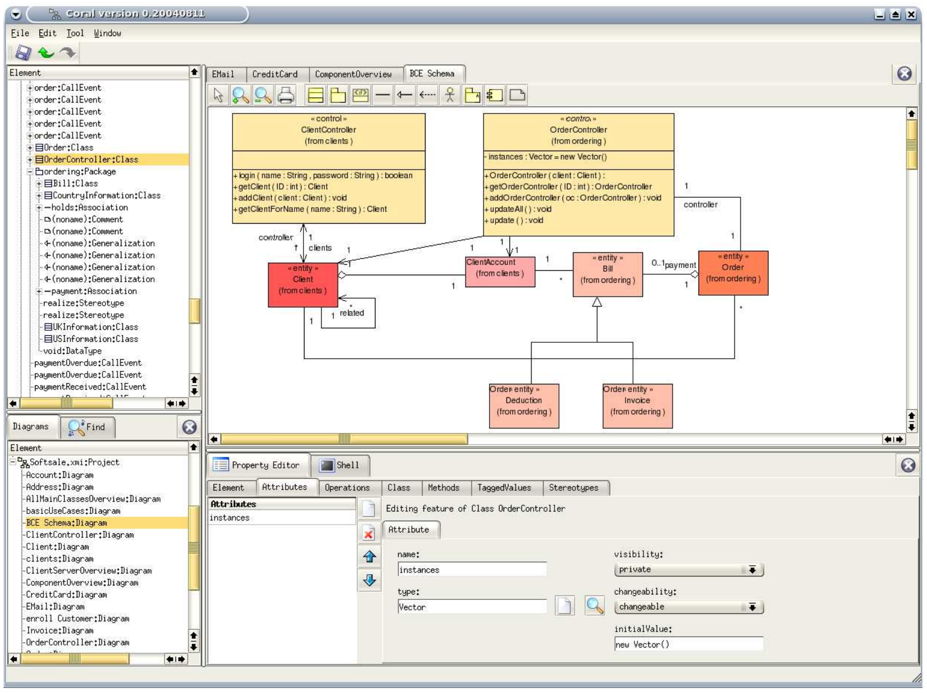
The scripting interface provides a highly flexible environment for automatic model generation, querying and transformation. Metamodels support predefined operations on specific elements, and there is no need to explicitly compile any scripts as they can be loaded on-the-fly from within Coral.

4 Conclusions

We have presented the Coral kernel, a metamodel-independent tool. In Coral, metamodels are first-class objects that can be created at runtime. This allows us to support new modelling languages without recompiling or even restarting Coral. Furthermore, we have made interesting progress in the dynamic combination of metamodels. Effort has been placed into making a Python-friendly interface to facilitate easy scripting for model transformation. However, it is possible to support other programming languages.

There are two ways to evaluate the Coral kernel: as a research tool or as a development tool. As a research tool, we are interested in a flexible software library that can be used to quickly create small prototypes to test new ideas in modelling technology. If we consider Coral a development tool we are interested in a robust and efficient library based on current standards.

Currently, Coral is definitely geared towards a research tool. The decision of using its own metamodelling language instead of MOF is an example of this. However, Coral is able to manage large models created using commercial tools efficiently. It supports the UML 1.1, UML 1.3, UML 1.4 and UML 1.5 metamodels. Model interchange can be accomplished using XMI 1.x, XMI 2.0 and XMI-DI [5] for diagram interchange.


Coral version 0.20040811
File Edit Tool Window
Element
EMail
CreditCard
ComponentOverview
BCE Schema
«control» ClientController (from clients)
+login (name : String , password : String) : boolean
+getClient (ID : int) : Client
+addClient (client : Client) : void
+getClientForName (name : String) : Client
«control» OrderController (from ordering)
- instances : Vector = new Vector()
+OrderController (client : Client) :
+getOrderController (ID : int) : OrderController
+addOrderController (oc : OrderController) : void
+updateAll ( ) : void
+update ( ) : void
controller
clients
related
«entity» Client (from clients)
ClientAccount (from clients)
«entity» Bill (from ordering)
0..1 payment
«entity» Order (from ordering)
Order entity » Deduction (from ordering)
Order entity » Invoice (from ordering)
Diagrams
Find
Property Editor
Shell
Element
Attributes
Operations
Class
Methods
TaggedValues
Stereotypes
Attribute

Editing feature of Class OrderController

Attributes: instances

name: instances
visibility: private
type: Vector
changeability: changeable
initialValue: new Vector()

Fig. 1. Coral can render diagrams stored in XMI-DI 2.0, for example models created with Gentleware's Poseidon. This screenshot shows the Softsale example model imported from Poseidon.

As an example of the compatibility and scalability of Coral, Figure 1 shows Coral rendering a UML 1.4 model with more than 20.000 model elements created with a commercial tool. The figure also reveals one of the interesting discoveries we had while developing Coral: Although the tool does not follow the OMG standards at the Meta-modelling level, it is fully compatible at the modelling level. That is, it is possible to develop a fully UML compliant editor and transformation tool without using MOF or the UML 2.0 infrastructure as basis for the tool.

Coral source code is available under an open source license at http://mde.abo.fi.

References

1. D. Akehurst, S. Kent, and O. Patrascoiu. A relational approach to defining and implementing transformations between metamodels. *Software and Systems Modeling*, 2(4), December 2003.
2. OMG. Object Constraint Language Specification, version 1.1, September 1997. Available at `http://www.omg.org/`.
3. OMG. MOF 2.0 Query / Views / Transformations RFP. OMG Document ad/02-04-10. Available at www.omg.org, 2002.
4. OMG. Meta Object Facility, version 2.0, April 2003. Document ad/03-04-07, available at `http://www.omg.org/`.
5. OMG. Unified Modeling Language: Diagram Interchange version 2.0, July 2003. OMG document ptc/03-07-03. Available at `http://www.omg.org`.
6. Ivan Porres. Model Refactorings as Rule-Based Update Transformations. In *Proceedings of the UML 2003 Conference*. Springer-Verlag, October 2003.
7. Dragos Truscan, João Miguel Fernandes, and Johan Lilius. Tool Support for DFD-UML Model-based Transformations. In *Proceedings of the ECBS 2004 Conference,* May 2004.

ADT: Eclipse development tools for ATL

Freddy Allilaire (freddy.allilaire@laposte.net) Tarik Idrissi (tarik.idrissi@laposte.net)

Université de Nantes
Faculté de Sciences et Techniques
LINA (Laboratoire d'Informatique de Nantes-Atlantique)
44322 Nantes Cedex 3, France

Abstract

This paper presents our work on the creation of an Integrated Development Environment (IDE) for ATL under Eclipse. ATL is a language for expressing model transformations. The ATL IDE proposes tools that may be found in traditionnal programming languages: syntax highlighting, code completion, wizard for the creation of project or debugger. These developed tools make it possible to create more easily transformations.

Keywords : MDA, ATL, IDE, Eclipse, Model transformation

1. Introduction

The MDA is an OMG initiative in response to the recent new software crisis. If the platform changes were rather rare before, they are much more frequent today. The idea of the MDA is that technologically neutral models could be implemented on a large variety of technologically different platforms in an automated way [1]. However tools supporting MDA basic operations are still rather rare. ATL proposes a partial answer to this situation. It is a language, engine and environment for model transformations developped by the ATLAS team in Nantes [2].

2. Concept

2.1 ATL

What is ATL ?

ATL stands for ATLAS Transformation Language. ATLAS is a recently created INRIA project located at the University of Nantes (LINA) and focusing on data-centric systems.

The ATL transformation language that is being developed by the ATLAS team is a hybrid language, i.e. a mix of declarative and imperative constructions designed to express model transformation as required by any MDA approach. It intends to answer the MOF/QVT RFP issued by OMG. It is described by an abstract syntax tree (a MOF meta-model) and a textual concrete syntax. Any transformation such as PIM (Platform Independent Model) to PSM (Platform Specific Model) can then be written in ATL.

What is an ATL transformation ?

An ATL transformation can be decomposed into three parts : a header, helpers and rules.

The header is used to declare general information such as the module name (it is the transformation name : it must match the file name), the input and output meta-model and potential import of needed libraries.

Helpers are subroutine that are used to avoid code redundancy. We can imagine a helper for translating UML visibility kind to MOF visibility kind in a UML to MOF transformation. It is easy to guess the utility of this helper when we know all UML and MOF elements that define a visibility.

Rules are the heart of ATL transformations because they describe how output elements (based on the output meta-model) are produced from input elements (based on the input meta-model). They are made up of bindings, each one expressing a mapping between an input element and an output element.

The ATL project is part of the GMT Eclipse project. That is why Eclipse has been chosen to be used as an IDE for ATL. In the next part of the paper we are going to present the Eclipse tools for ATL.

2.2 Eclipse

Eclipse was created by OTI and IBM teams responsible for IDE products. Eclipse [3] is a platform that has been designed for building integrated web and application development tooling. The value of the platform is that it encourages rapid development of integrated features based on a **plug-in** model. Eclipse has a wide community of tool developers.

Eclipse provides a common user interface model for working with tools. It is designed to run on multiple operating systems while providing robust integration with each underlying OS.

At the core of Eclipse there is an architecture for dynamic discovery, loading, and running of plug-ins. The platform handles the logistics for finding and running the right code. The platform UI provides a standard user navigation model. Each plug-in can then focus on a small number of well-defined tasks.

Open architecture

The Eclipse platform defines an open architecture so that each plug-in development team can focus on their area of expertise. If the platform is correctly designed, new features can be added without impact to other tools. Newly developed tools can plug into the workbench using well defined hooks called **extension points**.

The platform itself is built in layers of plug-ins, each one defining extensions to the extension points of lower-level plug-ins, and in turn defining their own extension points for further customization. This extension model allows plug-in developers to add a variety of function to the basic tooling platform. The platform manages the complexity of different runtime environments. Plug-in developers can focus on their specific task instead of worrying about these integration issues.

2.3 EMF

EMF [4] (Eclipse Modelling Framework) is a modeling framework and code generation facility for building tools and other applications based on a structured data model. EMF is used to define and implement structured data models. A data model is simply a set of related classes used to handle the data which you want to deal with in your application. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Models can be specified using annotated Java, XML documents, or modeling tools like Rational Rose, then imported into EMF. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications.

3. Tools supporting ATL transformation

The ATL project will progressively provide a complete environment based on Eclipse for developing, testing and using model transformation programs. The environment under Eclipse for ATL is called ADT (ATL Development Tooling). Currently, the following items have been developed:

- A project kind and a project creation wizard
- A text editor with syntax highlighting and bracket matching
- A content outline and a property view associated to the editor
- Smart icons for all ATL specific elements
- A builder associated to the project kind
- A perspective
- Compilation error report
- A debugger

The development of the following items has started and will be completed soon :

- Rename refactoring (in near feature)
- Execution error report
- Auto-completion and content assist for the editor

3.1 A project kind, a project creation wizard and a perspective

Let us start by explaining the concepts of project and perspective in Eclipse.

A project enables to group, organize and manage related elements: in a Java project we can find packages, their respective classes and an associated builder (Java builder) allowing to compile easily each Java class.

A perspective is a visual container for a set of views and editors : in the Java perspective there is among others things a package explorer view (to easily navigate through Java classes) and Java editors (each one editing a Java file with syntax highlighting and other advanced features).

We have defineed our own project kind named “ATL Project” with its specific icons. It is possible to create an ATL project by using the ATL project creation wizard.

An ATL project is automatically associated to the ATL builder. This makes building the ATL transformations contained by an ATL project quite easy: right click on the project and select build. The build operation can even be performed automatically at resource save time if the user activates the appropriate option in his preferences.

Others facilities are offered for an ATL transformation file: deleting or renaming it causes its associated compiled file to be deleted or renamed automatically. When moved in any other folder, its associated compiled file will be moved too to the same folder.

Moreover, ATL defines its own perspective made up of the ATL editor, the ATL outline associated to this editor, the problem view used to report compilation errors.

3.2 Editor

We know how useful can be an editor to help the developer in his task. That is why the first component developed for the ATL language was an editor. The features currently provided by the editor for ATL (that you can see in the middle of **figure 1**) are:

- Syntax highlighting fully customisable through preference pages; the user may change the colour of the editor background, ATL keywords , ATL types and so on.

- Bracket matching: when the user types in an opening bracket, the ATL editor automatically adds its closing and indents the content between those two brackets by offsetting.

The development of the auto-completion feature, that will allow the user for example to dynamically get all the available methods for a variable according to its type, has started and will be available soon. The final purpose is to make that editor support all major features provided by the Eclipse Java editor.

Editors often have corresponding content outliners that provide a structured view of the editor contents and assist the user in navigating through the content of the editor. Thus, like the Java editor, the ATL editor always gets an outline.

For the outline too, the purpose is to make it support all major features supplied by the Java outline. Presently the functionalities related to the outline are the following:

- Selecting an element in the outline causes the area of this element in the editor to be highlighted and the editor vertical rule to be updated (for example an item representing a helper will highlight this one in the editor and will make the editor vertical rule lying down from the helper starting line to its end line).

- When the editor cursor position changes, by scrolling up or down with the keyboard or by clicking with the mouse, the element corresponding to the editor current position is selected in the outline

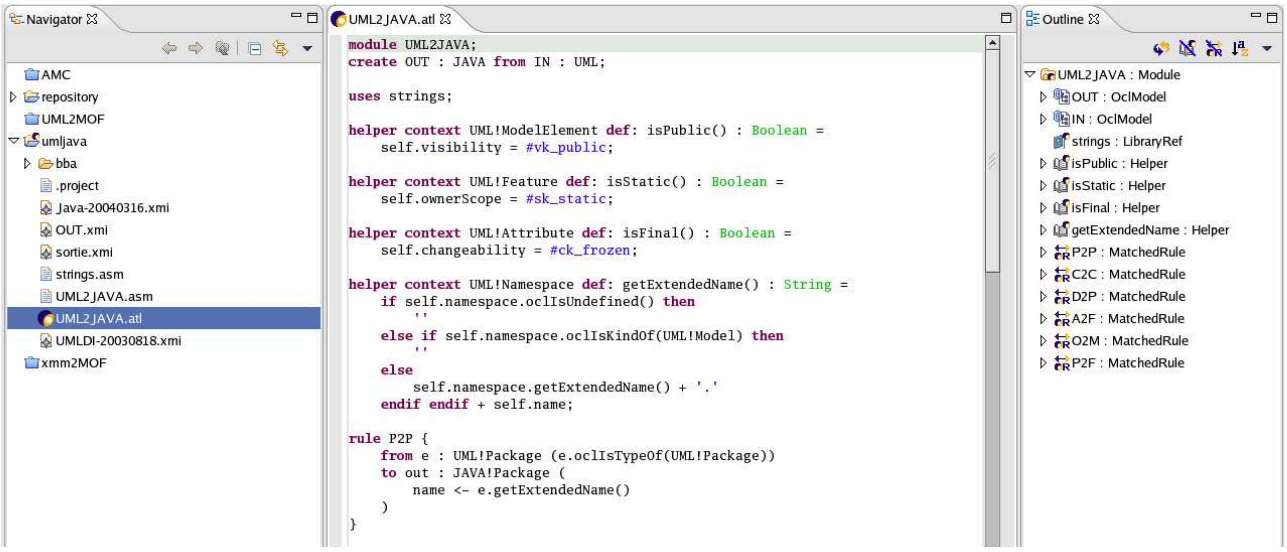
- Clicking on an object in the outline causes information such as its location, its name and so on to be displayed in the property view (in Eclipse a view is a frame used to display information user friendly and from which we can carry out actions).

- It is possible to sort and filter the content of the outline, e.g. hide helpers or rules

The specificity of the ATL outline is that while Java outline only contains the class, its attributes, methods and subclasses, ATL outline contains the whole Abstract Syntax Tree (AST). It is then possible to navigate through basic elements such as expressions, operators and so on. This is due to the fact that the ATL outline was not only designed to support traditional outline facilities but also to play an important role in the debugging task. This needs some explanations. Breakpoints are very important in the debugging process and in many languages as Java, breakpoints are placed on lines. In ATL, breakpoints may be placed on almost any element of the AST. And in that respect, the ATL outline will play an essential role. Indeed, adding a breakpoint can be done by right clicking on an element and selecting “A dd breakpoint”. This al lows fine grained control of the execution process.

The ATL outline (that you can see on the right of **figure 1**) is also designed to support rename refactoring. This feature is not completely available yet but the principle will be the same as for breakpoints: selecting an element in the AST, right clicking and selecting “rename” will open a frame enabling to carry out the refactoring.

The outline is designed according to the MVC (Model View Controller) pattern. As for the Model part it is not made up of traditional Java classes but is an EMF based Model. The ATL meta-model is loaded by EMF providing then tools that enable to deal easily with ATL model elements.



```
module UML2JAVA;
create OUT : JAVA from IN : UML;

uses strings;

helper context UML!ModelElement def: isPublic() : Boolean =
    self.visibility = #vk_public;

helper context UML!Feature def: isStatic() : Boolean =
    self.ownerScope = #sk_static;

helper context UML!Attribute def: isFinal() : Boolean =
    self.changeability = #ck_frozen;

helper context UML!Namespace def: getExtendedName() : String =
    if self.namespace.oclIsUndefined() then
        ''
    else if self.namespace.oclIsKindOf(UML!Model) then
        ''
    else
        self.namespace.getExtendedName() + '.'
    endif endif + self.name;

rule P2P {
    from e : UML!Package (e.oclIsTypeOf(UML!Package))
    to out : JAVA!Package (
        name <- e.getExtendedName()
    )
}
```

Figure 1 – ATL Editor with its content outline

3.3 Execution of transformation

In order to execute ATL transformations, there are two ways like for Java programs : run and debug.

Debug plug-ins of Eclipse allow to extend the plateform in order to launch correctly your program, with parameters of the user.

Before launching a transformation, one needs to create a launch configuration. One should specify parameters required for the execution of the transformation. Then it is possible to run a transformation using a generic ATL launch configuration that derives most of the launch parameters from the ATL project and the workbench preferences. It is however possible to override the derived parameters or specify additional arguments.

There is two different ATL launchers: ATL configuration and Remote ATL configuration.

ATL configuration

The ATL Configuration defines the transformation to be executed. This configuration needs the name of the project where your transformation is located, the name of your transformation and the model handler (for the moment MDR or EMF). This configuration needs also the name and the path of the models used in the transformation (INPUT MODEL, OUTPUT MODEL, INPUT METAMODEL, OUTPUT METAMODEL and LIBS). For the moment, the name of your model should be the same between configuration and ATL transformation file. But soon, thanks to the repository, this binding will be automatically achieved.

Example : *create OUT : Java from IN : UML;* in your transformation implies in this tab: the couple IN (model) and UML (metamodel) in the IN part and the couple OUT (model) and Java (metamodel) in the OUT part.

Currently, we are working on the repository where all metamodels will be stored, the metamodels needed will be found in the repository and link with launch configuration automatically made.

Remote ATL configuration

This launching configuration is useful when you want to launch the debuggee (the debugged program) out of Eclipse (for example, if your transformation needs injector or extractor, the ATL configuration cannot be used for the moment but it is possible with Remote ATL configuration). Otherwise, you can use the ATL configuration when you have a project with input and output models, input and output metamodels; it is easier to use than remote ATL configuration.

This configuration just needs the name of the project and the name of the ATL transformation and connection properties : hostname (for example: localhost) and port.

If all the parameters are given (project, transformation, model handler, input and output model), you can apply the configuration, else a message error at the top of the dialog displays what is wrong in your configuration (for example, the path of your model is empty).
When the configuration is completed, the launcher is ready to do its work. ATL Transformation can be runned or debugged.

Running your programs

In run mode, the program executes, but the execution may not be suspended or examined.

When the transformation has finished its work, output model has been created or updated. In the next version of the ADT, execution errors will be displayed in the Eclipse console.

Debugging your programs

In debug mode, execution may be suspended and resumed, breakpoints added, and you can see the variables of the stackframes.

There are several important views in the debug perspective.

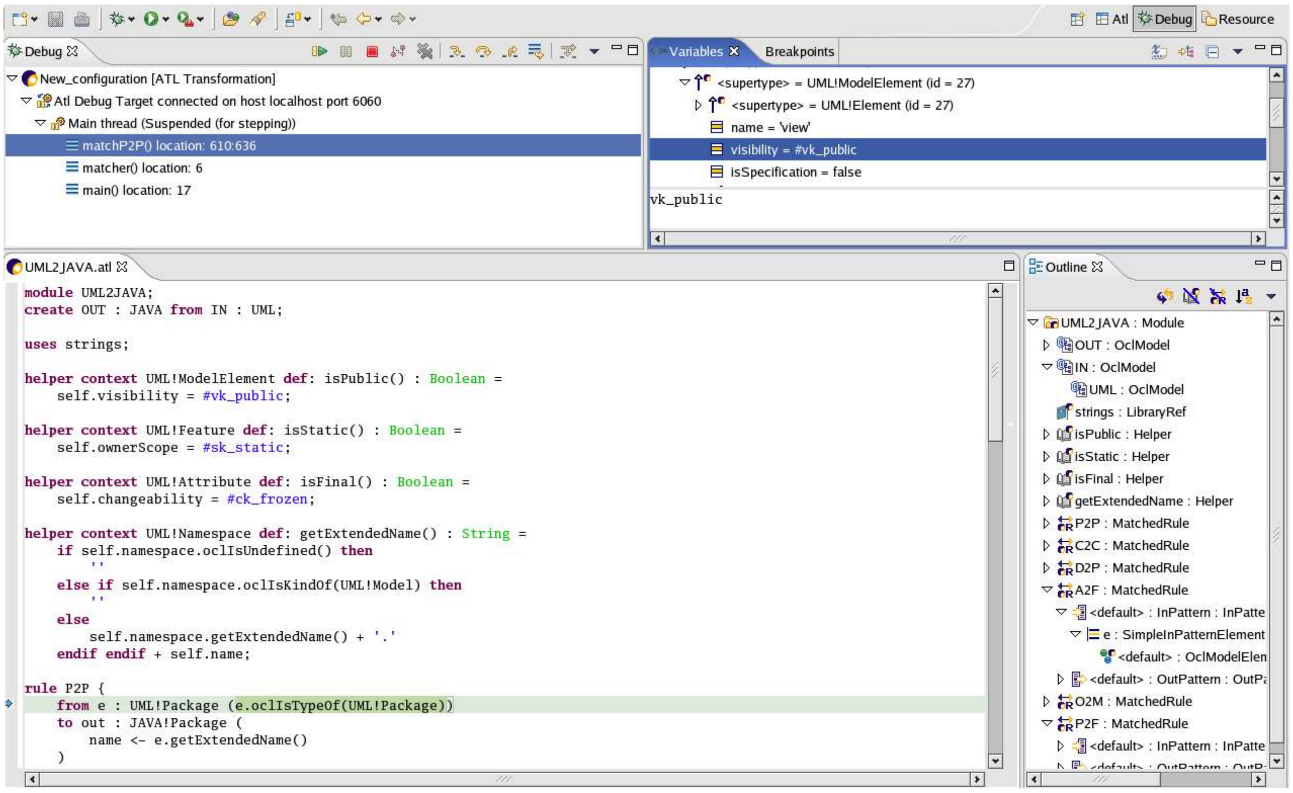
In the **Debug view** (top left of **figure 2**), you see the different configurations launched with their state. For each configuration, there is the debug target, threads and stackframes. At the beginning, the transformation is suspended. When the breakpoint is hit, execution is suspended. Notice that the process is still active (not terminated) in the Debug view.
You can step through the code with several action. With the action **Step Over**, the execution will continue at the next line in the same rule. You can also use **Step Into** or **Step Return** to step through the code. You can end a debugging with action Terminate, to step over the code until the transformation completes or **Resume** to allow the program to run until the next breakpoint is encountered or until the program is completed. When an exception is raised, you can see it on the debug view.

In the **Editor view** (bottom left of **figure 2**), you can see the code being debugged, the breakpoint. On each step, you see the highlighted text changed.

The **Variable view** (top right of **figure 2**) displays the values of the variables in the selected stack frame. You can expand the tree in the Variables view. The variables in the Variable view will change when you step in the Debug view.

The **Outline view** is useful in debug context to put a breakpoint. In fact, breakpoints are added with this view, they are added on the element selected.

The **Breakpoint view** lists all the breakpoints you have set in the workbench projects. Breakpoints can be enabled, disabled, deleted, added and located.



```
New_configuration [ATL Transformation]
  Atl Debug Target connected on host localhost port 6060
    Main thread (Suspended (for stepping))
      matchP2P() location: 610:636
      matcher() location: 6
      main() location: 17
```

```
<supertype> = UML!ModelElement (id = 27)
  <supertype> = UML!Element (id = 27)
  name = 'view'
  visibility = #vk_public
  isSpecification = false
vk_public
```

```
module UML2JAVA;
create OUT : JAVA from IN : UML;

uses strings;

helper context UML!ModelElement def: isPublic() : Boolean =
    self.visibility = #vk_public;

helper context UML!Feature def: isStatic() : Boolean =
    self.ownerScope = #sk_static;

helper context UML!Attribute def: isFinal() : Boolean =
    self.changeability = #ck_frozen;

helper context UML!Namespace def: getExtendedName() : String =
    if self.namespace.oclIsUndefined() then
        ''
    else if self.namespace.oclIsKindOf(UML!Model) then
        ''
    else
        self.namespace.getExtendedName() + '.'
    endif endif + self.name;

rule P2P {
    from e : UML!Package (e.oclIsTypeOf(UML!Package))
    to out : JAVA!Package (
        name <- e.getExtendedName()
    )
```

```
UML2JAVA : Module
  OUT : OclModel
  IN : OclModel
    UML : OclModel
  strings : LibraryRef
  isPublic : Helper
  isStatic : Helper
  isFinal : Helper
  getExtendedName : Helper
  P2P : MatchedRule
  C2C : MatchedRule
  D2P : MatchedRule
  A2F : MatchedRule
    <default> : InPattern : InPatte
      e : SimpleInPatternElement
        <default> : OclModelElen
    <default> : OutPattern : OutPa
  O2M : MatchedRule
  P2F : MatchedRule
    <default> : InPattern : InPatte
```

Figure 2 – ATL Debug perspective with the ATL Editor, Debug, Variable and Outline view

4. Conclusion

The goal of the ATL Development Tooling (ADT) is to simplify and assist the creative task of ATL programming and debugging. ATL is a transformation language and Eclipse allows us to develop an environment powerful and complete for ATL. But ADT should rapidly grow up because it will be necessary to meet more and more user requirements. For example, a repository will be created and accessible under Eclipse. Metamodels or transformations will be found in this MDA component repository. Issues of naming, remote access, encapsulation and typing will also be handled by this repository. Programming model transformations is a hard task, but we also have to seriously handle many other related tasks like reusing, specifiying and checking transformations. This paper has briefly presented the curent state of tools supporting transformation development and debugging, which is an important part of the work of the transformation programmer.

5. Acknowledgements

We thank Jean Bézivin, Frédéric Jouault and the members of the ATL Project for multiple advices and comments on this work.

Reference

[1] J Bézivin : From Object-Composition to Model-Transformation with the MDA. TOOLSUSA 2001, Santa Barbara, USA2001
http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda/TOOLS.USA.pdf
[2] J Bézivin, G Dupé, F Jouault, G Pitette, and E J Rougui : First experiments with the ATL model transformation language: Transforming XSLT into XQuery
http://www.softmetaware.com/oopsla2003/bezivin.pdf
[3] Eclipse Platform Technical Overview
http://www.Eclipse.org/whitepapers/Eclipse-overview.pdf
[4] Using EMF http://www.Eclipse.org/articles/Article-Using EMF/using-emf.html

Model-Driven Testing with UML 2.0

Zhen Ru Dai

Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
`dai@fokus.fraunhofer.de`

Abstract. The UML 2.0 Testing Profile provides support for UML 2.0 based model-driven testing. This paper introduces a methodology of how to use the profile in order to transform an existing UML system design model for tests. For the formalization of the proposed methodology, the QVT transformation rules defined by CBOP/IBM/DSTC are considered.

1 Introduction

The Model-Driven Architecture (MDA) is not only about system modelling throughout the abstraction levels in terms of platform independent system modelling, platform specific system modelling and system code generation [1, 2]. The MDA abstraction levels can also be applied to test modelling [3].

Due to increasing complexity of today's software systems, the early integration of testing into the development process becomes more and more important. By doing so, design mistakes and implementation faults can be detected in an early stage of the design process. This allows reduction of time and costs. Additionally, the developed tests can be executed against the developed system after it has been released to the customer in order to check its correct behavior in the customer's target environment.

The Unified Modeling Language (UML) is a visual language to support the design and development of complex object-oriented systems. With the growing system complexity the need for solid testing increases. But UML itself, even the newest version 2.0 [4, 5], provides no means to describe a test model. Thus, a UML 2.0 profile for the testing, called the *UML 2.0 Testing Profile* (U2TP) [6], has been defined which has become an official OMG standard since March 2004. U2TP bridges the gap between designers and testers by providing a means for using UML for both system modeling and test specification. This allows a reuse of UML design documents for testing and enables test development in an early system development phase.

According to the philosophy of MDA, the same modelling mechanism can be re-used for multiple targets [7]. Strict distinguishment should be made between platform independent and platform specific system models before generating executable system codes. Within these three abstraction levels, transformation techniques are applied. Similarly, test models can be specified platform independently and platform specific before generating executable test codes. Researches have been made on transformation between the different system or test development abstraction levels (vertical arrows in Figure 2) [8–10]. But only few research has been done for the transformation between system models and test models (horizontal arrows in Figure 2).

In this paper, we introduce a methodology of how to apply U2TP concepts to an existing UML system design model effectively in order to retrieve a test design model. The methodology is concretized by transformation rules which are formalized in Query/View/Transformation (QVT) rules defined by CBOP/IBM/DSTC [11].

The paper is structured as follows: After a short introduction about model-driven testing and UML 2.0 Testing Profile in Sections 2 and 3, the methodology is provided in Section 4, where different test aspects of UML 2.0 Testing Profile are discussed. In Section 5, a transformation example is outlined. Section 6 summarizes and concludes this paper.

2 Approaches to Model-Driven Testing

The philosophy of MDA can be applied both on system modelling and test modelling. As shown in Figure 1, platform independent system design models (PIM) can be transformed into platform specific system design models (PSM). While PIMs focus on describing the pure functioning of a system independently from potential platforms that may be used to realize and execute the system, the relating PSMs contain a lot of information on the underlying platform. In another transformation step, system code may be derived from the PSM. Certainly, the completeness of the code depends on the completeness of the system design model.

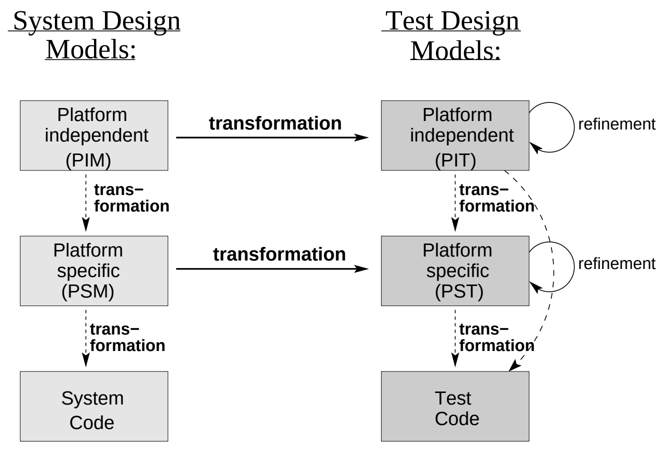


Fig. 1. System Design Models vs. Test Design Models

The same abstraction in terms of platform independent, platform specific modelling and system code generation can be applied to test design models.

Furthermore, test design models might be transformed from system design models directly. This enables the early integration of test development into the overall development process. Once the system design model is defined at PIM level, a platform independent test design model (PIT) can be derived. This model can be transformed either directly to test code or to a platform specific test design model (PST) [12]. The same transformation technology can be used for deriving PSTs from the PSM. After each transformation step, the test design model can be refined and enriched with test specific properties. Although the transformed test design model may already contain static and dynamic aspects, the behavior has to be completed in order to cover unexpected system behavior as well. Also, test issues such as e.g. test control and deployment information has to be manually added to the test design model. At last, the test design model can be finally transformed into executable test code from either PST or PIT.

3 The *UML 2.0 Testing Profile* (U2TP)

The UML 2.0 Testing Profile provides concepts to develop test specifications and test models for black-box testing [13]. The profile introduces four logical concept groups covering the aspects [6]: *test architecture, test behavior, test data* and *time*. Together, these concepts define a modeling language for visualizing, specifying, analyzing, constructing and documenting a test system. In the following, the U2TP concepts are introduced (Figure 2).

Test Architecture Concepts One or more objects can be identified as the *System Under Test* (SUT). *Test components* are objects within a test system which can communicate with the SUT or other components to realize the test behavior. The *test context* allows users to group test cases, to describe a corresponding *test configuration*, i.e. the connection between test components and the SUT, and to define the *test control*, i.e. the required execution order of the test cases. Arbitration is a means for evaluating an overall verdict for a test context. A tester can either use the default arbitration or define their own arbitration scheme using an *arbiter*. The *scheduler* controls the test execution and test components. It is responsible for the creation of test components, a synchronized start of the different test components, and the detection of test case termination.

Test Behavior Concepts A *test objective* defines the aim of a test. Herefore, UML Interaction Diagrams, such as State Machines and Activity Diagrams can be used to define test stimuli, observations, test control/invocations, coordination and actions. The normative test behavior is specified in a *test case*, which is an operation of the test context specifying how a set of co-operating components interact with the SUT to realize a test objective. When normative test behavior is defined, focus is given to the definition of unexpected behaviors which is achieved through specification of *defaults*. A *validation action* is performed by a local test component to inform the arbiter about its local test verdict. A test *verdict* shows the result of the executed test. Possible test verdicts are `pass`, `inconclusive`, `fail`, and `error`.

Test Data Concepts In the UML 2.0 Testing Profile, *wildcards* are used to handle unexpected events, or events containing many different values. The profile introduces wildcards allowing the specification of: (1) Any value and (2) Any or omitted values. *Data pools* are associated with test context and include concrete test data. *Data selectors* are operations to retrieve test data from the data pool or data partitions. The notion of *coding rules* allows the tester to define the encoding and decoding of test data when communicating with the SUT.

Time Concepts The time concept group defines concepts to constrain and control test behavior with regard to time. *Timers* are needed to manipulate and control test behavior as well as to ensure the termination of test cases. *Time zones* are used to group components within a distributed system, allowing the comparison of time events within the same time zone.

Test Architecture Concepts	Test Behavior Concepts	Test Data Concepts	Time Concepts
▨ SUT	▓ Test objective	▓ Wildcards	▦ Timer
▦ Test component	▨ Test case	▦ Data pool	▓ Time zone
░ Test context	▦ Defaults	▦ Data partition	
▦ Test configuration	▓ Validation action	▓ Data selector	
▦ Test control	░ Verdicts	▓ Coding rules	
Arbiter			
Scheduler			

░ Mandatories	▨ Derivable Mandatories	▓ Optionals	▦ Derivable Optionals

Fig. 2. U2TP Concepts & A Methodology on Test Design Model Development

4 A Methodology on Model-Driven Test Development

In this section, we introduce a methodology for using the UML 2.0 Testing Profile effectively after having received a detailed system design model which is to be tested [14]. In the following, we determine *system design model* to be the UML 2.0 system model in UML and the *test design model* to be the UML 2.0 model using U2TP concepts.

Having a system design model, a tester may have to specify tests for the system. This can be done by extending the system design model with U2TP concepts. The following aspects must be considered when transforming a system design model into a test design model:

First of all, define a new UML package as the test package of the system. Import the classes and interfaces from the system design package in order to get access to messages and data types in the test specification. Next, start with the specification of the *test architecture* and continue with *test behavior* specifications. Test data and time are mostly already comprised in either the test architecture (e.g. timezone or data pool) or test behavior (e.g. timer or data partitioning) specifications.

Below, issues regarding test architecture and test behavior specifications are listed. They are subdivided into two categories: mandatory issues and optional issues (Figure 2 on page 4). *Mandatories* are issues which are essential for a test design model with U2TP. The most important mandatory issues are e.g. SUT and test components. *Optional* issues are specific to test requirements and are therefore not always needed for the test design model specification. Optional issues are e.g. test control and timers. Additionally, there are both mandatory and optional concepts which can be derived directly from existing system design diagrams[1].

In the following, the mandatories and optionals are listed and possible derivations outlined. A test design model based on U2TP may use all UML diagram types for test specification. Depending on the given system design diagram types, different test design diagram types can be transfered. Therefore, in the methodology, we also point to the diagram type feasable for the derivations. These derivations are used for the test design model transformation in Section 5:

1. Test architecture:
   i. Mandatory:
      - Assign the classes (in a Class Diagram) or objects (in an Object Diagram) you would like to test to *SUT* class/object.
      - Specify a *test context* class listing the test attributes and test cases, also possible test control and test configuration.

   ii. Optional:
      - Depend on their functionalities, test components have to be defined. Group the classes/objects (except the SUT) to *test component* classes/objects. Test components are not needed in unit tests.
      - In order to define the ordering of test case execution, specify the *test control*. If there are Activity Diagrams given in the system design model, each activity illustrates one test case and the activity flow describes the test flow in the test control specification. If there are Use Case Diagrams provided, each use case depicts one test case which should be stringed together for the test control specification. If neither Activity Diagram nor Use Case Diagram exist in the system design model, string the test cases together for the test control specification. In a more complex test control specification, loops and conditions should also be used.
      - *Test configuration* are easily retrieved by means of existing Interaction Diagrams. Whenever two components exchange messages with

[1] A detailed case study on the methodology can be found in [14].

each other, assign a communication channel between the components. If there is no Interaction Diagram provided, connect the test components and SUT to an appropriate *test configuration* so that the configuration is relevant for all test cases included in the test suite.
    - Assign *timezones* to the components if the test system is a distributed system.
    - Provide *coding rule* information.

2. Test behavior:
   i. Mandatory:
    - For the specification of *test cases*, take given Interaction Diagrams from the system design model. Change (i.e. rename or group) the instances and assign them with stereotypes according to their roles (i.e. test component or SUT). If there are Use Case Diagrams or Activity Diagrams provided in the system design model, the use cases and activities are specified in additional Interaction Diagrams. Thus, for each use case or activity, a test case should be specified.
    - Assign *verdicts* at the end of each test case specification. Usually, the verdict in a test case is set to pass.

   ii. Optional:
    - Define *test objectives* for each test case that is to be specified.
    - System behavior which are not used for the tests should be taken for *default* specifications. Herefore, Interaction Diagrams like Sequence Diagrams, State Machines or Activity Diagrams should be used. Use *wildcards* to catch unexpected behavior. Verdict settings in a default are either fail or inconclusive.
    - *Timers* should be derived from time constraint specifications within a Sequence Diagram or State Machine.

UML 2.0 Testing Profile provides default arbitration and scheduling mechanisms which by default should be implemented by the tool vendor. Additionally, the profile also provides the tester the means to specify his own arbiter and scheduler. To do so, the tester needs additional diagrams in order to describe the behavior of the arbiter and the scheduler. Furthermore, the tester should also consider the modification in the whole test architecture.

5 Test Design Model Transformation

Figure 3 shows the meta-model based transformation for the test design model transformation. Herein, the source meta-model is the UML meta-model and the target meta-model is the U2TP meta-model. In the methodology (Section 4), classes and objects are grouped together in order to define test components or SUT. Such mechanisms cannot be performed by transformations. Thus, for our transformation approach, we have to define those mechanisms in order to provide the tester a means to group or delete elements[2], reference test behavior fragments etc. These mechanisms are called test directives and its meta-model is the *Test Directive Meta-Model.* Transformation rules are applied on both the UML meta-model and the Test Directive Meta-Model to create an instance of the U2TP Meta-Model. All three meta-models are based on MOF.

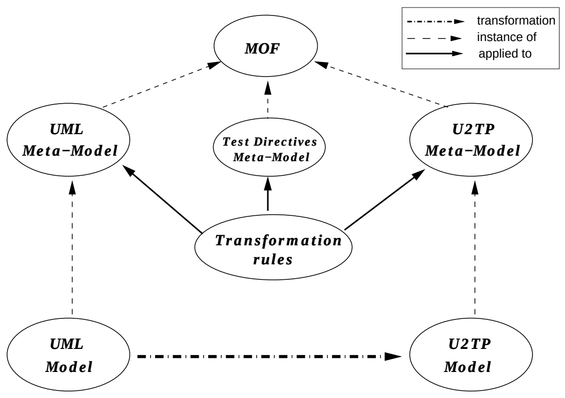


Fig. 3. Meta-Model Based Transformation

The transformation of the UML model to U2TP model is specified by a set of rules defined in the transformation meta-models [15] according to the QVC specification from CBOP/IBM/DSTC[11]. The introduced transformation language is aspect-oriented, declarative and pattern-based. It shows concepts for specification of rules, patterns and tracking relationships. *Transformation rules* are used to describe a correspondence between patterns of elements in the source model(s) and the elements to be created in a target model. *Patterns* are reusable definitions. When used in the source of a rule, a pattern is a query. When it is used in the target, it acts as a template for model elements. *Tracking relationships* associate the source model elements with the target model elements.

In the following, we will show a small example of our test design model transformation: Let us assume that we have an existing Object Diagram from the system design model and want to perform system test on this model. For the transformation for test components, the methodology says (in the test architecture optionals in Section 4): *Depend on their functionalities, test components have to be defined. Group the objects (except the SUT) to* test component *objects.* Thus, besides the Object Diagram, we also need a grouping mechanism, which should be provided by the Test Directives Meta-Model. A grouping mechanism is applied to at least two objects in the diagram.

Figure 4 shows how the transformation can be performed on instance level. On the left upper corner, a UML package with three objects is shown. In the left lower corner, the relationship between the objects which should be grouped

[2] Elements are UML elements such as classes, objects, instances etc.

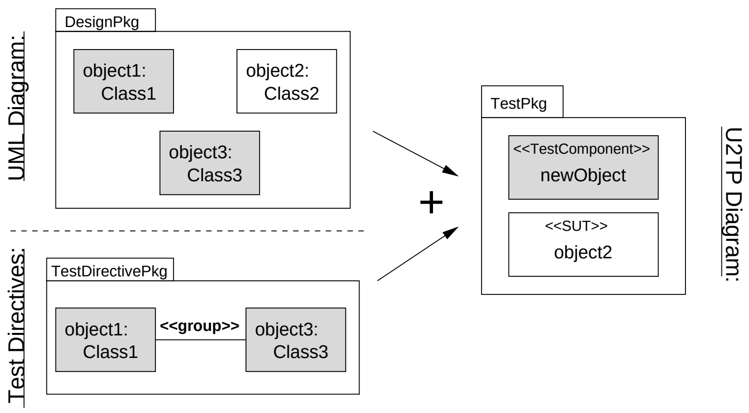


Fig. 4. Test Component Transformation

to a test component is specified in a test directives model. The grouping notation is an association between the objects with the stereotype <<group>>. In this example, only object1 and object3 should be grouped into one test component. Therefore, after the transformation in this example, the output test model consists of one test component and one SUT instance. Of course, two test components could also be specified, depending on the choice of the transformation rules. The stereotypes <<TestComponent>> and <<SUT>> are U2TP notations. By performing appropriate transformation rules on the different system design diagrams, test architecture and behavior can be specified for the test design model.

6 Summary and Outlook

In this paper, we have presented a methodology of how to derive a U2TP test design model from an existing UML system design model. Furthermore, the methodology can be formalized by defining transformation rules from system design diagrams to test design diagrams. For the transformation, we chose the QVT specification from CBOP/IBM/DSTC.

The definition of the transformation rules is not fully completed. Thus, we shall complete this work first. Unfortunately, due to lacking tool support for UML 2.0 and U2TP at time, we are not able to proof our model transformation rules. Thus, in our future work, we plan to investigate in tools which support the U2TP concepts and automated derivation of test design models from system design models.

Acknowledgement

Many thanks to Mr. Keith Duddy from DSTC for the various intensive discussion sessions.

References

1. OMG: (Model-Driven Architecture (MDA)) http://www.omg.org/mda/.
2. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture–Practice and Promise. Addison-Wesley Pub Co (2003)
3. Gross, H.: Testing and the uml – a perfect fit. Technical report, Fraunhofer IESE Report 110.03E (2003)
4. http://www.omg.org/uml.
5. Born, M., Holz, E., Kath, O.: Softwareentwicklung mit UML 2. Addison-Wesley (2004)
6. U2TP Consortium: UML 2.0 Testing Profile. (2004) Final Adopted Specification at OMG (ptc/04-04-02).
7. Siegel, J., the OMG Staff Strategy Group: Developing in omg's model-driven architecture. OMG white paper (2001)
8. OMG: MDA Guide Version 1.0. (2003)
9. Bézivin, J.: From object composition to model transformation with the mda. In: IEEE TOOLS-39, Santa Barbara, USA, TOOLS (2001)
10. Born, M., Schieferdecker, I., Gross, H.G., Santos, P.: Model-driven development and testing – case study (2003) http://www.fokus.fraunhofer.de/mdts/.
11. CBOP/DSTC/IBM: MOF Query/Views/Transformations, 2nd Revised Submission (ad/04-01-06). OMG. (2004)
12. Schieferdecker, I., Din, G.: A meta-model for ttcn-3. 1st International Workshop on Integration of Testing Methodologies (ITM 2004) (2004)
13. B.Beizer: Black-Box Testing. John Wiley & Sons, Inc (1995)
14. Dai, Z.R., Grabowski, J., Neukirchen, H., Pals, H.: From Design to Test with UML. Testing of Communicating Systems (Editors: R. Groz and R. Hierons) – 16th IFIP International Conference, TestCom2004, Oxford, Proceedings. Lecture Notes in Computer Science (LNCS) 2644, Springer, pp. 33-49 (2004)
15. Duddy, K., Gerber, A., Lawley, M., Raymond, K., Steel, J.: Model Transformation: A declarative, reusable patterns approach. (In: 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)) pp. 174–185

Model Abstraction versus Model to Text Transformation

Jon Oldevik, Tor Neple, Jan Øyvind Aagedal

SINTEF Information and Communication Technology, Forskningsvn 1, N-0314 Oslo, Norway
{jon.oldevik|tor.neple|jan.aagedal}@sintef.no

Abstract. In this paper we discuss the principal differences between model to model transformation and model to text transformations, and sketch how these are pertinent to different model abstraction levels. We also try to clarify characteristics of model abstraction levels. Finally, we emphasize the role of model to text transformations in model-driven development.

1 Introduction

In system development the main goal of the activity is production of a running system. At this point in time, the most important assets for the running system are the developed or generated code that is compiled and executed. The importance of this has been recognized also by OMG, as signalled by the MOF Model to Text Transformation RFP [1]. This can also be seen in the tools that support model driven engineering today; transformation between model abstraction levels has not been in focus so much as support for generating implementation code. Within the current OMG MDA® regime, this has changed. The main focus has been the QVT standardisation for supporting transformations between models. We now need to address how the model to text transformation should integrate with this and provide a convenient and standard way of generating code from models on different levels.

2 Model Abstraction Levels

When looking at the MDA Guide [2], it is clear that OMG's current visions of model-driven architecture are quite open for interpretation. Its specific focus on different abstraction levels, the Computational Independent Model (CIM), Platform Independent Model (PIM) and the Platform Specific Model (PSM), does not reflect a natural distinction of abstraction levels. In principle, these can be models at any abstraction level, depending on your definition of platform.

A very stringent view of this is to consider platform-independent models to *be any models that still have variability with respect to the target execution platform.* A common interpretation of the platform-specific model is based on a loose definition, in the area of *a model that has implications related to a specific implementation technology, such as EJB.*

So, the separation between PIM and PSM points out some open issues. When does a PIM become a PSM? What is the difference between a PSM and the code representing the system? As long as there is no real way of executing a UML model, the most accurate model of the system will be the implementation code.

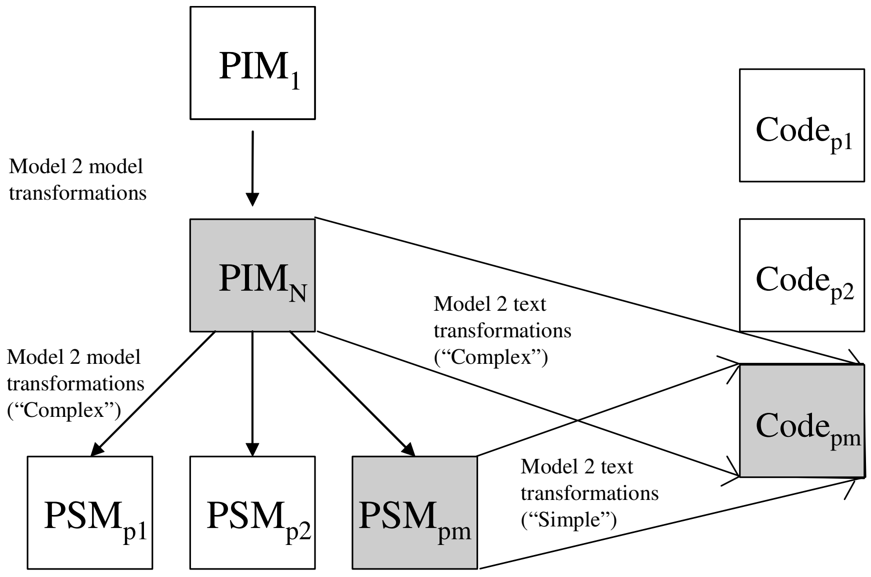

PIM_1
Model 2 model transformations
PIM_N
Model 2 model transformations ("Complex")
Model 2 text transformations ("Complex")
Model 2 text transformations ("Simple")
PSM_{p1}
PSM_{p2}
PSM_{pm}
$Code_{p1}$
$Code_{p2}$
$Code_{pm}$

Figure 1 Transformations – From Model to Text

As illustrated by Figure 1 the transformation from model to text can be achieved from different abstraction levels. It is possible to generate code from a quite high-level architecture model, which can be considered a PIM. The transformation logic then will be of high complexity to bridge the detail gap from the PIM to the code. On the other hand, it is possible to create a platform specific model, based on the PIM and then generate textual code from the PSM. The complexity will then be in the model transformation between the PIM and the PSM. The mapping from the PSM to the code should be a simple matter, since the PSM and the code should be closer to semantic isomorphism.

Textual transformations should be possible from any model abstraction level. For a transformation architect, the challenge is to find the appropriate level, and to design the transformations. The complexity of transformations will increase proportionally to the abstractness of the models.

So, the exact timing, or level of model detail appropriate for transforming from PIM to a PSM rather than PIM to text is not given. Whatever level chosen, the transformation to text needs to be done at some point.

3 From Model to Text

The process of transforming a model to text can be required from any model abstraction level in a system development process; from business models, requirement models, high level architecture models, or detailed design models.

Today, there are numerous approaches for achieving this, implemented in different case tools, MDA tools, etc. In practice, however, the way of doing this is more or less the same:

- There is some kind of implicit or explicit representation of a metamodel (such as the UML metamodel)
- There is some kind of imperative language, such as a scripting or programming language, to write text generators within.

In case tools, there is typically an internal UML metamodel and a specific scripting language for writing text generators. For example, in Poseidon, they use the Velocity Template Language (VTL) to access the internal Java UML API. In IBM Rational Rose, they use Visual Basic extensions to access the internal Rose UML metamodel. Other approaches, such as in UML Model Transformation Tool (UMT), use externalized MOF/UML data on XMI form and some implementation language, such as XSLT, Java, VTL or other for text generation.

Consequently, we can see that there are many scripting languages used, such as XSLT, NiceXSL, Java Server Pages, Jython, VTL, JET, etc. They all target more or less the same problem area, although they have different strengths and weaknesses. The motivation for the MMTT RFP is to try to reach a more standardised way of achieving this, and hopefully leverage tool interoperability. We see today that models are used to generate all kinds of textual output, from different kinds of models. For example, requirements documentation and test cases from requirements models, API and system documentation from design models, as well as implementation code. So, we are looking for a standard that can provide generation of not only source code, but also text for humans.

4 The MMTT Language

A standardised language for model to text transformation needs to have a certain set of characteristics. There will be tradeoffs between expressive power and ease of use. One could argue that writing model to text transformations is not something that will be done by every developer, in fact probably only one or a few persons in a development department will have to deal with this task. Some will probably only use the standard generation scripts for standard platforms that come with the MDA tools. So, the ease of use issue should probably not be considered a limiting factor, but as simple as possible is still the preferred path.

The RFP asks for a language for transforming MOF models to text, which reuses the QVT language specifications. The resulting QVT standard language is therefore the natural extension point.

It is therefore natural to look into what is needed in addition to what is in the QVT proposal (QVT-Merge proposal [3]). QVT will include OCL expressions and the ability to create complex queries for model elements. It lacks, however, the ability to produce text output to files. Creating a specialisation of QVT *TransformationRule*, similar to a *Mapping*, except with the ability to create output, is one possibility.

Creating a "wish-list" for MMTT is only hard in having to try to make some limitations. For instance, there are many features in programming languages such as Java or in the surrounding libraries that would be useful, but we cannot standardize all of these. In the following we have tried to list the features that we mean are needed in model to text transformations:

- The ability to produce output to files from model elements. The language should allow for multiple target files from a model element.
- The ability to iterate model element sets. This is already an integral part of QVT.
- The ability to manipulate text, i.e. text functions such as strcmp, toUpper, toLower, strcat, substring. Some of these are already part of OCL and thus QVT.
- The ability to write and reuse functions. This is directly supported in QVT since *TransformationRule* is a subtype of *Operation*.
- The ability to interact with system services or libraries, for instance getting hold of the current date and time. This can be supported by extending the standard library of QVT with some extended necessary functionality.
- Support for parameterized transformations, where parameters will be provided at transformation time, for instance definition of package names in the Java language.
- The ability to include boilerplate text (such as copyright statements) into the resulting text files

Roundtrip engineering and reverse engineering issues are optional requirements in the MMTT RFP. However, these are indeed essential. These issues are related to MMTT, but are somewhat a different matter. However, the need for tracing what model element has generated what text-artefact is clear in a round-trip engineering context, and is a part that should be handled by the MMTT technology. The QVT standard library operations *markedAs* and *markValue* can be used to support some aspects of traceability.

The optional requirement of detecting and handling hand made changes in target files is more closely related to the subject. One may argue that this is outside the scope of a language for model to text transformation, but it is an essential requirement to a tool providing transformation capabilities.

Figure 2 shows a possible extension to the QVT-Merge submission. Here, we have added a *TextTransformation*, which specializes the *Mapping* class from the metamodel. In addition, it overrides the *body* AssociationEnd, which can be a *CompoundExp*. We have also introduced a *FileExpression* class, which provides a context for output to a file.

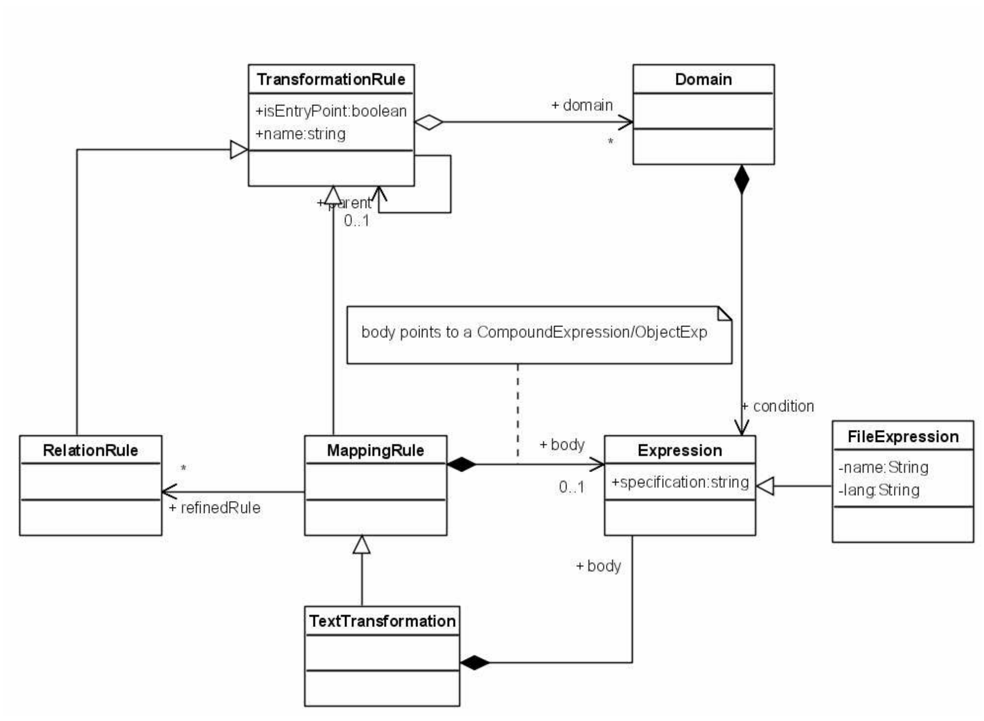


Figure 2 QVT-Merge metamodel specialisation

In the example below, a possible concrete syntax for the *TextTransformation* is shown. In this example, there are two file contexts within the TextMapping, providing two Java file outputs for each class in the source model. An important issue is how to standardise this part of the language. A possible approach is to allow hooks for embedding different languages within the MMTT. This is similar to how scripting is supported within HTML. As the simple example below shows, text mappings easily become complex and hard to read. In order to get the expressiveness needed, a language based on OCL-like constraints will not be sufficient. Additional imperative language constructs are needed.

```
Textmapping Simple_Class_to_Java
{
  domain { (SM.Class)[name = n, attributes = A] }
  body {
     let package_postfix = "qvt.org";
     let output_dir = "c:\test"
     file f [dir=output_dir, fname=n, ext="java", lang="MOFScript"] {
        print ("public class" + fname + "{");
        A->iterate(a | Simple_Attribute_To_Java (a, this));
        print ("}");
     }
}
TextMapping Simple_Attribute_To_Java
{
  domain {(SM.Attribute)[name = n, type = t]}
  domain {(File f)}
  body {
    f.print ("private " + Simple_Type_to_Java_Type(t.getName()) +
            " " + n.toLower() + ";")
  }
}
String Simple_Type_To_Java_Type (String typename){
   // Need logic to check different values and return different
   // type names based on this.
}
```

If MMTT standardises one language, lets say MOF Scripting Language (MOFScript), and provides extension points for other languages, this would lead to a flexible model.

5 Conclusion

Since it is really not defined how platform-specific a PIM needs to be to become a PSM, MMTT technology needs to have enough power to perform quite complex transformation tasks.

MMTT and QVT are closely related by topic, and the QVT language will provide many of the needed MMTT features. It is therefore natural to use the QVT standard as an extension point for creating MMTT. The additional features needed are already present in the group of transformation languages used in tools today; the key issue will be to define the set of wanted features for standardisation.

There will be tradeoffs between language usability and expressiveness. At one point, model transformation writing is not for everyone to worry about. On the other hand, if it gets too complex, no one will use it, leaving the whole process of standardisation pointless.

Acknowledgements. The work reported in this paper is carried out in the context of MODELWARE, an EU IP-project in FP62003/IST/2.3.2.3.

References

1. *MOF Model to Text Transformation Language - Request For Proposal*. 2004, Object Management Group: Needham,http://www.omg.org/cgi-bin/apps/doc?ad/04-04-07.pdf.
2. Miller, J. and J. Mukerji, eds. *MDA Guide Version 1.0.1*. 2003, Object Management Group: Needham.
3. QVT-Merge Group, *Revised submission for MOF 2.0 Query/Views/Transformations RFP*. 2004, Object Management Group,http://www.omg.org/cgi-bin/apps/doc?ad/04-04-01.pdf.

MOLA Language: Methodology Sketch

Audris Kalnins, Janis Barzdins, Edgars Celms

University of Latvia, IMCS, 29 Raina boulevard,
Riga, Latvia
{Audris.Kalnins, Janis.Barzdins, Edgars.Celms}@mii.lu.lv

Abstract. The paper demonstrates the MOLA transformation program building methodology on an example. The example shows how to obtain self-documenting model transformation programs in MOLA by means of standardized comments. The proper usage of loops in MOLA is also discussed.

1. Introduction

There is no doubt that model transformation languages and tools are the key technology elements for MDA. Due to OMG initiatives, currently there are several proposals for model transformation languages, both as responses to OMG QVT RFP [1,2] or "independent" ones [3,4]. Among the independent languages there is also the MOLA language proposed by the authors of this paper [5,6]. Each of the proposed languages has its strengths and weaknesses, there is no clear adoption of any of the languages in the MDA community yet. The main distinguishing feature of MOLA is a natural combination of traditional structured programming in a graphical form with pattern-based rules. Especially, the rich loop concepts in MOLA enable the iterative style for transformation definitions, while most of other languages rely on recursion. A more detailed comparison of MOLA to other MDA languages is provided in [5,6].

Transformation languages have two essential requirements. On the one hand, transformations should be easy to write – to implement the intended algorithms in an adequate manner. On the other hand, transformations should be easy readable by much broader user community – those wanting to apply a transformation to their models in a safe and controllable manner. Transformation readability has been one of the design goals of MOLA.

The only way to evaluate different languages is to compare them on generally accepted benchmark examples. Since transformation development actually is a completely new domain, there are no proven methodologies and design patterns, as there are in more classical domains.

The goal of this paper is to analyze the MOLA language from the above-mentioned perspectives. Using one of the standard benchmark examples - Class to Relational Database transformation, it will be shown how the readability can be achieved in MOLA, including also standardized comments. Transformation design methodology will also be sketched, especially the proper use of loops. Certainly, the paper does not claim to provide a methodology for MOLA-based system design, just some advices how the model transformations themselves should be programmed in MOLA.

2. Brief Overview of MOLA

This section gives a very brief overview of the MOLA language. A more complete description of MOLA is to be found in [5,6]. Authors also hope that the example in section 4 will help significantly to understand the language.

A MOLA program, as any other transformation program, transforms an instance of **source metamodel** into an instance of **target metamodel**. These metamodels are specified by means of UML class diagrams (MOF compliant).

More formally, source and target metamodels are part of a transformation program in MOLA. But the main part of MOLA program is one or more MOLA diagrams (one of which is the main). A **MOLA diagram** is a sequence of **graphical statements**, linked by arrows. It starts with a UML start symbol and ends with an end symbol.

The most used statement type is the **loop statement** – a bold-lined rectangle. Each loop statement has a **loop head** – a special statement (grey rounded rectangle) containing the **loop variable** and the **pattern** – a graphical condition defining which instances of the loop variable must be used for iterations. The pattern contains **elements** – rectangles containing `instance_name:class_name` – the traditional UML instance notation, where the class is a metamodel class. The loop variable is also a special kind of element, it is distinguished by having a **bold-lined** rectangle. In addition, a pattern contains metamodel associations – a pattern actually corresponds to a metamodel fragment (but the same class may be referenced several times). Pattern elements may have attribute constraints – OCL expressions. Associations can have cardinality constraints (e.g., NOT). The semantics of this loop statement (called the **FOREACH** loop) is natural – the loop is executed **once** for each instance of the loop variable, where the condition is true – the pattern elements can be matched to existing instances and attribute constraints are true on these instances. There is also another kind of loop – **WHILE** loop, which is denoted by a 3-d frame and continues execution while a valid loop variable instance can be found (it may have also several loop heads). Loops may be nested to any depth. The loop variable (and other element instances) from an upper level loop can be referenced by means of a **reference** symbol – the element with @ prefixed to its name.

Another widely used statement in MOLA is **rule** (also a grey rounded rectangle) – a statement consisting of pattern and actions. These actions can be **building actions** – an element or association to be built (denoted by red dotted lines) and **delete actions** (denoted by dashed lines). In addition, an attribute value of an element (new or existing) can be set by means of **attribute assignments**. A rule is executed once – typically in a loop body (then once for each iteration). A rule may be combined with a loop head, in other words, actions may be added to a loop head, thus frequently the whole loop consists of one such combined statement.

To call a subprogram, a **call statement** is used (possibly, with parameters - instances in the same reference notation). A subprogram, in turn, may have one or more **input parameters**. The same loop statement notation can be used to denote control branching – with a guard statement instead of loop head.

In this paper an additional MOLA element – **standardized comments** are introduced. These comments are text boxes associated to a MOLA diagram (its start symbol) and its statements. Comments can contain any text, but references to loop variables are shown in **bold**, and references to other elements – in *italic*. The comment for the whole diagram is intended to describe its informal pre- and post-conditions. The comments to separate statements are meant to describe their goal in an informal way.

Our goal is to make a MOLA program self-documenting, i.e., so easy readable that any one can ascertain that a MOLA program actually performs the intended transformation. Our experience shows that a well-written MOLA program with such comments is self-documenting really and we hope that the example in section 4 confirms this.

3. The Benchmark Example

The most popular transformation benchmark example – transformation of UML class model to relational database is used here. There are several versions of this example originally proposed by OMG – nearly each paper uses its own version. We use here the version from the QVT-P proposal [1].

The source metamodel is a significantly simplified fragment of the UML class diagram metamodel, it is visible in the upper part of Fig. 1. The target metamodel is a simplified relational database metamodel, it is given in the lower part of Fig.1. Next, the precise informal specification of the transformation task will be given (since there are some minor deviations from [1] due to some inconsistencies in it).

Any persistent *Class* (with *kind*="*persistent*") must be transformed into a database *Table*. In addition, a (primary) *key* is built for this *table*. *Attributes* of the class, which have a primitive data type, must be transformed into *columns* of the corresponding *table* (we assume here that types in UML and SQL coincide). *Attributes* whose type is a *class*, must be "drilled-down": primitively-typed attributes of this new class are added as columns to the table for the original class. Class-typed attributes are processed as before. The process is repeated until no new columns can be added to the table for the original class. In other words, a transitive closure is performed, which finds all "indirect" attributes of the class. The added columns have compound names consisting of all attribute names along the path. One special issue must be reminded here: several attributes of a class may have the same class as a type, in this case the added columns are duplicated for each of them (they have unique names!). In other words, any path leading to a primitively-typed attribute results into a separate column.

For primitive-typed "direct" attributes of a persistent class with kind="primary", the corresponding columns are included in the relevant (primary) key. An association (with multiplicities ignored, but direction taken into account) is transformed into a foreign key for the "source end" table. The same table is extended with columns corresponding to columns of the (primary) key at the target end. For both "primary" and "foreign" columns their kind is set accordingly.

4. MOLA Solution

4.1. Building the Workspace Metamodel

The first step in building a MOLA program (transformation) is to define the **workspace** metamodel (see Fig.1). This metamodel includes both the **source metamodel** (light yellow classes – the upper part) and the **target metamodel** (dark yellow classes – the lower part). Both metamodels are taken from the problem domain without modifications – they describe the corresponding input data (source model) and the result (target model) of the transformation.

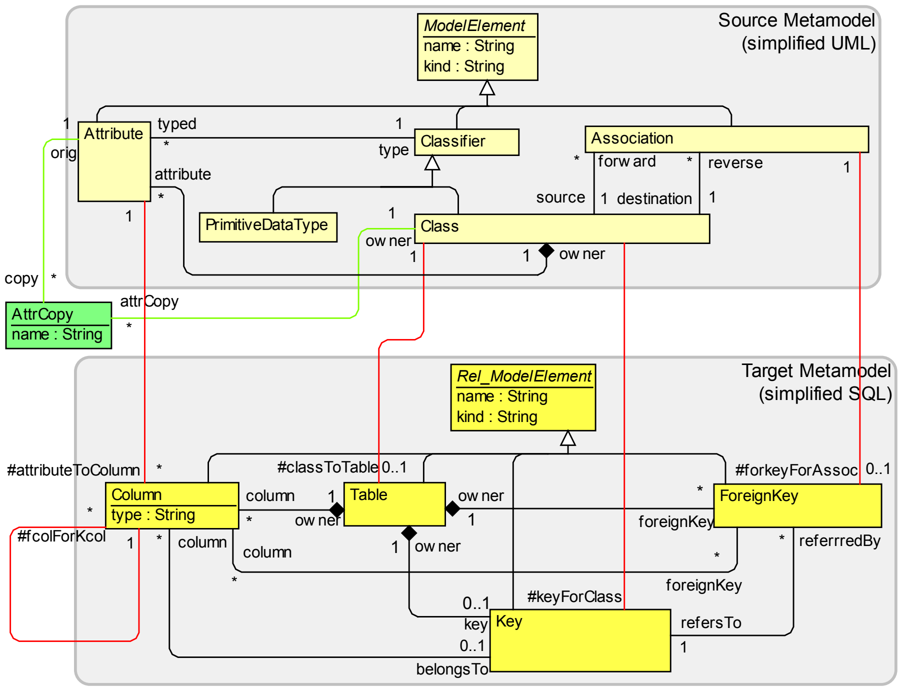


Fig. 1. The workspace metamodel

However, some elements typically are added to the workspace metamodel. First, there are **mapping associations** – associations linking classes in the source and target metamodels (red lines in Fig.1). They serve two different purposes – on the one hand, they document relations between the corresponding source and target elements of the transformation (e.g., *Class* and *Table*, *Attribute* and *Column*, etc.) and thus enable the **traceability** at the instance level (which Table was obtained from which Class). On the other hand, they have a technical role in MOLA – after being built by one rule, they frequently are used in patterns of subsequent rules. It is recommended in MOLA to start the role names of mapping associations with "#".

Another possible metamodel extensions are temporary classes – *AttrCopy* in the example and temporary associations (associations linking *AttrCopy* to base classes of the metamodel, all temporary elements are in green color in the example). This temporary class will be used to store copies of an attribute – indirect attributes. Temporary elements serve as a "workspace" for transformations, they have instances only during the transformation execution, and they are not supplied at input and are discarded at output. Base metamodel classes may have also temporary attributes added (attributes which have value only during the transformation execution) – this example does not use them.

4.2. MOLA Program Implementing the Transformation

The transformation is specified in MOLA by means of one main diagram (Fig. 2) and four subprograms (subdiagrams) – Fig. 3 to 6. The implemented transformation corresponds to its informal specification in a quite straightforward manner. The specification requires to perform a transitive closure – to find all indirect attributes of a class, and with duplicates included (therefore attribute copying is required). We use an idea that each instance of indirect attribute actually is a path in the "instance graph" from the "root class" to an attribute. The iterative algorithm (Fig. 3) for finding all indirect attributes of a class is inspired by the well known algorithm for finding all paths from a node.

All diagrams (Fig. 2 to 6) are annotated by standardized comments and, we hope, will require no other explanations.

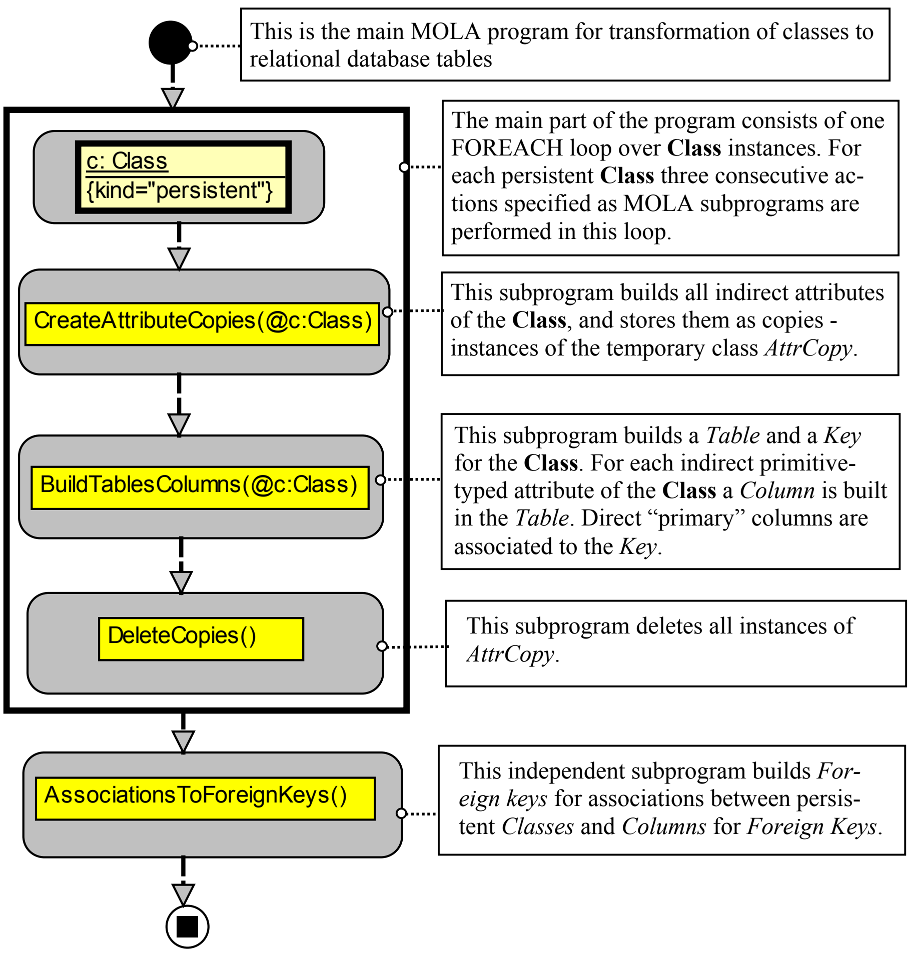
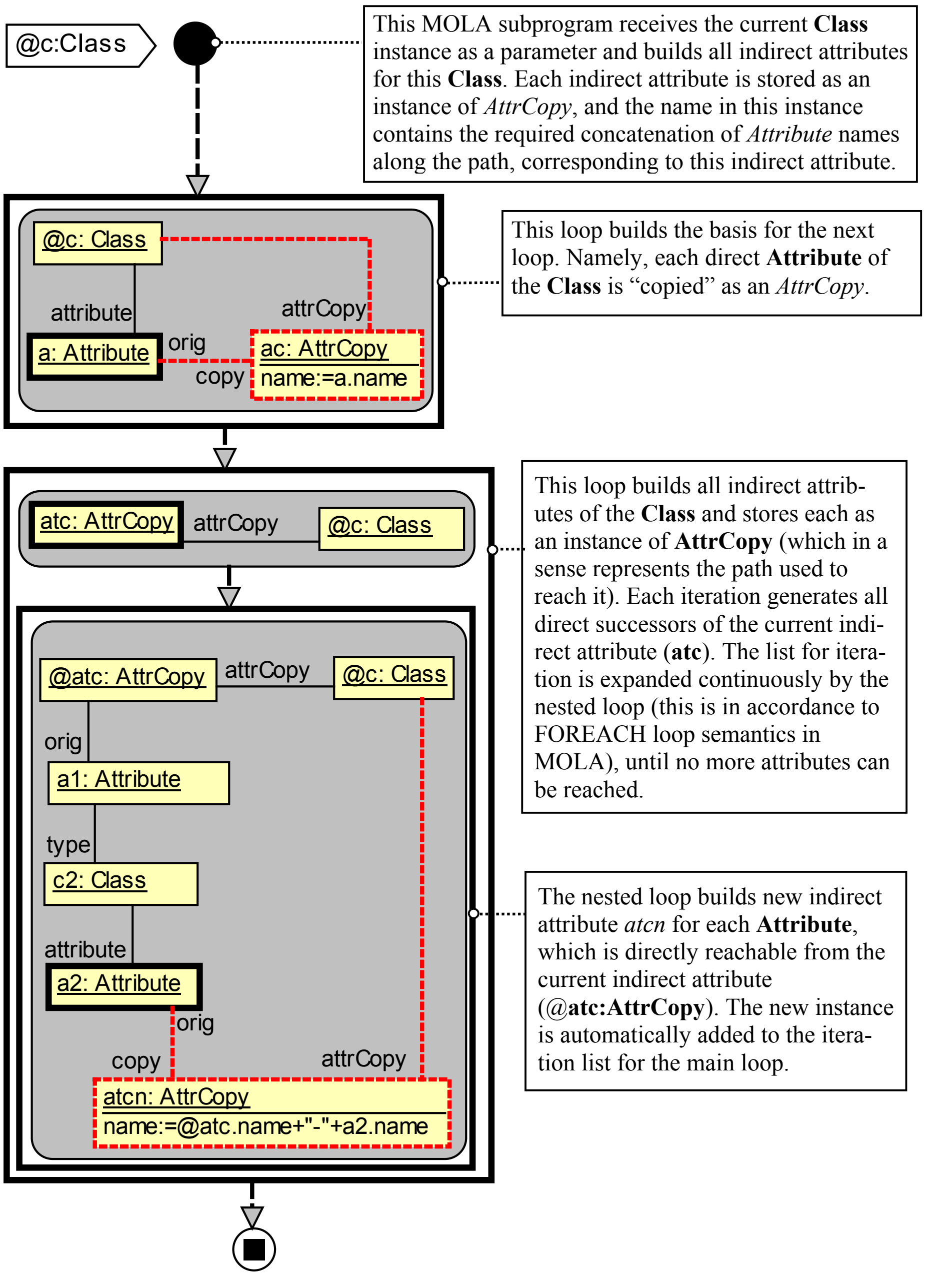

This is the main MOLA program for transformation of classes to relational database tables
c: Class
{kind="persistent"}
The main part of the program consists of one FOREACH loop over Class instances. For each persistent Class three consecutive actions specified as MOLA subprograms are performed in this loop.
CreateAttributeCopies(@c:Class)
This subprogram builds all indirect attributes of the Class, and stores them as copies - instances of the temporary class AttrCopy.
BuildTablesColumns(@c:Class)
This subprogram builds a Table and a Key for the Class. For each indirect primitive-typed attribute of the Class a Column is built in the Table. Direct "primary" columns are associated to the Key.
DeleteCopies()
This subprogram deletes all instances of AttrCopy.
AssociationsToForeignKeys()
This independent subprogram builds Foreign keys for associations between persistent Classes and Columns for Foreign Keys.

Fig. 2. The main diagram of the transformation



This MOLA subprogram receives the current **Class** instance as a parameter and builds all indirect attributes for this **Class**. Each indirect attribute is stored as an instance of *AttrCopy*, and the name in this instance contains the required concatenation of *Attribute* names along the path, corresponding to this indirect attribute.

This loop builds the basis for the next loop. Namely, each direct **Attribute** of the **Class** is "copied" as an *AttrCopy*.

This loop builds all indirect attributes of the **Class** and stores each as an instance of **AttrCopy** (which in a sense represents the path used to reach it). Each iteration generates all direct successors of the current indirect attribute (**atc**). The list for iteration is expanded continuously by the nested loop (this is in accordance to FOREACH loop semantics in MOLA), until no more attributes can be reached.

The nested loop builds new indirect attribute *atcn* for each **Attribute**, which is directly reachable from the current indirect attribute (**@atc:AttrCopy**). The new instance is automatically added to the iteration list for the main loop.

Fig. 3. Subprogram *CreateAttributeCopies*

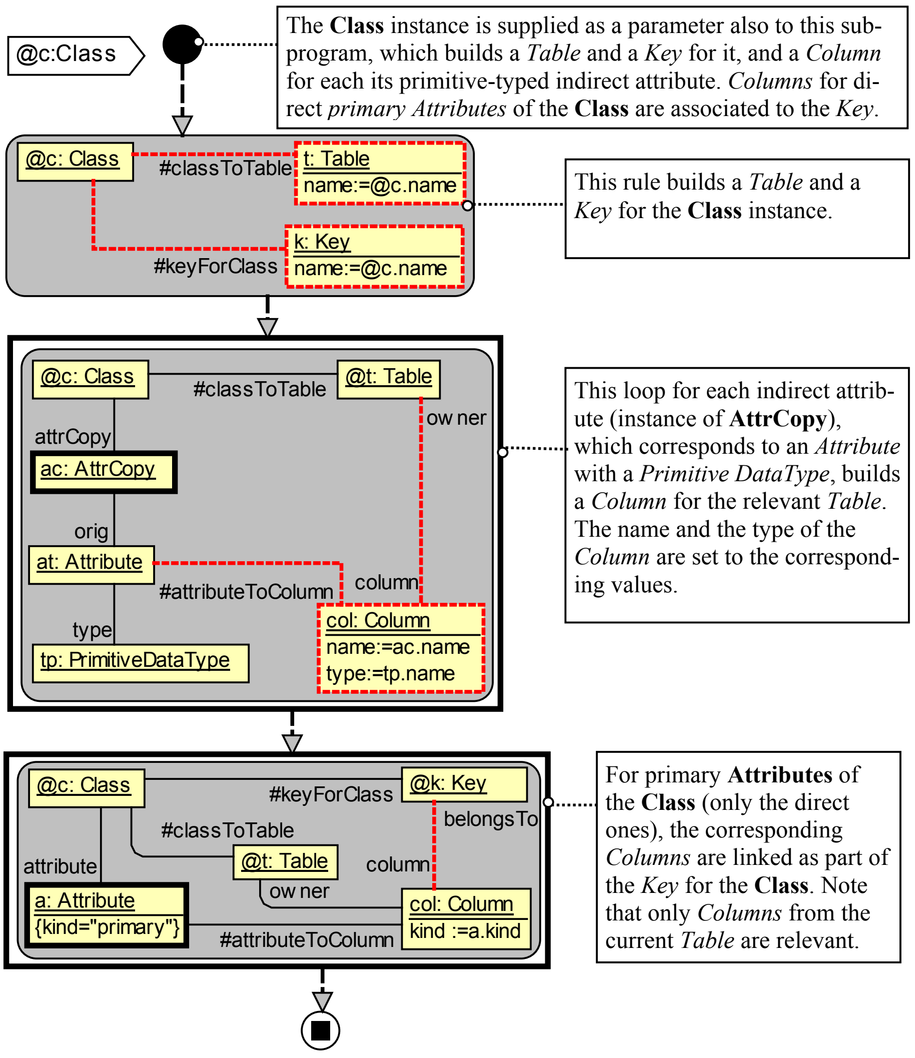

@c:Class
The **Class** instance is supplied as a parameter also to this subprogram, which builds a *Table* and a *Key* for it, and a *Column* for each its primitive-typed indirect attribute. *Columns* for direct *primary Attributes* of the **Class** are associated to the *Key*.
@c: Class
#classToTable
t: Table
name:=@c.name
k: Key
name:=@c.name
#keyForClass
This rule builds a *Table* and a *Key* for the **Class** instance.
@c: Class
#classToTable
@t: Table
owner
attrCopy
ac: AttrCopy
orig
at: Attribute
#attributeToColumn
column
type
tp: PrimitiveDataType
col: Column
name:=ac.name
type:=tp.name
This loop for each indirect attribute (instance of **AttrCopy**), which corresponds to an *Attribute* with a *Primitive DataType*, builds a *Column* for the relevant *Table*. The name and the type of the *Column* are set to the corresponding values.
@c: Class
#keyForClass
@k: Key
belongsTo
#classToTable
attribute
@t: Table
column
owner
a: Attribute
{kind="primary"}
#attributeToColumn
col: Column
kind :=a.kind
For primary **Attributes** of the **Class** (only the direct ones), the corresponding *Columns* are linked as part of the *Key* for the **Class**. Note that only *Columns* from the current *Table* are relevant.

Fig. 4. Subprogram *BuildTablesColumns*

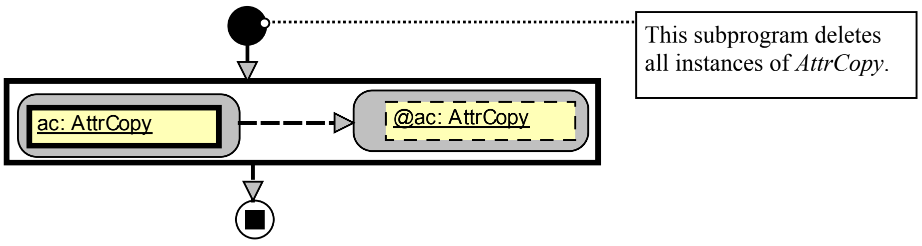
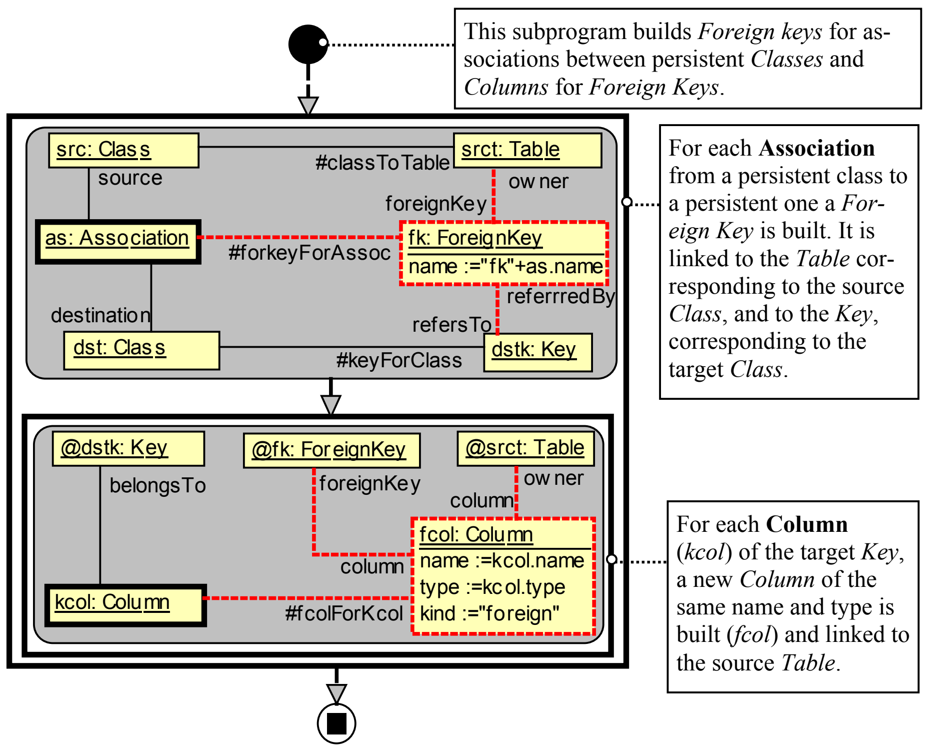


Fig. 5. Subprogram *DeleteCopies*


This subprogram builds *Foreign keys* for associations between persistent *Classes* and *Columns* for *Foreign Keys*.

For each **Association** from a persistent class to a persistent one a *Foreign Key* is built. It is linked to the *Table* corresponding to the source *Class*, and to the *Key*, corresponding to the target *Class*.

For each **Column** (*kcol*) of the target *Key*, a new *Column* of the same name and type is built (*fcol*) and linked to the source *Table*.

Fig. 6. Subprogram *AssociationsToForeignKeys*

5. Some Remarks on MOLA Methodology

Using the previous example as a basis, some elements of transformation program design methodology in MOLA will be provided.

Firstly, in MOLA, like most model transformation languages, a top-down approach should be used. Namely, the most coarse-grained elements of the source model (in our case, *Classes*) must be processed first. Only in this way, the mapping associations built for transforming them (e.g., *#classToTable*) can be used in patterns for transformations of contained elements (*Attributes*) or related ones (*Associations*). Thus, there is no need to repeat higher-level constraints (e.g., {kind="persistent"}) at lower level.

Since the main "processing element" in MOLA is loop, a correct design of loops is of prime importance. Typical algorithm steps frequently contain statements "for each A do …" and FOREACH loops (the loops used in section 4) should be used to implement these steps. Besides being a natural formalization of the step, FOREACH loops are easier to use, because their semantics already includes "iterate once for each element …" and there is no need for marking instances already processed. In most cases, the possible infinite loop problem is also eliminated due to a finite set of instances of the class (all loops in Fig. 2, 4, 5, 6). However, MOLA FOREACH loop can have its instance set replenished dynamically (the second loop in Fig.3), in this case additional considerations should be used (during the building of indirect attributes we assume that no class is used as the type of its own indirect attribute). On the contrary, WHILE loops should be used for steps which are a mix of iteration and recursion (such as the moving of transition ends during the flattening of a UML statechart in [5]), there the use of several loop heads per loop enables a natural and compact at the same time formalization for this kind of algorithm step.

Yet another important design element in MOLA is the selection of loop variables so that patterns in loop heads do not become complicated. Especially, for nested loops the deepest repeating element must be used. Use of referenced elements from upper level loops helps to simplify patterns in nested loops (see the nested loop in Fig. 6).

Actually, there are more design hints in MOLA and eventually "GOF-style design patterns" could be defined, but this is a topic of another paper.

And finally, nearly any non-standard transformation element can be described in MOLA using low-level facilities such as temporary classes and associations.

6. Conclusions

We have shown that by selecting an appropriate design style, the transformation programming in MOLA is relatively simple, as it is demonstrated by the complete example in section 4.

By adding standardized comments (even quite short ones, as in section 4), the readability of MOLA programs really reaches the level of self-documenting – one of main goals for the design of MOLA.

The implementation of MOLA is expected not to be very complicated due to relatively simple constructs in it. The implementation efficiency is also expected to be high enough – if the programming guidelines from section 5 and some more natural assumptions are observed, typical pattern matching problems of graph transformations are avoidable in MOLA.

References

1. QVT-Merge Group. MOF 2.0 QVT RFP, Revised submission, version 1.0. OMG Document ad/2004-04-01, http://www.omg.org/cgi-bin/doc?ad/2004-04-01
2. Compuware, SUN. MOF 2.0 Query/Views/Transformations RFP, Revised Submission. OMG Document ad/2003-08-07, http://www.omg.org/cgi-bin/doc?ad/2003-08-07
3. Willink E.D. A concrete UML-based graphical transformation syntax - The UML to RDBMS example in UMLX. Workshop on Metamodelling for MDA, University of York, England, 24-25 November 2003
4. Agrawal A., Karsai G, Shi F. Graph Transformations on Domain-Specific Models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, 2003
5. Kalnins A., Barzdins J., Celms E. Model Transformation Language MOLA. Proceedings of MDAFA 2004, University of Linkoping, Sweden, 2004, pp.14-28. (see also http://melnais.mii.lu.lv/audris/MOLA_MDAFA.pdf)
6. Kalnins A., Barzdins J., Celms E. Basics of Model Transformation Language MOLA. Proceedings of WMDD 2004, Oslo, 2004, http://heim.ifi.uio.no/~janoa/wmdd2004/papers/kalnis.pdf

Automated Generation of Metamodels for Web service Languages

Behzad Bordbar and Athanasios Staikopoulos

School of Computer Science, University of Birmingham, Birmingham B15 2TT, UK
B.Bordbar@cs.bham.ac.uk, A.Staikopoulos@cs.bham.ac.uk

Abstract: Recently, the application of the MDA to Web services has received considerable attention. In the MDA, models are instances of the MOF based metamodels. Model Transformation, which is a key feature of the MDA, can carried out via defining Transformation Rules between two MOF compliant metamodels. As a result, finding MOF compliant metamodels for languages is an essential prerequisite for model transformation.
This paper presents a semi-automated, tool-based method for the generation of MOF compliant metamodels for languages, which are specified via XML Schema Descriptions (XSD). We demonstrate that our approach can easily be implemented using existing XML Schema integration tool and UML CASE tool. To explain the approach, the paper sketches the stages involved in the generation of a metamodel for Web Service Description Language (WSDL) and compares the resulting metamodel with an existing metamodel for WSDL.

1. Introduction

Web services are Web- based enterprise application that use XML [19] based standards and transport protocols to communicate with each other in a platform and a programming-language independent manner. Applying Model Driven Architecture (MDA) [6][8][13] to Web services design has recently received considerable attention [1][8][3][4]. In particular, [1][8] study the Model Transformation for Web services and present a set of case studies involving the transformation of Web services models to various implementation platforms such as Java, Web Services Description Language (WSDL) [18] and EDOC [12].

Currently, there are a number of specifications and vocabularies defined and expressed in terms of the Extended Markup Language (XML) such as the Web Services Description Language (WSDL) [18] for Web Services. Such languages are XML extensions and are defined accordingly to a well-formed structure, the XML Schema. Therefore, an XML schema defines the language in the same respect where MOF is used to define the UML language. Considering the similarity it would be very beneficial within the domain of transformations to represent the XML family of languages such as Web Services in a MOF compliant metamodel.

In the MDA, each model is based on a specific *metamodel*, which defines the language that the model is created in. All metamodels within MDA, are based on a unique metamodel called Meta Object Facility (MOF)[14]. As a result, Model Transformations can be carried via defining Transformation Rules between two MOF compliant metamodels [1][3][6]. Consequently, there are two stages involved in any Model Transformation

- introducing MOF compliant metamodels for source and destination languages
- specifying Transformation Rules between metamodels.

This paper, which only deals with the first bullet point, aims to present a semi-automated, tool-based method for the generation of MOF compliant metamodels for languages, which are based on XML Schema Descriptions (XSD) [22] specification. In particular, Web Service languages such as WSDL [18], UDDI [11], SOAP[20], WSCI [21] and BPEL4WS [10] are examples of such languages. In general introducing a metamodel for each of the above languages involves identifying the concepts involved in the language and their relationship. Often, the starting point is reading and understanding the specification of such languages, which are published by organizations such as W3C [17] and OASIS [9]. The next step is to create a conceptual model involving the model element of the language and their relationship. However, specification of all above languages includes an XML Schema Description (XSD), which is a meta-language representing various features for constructing and formalising the vocabulary and grammar of the XML model of the language. The current paper explores the idea of using the XSD representation of the language and generating MOF compliant metamodel for the language. The paper sketches an implementation of our method via *hyper*Model [6], an XML schema design tool, and Poseidon for UML [16]. We shall also apply our method to create a metamodel for WSDL and compare the result with a WSDL metamodel presented in [1].

The paper is organised as follows. The next section is a brief review of concepts used in the paper. Section 3 present the core of our approach and sketches the implementation via *hyper*Model and Poseidon UML tool. Section 4 is a case study involving the creation of a metamodel for WSDL. Section 5 sketches the future wrok. Finally, section 6 presents a conclusion.

2 Preliminaries

Kurtev and van den Berg [7] identify four MDA Model Transformation scenarios. Three of the scenarios studied in [7] make direct use of the definition of the Transformation Rules between metamodels. In particular, in the context of Web services, model transformations can be carried out via defining Transformation Rules between two MOF compliant metamodels [1][3][6]. Figure 1, depicts an example of the use of Transformation Rules for model transformation [1].

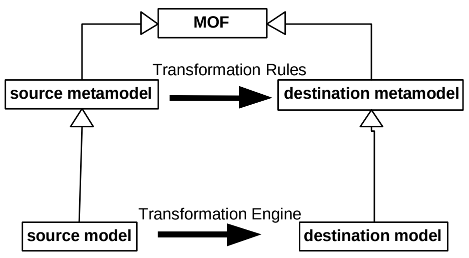


Figure 1: Using Transformation Rules in the MDA

As a result, defining a metamodel is one of the main steps in the process of the Model Transformation. In this paper, we are dealing with the creation of metamodel for languages for which the XML Schema Description (XSD) is available. This section presents a brief introduction on various concepts involved in the Model Transformation for XML based languages.

2.1 XML, XMI and XSD

The Extended Markup Language (XML) [19] is a cross-platform, text based W3C [17] standard for interchanging, structuring and representing data. One of the main characteristics of the XML is its extensibility mechanism and its flexibility to define complicated tree hierarchical structured data. In addition, XML can be used as a meta-language, allowing the generation of a whole family of XML languages. Such languages may be specialised in specific domains such as Web Services with WSDL [18], UDDI [11], BPEL4WS [10], Ontology with RDF and model interchange formats with XMI [15].

The XML Schema Definition (XSD) [22], which is also a W3C standard, is an XML language for describing XML documents. It offers a set of features both for specifying and formalising the vocabulary and the grammar of XML documents, and to impose various constraints on their content. In this way, XSD provides a validating mechanism, allowing computer programs to validate and check the XML document for well-formedness.

The XML has also been used to create a common interchange format between UML tools for interchanging models and metadata. The XML Metadata Interchange (XMI) [15] is a format introduced by the OMG, combining the rigor of the MOF models with the XML definition semantics.

2.2 Transformations between XML and UML

The XML Metadata Interchange (XMI) is designed to facilitate the interchange of data and metadata expressed via the MOF. As a consequence, the XMI specification defines a number of mapping rules that specify how to generate XML Document Type Definition (DTD) and XSD schema from class diagrams. The XMI also specifies methods of producing MOF models from such input formats. The automatically generated DTDs and XML Schemas are based on the MOF defined rules and allow the MOF-based models to be serialized validated and interchanged among different tools without controversies. This makes XMI a necessary intermediate medium standing between MOF models and XML representations. Therefore any transformations from XML to MOF/UML need to be based or extend XMI. The transformation from an XML Schema or DTD to an XMI format can be performed using the Extensible Stylesheet Language (XSLT).

One of the key feature of the XMI is that it provides *parameterised mapping*, i.e. by choosing different mapping parameters, it is possible to define different mappings from a UML model to its schema representation. For example, it is possible to choose between mapping a class attribute to an XML attribute or a an XML element.

3 A tool-based approach to metamodel generation

A language metamodel defines the model elements of the language, specifies the semantics of language and relationship between various model elements. As a result, the modeller often starts by understanding the language description by studying its specification and creating a conceptual model involving the entities of the language and their relationship. Currently, there is no systematic way of creating such conceptual models. Figure 2 depicts the outline of our approach, which aims to address this issue. To create a MOF metamodel, we shall start from the XSD Schema representing the language. The XSD documents, for most Web service languages are included and published in their specifications, available from W3C www.w3.org or OASIS www.oasis-open.org web pages. As depicted in Figure 2, an *XML transformation tool* can be used to covert the XSD document into the XMI format, which can in turn be imported by a *UML tool* as a class diagram. As a result, the transformation from an XML Schema to a UML Model is a fully automated process, which is carried out via CASE tools. The UML model presents a clear, high-level view of the involving concepts and their relationship. At this point, the *Modeller* begins refining the UML Model by consulting the *Language Description*. However, unlike the ad hoc approach, the created UML Model can guide the refinement of the model by pointing out the existing model elements that the modeler needs to inquire about.

3.2 Implementation

*hyper*Model [6] is an XML schema design and integration tool, offering various UML modeling capabilities. *hyper*Model is offered as a free plug-in to Eclipse workbench [2][1] allowing the transformation of XML vocabularies and schema into XMI 1.0 format. To implement our method, we start by opening the XSD document of the language in *hyper*Model. In *hyper*Model creating an XMI document from an XSD document is at a click of a mouse. It is possible to view the XMI model as a UML class diagram in *hyper*Model/Eclipse. However, in order to have greater flexibility in editing and refining of the model, we import the XMI document into a separate UML tool, for example Poseidon for UML [16]. In the next section, we shall apply our method to generate a metamodel for WSDL. We shall also compare our metamodel with the WSDL metamodel presented in [1]

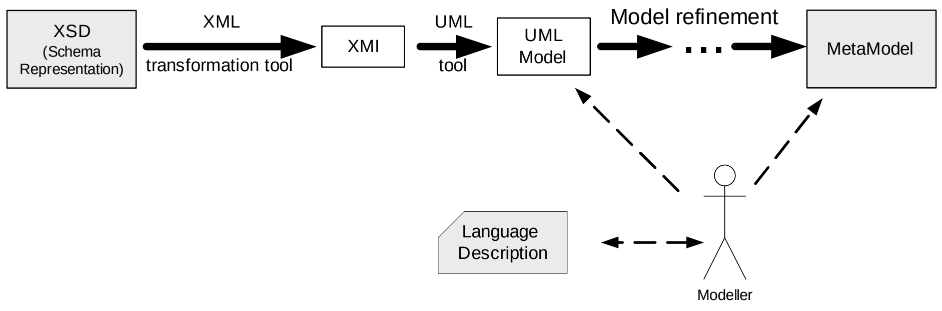


Figure 2 : Generating metamodels from XSD

4 Case study: a metamodel for WSDL

The Web Service Description Language (WSDL) [18] describes the syntax and semantics necessary to call up services. The language specification [18] contains the XSD for the WSDL. Figure 3 depicts a part of the first version of the WSDL metamodel created by *hyper*Model. Figure 4 depicts the refined version of the metamodel on which the following changes are made.

The XML provides an extensive mechanism for documenting and extensibility features. Some of the elements in the metamodel of Figure 3 are specific to XML and have no equivalent in MOF. For example, `tExtensibleAttributesDocumented` allows future extensions of the WSDL by adding new attributes from other XML namespaces. To create a MOF compliant metamodel, in the refined version, all such elements are deleted. Similarly, there are various stereotypes, for example `<<XSDattribute>>`, which are created from an XML tag representing *XSD attributes* , which are also deleted.

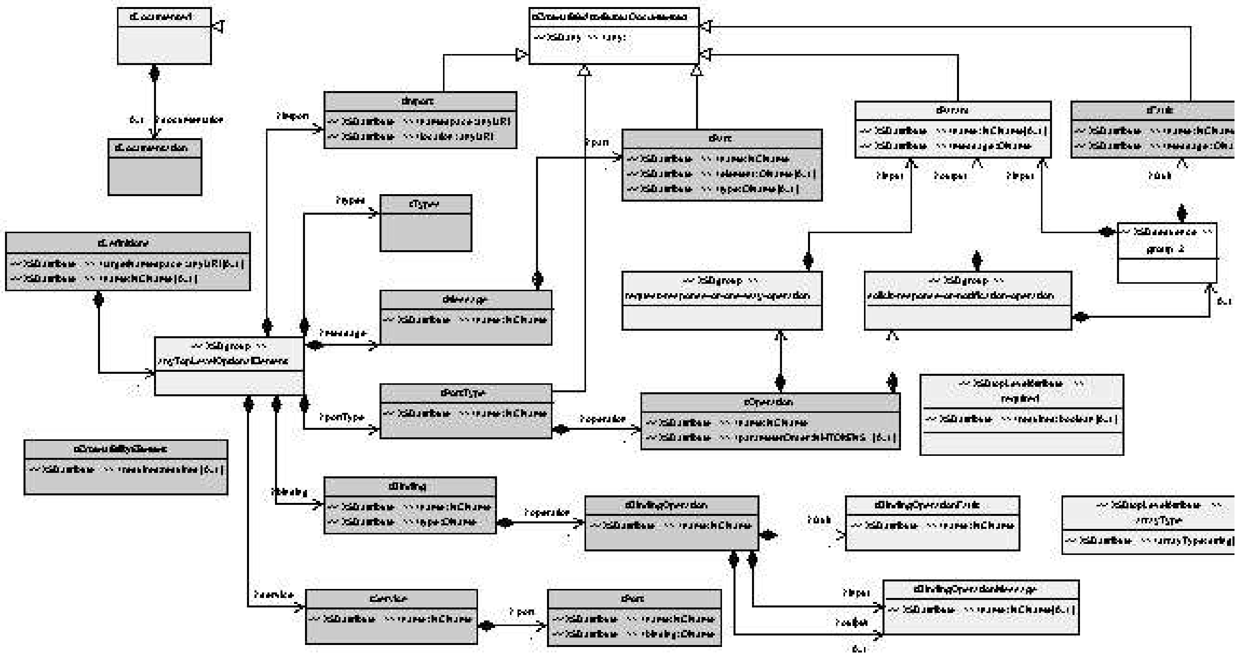


Figure 3: Initial WSDL Metamodel, version 1

The metamodel of Figure 4 contains the model element `group_2`, see the top-right corner of the picture, which is the translation of the following piece of XSD code.

```
- <xsd:group name="solicit-response-or-notification-operation">
- <xsd:sequence>
  <xsd:element name="output" type="wsdl:tParam" />
- <xsd:sequence minOccurs="0">
  <xsd:element name="input" type="wsdl:tParam" />
  <xsd:element name="fault" type="wsdl:tFault" minOccurs="0"…/>
  </xsd:sequence>
```

Creation of this metamodel element is a direct result of the XSD tag `</xsd:sequence>`, which means the elements within its scope must appear as a sequence, see [19]. This is a feature exclusive to XML. Eliminating such model element requires refactoring of the diagram, which can be easily done by redirecting each association of the model element `group_2`, to its source, *solicit-response-or-notification-operation*. This results in the metamodel of Figure 5. For the rest of the section, we shall compare the metamodel of Figure 5 created via our method and the WSDL metamodel presented in [1], depicted in Figure 6.

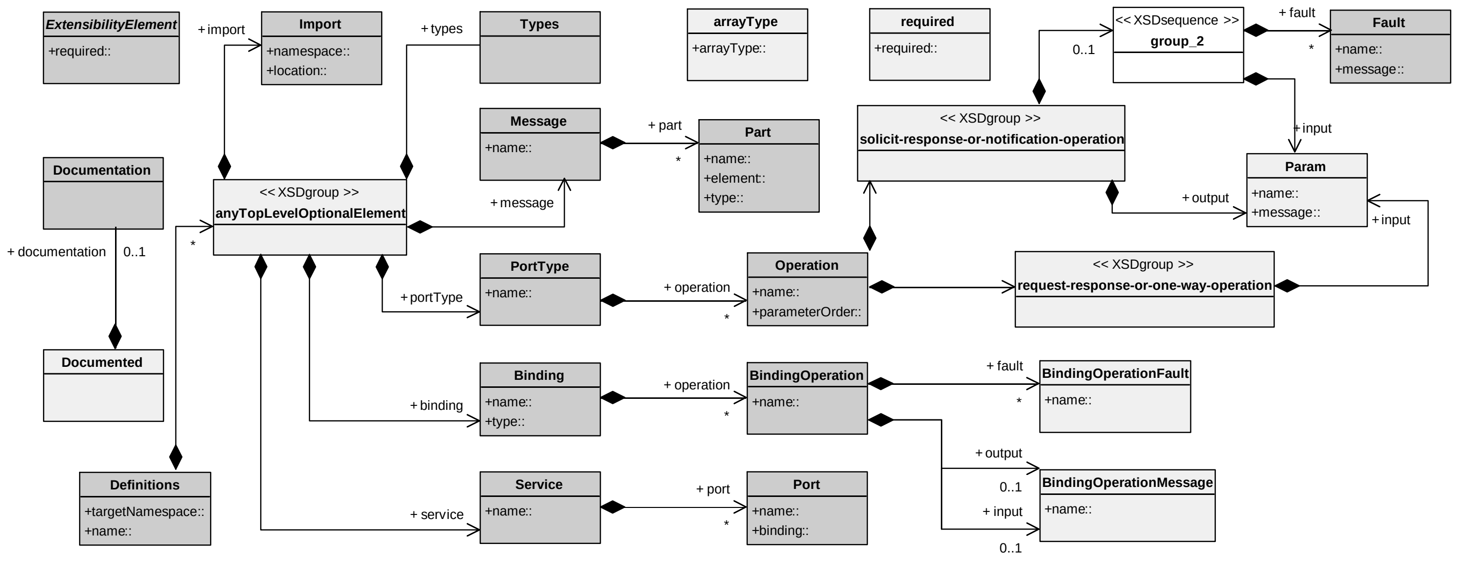


Figure 4 : Refined WSDL metamodel, version 2

There are clear similarities between the two metamodels. The gray shaded metamodel elements in Figure 5 are directly appearing in the other model. Figure 5 is more detailed and contains more elements. However, the authors of [1] clarify that the paper presents only a simplified version of their metamodel.

There are also a number of elements in Figure 6 which are not in our metamodel. Most notably, `input` and `output` are modeled as separate WSDL types in Figure 6, where in our case, they are modeled as metamodel *attribute ends*, which are of type parameters (`Param`). This correspond to the following line in the XSD document for the WSDL

```
<xs:element name="input" type="wsdl:tParam" />
```

In fact, we noticed that the XSD description of the WSDL does not define the types `inuput` or `output`. However, WSDL documentation [18] mentions phrases “output element” and “input elements” in numerous occasions. As a result, it is very natural that the authors [1] included `input` and `output` as model elements.

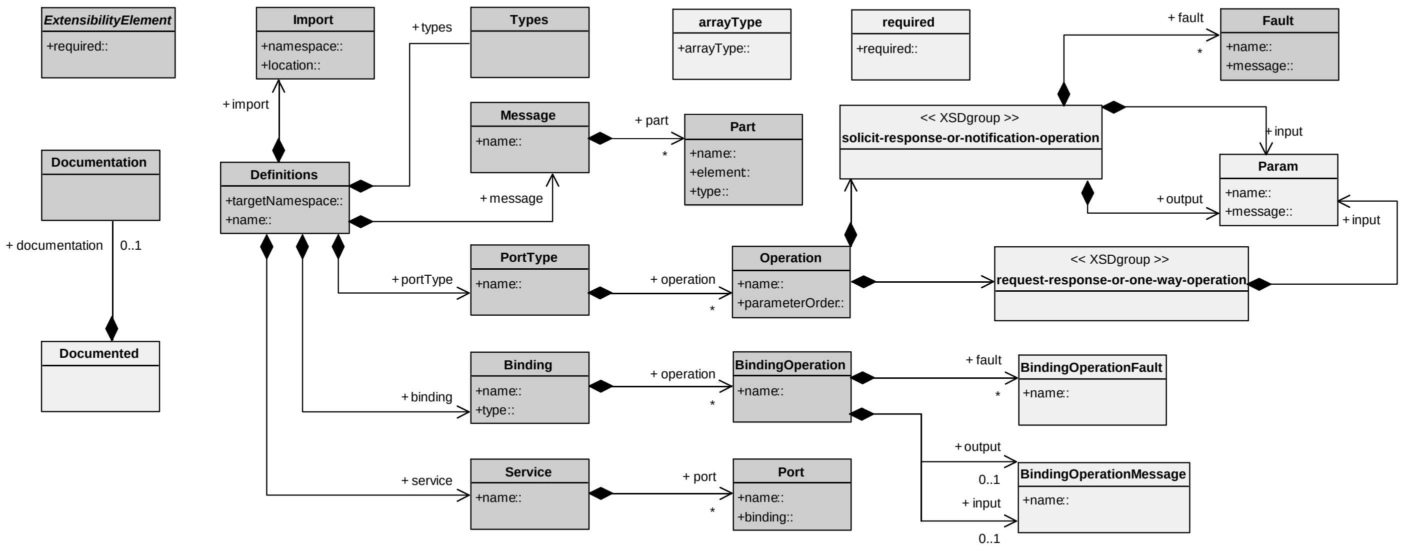


Figure 5: WSDL metamodel, final version

From the conceptual point of view, there is hardly any difference between the two metamodels[1]. From the model transformation point of view, the advantage of choosing one metamodel over another is not clear to us and remains a subject for future research.

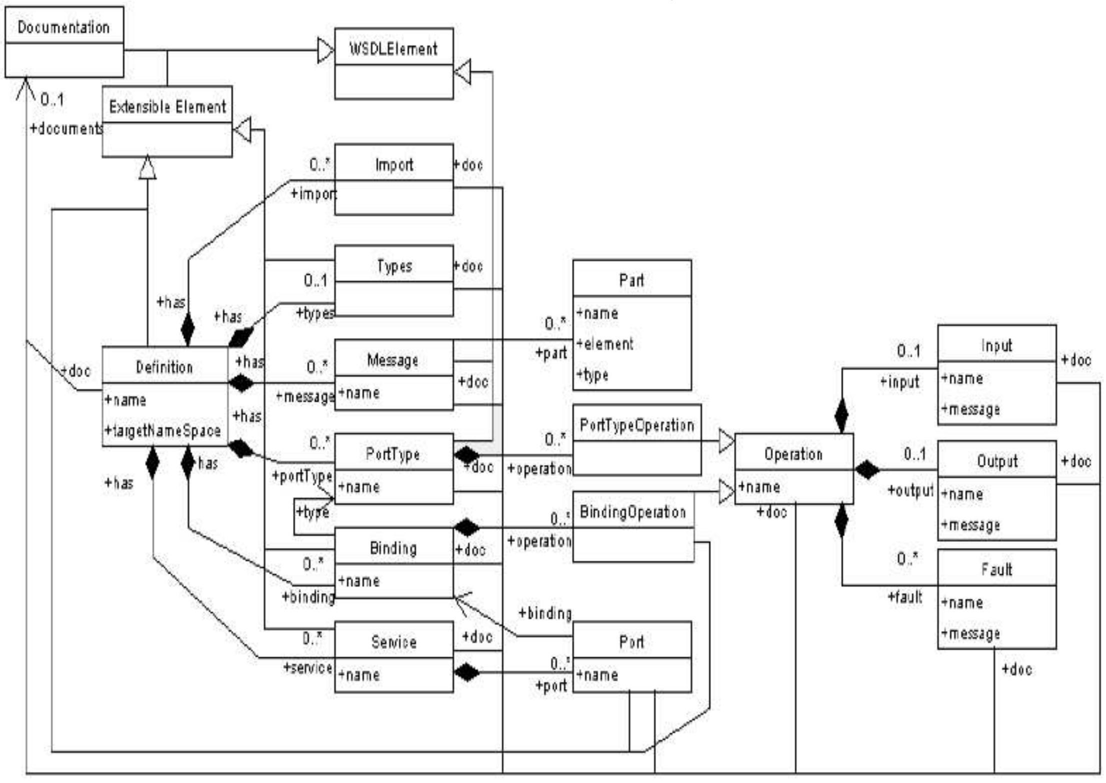


Figure 6 : WSDL metamodel, copied from[1]

[1] This is subject to including parameters (`Param`) in the metamodel of Figure 6.

5 Future works

*hyper*Model is a powerful tool for Web service integration and XML Schema design. However, the transformation from XSD to XMI is carried out in rigid form. It is important to make use of the *parameterized mapping* facilities of the XMI and be able to choose parameters to alter the transformation map. Moreover, the UML model created from the schema in *hyper*Model/Eclipse is only partially editable, which forces us to use another UML tool to edit and refine the model.

We have applied our method to generate metamodels for a number of Web service languages. Currently, the refactoring part of the process, which is at the heart of our approach, is performed manually. There is a clear scope for research into the automation of such refactoring activities. We are currently implementing the above method as an integrated UML tool, which particularly aims at the following

- providing greater flexibility in the transformation from XSD to XML, by allowing the modeller to choose the mapping of model elements
- producing better edit and viewing facilities to assist the modeller
- automating the refactoring of the model

6 conclusion

This paper presents a semi-automated method of generating metamodels for languages, which are specified via XML Schema Description (XSD). The method presented starts by creating an XMI document from the XSD specification of the language. The XMI model, which can be imported as class diagram in a UML tool, provides a high level view of the concepts involved in the language and their relationship. Such model is subsequently refined to create a metamodel for the language. The process of refinement may require refactoring of the model to eliminate some elements, which exclusively correspond to XML model elements and have no equivalent in MOF. Our method is particularly suitable for Web service languages and the paper sketches the generation of a metamodel for Web Service Description Language (WSDL).

References

[1] J. Bezivin, S. Hammoudi, D. Lopes, F. Jouault, *An Experiment in Mapping Web Services to Implementation Platforms*, Atlas Group, INRIA and LINA University of Nantes, Research Report, March 2004

[2] Eclipse project, www.eclipse.org

[3] D. S. Frankel, Model Driven Architecture, *Model Driven Architecture: Applying MDA to Enterprise Computing*, OMG Press, ISBN: 0471319201, January 2003

[4] D. S. Frankel, *White Paper: Using Model Driven Architecture to Develop Web Services*, IONA Technologies PLC, Second Edition, April 2002

[5] *hyper*Model, www.xmlmodeling.com/hyperModel/index.html

[6] A. Kleppe, J. Warmer, W. Bast , *MDA Explained. The Model Driven Architecture: Practice and Promise*, Addison-Wesley, ISBN: 321-19442-X, April 2003

[7] I Kurtev and K. van den Berg, *Unifying Approach for Model Transformations in the MOF Metamodeling Architecture*, Proceedings of the 1st European MDA Workshop, MDA-IA, University of Twente, the Nederlands, March 2004

[8] D. Lopes, S. Hammoudi, *Web Services in the Context of MDA*, University of Nantes, France, 2003

[9] OASIS, available from http://www.oasis-open.org/

[10] OASIS, Business Process Execution Language for Web Services (BPEL4WS), available from OASIS site
[11] OASIS, *Universal Description Discovery & Integration (UDDI)*, Version 3, available from OASIS site
[12] OMG, *Enterprise Collaboration Architecture (ECA) Specification*, Object Management Group, Version 1.0, February 2004
[13] OMG, *Object Management Group*, Available from http://www.omg.com
[14] OMG, *Meta Object Facility (MOF) Specification*, Object Management Group, Version 1.4, April 2002, available from OMG site
[15] OMG, XML Metadata Interchange (XMI), available from OMG site
[16] Poseidon for UML, www.gentleware.com/
[17] W3C, World Wide Web Consortium, www.w3.org
[18] W3C, *Web Services Description language (WSDL) Version 2.0*, W3C Working Draft, November 2003
[19] W3C, *Extensible Markup Language (XML) 1.0*, Third Edition, W3C Recommendation, Available from http://www.w3.org/TR/2004/REC-xml-20040204, February 2004
[20] W3C, *Simple Object Access Protocol (SOAP)*, Version 1.2, W3C Recommendation, Available from http://www.w3.org/TR/soap12-part1, June 2003
[21] W3C, *Web Service Choreography Interface (WSCI) 1.0*, W3C Note, Available from http://www.w3.org/TR/wsci, August 2002
[22] W3C, XML Schema Primer

Reports from Breakout Sessions

EMWDA Working Group on Lossy Transformations

Discussion Report

Editor: Anneke Kleppe

People Attending

Ed Willink, Jim Steel, Joao Paulo Almeida, Octavian Patrascoiu, Dave Akhurst, Julian Johnson, Audris Kalnins, Olivier Le Merdy, Kevin Dockerill, and Anneke Kleppe.

Lossy Transformations

We started of with a number of different interpretations of the term ‘Lossy transformation’. ‘Lossy’ could mean either of:

1. Necessary information is not present in input model (such as requirements or design intent that were never modelled)
2. Potential corruption by a transformation that does not satisfy its specification, or whose implementation is flawed (*invalid* transformation).
3. Deliberate discard of information in the input model, for instance because it cannot be represented in the output language (*partial* transformation).
4. Lack of traceability, i.e. the elements of the output model cannot be linked to the elements in the input model they were generated from.
5. Lack of reversibility, i.e. the input model cannot be restored from the output model.
6. Loss of info on how or why the transformation is executed.

To get a better grip we propose the use of the term *invalid* (versus *valid*) for the second meaning, and the term *partial* (versus *complete*) for the third meaning. A transformation T may be invalid because either T does not apply to all possible input models or because there is a gap between the specification of T and implementation of T.

With regard to option one, incomplete input, we considered this to be the responsibility of the transformation it self. Either it should issue a warning before executing or it should not execute at all. On the topic of option four, traceability, we concluded that traceability is not a theoretically difficult issue. It is feasible, although in practice one may need very large machines to run the transformations on. In the discussion on option five, reversibility, it was amazing to see that none of us found this to be a very big issue. There were no dissenters from the perception that reversibility is only relevant for approximately 10% of the transformations. In the discussion on option six, loss of info on how or why the transformation is executed, there remained on open question: if a compound transformation fails (is *invalid*), how can you find the element that causes the failure?

Transformation Use Cases and Semantics Preserving Transformations

Tracy Gardener, in her keynote, presented a list of possible use cases for transformations. We discussed two of them in more depth in order to see whether they would need different types of transformations. We discussed pattern expansion and PIM to PSM transformations. Our conclusion was that the differences were not very large. For pattern expansion the transformation can be called ‘in-place’, which means that the source and target model are the same in some meaning, at least they are written in the same language. at this point in the discussion it became clear that we need to define equivalence of systems (and after that also of models) before we are able to define what a semantics preserving transformation is. Another conclusion was that parameterisation of transformations should be possible for any type of transformation.

EMWDA Working Group on ‘MDA is the Wrong Answer?

Discussion Report

Editor: Jos Warmer

People Attending

Oliver Sims, Andrew Watson, Jos Warmer,Tracy Gardner, Dave Pilfold, Tony Mallia, Ian de Beer, Marcus Alanen, Nelly Bencomo, Lea Kutvonen.

Introduction

Starting with “MDA is the wrong answer”, we came up with a list of problems that the participants want (or expect) to solve using MDA, and why MDa will be able to achieve this. At the same time a list was made up of perceived shortcomings of MDA. This contains arguments that the participants often encounter during their discussions on MDA with other people.

What problems do we want to solve

We came up with the following problems that the participants want (or expect) to solve using MDA: making software development less tedious, get a higher quality, documentation and application generation, solving the lack of enough skilled programmers, making less IT projects fail, structurally survive technology changes, provide better support for collaboration and integration, building better product lines, maintaining relationship between different domains, and avoiding to get into the same problem again and again.
This is quite am impressive list, suggesting that MDA has to be a silver bullet after all, if it can really solve all of these problems. The group discussed how and why MDA would be helpful in solving such a wide variation of problems.

Why would MDA solve these problems?

A crucial characteristic of MDA is that transformation specifications explicitly define the relationships between many of the artefacts that are produced during a software development project. Knowing this relationship and having tools to validate them and/or resolving conflicts between them allow for better (i.e. better quality, better productivity, better documentation, etc.) product lines. For the same reason, MDA can be the glue between all of the different issues and views within a software project.
Working from the modeling level, MDA allows people to work at a higher, technology independent, abstraction level. This provides good support for dealing with the quick technology changes that are typical for the IT world.
Current measurements are not too many, but they do indicate that savings of 40% over the complete software development life cycle can be achieved using MDA. If this is the case for MDA in its current, rather immature state, then the saving might eventually be much bigger.

Reasons why MDA would not work

Many of the participant have encountered scepticism about MDA. We have made a list of typical reasons why people don’t want to use MDA. The participants do not agree with these, but it is important to know the arguments and be able to counter argument them.

No appropriate languages. The group agreed that UML has many shortcoming. It is both too complicated, and too low level.

A solution can be found by using UML profiles to define higher level languages, or by defining new and higher level modeling languages.

Tools. There is a lack of tools and the big vendors are not offering MDA tools yet. This situation is remedied by Microsoft, who is actively building support for model driven development in their Visual Studio 2005 product. Although they do not use the term MDA (copyrighted by the OMG) they support many of the major ideas behind it.

The MDA community, needs to develop more and better tool support. The group sees both a place for open source tools, for easy experimentation and having a low threshold, and for vendor tools, to get a credible and well-supported market place. There was a slight fear in the group that vendors would try to come up with all types of proprietary model, thus making defeating the goals of MDA.

No incentive. For various reasons people simply have no incentive to use MDA. This can be a lack of the will to change and fear of losing your jobs. From a management perspective software development often isn't their major concern, day to day operations is. Therefore MDA does not seem to be relevant.

The group felt that, given the lack of enough gifted programmers, MDA will not cost jobs. It will merely change the type of work that people do. Also, when new possibilities arise, business always has an increasing demand. Therefore lose of jobs seems unlikely.

It didn't work before. People often see MDA as being CASE in different words. CASE and e.g. Shlear-Mellor failed to deliver their promises, why would MDA do any better?

An answer to this is that MDA is much more than just code generation that CASE offered. MDA offers the infrastructure such as a standardized MOF, standardized metamodels, standardized transformation languages (QVT) etc. that were lacking in the CASE era. This allows developers to look at all pieces and enables them to easily change whatever they want.

MDA is not applicable for my domain. People talk about certain domains, especially GUI, and decide MDA can never work for these.

There are already MDA tools out that do a good job in the GUI area. This type of argument can only be countered by giving concrete examples.

MDA is too vague. An often heard complaint is that MDA is too vague. Everyone and every tool is free to call itself MDA. It become unclear what MDA really is.

This is a very true complaint. However, the OMG has just started an initiative to clarify the meaning of MDA much better, including a list of requirements that a tool need to fulfil to allow it to be called an MDA tool.

Conclusion

The first and main aspect that needs work for MDA to work as promised are better tools. There was consensus in the group that MDA without tools will never work. Of course, the tools can only be made to work effectively is we also have available UML profiles, higher level modeling languages, meta-models, models of architectures, QVT standards, etc.

The group agreed that MDA is still in its early years and that we need time to make it all work as promised. Even after hearing all the reasons why it would not work, we are still convinced that MDA holds much promise and will certainly mean an important advance the IT world.

Types in MDA

Jim Steel

Irisa, Campus de Beaulieu, 35042 Rennes, France

Abstract. The following people participated in the breakout session held on September 8 as part of the 2nd European Workshop on Model-Driven Architecture, and it is their ideas that are represented herein, and the author's hope that their ideas have been adequately expressed: Marcus Alanen, Rasmus Fogh, Val Jones, Girish Maskeri, Jim Steel, Laurence Tratt, Andrew Watson, Ed Willink.

The MDA paradigm is, at its core, an architecture designed around the definition of languages and transformations between them. However, while specifications such as the MOF discuss some aspects of language definition, particularly structural aspects, there is considerably less discussion of the type systems that may exist in these langauges.

In considering this issue, a number of questions become apparent. Firstly, in this MDA world, what are the concepts that we wish to characterise by a type, what sort of structures do we want to use to describe these types, and under what circumstances may something of one type be used in a situation demanding a different type? The most compelling example for answering these questions is perhaps that of sequencing model transformations; how do we know if the output of one transformation is acceptable as input to another?

At the most general level, a platform's type system can be characterised by answering two questions. What is a type, and what is the substitutability relationship (if any) between two types? Further issues such as type induction can, in most cases, be considered ancillary. Fortunately, there is a large and thorough body of ongoing research into type systems, so answers to these questions are many. However, there is a perceived gap between the more mathematical models of type theory presented in the literature and those witnessed in the "real-world" programming languages.

There are a number of possible definitions to take for type. Taken broadly, it may be characterised as "suitability for some purpose", or more specifically as a "domain of interesting instances". The latter definition raises the question of whether types are defined extensionally or, as is more common, intensionally. The former, as a closed-world assumption, can have both simplifying and limiting consequences for implementation. The latter approach includes techniques such as structural-level typing, such as those used in inheritance-based or structural-conformance typing. Also worth considering are type systems including awareness of semantic domains. For example, under what circumstances is one process definition substitutable for another?

At a practical level, there are many possible choices available for typing MDA models. Using constraints as types is the most general and perhaps the most powerful, but may not be the most usable formalism for the modeller. Using patterns is also powerful, and has gained popularity in recent times. Using classes is a very familiar technique, but is limited in its expressive power in some cases. Alternatively, perhaps some hybrid of these would be useful, such as attaching constraints to classes, or adding limits to the numbers of class instances allowable.

From a wider viewpoint, having modelled type systems, there may also be a need to reason about them in comparison to one another. That is, is a given type system appropriate as a target for some mapping? A concrete example might be seen in numerical analysis, where an implementor might demand some assurance that the type system of a platform will provide sufficient numerical precision. This sort of comparison might be thought of as "type systems for type systems", since it raises similar typing substitutability questions with respect to the type systems themselves.

In conclusion, the consideration of type systems within MDA is one that has, to date, been given little treatment, and remains an avenue for research. We have asked many questions here, and provided few answers. These open questions include consideration of techniques for modelling type systems, including the notions of type and substitutability, using MDA formalisms, and in what situations these might be used.

Report from the EWMDA-2 Working Group on “MDA and Reuse”

Julian Johnson (BAE Systems, UK), **Kevin Dockerill** (BAE Systems, UK), **Joao Paulo Almeida** (Univ. of Twente, The Netherlands), **Peter Linington** (Univ. of Kent, UK), **Zhen Ru Dai** (Fraunhofer Fokus, Germany), **Oliver Sims** (Sims Architectures, UK), **Salim Bouzitouna** (LIP6, France), **Ian de Beer** (UCSSM, South Africa), **Nelly Becomo** (Lancaster Univ., UK), **Antonio Vallecillo** (Univ. of Málaga, Spain)

The group met from 13:30 to 14:15 to discuss the topic of MDA and re-use, originally to cover two main issues: (1) How to reuse in MDA, and (2) Does MDA facilitate reuse? It was agreed to concentrate on the first one, in order to be more focused and make the best use of the time allocated for discussions.

Before we went into the discussions, we felt it was useful to review the participants’ definition of the terms “MDA” and “re-use”. Thus, it was agreed that **MDA** includes (at least) the following concepts: Models and Transformations; Platform independence / PIM; separation of concerns: Business / Technology; Moving Intellectual Property (IP) from code to models; Design Explicitness / visibility. With regard to **re-use**, it was agreed that it might imply re-exploitation of IP across two or more applications/programmes, in a maintained way. Furthermore, re-use not only implies reuse of design/models, but also reuse of knowledge, skills etc. Finally, it was noted that re-use does not mean using many times in the same context, but using in different contexts (quote due to Clemens Szyperski?).

Once the basic terms were discussed and their scope and meaning was generally agreed, we moved onto the main points that the Workshop organizers suggested for discussion.

The first approach was to identify **re-use contexts**. The following list of potential re-use contexts was produced:

- Product line oriented (families of application types), component-based application development
- Same application PIM used for multiple target platforms (via PSM’s)
- Same platform / architecture model used for different applications
- Reuse of IP in capturing an organisation’s test approach as a transformation (then used to generate test info / test cases / configurations, etc.)
- Aspects / cross-cutting concerns: security, audit trails, … MDA transformations may support the development for certain aspects
- Exploitation of legacy code / applications, via modelling of wrapping

It was agreed that the **solution** might be based on applying all principles and technologies associated with MDA, with particular emphasis on

- Separation of concerns
- PIM / PSM’s
- Reusable components/transformations
- Model / component repositories

This solution has clear **advantages**, such as: (a) Most cost effective development and maintenance of existing and new applications; (b) improved knowledge and IP re-use; and (c) Explicit capture of IP. However, it also presents some **disadvantages**, such as potential performance issues; the challenges of designing for reuse; cost; and it also may have organisational implications.

After that, the group also identify a set of **issues** related to re-use such as the following:

- In order to re-use an artefact we need to be able to perform a set of tasks, such as: find it; get to it to use; maintain it; manage its lifecycle; and properly handle the possible organisational changes it might imply
- Designing for re-use may incur extra costs (development, maintenance, consequential performance, etc.). Moreover, there are difficulties in predicting use contexts
- It was also agreed that re-use implies using without changes, otherwise it is not re-use. However, some products and artefacts allow customisation and extensibility in order to facilitate re-use
- There is also the difficulty of integrating legacy systems and applications into the MDA chain

Finally, some **further work** needs to be done. In particular, the following tasks were identified:

- What are the [abstraction] criteria for development of PIM's and PSM's, an abstract platform, etc?
- There is a need for MOF repository support that is integrated with other tools
- MDA support for aspect development

EMWDA Working Group at Kent. September 8th 2004

Workshop: How to Sell MDA

David Pilfold, Domain Solutions Ltd, http://www.ooagenerator.com

Abstract

The Object Management Group's (OMG's) Model Driven Architecture (MDA) approach to software project delivery is based on producing analysis models that separate the *problem* from the *implementation technology*. For example separation of the analysis of Enterprise Resource Planning (ERP), which attempts to integrate all departmental functions across a company, from the technology e.g. J2EE or .NET platforms etc.

MDA promises many benefits in terms of raising the abstraction level of system description and future-proofing systems against changes in technology and also delivering customisable projects and off-the-shelf architectures. However, MDA usage is still in its infancy and has a long way to go before it becomes widely accepted, this is shown by a lack of mainstream MDA projects. As MDA is a huge paradigm shift in terms of how software projects are realised it is essential to create a defined transition from the early adopters to mainstream customer use. This workshop addressed these issues of how to sell MDA.

The workshop firstly discussed the characteristics of markets (early adopters, cost-cutters etc.) and identified which market types (finance, government etc) would be potential users of MDA. The consensus on this was that MDA applies to many market types. Secondly, the workshop discussed what needed to be realised before MDA could be widely accepted; MDA itself needs a better profile description and supporting software tools and standards need to be defined. The remainder of the workshop focussed on identifying the MDA stakeholders and the messages to use (and not use) in selling to these stakeholders.

The MDA stakeholders were identified as business managers, to developers and testers and also standards bodies, research and education institutes and tools vendors. Workshop members believe that the MDA sales approach and message has to be tailored to sell the benefits at different levels. These benefits range from the return on investment that can be expected to the removal of tedious coding work.

In summary it will be hard for MDA to cross the chasm from early adoption to mainstream until it is better defined and supported. A targeted sales message and mechanism to deliver it should make this leap achievable.