Model-Driven Engineering of Information Systems
Principles, Techniques, and Practice

Model-driven engineering (MDE) is the automatic production of software from simplified models of structure and functionality. It mainly involves the automation of the routine and technologically complex programming tasks, thus allowing developers to focus on the true value-adding functionality that the system needs to deliver. This book serves as an overview to some of the core topics in MDE; the volume is broken into two sections offering a selection of papers that help the reader not only to understand the MDE principles and techniques but also to learn from practical examples.

Also covered are the following topics:
• MDE for software product lines
• formal methods for model transformation correctness
• metamodeling with Eclipse eCore
• metamodeling with UML profiles
• test cases generation

This easily accessible reference volume offers a comprehensive guide to this rapidly expanding field. Edited by writers with experience in both research and the practice of software engineering, Model Driven Engineering of Information Systems: Principles, Techniques and Practice is an authoritative and easy-to-use reference, ideal for both researchers in the field and students who wish to gain an overview to this important field of study.

ABOUT THE EDITORS
Dr. Liviu Gabriel Cretu holds a full-time position of associate professor in business information systems at Alexandru Ioan Cuza University of Iași, Romania, where he coordinates the activities of the Software Engineering Lab, Centre for Advanced Information Systems, Financial Reporting and Accounting. As an IT consultant specialized in software architecture and enterprise architecture, he has developed enterprise systems for very large organizations, including the European Commission. He is a researcher, developer, and author, having published numerous peer-reviewed articles in the fields of enterprise systems, semantic web, enterprise architectures, model-driven engineering, service-oriented architecture, and business process management. He received his PhD in business information systems from the Alexandru Ioan Cuza University of Iași.

Dr. Florin Dumitriu is leading the Department of the Accounting, Business Information Systems and Statistics at the Alexandru Ioan Cuza University, Iași, Romania. He lectures in Systems Analysis and Design and Human Resource Management Systems at the same university. He is a researcher, having published numerous peer-reviewed articles in the fields of systems analysis and design, global software development, agile software development, distributed databases, and business process management. He received his PhD in business information systems from the Alexandru Ioan Cuza University, Iași, Romania.
MODEL-DRIVEN ENGINEERING OF INFORMATION SYSTEMS
PRINCIPLES, TECHNIQUES, AND PRACTICE
ABOUT THE EDITORS

LIVIU GABRIEL CRETU, PhD

Dr. Liviu Gabriel Cretu is currently coordinating the activities of the Software Engineering Lab, Centre for Advanced Information Systems, Financial Reporting and Accounting, Alexandru Ioan Cuza University, Iași, Romania. He lectures on Software Engineering and Information Systems Architecture at the same university, and he is a software architect consultant with a consistent portfolio of clients across Europe. He is a researcher, developer, and author, having published numerous peer-reviewed articles in the fields of enterprise systems, semantic web, enterprise architectures, model-driven engineering, service-oriented architecture, and business process management. He received his PhD in business information systems from the Alexandru Ioan Cuza University, Iași, Romania.

FLORIN DUMITRIU, PhD

Dr. Florin Dumitriu currently leads the Department of Accounting, Business Information Systems and Statistics at Alexandru Ioan Cuza University, Iași, Romania. He lectures on Systems Analysis and Design and Human Resource Management Systems at the same university. He is a researcher, having published numerous peer-reviewed articles in the fields of systems analysis and design, global software development, agile software development, distributed databases, and business process management. He received his PhD in business information systems from the Alexandru Ioan Cuza University, Iași, Romania.
CONTENTS

Acknowledgment and How to Cite ............................................................... ix
List of Contributors ...................................................................................... xi
Introduction ................................................................................................ xv

Part I: MDE Principles and Techniques
1. End to End Development Engineering ....................................................... 1
   Abdelgaffar Hamed and Robert M. Colomb

2. Model-Driven Engineering for Software Product Lines ....................... 51
   Jean-Marc Jézéquel

3. Quality Model for Conceptual Models of MDD Environments ......... 111
   Beatriz Marín, Giovanni Giachetti, Oscar Pastor and Alain Abran

4. Formal Model-Driven Engineering: Generating Data and Behavioral Components ................................................................. 141
   Chen-Wei Wang and Jim Davies

5. Category Theory and Model-Driven Engineering: From Formal Semantics to Design Patterns and Beyond .......................................... 173
   Zinovy Diskin and Tom Maibaum

Part II: MDE in Practice
6. Integrating Software Architecture Concepts into the MDA Platform with UML Profile ......................................................................................... 207
   Adel Alti, Tahar Khammaci, and Adel Smeda

7. Model Driven Engineering Using UML: A Pragmatic Approach ...... 229
   Liviu Gabriel Cretu

8. Model-Based Reuse for Crosscutting Frameworks: Assessing Reuse and Maintenance Effort ................................................................. 245
   Thiago Gottardi, Rafael Serapilha Durelli, Óscar Pastor López
   and Valter Vieira de Camargo
9. Using Built-In Domain-Specific Modeling Support to Guide Model-Based Test Generation ................................................................. 295
    Teemu Kanstrén and Olli-Pekka Puolitaival

10. Recovery and Migration of Application Logic From Legacy Systems ........................................................................................................ 321
    Wiktor Nowakowski, Michał Smiałek, Albert Ambroziewicz,
    Norbert Jarzebowski and Tomasz Straszak

Author Notes ........................................................................................................ 345
Index ..................................................................................................................... 347
ACKNOWLEDGMENT AND HOW TO CITE

The editor and publisher thank each of the authors who contributed to this book, whether by granting their permission individually or by releasing their research as open source articles or under a license that permits free use, provided that attribution is made. The chapters in this book were previously published in various places in various formats. To cite the work contained in this book and to view the individual permissions, please refer to the citation at the beginning of each chapter. Each chapter was read individually and carefully selected by the editor; the result is a book that provides a nuanced study of model-driven engineering. The chapters included examine the following topics:

- MDE is usually associated with software product lines where variability and variation points are the main concerns. The proposed E2EDE methodology in Chapter 1 guides the MDE process, explaining how various types of requirements may have a formal representation in the models by means of annotations with metadata. This way, the model will contain enough information for the automation of transformation decisions. The paper includes a case study where QVT is used to implement the mapping rules.
- The authors of Chapter 2 carry out a review of variability modeling methods and related tools, in order to organize the plethora of existing approaches into several classification dimensions, and provide representative examples of Model-Driven Engineering tools and algorithms exploiting them.
- Quality of the input models is as important as the transformation process. Chapter 3 presents a metamodel for quality assessment in MDE automation. This metamodel is developed using Eclipse MOF specification, then a procedure for defect detection is proposed.
- One of the main advantages of MDE is that correctness of complex software may be assessed even before the physical system is actually generated, at the level of abstract models with lower complexity. The authors of Chapter 4 show how formal techniques can be used to establish the correctness of model transformations.
- Chapter 5 applies mathematical theory to MDE where the later is regarded as just another particular instance of the engineering discipline. Two examples are presented, showing categorical arrangement of model management scenarios: model merge and bidirectional update propagation.
• Chapter 6 offers clear guidance for an MDA process implemented with UML and ATL. A metamodel for component-oriented software is discussed and then it is designed in UML using stereotypes and tags. A set of mapping rules is defined and implemented in ATL to transform components (PIM) into CORBA PSM.

• Systematic integration of Model Driven Engineering (MDE) principles within the software development process has proven to be a challenging task. Usually, one of the first questions arising is how to properly organize the MDE process itself. Chapter 7 introduces a pragmatic method to apply MDE using UML, the de-facto standard modeling language in software development. The proposed method is based on the metamodeling approach and uses UML profiles to encapsulate information about typed use cases. A Java example is then provided to validate and illustrate the method.

• In Chapter 8 we find an MDD approach aiming to facilitate the reuse of crosscutting AOP frameworks. Two experiments have been conducted to compare the proposed model-based reuse approach with the conventional way of reusing CFs, i.e., manually creating the reusable code. The authors also present productivity gains and reuse of metamodel components from one AOP framework to another.

• In Chapter 9, the authors propose an MDD approach to support automated test generation with domain specific concepts. A framework is provided for building test models using Java language together with a DSM language used to guide test case generation. A MBT tool is used to automatically create a DSM language and to apply the transformation of the test model.

• Chapter 10 presents an approach and a case study for the migration of legacy systems to new technologies. According to the authors, application behavior information can be extracted from any existing system by determining its observable behaviour and stored in the form of requirements-level models conformant to the RSL-AL—an extension of the Requirements Specification Language that serves as an intermediate language between the recovery and migration steps. Specific tools are then used to transform this model back to an application based on the new set of technologies.
LIST OF CONTRIBUTORS

Alain Abran
Department of Software Engineering & Information Technology, École de Technologie Supérieure, Université du Québec, 1100 Notre-Dame Ouest, Montréal QC, Canada H3C 1K3

Adel Alti
Department of Computer Science, Farhat Abbes University of Setif, Algeria

Albert Ambroziewicz
Warsaw University of Technology, Warsaw, Poland

Robert M. Colomb
Faculty of Computer Science and Information Systems, Universiti Technologi Malaysia, Skudai, Malaysia.

Liviu Gabriel Cretu
Business Information Systems Department, Alexandru Ioan Cuza University of Iasi, Bd. Carol I, no 11, Iasi, Romania

Jim Davies
Department of Computer Science, University of Oxford, Oxford, United Kingdom OX1 3QD

Valter Vieira de Camargo
Departmento de Computação, Universidade Federal de São Carlos, Caixa Postal 676, 13.565-905, São Carlos, São Paulo, Brazil

Zinovy Diskin
Network for Engineering of Complex Software-Intensive Systems for Automotive Systems (NEC-SIS), McMaster University, Canada and Generative Software Development Lab, University of Waterloo, Canada

Rafael Serapilha Durelli
Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo, Av. Trabalhador São Carlense, 400, São Carlos, São Paulo, Brazil

Giovanni Giachetti
Centro de Investigación en Métodos de Producción de Software, Universidad Politécnica Valencia, Camino de Vera s/n, 46022 Valencia, Spain
Thiago Gottardi  
Departamento de Computação, Universidade Federal de São Carlos, Caixa Postal 676, 13.565-905, São Carlos, São Paulo, Brazil

Abdelgaffar Hamed  
College of Computer Science and Information Technology, Sudan University of Science and Technology, Khartoum, Sudan

Norbert Jarzebowski  
Warsaw University of Technology, Warsaw, Poland

Jean-Marc Jézéquel  
Institut de Recherche en Informatique et Systèmes Aléatoire (IRISA), University of Rennes 1, 35042 Rennes, France

Teemu Kanstrén  
VTT, Oulu, Finland

Tahar Khammaci  
LINA, Université de Nantes, 2, Rue de la Houssinière, 44322, Nantes, France

Óscar Pastor López  
Universidad Politécnica de Valencia, Camino de Vera s/n, Valencia, Spain

Tom Maibaum  
Network for Engineering of Complex Software-Intensive Systems for Automotive Systems (NECSIS), McMaster University, Canada

Beatriz Marín  
Centro de Investigación en Métodos de Producción de Software, Universidad Politécnica Valencia, Camino de Vera s/n, 46022 Valencia, Spain

Wiktor Nowakowski  
Warsaw University of Technology, Warsaw, Poland

Oscar Pastor  
Centro de Investigación en Métodos de Producción de Software, Universidad Politécnica Valencia, Camino de Vera s/n, 46022 Valencia, Spain

Olli-Pekka Puolitaival  
F-Secure, Oulu, Finland
List of Contributors

Adel Smeda
LINA, Université de Nantes, 2, Rue de la Houssinière, 44322, Nantes, France

Michał Smiałek
Warsaw University of Technology, Warsaw, Poland

Tomasz Straszak
Warsaw University of Technology, Warsaw, Poland

Chen-Wei Wang
McMaster Centre for Software Certification, McMaster University, Hamilton, Canada L8S 4K1
Model-driven engineering (MDE) is an important research topic in the software engineering field. As an evolutionary step from computer-aided software engineering (CASE) methods, model-driven engineering techniques apply systematic transition from semantically rich but simplified models to other models that include more design and technical details. This approach has been reportedly applied in various fields, from embedded software to denotational semantics definition [1]. In this book we have focused our attention on identifying relevant papers proposing techniques and practical examples that could be useful for the development of information systems, the model-driven way.

Simply said, MDE is the automatic production of software from simplified models of structure and functionality. The key here is “simplified models”—the opposite of the detail-rich UML diagrams needed to execute forward and reverse engineering where the usual software design has to involve enough technical (platform-specific) details for the modeling tool to be able to generate some code. MDE is considered by some authors “the first true generational shift in programming technology since the introduction of compilers” [2], and it can profoundly change the way applications are developed [3]. It mainly involves the automation of the routine and technologically complex programming tasks, thus allowing developers to focus on the true value-adding functionality that the system needs to deliver, instead of continuously debugging various technical details inherent in the use of programming languages or specific technologies.

The core concepts used in MDE are: models and transformations. Bézivin offers a systematic analysis about the transition from “everything is an object” paradigm to “everything is a model” [4]. Just like in the object-oriented world, where an Object is an instance of a Class, in model engineering a Model is an instance of a Metamodel. Metamodels are intended to represent a set of related concepts and each metamodel defines a language for describing a specific domain of interest. If one can specify
Introduction

rules to transform metamodel A into metamodel B, then transformers may be implemented to generate new models in the target-language B from input models written in the source-language A. Similar ideas that MDE is all about models, meta-models and transformation rules may be found in [5] where the author names it the model-driven engineering "megamodel", as well as in popular works such as [6, 7, 8] including the MDA guide [9].

Models are created using domain-specific languages (DSL, sometimes called domain specific modeling languages—DSML—if it is a graphical language). DSLs are tailored to directly represent the concepts of an application domain as modeling primitives. To create a DSL one needs a domain analyst [10] which is a person who examines the needs and requirements of a collection of similar systems. Most DSLs are supported by a DSL compiler, called the application generator [11], that generates applications from DSL programs. Various modeling languages dedicated to MDE have been proposed, such as Alloy [12] and B [13]. While the main advantage of building a compiler or interpreter is that the implementation is completely tailored towards the DSL, an important problem is the cost of building such a compiler or interpreter from scratch, and the lack of reuse from other (DSL) implementations [14]. At the same time, the Unified Modeling Language (UML) remains the widely accepted general-purpose modeling language in the software industry today. The main advantage of UML is that it can be specialized to become a domain-specific language (DSL) using UML profiles. Thus, a standard UML model element may be categorized according to a different metamodel by means of stereotypes and tagged-values. There are many researchers who have identified the natural relationship between UML profiles and the metamodeling phase in MDE [15, 16], while a large number of papers are proposing domain-specific profiles such as for critical infrastructures [17], distributed service models [18], embedded systems [19], web services [20], and semantic web services [21].

Although there are still debates on the terminology, some preferring model-based software development [22], others model-driven development [2, 3, 23] or model-driven engineering [4], it is widely accepted that the Model Driven Architecture (MDA) [24] introduced by Object Management Group, is the reference implementation of MDE. MDA is often described as “the future of software engineering,” popular works offering
statements such as “MDA is about using modelling languages as programming languages rather than merely as design languages”[25].

The book is organized in two sections. Section one—MDE Principles and Methods—is a collection of papers needed to lay down the theoretical foundation of MDE for researchers and professionals alike. The second part—MDE in Practice—mainly focuses on real-world scenarios where MDE has been applied as well as some lessons learned along the process.

REFERENCES


Liviu Gabriel Cretu and Florin Dumitriu
PART I

MDE PRINCIPLES AND TECHNIQUES
1.1 INTRODUCTION

The complexity of producing large-scale software systems is increasing due to the increased complexity of requirements. Technologies are volatile for many reasons but enhancing the quality of services is among clear reasons justified by software providers. For example, java platform versions and Google Chrome browser has adopted new browsers technology. The functionality of browsers is already crafted (i.e. Mozilla) but putting it into a new fashion is because of security, performance, reliability, and etc. On other hand, service-based system (SOA) has emerged as new engineering discipline encourages organizations to integrate their systems in a seamless manner. These highlight questions like

1. How to extend the traditional methods (Code-based) in a longlived architecture to deliver these new businesses?
2. How to provide an effective integration with legacy systems?
3. If a decision is made to change technology (acquiring new quality such as security and performance) is the design easily adaptable?

The trend now is proposing model-based engineering approach which means separating concerns where software development is driven by a family of high level languages [1]. To this end abstraction level is raised above 3GLs which increases re-using theme and put software artifact into a situation of core asset. More importantly formalizing of these artifacts (i.e. metamodels) leads to realize the benefit of high degree of automation. This means a machinery of specification (i.e. UML), synchronization and management of these models are essential. Thus, software not a program became like an information system itself. Thereby the crafting of code is becoming a manufacturing process not a personal skill.

Model Driven Architecture (MDA) is a new software development method following that trend. It raises abstraction level and maximizes re-use. Using MDA, we will be able to work with software artifacts as assets which from the software engineering perspective is a major success factor for reliable and fast development. The philosophy of MDA is to do more investment on software artifacts (models) to increase their efficiency, leading to systematic and more powerful mechanisms for software re-use.

MDA is an aspect of the more general discipline of software reuse. The synergistic relationship among MDA and the longer-established areas of Design pattern and software product line engineering (SPL) [2] has been studied as part of the present research [3,4].

The problem of producing a complete solution from specification through to implementation is still a long standing research aim, and because of the mapping gap from PIM to PSM, E2EDE has emerged. Most of previous work in MDA has been on infrastructure and components. Therefore the major question in this paper is how to write a program in MDA?

End to End development engineering (E2EDE) is a new trend to software engineering, proposed to answer that question, which uses MDA methodology and exploits some experience from Software Product Line (SPL) and lessons from Design Pattern (e.g. nonfunctional requirements) to automate the development from specification through to implementa-
tion. In doing so, we need to investigate the relationships among MDA, SPL, and design pattern and how MDA can fit on them. Therefore the contributions of the paper are the following:

1. We present E2EDE to automate the mapping process as realization to MDA which is intended to produce products without entirely writing code.
2. We discuss the relationship between MDA, SPL, and design pattern and how MDA can fit in the both longer established re-using approaches.
3. We share some lessons learned and challenges of MDA software engineering in practice.

Domain engineering is the key concept utilized from SPL which realizes breeding of a number of products that have similarity and some sort of variability in features. Design pattern in the history of software engineering has concerned with linking the design with nonfunctional requirements. Therefore, Nonfunctional requirements is borrowed as a concept.

The paper is organized as follows: Section 2 describes MDA as a major method used in E2EDE. In Section 3 we explain the problem through example of mappings from QVT specification. The proposed E2EDE methodology is discussed at Section 4. The concepts of E2EDE are validated by case study at Section 5. Sections 6 and 7 describe the relationship among MDA, Design Pattern and SPL. Section 8 draws on the principle of MDA and shows a case of strategic PSM. In Sections 9 and 10 we discuss MDA and E2EDE implementation aspects respectively. In Section 11 the realistic value of our approach with existing platforms is investigated. A conclusion is presented in Section 12.

1.2 MODEL DRIVEN ARCHITECTURE (MDA)

MDA is a new development paradigm initiated by the OMG aimed at software development driven by model [1]. In this case, a Platform Independent Model (PIM) is used to specify application behavior or logic by using MOF or a MOF-complaint modeling language [3]. This step represents a
problem space in an application-oriented perspective. A Platform Specific Model (PSM) is used to realize a PIM. It represents a solution space from an implementation-oriented point of view. Therefore, a transformation from the problem space to the solution space is required. The automation of this process is the ultimate goal of MDA. Thereby, when we need to change the application, changes will be in only one part (PIM) without affecting implementation technologies (PSM). Conversely, when the platform such as SQL Server is changed retargeting a new platform for example new version (has enhanced feature), we need only to select the appropriate PSM and then regenerate the code not only without modifying PIM but also this time re-using most of the transformation. Productivity becomes higher and cost is reduced due to the increased reuse of models. In addition, maintenance becomes cheaper. It is worthwhile to observe here that MDA is working with models as assets that can be reused once the initial investment is made. MDA depends on a well established code-base.

1.2.1 MDA TRANSFORMATION PROCESS

The transformation from PIM to PSM is done by a mapping function, which is a collection of mapping rules. In this case some or all of the content of the target model is defined. It is expected that when MDA automates this process, development efficiency and portability would significantly increase. In addition, the mapping function can be repeated many times (re-used) for different applications using the same PIM and PSM metamodels. MDA also helps avoid risks of swamping the application with implementation detail which causes model divergence [5].

The steps of designing a system is to create a conceptual model by designers for application requirements and developing another implementation model to map the first into the second. But this might involve many sub activities. However, we can divide MDA into two major processes [1]:

1. Model to Model mappings: The mapping in this stage does not consider any specific characteristics or special cases that apply to technology or platform (called M2M). The result of this phase is still high level model but for code (PSM instances).
2. Bringing in a Particular Platform: The goal of this mapping (sometimes called M2T) is tailoring the conceptual model to specific technology. Different platforms have different features and constraints so step 1 will be refined to conform to features of one of the selected platform. The result in this phase is expected to be context dependant code expressed in a platform concrete syntax. In fact, we intended to use the word bringing to denote applying the principle of MDA in de-developing standard PSM.

1.2.2 METAMODELING

The conceptual model of the design language such as UML data model (i.e. class diagram) is called a metamodel, which has concept like a class. A particular design in a design language is called model instance like student class in the student record system. This model instance can be visualized by using UML model instance (i.e. object diagram) but also MOF has similar model instances metamodel. A metamodel defines a schema for database called a repository. The population of this repository is the model instance. Formation rules of the metamodel are expressed as constraint on the repository [6]. A metamodel represents syntax of a modeling language. If the metamodel tells the designer how to create a model instance, it is said to be concrete syntax [6]. If it does not, it is said to be abstract syntax. Therefore, sometimes rendering conventions augmented with abstract syntax to generate concert syntax like MOF instance specification [7].

Model-based design that relies on a repository (tables or data structure) for storing a complex object (design), is the key art behind the MDA approach. For example QVT mappings are specified as patterns on schemas, or metamodel [8]. In this way, information contained in the models is separated from the algorithms defining tool behavior, instead of being hard-coded into a tool. The algorithmic part of tool communicates with models via an abstract program interface (API), which affords the facility to create, modify and access the information in models [9]. Further, MDA tools can transform model instances into various forms. For example, the mappings from PIM to PSM takes PIM metamodel instances from the
instance repository and turns it into corresponding instances updating the target repository, moving from problem space to solution space.

This mapping activity is done using standard language independent of the source and target. The metamodel of the mapping from end to another end can also be re-used as an asset.

1.2.3 QUERY, VIEWS, AND TRANSFORMATIONS (QVT)

Two kinds of transformation are recognized in MDA community:

- Horizontal, that does not change the abstraction level, for example from PIM to PIM which is used when a model is enhanced, filtered or specialized (mapping from analysis to design),
- Vertical, that changes the abstraction level, for example projection to the execution infrastructure. Four types of these transformations are categorized in [10].

There are tools to specify such mappings, such as query-view-transform QVT [8]. QVT is an OMG standard which helps us to specify rules for the transformation function. QVT uses the concept of predicate (expression evaluated to true or false) and pattern (set of expressions) in much similar way as prolog programming language.

The intended scenario of writing a program using MDA will be demonstrated first by an example then below in our case study where the mapping task is the major activity. Generally the application development is a process involves many transformations so vertical and horizontal or combination of them might be used.

1.3 MDA EXAMPLE AND MAPPING PROBLEM

We will use the QVT specification [8] example of object to relational mappings in order to understand where the problem is in this context. For sake of simplicity we focus on part of the mappings between PIM (object model) and PSM (relational model). This example shows the mappings take place between simplified UML2 metamodel in Figure 1, the PIM, and the PSM in Figure 2.
FIGURE 1: Simplified UML2 metamodel from QVT [8].
FIGURE 2: Simplified relational database model from QVT specification [8].
The mapping between conceptual models and relational schema is well established in the database. The general idea is that classes map to tables, packages map to schemas, attributes to columns, and associations to foreign keys. We will discuss part of this mapping informally then we will show simple QVT rules for that.

In Figures 3 and 4 examples we have a relation (use to specify rules of mapping source to target) named PackageToSchema, and ClassToTable. Both have two domains that will match elements in uml (our PIM) and rdbms (our PSM). The relation ClassToTable specifies the map of a class which has attribute name with value equal the variable “cn”. All classes instances in uml repository will populate this variable one by one. For example if the model instance of our PIM is student record system, these will be person, lecturer, student, etc (M1). If the precondition is satisfied by this way the enforce clause is very similar to a checkonly clause. If there is an instance in the rdbms repository satisfying the pattern expression, then the enforce clause behaves as a predicate, with the value true. If no such instance exists, then the QVT engine will create one.

The mapping takes structural patterns in the M1 PIM model (problem domain) instance into in some cases quite different structural patterns in the M1 PSM model (implementation domain). The patterns are described in the M2 metamodel. Thereby that mapping is grasped as a part of the process of specifying and implementing the system. This process in most traditional software engineering methods is done manually. The ultimate goal of mapping is having a way to be able relate M0 instances of the PIM to M0 instances of the PSM. But the standard document of MDA [1] does not specify how to do that.

MOF specification has described how to create instances from MOF-based metamodel. When we talk about MOF we mean M2 level in the OMG hierarchy. Both create and destroy methods for objects and links are specified as MOF standard operations for creating and deleting dynamic objects [7]. So the objects created and destroyed by these methods are of kind M1 objects for M2 metaclasses. By this way an instance model will be created for M2 metamodel elements similar like having student, lecturer, and course, etc., instances. UML standard also specifies methods for creating instance model for M1 objects which are M0. Hence since like the instance model of record system is UML instance model then it should be
able to create m0 objects using UML model instance inherited capabilities: create and destroy of objects and links.

The intended scenario for mapping is that an application using application terms have a facility to create objects which is PIM instance model. On other hand the implementation model that is PSM creates objects corresponding to that objects using PSM concepts and terms which represents a design vocabulary. To this end still in the MDA development (i.e. using MDA to solve problems) the question is how to do this process which requires finding concrete technical mapping methods.

A general method to approach this problem is needed whereby one could develop application without entirely writing code. Here the task for developer/modeler is a function of specification where application requirements, implementation and mapping metamodels are presented using design languages such as UML/MOF and mapping language such as QVT. In this case mapping and synchronization among models are performed by toolset.

1.4 PROPOSED E2EDE DESIGN & IMPLEMENTATION

This section demonstrates key points of E2EDE and explores most important aspects that should be addressed.

![relationPackageToSchema /* map each package toa schema */

{  
  Domain uml p:Package {name=pn}
  Domain rdbms s:Schema {name=pn}
}

FIGURE 3: Mapping package to schema QVT rule.
1.4.1 INTRODUCTION

End to End development engineering (E2EDE) is a novel paradigm intended to automate software development from the specification end (i.e. object model) to the implementation end (i.e. relational model) using the MDA approach. The central issue is filling the mapping gap between PIM and PSM in MDA.

Theoretically E2EDE is inspired from an investigation of the synergistic relationship among MDA, SPL, and design patterns as we will see in Sections 6 and 7. The rationale behind establishing this relationship was from literature SPL and design patterns have long history of re-use software development. So they are longer established reuse methods. MDA is a more recent stream of re-use. E2EDE engineering is going to exploit this relationship to achieve its mapping goal. The key concept in SPL is variability which gives customization or configuration options. Variant feature is a place in the software artifact can be populated by at least one variant at a time from a set of variant. For example, if color is variant feature then Red is one variant. We conceive design decisions as variation points and the design choices as variants populating these points. This comes from the observation that PSM as a design artifact has different structures most properly lead to different architectural qualities. Therefore, to model

```
relationClassToTable /* map each persistent class
to a table */
{
    Checkonly domain umlC:Class {
        name=cn
    }
    Enforce domain rdbmst:Table {
        name=cn
        when { PackageToSchema(p, s); }
        where { AttributeToColumn(t, c); }
    }
}
```

FIGURE 4: Mapping class to table QVT rule.
design decisions we need to represent variability explicitly in the PSM. The study of the design pattern approach highlights the importance of the relationship between design and requirements, specifically nonfunctional requirements, which is proposed to be modeled in the PIM. Still there is a research gap in these areas on how to map nonfunctional requirements with design decisions systematically. The problem has been looked at from one dimension, for example SPL has concerned only with variability without considering NFR such as [2,3,11, 12] while design pattern has recognized the impact of NFR dimension without variability in an explicit way [13-15].

However, the benefit here is PSM construction could be automated effectively because of consideration of design quality and management of single PSM. Hence, documenting variability in architecture and modeling nonfunctional requirements explicitly will become major activities during the development process. This section demonstrates key issues arising when we tackle E2EDE. Further, these issues have been applied to a selected case study to evaluate the possibility of the proposed engineering approach.

The ultimate goal of E2EDE is to provide a method of generating a solution from one specific source to a specific target like for example, from object-model to relational model. The advantages of E2EDE are reflected in the modeling support for the concepts in the domain and the ability to do more than general-purpose languages do, in addition to reduction of cost.

1.4.2 THE STEPS OF E2EDE PROCESS

In this section we will see the main steps of E2EDE. It will be detailed step by step and finally summarized as shown in Table 1.

1.4.2.1 MODELING VARIATION POINTS IN PSM (TASK A)

The key concept is to document variability in the PSM, which can be thought of as an abstract data type similar to the logical level in database
systems. Usually, a solution is specified firstly at high abstraction level before rendered into a database technology. Variability in this PSM exposes different design decisions from the design space. The design decisions we mean in this context are architectural elements. Since it is possible to create different implementations from a generic specification, variability management concepts and techniques from SPL experiences are utilized to document design decision variants explicitly in the architecture. For example we will see in Section 5 two types of connection: Topic (indirect) and Queue (direct) for a messaging system. The variability difference is that in topic multiple subscribers receive a message while in queue only one subscriber is allowed to receive. This variability can be modeled as two different structures at PSM or formally as variants populating the connectionType variation point.

Since standard UML does not have a variability concept, modeling variability in a PSM we need to use a profile to allow us to specify the variability ontology. A profile is a special domain language used as an extension mechanism to UML model elements while keeping their syntax and semantic intact. Proposed metamodels and profiles in the literature

<table>
<thead>
<tr>
<th>TABLE 1: The steps of E2EDE process.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tasks</strong></td>
</tr>
<tr>
<td>A. Modeling Variation Points in PSM</td>
</tr>
<tr>
<td>B. Analyzing Variability and categorized based on PSM/mapping, and functional/non-functional</td>
</tr>
<tr>
<td>C. Developing Profiles for variability &amp;NFRs</td>
</tr>
<tr>
<td>D. Modeling of Non-functional Requirements in PIM and classified into package/class level</td>
</tr>
<tr>
<td>E. Developing model-model mapping rules.</td>
</tr>
<tr>
<td>F. Packaging mappings variability rules</td>
</tr>
<tr>
<td>G. Implement the system</td>
</tr>
</tbody>
</table>
such as [2,12] can allow an architect to identify specific variation points, constraints and dependencies that indicate different relationships between variation points (VP) and variants (V), VP and VP, etc.

Because we are using design model (i.e. class diagram) the proposed profile here is different from these because there is no need for dependency and constraints concepts. They are built-in mechanisms whose semantics is specified with the mapping process and NFRs. Also, there is no need for open and closed concepts which gives the ability to add new variant or variation points because all are closed in this situation (MDA works above 3 GLs).

Therefore, developing a suitable variability MOF-profile is an essential part for the solution presented by E2EDE. In fact there are alternative ways to model variability and nonfunctional requirements concepts using a profile. The method we have chosen will help produce a working system.

The variability ontology needed includes the concepts variant indicated by <<V>> stereotype, variation point indicated by <<VP>>, and an ID tag attribute to identify each VP.

In Figure 5 the UML metaclass class is extended to represent variant and variation point. A tagged value extension mechanism is used to model identifier and type meta-attributes. Tagged values are additional meta-attributes assigned to a stereotype, specified as name-value pairs. They have a name and a type and can be used to attach arbitrary information to model elements. For instance, if we need to model ConnectionType (the two kinds of connections in messaging system) variation point we use the stereotype <<VP>> and for its variants Queue and Topic we use two <<V>> at class level. Then the tag for ConnectionType will be VPID = 1 and default can take the value Direct. The effect tag of variant specifies design decision consequences like resource consumption.

1.4.2.2 VARIABILITY ANALYSIS (TASK B)

A taxonomy for variability has emerged from our analysis of variability in software architecture artifacts. They could be called Nonfunctional variability, Functional variability and Mapping variability. SPL has been focused mainly on functional variability. An extensive analysis of this
can be found in [16]. Although our proposed method includes this sort of variability, it highlights the influence of Nonfunctional variability in the design. Most of the issues discussed in Section 6.3 are of this kind. Mapping variability can be seen in the problem of mapping superclass/subclass structures from object model into relational model. It is not like the others because the variation points are in the transformation, not in the PSM (i.e. the mappings are parameterized). In this case the mapping is from an object model as a source to the relational model as the target. The former specifies objects and relationships between them which may includes superclass-subclass relationship in a class diagram, while the latter specifies relations and their structure. The metamodels and mapping using QVT of these are well described in [8]. The target does not include a structure corresponding to superclass-subclass in the source. To solve this deficiency four options are suggested for this mapping in the standard database literature [17].

Generally these options can be classified into single-relation and multiple-relation approaches named SR and MR respectively. In the SR approach a table for superclass attributes will be created with subclass attributes included as optional while in MR approach table for each sub-
class will be created. Two implementation techniques are available for both. SR can be implemented by introducing one type attribute indicating the subclass to which each tuple belongs (null values will introduced), or multiple boolean type attributes can be used (allowing overlapping subclasses). MR has as one option with super class attributes duplicated in each subclass table and another option to share a key among superclass and subclass tables.

For example, the option of one table for the superclass with subclass attributes included as optional is a good design in terms of performance for SQL navigation, at a cost of increased table space and increased integrity checking.

1.4.2.3 MODELING OF NON-FUNCTIONAL REQUIREMENTS (TASK D)

The E2EDE methodology considers NFRs as first class objects which allow a PIM metamodel to be more informative. The separation of concerns (i.e. PIM-PSM) of MDAeffectively supports their representation.

Functional requirements are functions that the developed software must be capable of performing, while nonfunctional requirements (NFRs) inform the design choices as to how functional requirements are going to be realized in software products [16]. There is no one agreed definition because of the extremely diverse nature of NFR. In fact, practices like in design pattern shows a single NFR can have different semantic interpretations (impact on implementation) within the same application. These can be called impact factors. For example in our case study, connection types, session types, and message types are impact factors affecting performance positively or negatively. There is confusion in term usage where a term sometimes refers to the nature of the requirement and sometimes refers to the design decisions. We will be using the term NFR to denote the nature of the requirement so a PIM metamodel is the place where we can define specific NFR types.

The difficulty of modeling and integrating explicitly NFRs (additional constraints) within the context of functional requirements is the fact that NFR affects the system as whole [18]. Non-functional requirements