Transformations Co-Evolution in Response to Meta-model Evolution: A Systematic and Automatable Approach

By

Shehla Zeb

304-FBAS/MSSE/F09

Department of Computer Science and Software Engineering

Faculty of Basic and Applied Sciences

International Islamic University Islamabad

2014

- Software engineering.
- Computer software.
- Meta-models.

Transformations Co-Evolution in Response to Meta-model Evolution: A Systematic and Automatable Approach

Researcher
Shehla Zeb
304-FBAS/MSSE/F-09

Supervisor
Atif Aftab Ahmad Jilani
Assistant Professor
Fast-NU Islamabad

Co-Supervisor
Miss.Saima Imtiaz
Lecturer DCS/SE
IIUI

Department of Computer Science and Software Engineering
Faculty of Basic and Applied Sciences
International Islamic University Islamabad
2014

بسم الله الرحمن الرحيم

Department of Computer Science and Software Engineering
International Islamic University Islamabad

Date:__________

It is a certificate that we have read the thesis submitted by Miss. Shehla Zeb and it is our decree that this dissertation of satisfactory standard to certify its acceptance by the International Islamic University Islamabad, for MS degree in Software Engineering.

<u>COMMITTEE</u>

External Examiner
Dr. Amir Nadeem
Associate Professor
MAJU, Islamabad

[illegible]

Internal Examiner
Muhammad Nasir
Lecturer, DCS & SE
IIUI Islamabad

[illegible]

Supervisor
Mr. Atif Aftab Ahmad Jilani
Assistant Professor

[illegible]

Co-Supervisor
Miss. Saima Imtiaz
Lecture, DCS & SE
IIUI Islamabad

[illegible]

Acting Chairperson
Miss. Asma Batool
DCS & SE, IIUI Islamabad

[illegible]

A Thesis submitted to Department of Computer Science and Software Engineering, International Islamic University, Islamabad
As a partial fulfillment for the award of the
Degree of MSSE

Dedication

I would like to dedicate this research work to my Late Uncle, Khan Meher Dad, and my parents. I have no dubiety in my mind that without their persistent encouragement, provision and counsel I could not have completed this process.

Acknowledgment

I would like to acknowledge the inspirational instructions and guidance of Mr.Atif Aftab Ahmad Jilani who gave me the initial impetus to study Model Driven Engineering. I would like to thank him along with Dr. Uzair, Dr. Zohaib and Miss Saima Imtiaz for their kind support, continuous guidance and worthy co-ordination. Their persistent support and inspirational guidance made me able to go through the entire process successfully. They have been very generous in their support and many times, they have pervade some worthy impressions, views, criticism and recommendations. Lastly, I would like to thank Mr.Jeff Gray whose valuable comments, moral support and suggestions were obliging and worthwhile.

Thank you all for being so kind and gentle

Declaration

I hereby declare that the research presented in this thesis is my own work, excluding where otherwise acknowledged and that the thesis is my own composition. No part of the thesis has been previously presented for any other degree.

Date: [redacted]

[redacted]

Full Name

Table of Contents

List of Figure

List of Tables

Abstract

In the context of model driven engineering, models and transformations are treated as first class citizens during the software engineering process. These models and transformations are principally based on meta-models. Like any software system, meta-models are subject to evolutionary changes due to several reasons. When a meta-model evolves, all the dependent artifacts, e.g., models and transformations become inconsistent with respect to the new version of the meta-model. Consequently, the co-evolution of the dependent artifacts becomes essential. The researchers' divide the problem of co-evolution into three sub problems based on the relationship of meta-model and its related artifacts. They addressed each problem separately. The researchers investigated the co-evolution of meta-model and models intensely. However, they paid less attention to the co-evolution of meta-model and transformations and the co-evolution of meta-model and editors. Due to the intrinsic nature of transformations, the researchers have not sufficiently investigated the co-evolution of meta-model and transformations. Few of the researchers used the difference based and operator based approaches for transformations co-evolution. Therefore, these approaches have the inherent issues with some additional issues particular to the transformations co-evolution problem. For example, these approaches do not take into account the difference in the nature of models and transformations. They ignored to contemplate the intelligence the transformations employ for generating target model elements. These aforementioned issues make the transformations co-evolution process cumbersome and impractical. This research proposes a systematic and automatable approach to transformations co-evolution that overcomes the issues of the existing approaches. The proposed approach employs a relationship based change propagation mechanism, which considers the model element usage by transformations and the intelligence that transformations employed to generate target elements. It uses the change model and trace model for analyzing the impact of model element change on transformations. It also ensures the correct co-evolution, which might not always be possible in case of the existing employing existing co-evolution approaches.

The evaluation showed promising results when applied on the considered examples. Total nine well-known changes are introduced to the meta-model and its impact on transformations is analyzed. The extension of trace meta-model with the elements of considered transformation language assists in identifying the exact part of transformation, which is impacted by the change, therefore, ensuring the correct change propagation. Our approach is significantly different from the existing approaches in detecting and propagating change to transformations. To signify the difference and novelty of the approach, the proposed approach is compared with the existing approaches. The comparison showed that the exiting approaches ignored the impact analysis activity, which is crucial part of the co-evolution process to determine the cost of change and to identify the model element usage by the transformations. The comparison also demonstrated the difference in the ways that two versions of meta-model is compared. The evaluation signified the proposed approach's generality, correctness and applicability on a variety of transformations languages and meta-models. During evaluation, it is observed that traces for the model elements, e.g., associations that are transformed implicitly by the transformations are not captured by trace meta-model and requires further investigation. We applied the proposed approach on three different examples to demonstrate its applicability. The transformations in each example are specified using different dedicated transformation languages.

Chapter. 1 Introduction

1.1 Introduction

The key objectives of Model Driven Engineering are to improve productivity, quality and cost effectiveness of software applications. Seeing models as first class citizens of the software development process, and employing model transformations to automate the implementation accomplish these objectives.

MDE is gains attention with time as a technique to leverage intellectual property and business logic malleable to technological advancements via shifting focus from code to modeling [1]. MDE is based on three main concepts, i.e., meta-models, models and transformations. Meta-models lie at the heart of MDE. Models conform to meta-model. Transformations are defined based on some source and target meta-model and are employed to generate target models from source models [2]. Meta-models evolve over time because of various reasons such as design refinements and requirements changes [3, 4]. Meta-model evolution might influence its related artifacts and they, therefore, need to be adapted to the evolved meta-model [2, 5]. Changing meta-model and propagating those changes to its related artifacts are termed as evolution and co-evolution respectively by the researchers [6, 7]. Researchers investigated the model co-evolution largely during last several years. However, transformations co-evolution does not only consider the "DomainConformence" relationship, it also takes into account the intelligence that transformations employed for producing target model elements [8]. Therefore, the problem of transformations co-evolution is intrinsically more complex than the problem of co-evolution of models and researchers paid less attention to transformations co-evolution.

To co-evolve transformations, few approaches have been proposed until now. Generally, most of them tend to employ techniques devised for model co-evolution [2]. The approaches proposed to transformations co-evolution focused on devising migration strategies [8, 57]. They assumed that the meta-model change history is available which might not always be possible. Some of the existing approaches overlooked the fact that the impact of meta-model evolution on transformations not only depends on the type of changes but also on the usage of elements in transformations. Many of the existing approaches to transformation co-evolution employed the change classification scheme proposed for model co-evolution [9]. While the change classification scheme proposed for transformation co-evolution, classifies meta-model changes from adaptation automation point of view [8]. This classification scheme does not reflect the impact of changes on transformations.

In this dissertation, we propose a systematic and automatable approach to transformations co-evolution. The approach supports both meta-model evolution and transformations co-evolution. A traceability meta-model is proposed to capture and store dependency relationship between meta-model and transformations. The trace links illustrate the usage of model elements in transformations and assist in change impact analysis. A change meta-model is employed to capture changes introduced to the meta-model. This helps in identifying which type of changes are introduced to the meta-model. A change classification scheme is proposed to classify changes according to its impact on transformations. In addition, we propose a transformation classification scheme, which will assist in determining the required steps to co-evolve transformations. The main steps of the proposed approach are:

- Establish relationship between model elements and transformations and capture it as trace model (that conforms to trace meta-model).

- Capture and store the changes, introduced in the new version of meta-model, as change model (that conforms to change meta-model).
- Classify the meta-model changes according to the proposed classification scheme.
- Analyzing the impact of model element change on transformations by using the change model and traces.
- Classify transformations based on the proposed transformations classification scheme.
- Employ relationship based change propagation mechanism to propagate changes to transformations.

To validate the generality, correctness and applicability of the proposed approach, the approach is applied on three transformation examples. These examples are taken from Atlas Transformation Language Zoo (ATL Zoo) [90]. The transformations of these examples are specified using Atlas Transformation Language (a transformation specification language) abbrivated as ATL. We converted the transformations of two examples from ATL to Query View and Transformations (QVT) and Kermeta (these are also transformation specification languages), so, that we can demonstrate the applicability of the proposed approach on a variety of transformations languages. The meta-model defined in these examples are either MOF based or EMF/Ecore based, which will validate that the proposed approach is meta-model independent.

The remaining part of this chapter is organized as follows: section 1.2 defines a problem statement for this research. Section 1.3 presents motivation for this research. Section 1.4 states research objectives. Section 1.5 presents research questions we will investigate in this research. Section 1.6 describes the research method we followed for this research. Section 1.7 describes the proposed solution for transformations co-evolution. Section 1.8 discusses the contribution of this research. Section 1.9 presents the results of the research. Section 1.10 presents the outline of the dissertation.

1.2 Problem Statement

Like every type of software component, meta-models are also subject to evolutionary changes, for example, if a new version of meta-model is released; than all the artifacts based on old version of meta-model become inconsistent with respect to the new version of meta-model and need to co-evolve/migrate accordingly.

Let suppose some people are using software that uses old version of meta-model. Then, with the release of new version of meta-model all the artifacts, e.g., models and transformation rules, etc., based on the old version of the underlying meta-model become inconsistent and they all will rely on the availability of migration rules to adapt/co-evolve their models and transformation rules contents to new version. On the side of the software developer, the editor of the meta-model should be aware of the way in which changes to the meta-model layout will affect the compatibility to existing data in terms of meta-model compliance and interface compatibility, so that the tools that the modeling software offer can still be used by customers of the previous versions. This problem does not only affect the relationship of the software vendor to a customer. Since different persons can take the roles of the meta-model editor and the software engineer who implements the modeling tools, the knowledge of the impact of the meta-model changes is also relevant for the internal software development process.

1.3 Motivation

Meta-models lay at the center of model driven engineering idea. Both models and transformation rules are tightly coupled with meta-model. Models conform to meta-model and transformation rules are specified upon meta-model. Like any other software system, meta-models are also under the pressure of evolutionary changes due to design refinements and emerging requirements.

These evolutionary changes cause inconsistency in models and transformation rules with respect to the new version of meta-model. Let suppose an organization is using software that creates models based on old version of meta-model say MM. Now a new version of meta-model is released let suppose MM'. Now the models that conform to meta-model MM and the transformation rules based on it became inconsistent with respect to meta-model MM'. To use models and transformation rules with new version of meta-model they need some migration/co-evolution mechanism.

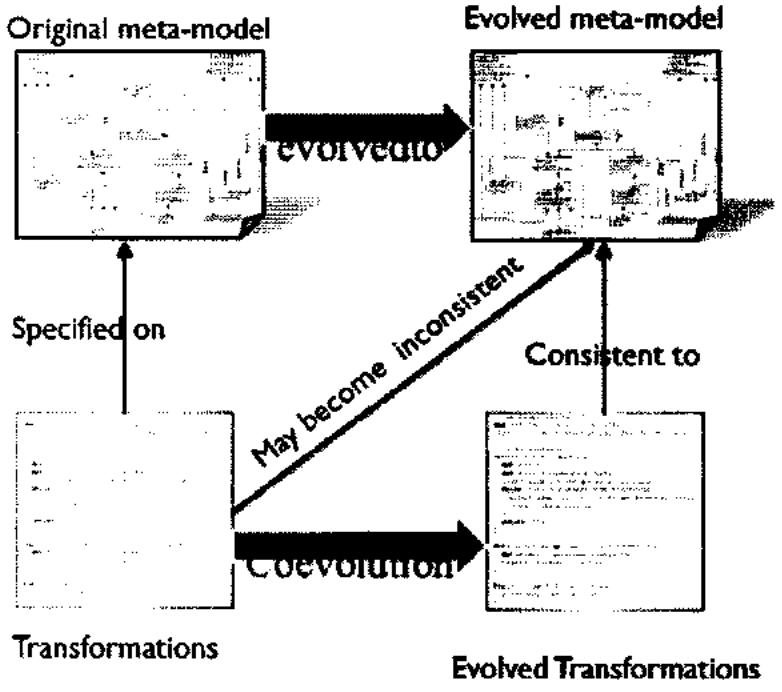


Figure 1.1 meta-model evolution problem

This research aims to provide the co-evolution mechanism that would assist user automatically in co-evolution/migration of legacy transformation rules to the new version of meta-model.

1.4 Objectives

The key objectives of this research are:

- To evaluate the impact of meta-model evolution on model transformation rules
- To model relationship between model transformations and meta-models.
- To provide mechanism for legacy model transformations adaptation for new version of meta-model upon which they are specified.
- Provide a systematic and automated approach that supports the whole co-evolution. process from meta-model evolution to transformations co-evolution.
- To assist user automatically in adapting model transformation rules.

- To perform the impact analysis activity for identifying the exact parts of the transformations that are affected by the model element change.
- Evaluate the proposed Approach for transformations co-evolution by applying on the available examples.

1.5 Research Questions

Meta-model evolution is an obvious and essential step in the development life cycle of meta-models. However, this evolution affects the entire set of artifacts depending on the old version of the meta-model. Transformation rules are one of the important artifact affected by meta-model evolution. Therefore, adaptation/co evolution of transformation rules is an essential step after meta-model evolution. The problems we have to investigate here are:

- How to detect the impact of meta-model evolution on transformations?
- How to co-evolve transformations with the evolution of meta-model?
- Can we automate the process of evolution? If yes, up to what extent can we automate this process?

1.6 Research Method

To explore the problem outlined above, we conducted this research by employing the method outlined in Figure 1.2 and discussed in this section. The shaded squares represent the three main phases of the research, which are discussed below while the simple squares represent the inputs and output to these phases.

First, the analysis phase is carried out by studying the existing co-evolution approaches, change classification schemes, co-evolution strategies, traceability techniques employed in different related research areas, change categorization schemes and co-evolution in other related areas. This has led to the identification of many important requirements like change classification scheme particular to the transformation co-evolution process, traceability support for co-evolution problem and transformation classification scheme that has not been recognized and employed previously in the co-evolution research.

The designing phase involved proposing and designing a systematic and automatable approach for managing co-evolution of meta-model and transformations. The co-evolution issues identified in the analysis phase are resolved by designing an automated and systematic approach and the identified requirements for a novel approach are fulfilled.

The evaluation phase comprised assessing the proposed approach for managing transformation co-evolution by applying it on three examples chosen from Atlas Transformations Language Zoo (ATL Zoo). To alleviate possible threats to validity of the research the examples used in the evaluation phase is different from the examples used for demonstration purpose. In addition, transformations in each example are specified using different transformation language. The strengths and weaknesses of the approach are synthesized from it applicability on different examples, particularly from how accurately the changes are propagated, which type of changes are identified and how accurately transformations are co-evolved.

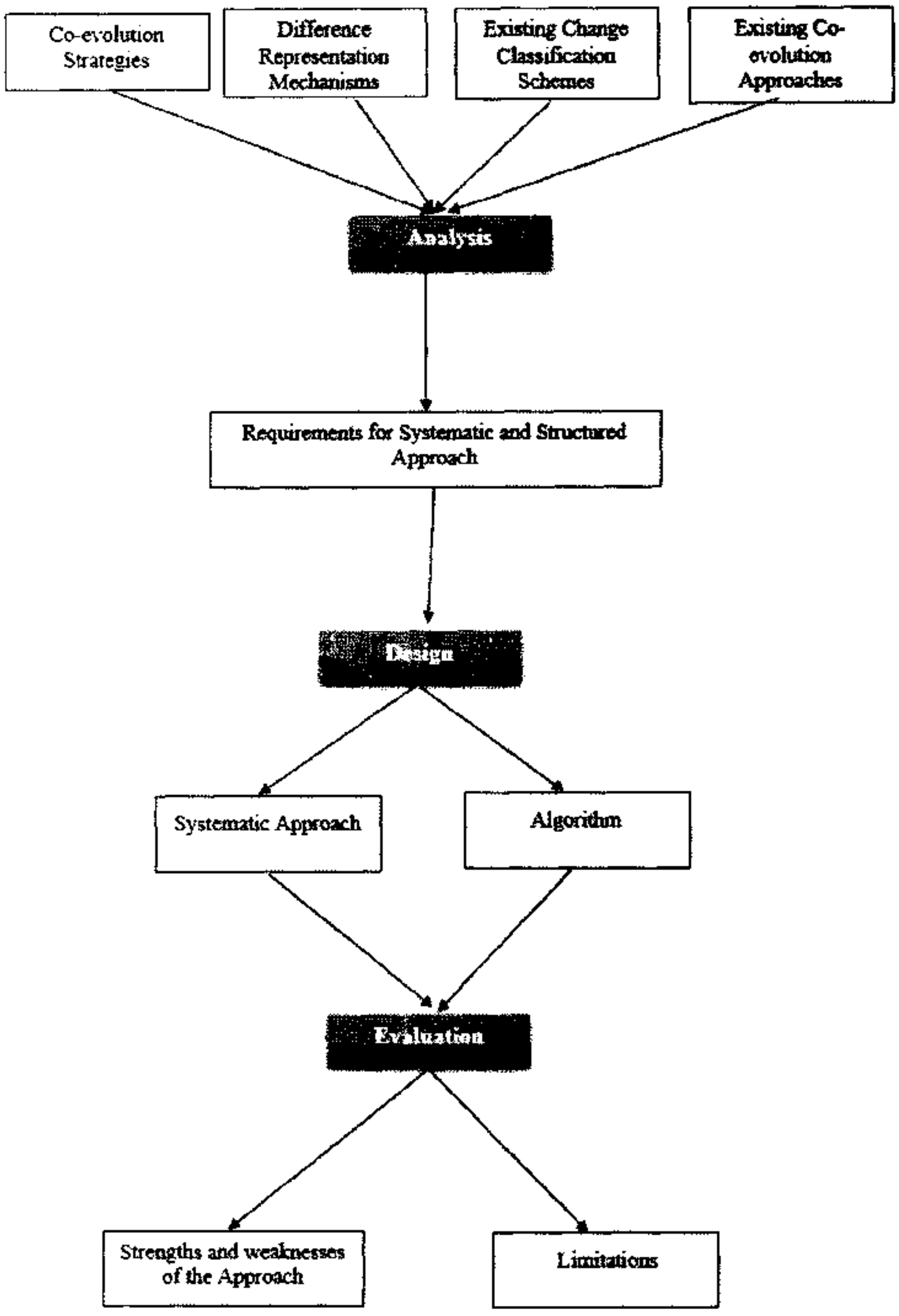

Co-evolution Strategies
Difference Representation Mechanisms
Existing Change Classification Schemes
Existing Co-evolution Approaches
Analysis
Requirements for Systematic and Structured Approach
Design
Systematic Approach
Algorithm
Evaluation
Strengths and weaknesses of the Approach
Limitations

Figure 1.2 Research Method

1.7 Proposed Solution

We proposed a systematic and automatable approach that is capable to co-evolve transformations after meta-model evolution automatically. The proposed approach is capable to detect, visualize and represent model element changes, classify these changes according to its impact on transformations, and establish a dependency relationship between model elements and transformations, classify transformations according to the required co-changes and finally co-evolving transformations. The proposed approach is automatable. This approach relies on the following principle steps:

- **Mapping Elements:** In this step, Model Elements are mapped on transformations to make their relationship explicit and capture it as trace model conforming to trace meta-model.
- **Change Detection and Classification:** In this step, changes that are introduced to the meta-model are captured and stored as change model conforming to change meta-model. After change detection, model element changes are classified according to the proposed classification scheme. This assists in identifying which type of changes are introduced to the meta-model.
- **Impact Analysis:** In this step, the impact of model element changes on transformations are analyzed by using the change model and traces.
- **Transformations Co-evolution:** During co-evolution step, transformations are first classified based on the proposed classification scheme and then the relationship based change propagation mechanism is employed to propagate changes to transformations.

In first step, "Consistency" relationship between model elements and transformations is made explicit and formalized via establishing trace links that would conform to "Trace" meta-model. To identify the type of changes introduced to the meta-model, meta-model changes are captured as change model conforming to the change meta-model by comparing the old and new version of meta-model. Once the changes are identified, these changes are then classified according to the proposed classification scheme. The impact of each change is then identified by mapping changes on trace links. The impacted transformations are then co-evolved accordingly. An algorithm is designed to support the proposed approach that takes the impacted transformations and asks for the desired changes along with the location of change.

1.8 Thesis Contribution

This research proposes a systematic and automatable approach for managing co-evolution of transformations in response to meta-model evolution. The approach covers the entire co-evolution process starting with the detection of changes in the new version of meta-model, making consistency relationship explicit via establishing traces, analyzing change impact, propagating changes accurately and systematically and co-evolving transformations. An algorithm is constructed to automate the co-evolution process. The algorithm supports all defined steps starting from difference identification to migrating transformations.

To detect the impact of meta-model change on transformations and to propagate the meta-model change accurately, we formalized a relationship-based mechanism. Following the MDE principles, the mechanism comprised a trace meta-model, which defines all the concepts necessary to formalize consistency relationship and make it explicit. Currently there is no mechanism to support traceability in the context of meta-model evolution. We provide traceability support for problem of co-evolution of meta-model and transformations in this research.

For change detection, representation and visualization, this dissertation proposed a change meta-model along with the change classification scheme. The change meta-model defines all mandatory concepts related to the change as well as the meta-model (since meta-model is subject to evolutionary pressures and the change is introduced mainly to the meta-model).

For determining which type of action is required to co-evolve transformation after meta-model evolution, we proposed a transformations classification scheme. Previously, there is no classification scheme proposed for transformations classification by the researchers. Here, this research proposed a transformation classification scheme to facilitate the identification of co-changes and the process of transformation co-evolution. The key benefits of our approach to transformation co-evolution are:

- It minimizes the effort and time required to transformation co-evolution and saves money.
- It guarantees the synthetic correctness of transformations after co-evolution.
- It provides a complete process which covers all steps of meta-model and transformations co-evolution from change detection to transformations co-evolution.
- The presented change classification scheme classify changes based on its impact on transformations.
- The successful employment of the concept of traceability for transformations co-evolution illustrated significant results.
- The concept of relationship based change propagation for meta-model and transformations co-evolution improved the significance and correctness of the proposed approach.
- The execution of Impact analysis activity assists in identifying the exact parts of transformations that are impacted by the model element change, therefore, ensuring correct propagation of change to transformations.
- Capturing changes as change model and employing change model to analyze the impact of change on transformations showed promising results.
- The proposed algorithm and prototype validated that the proposed approach is automatable.

1.9 Results of the research

This research proposed a systematic and automatable approach to meta-model evolution and transformations co-evolution. Our approach supports the entire process starting with the detection of changes between two meta-model versions, establishing a relationship between model elements and transformations, analyzing the impact of change and propagating the changes to the transformations.

The approach is generic enough to be applicable on transformations written in any dedicated transformation language. It is independent of the meta-model as well. The approach is applicable on any type of meta-model either EMF/Ecore based or MOF based. This makes our approach distinct and better from the existing approaches for transformations co-evolution. Our approach is automatable and implementable. We provide an algorithm, to support the complete co-evolution process.

The proposed approach is evaluated, by applying it, on three different examples that are obtained from the Atlas Transformations language ZOO (ATL ZOO). The examples comprised the source and target meta-models, transformations description and transformations rules specified in ATL. To validate the generality of the approach, transformations of the one example are re-written in Qvt and the other example are re-written in kermeta.

To evaluate the proposed approach, total nine well-defined and obvious changes reported by the industry as well as academia were introduced to the meta-models in the considered examples. During evaluation, it is observed that all the changes are captured and recorded accurately by change model. The change meta-model captured sufficient information about the change, to use later on during impact analysis. The classification of changes assisted not only in identifying which type of changes are introduced, but also in determining the category of impacted transformations. The evaluation also demonstrated that the change classification could not be generalized for all co-evolution scenarios, because the impact of model element change relies not only on the type of change but also on the model element usage by transformations. The change classification varies from scenario to scenario depending on the type of transformation language used and on the model element usage by transformations.

The trace model captured the relationship between model element and transformations. The evaluation showed that the extension of trace meta-model with the meta-model elements of underlying transformation language helps in identifying the exact part of transformation, which is impacted by the change, therefore, ensuring the correct change propagation. The trace models are used during impact analysis to identify the impacted part of transformations and during transformations co-evolution to propagate changes. However, the trace meta-model is not capable to capture the implicit traces between model elements and transformations, i.e., the implicit relationship between model element and transformations is not captured and requires more attention and investigation.

The execution of impact analysis for identifying the affected transformations demonstrated substantial benefits. First, it located the exact part of the transformation impacted by model element change. Second, it helped in determining the model element usage by the transformations. Third, it assisted in taking into account the intelligence employed by the transformations to transform a source model element into target model element.

Our approach is considerably different from the existing approaches in detecting as well as in propagating changes to transformations. To indicate the difference and demonstrate the uniqueness of the approach, the proposed approach is compared with the existing approaches. The comparison of our approach with existing approaches revealed that the exiting approaches do not perform the impact analysis activity, which is crucial part of the co-evolution process to determine the cost of change and to identify the model element usage by transformations. The comparison also demonstrated the difference in the ways; the two versions of meta-model are compared. Our approach employs a dedicated meta-model based mechanism to identify the changes introduced in the new version of meta-model. While the existing approaches, either used EMFCompare, to find the difference between two versions of the meta-model or assume that meta-model change history is already available. In addition, we proposed a dedicated classification scheme to classify model element changes, while the existing approaches used the classification scheme devised for model co-evolution process. Thus, the evaluation validated the proposed approach's generality, correctness and applicability on a range of transformation languages as well as meta-models.

1.10 Dissertation Outline

The remaining dissertation is ordered in a manner described in this section. Chapter. 2 establishes and founds the ground for this research by defining basic concepts and terminologies; describing the tools, standards and methods used in the context of model driven engineering; reviewing the principles, guidelines and practices the organizations, that practices MDE, are employing.

Chapter. 3 reviews the existing research related to meta-model evolution either theoretical or practical. The review then describes the research related to the particular category of meta-model evolution problem. The chapter then reviews the approaches related to the traceability, requirements management, model consistency management and model synchronization. The review then discusses change classification schemes, difference representation mechanisms and migration strategies presented in the literature and highlighted the potential issues and challenges related to it.

Chapter. 4 narrows down the focus of this research by formalizing the problem statement for this dissertation. Identifying the key issues that this research work targets, the remaining dissertation then focus only on the development of a systematic and automatable approach for transformations co-evolution problem.

Chapter. 5 describes a systematic and automatable approach for transformations co-evolution, including trace meta-model that is used to establish relationship between model elements and transformations as well as for relationship based change propagation, and change meta-model that captures the difference between the two version of meta-model. The change classification scheme proposed by this research classifies changes according to its impact on transformations. This research then proposed an algorithm to perform the proposed steps automatically.

Chapter. 6 discusses the case studies on which the approach is applied and evaluated. The UML2JAVA example (ATL transformation example), the PetriNet2PNML example (Qvt transformation example) and the Class2Relational example (Kermeta transformation example) are discussed in detail. ATL, kermeta and Qvt are three different languages to define transformations. The chapter then discussed the considered changes incorporated in the new versions of the underlying meta-models.

Chapter. 7 evaluates the approach proposed by applying on the three examples discussed in theChapter. 6. The results were discussed after the evaluation. The chapter then assesses the approach by comparing it with the existing approaches. The limitations and advantages of the approach, identified and observed during evaluation process, are also specified in this chapter.

Chapter. 8 summarizes the accomplishments of this research and discusses the results. This chapter ends up with concluding the entire research and discussing the future work.

Chapter. 2 Background

2.1 Introduction

This chapter presents background knowledge for the underlying research, introducing MDE related concepts and reviews MDE Literature. MDE is a standardized approach to software engineering in which models are first class entities and the software engineers specify, design, implement and deploy software systems through a series of models. Section 2.2 gives an overview of model driven software engineering. Section 2.3 introduces basic terminologies and principles related to MDE. Section 2.4 surveys principles standards, guidelines and methods used in Model Driven Engineering. Section 2.5 summarizes the entire chapter.

2.2 Model Driven Engineering (MDE): An Overview

Lifting up the level of abstractions is proved to be effective several times to improve the efficiency and descriptive power in software engineering process [10, 11]. In this manner, coding languages became more advanced from pure machine-readable commands to generally utilized object-oriented languages like Java and C++ that are general purpose in nature. MDE is the paradigm that pursues this drift and simplifies the historically full-fledged abstraction process. The fundamental objectives of MDE are to raise productivity, improve software quality and enhance the manageability of larger software projects [12, 13]. Abstract models present high level view of programming code. So far, models are used to present the developer's mental conceptualization, which allow describing the system functionality rather than focusing on its implementation details. This assists spontaneous and quick implementation of software systems, which are complicated in nature.

Models are considered as first class development citizens that are typically equipped with code generators. These code generators generate programming code, which is executable [14, 15]. Thus a model is a fundamental component of the specification of software system. Models are no more used only for the purpose to document the system [12]. Code generation makes the idea, of the employment of the random implementation technologies and diverse design patterns; which are specified once, realistic. For minimizing the risk to produced erroneous code, the available code generators can be reused [16]. In recent years, MDE has taken a front position in advancing a new paradigm shift in the field of software engineering. Figure 2.1 presents an overview of the MDE approach.

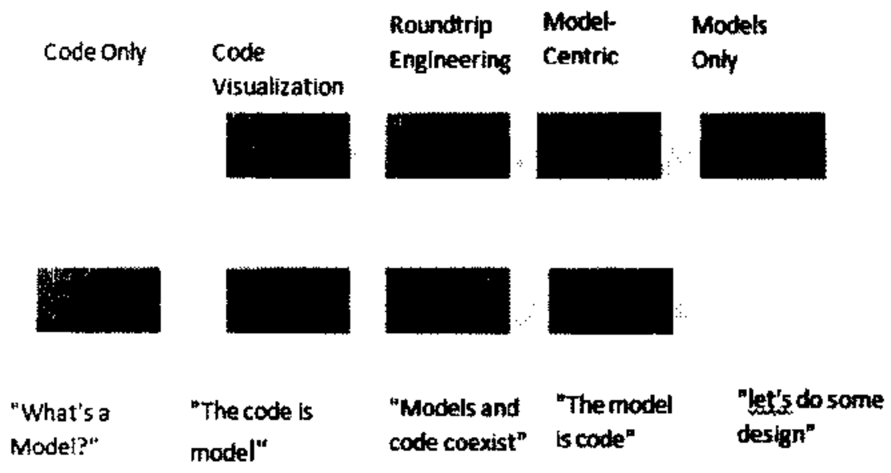


Figure 2.1 An overview of model driven engineering approach

2.3 MDE basic Terminologies and Principles

MDE presents development approaches and methods to software engineering with focus on minimizing the abstraction gap between problem area and solution area. The use of models geared the complexity of bridging the abstraction gap. Models describe complex system at multiple levels of abstraction and from different viewpoints.

Software engineers applying MDE techniques to develop software applications have to work with artifacts like models, meta-models and transformations in addition with traditional software engineering artifacts like code and documentation. In addition, MDE entails new development activities like model management. This section explains artifacts and activities associated with MDE.

2.3.1 Models

Dating back to the earliest programming days, modeling has a prosperous convention in the software engineering world. Current advancements included languages and tools, which let the modelers to articulate system viewpoint from various perspectives to software architect and developers in such manners, which can be plotted into code of some programming language that can be assembled for any specified o/s platform [16].

Models are considered as key artifacts from which components of software system can be automatically generated. "A Model is an abstraction of something for the purpose of understanding it before building it"[17]. A model allows the description of a family unit of problems for a particular domain having the abstraction level carefully selected in order to discard irrelevant and constant details and separates explicit important details. This results in reducing complexity making things simple and clear[12, 13]. A model may have either a textual or a graphical representation, depending on the type of language used. Some languages supports both textual and graphical representation, e.g., an ecore while other supports either graphical, e.g., UML or textual e.g. FORM.

Models can be used to describe a real world problem, e.g., Library providing a lending services or a computer based system, e.g., software and hardware specification for Library lending services. In other words, model can be used to illustrate both the domain and the machine.

Models are employed in a variety of modes, i.e., to estimate system characteristics, to reason about particular features when changes are introduced in a system and to communicate key attributes to the stakeholders. Models can be developed either to implement a physical system or derived from existing system to understand its behavior.

Models are at the heart of MDE. It increases the software development abstraction level from code to models with a huge prominence on keeping the software engineer focus on problem domain rather than on solution domain [12, 18]. OMG categorized models used in MDE to three type's i.e. [19]

- **Computational independent model (CIM)** that is called business model or domain model as well since, it employs the terminologies and expressions well-known to the concerned experts. It defines precisely the functionality of the system, i.e., what functions the system is expected to perform and hide all the technological specifications so that it can be independent of the details about how that system will be implemented. The CIM perform a crucial role in minimizing the space between domain experts and the information technologists who are accountable for software system implementation. There must be traces between CIM specification to the PIM and PSM constructs that employ them and vice versa.
- **Platform independent model (PIM)** reveals adequate independence level, which assists in its mapping to one or few platforms. The abstraction is usually done via specifying different sets of services in such a way, which omits technical particulars. Further models realize these services in a platform specific way.
- **Platform Specific Model (PSM)** merges the system requirements in the PIM with the particulars/information necessary to lay down how the system makes the use of a particular type of platform. However, if the PSM does not have sufficient details required to generate an implementation of that platform, it is deemed abstract illustrating its reliance on some supplementary explicit or implicit models that includes sufficient information.

2.3.2 Modeling Languages

In MDE, models are structured defined by using some modeling language. A modeling language specifies the structure and semantics of a set of models. Modeling language itself is specified by model termed as meta-model. A model conforms to meta-model only if it uses the concepts specified by the meta-model and satisfies the set of constraints defined by the meta-model [20]. A meta-model encompasses three types of constraints in general [21].

- **Concrete Syntax:** It describes the specific representation of the modeling language, covering encoding and/or visual appearance. It defines set of notations of language used to create models that conforms to that language's meta-model. A standardized concrete syntax facilitates communication. It provides a way to present a concept. The concrete syntax may be textual or graphical.
- **Abstract Syntax:** It expresses the structure of the language and the means the different primitives can be combined together, independently of any particular

representation or encoding. The concepts used by the language are specified by the abstract syntax, e.g., Data types, Attributes, classifiers etc. The meta-model illustrates abstract syntax of the underlying language.

- **Semantics:** It describes the meaning of elements defined in the language and the meaning of the various ways of combining them. It specifies the meaning of concepts for a specific domain.

The concrete syntax, abstract syntax and semantics defines any modeling language. This is a common practice used in MDE to define modeling languages although many other ways to define modeling language does exist. The industry and academia uses the UML as a common modeling language.

UML: UML is employed, as the primary language of modeling by the Practitioners [22]. It is an OMG's standard defines for modeling software or non-software system's structure, behavior, architecture and business processes and data structure. It describes concepts and their relationships. The meta-model defines the abstract syntax of the language while semantics is defined using natural language.

UML is use to model diverse application domain including banking, health care, aerospace, internet, etc. UML comprises best engineering practices proven booming in the modeling of huge and intricate software applications.

Based on a four-layer metamodel structure, the UML architecture comprises the user objects, model, metamodel, and meta-metamodel. The UML metamodel is not an implementation model. It is a logical model the benefit of which is that, it accentuates declarative semantics and restrains particulars of implementation. As implementations make the use of the logical meta-model, therefore, it must be conformant to the meta-model semantics, and ought to be capable to import and export complete and partial models. The drawback of a logical model is the shortness of the imperative semantics needed for precise and proficient implementation. Therefore, for the tool developers the meta-model is escorted with implementation notes.

In meta-model hierarchy, UML lies within the meta-model layer. It is moldered into various logical packages, i.e., Foundation package, Behavioral Elements package, and Model Management. They are than further decomposed into sub-packages. The UML meta-model is specified in a semi-formal way via rules for well-formedness, abstract syntax and Semantics. A model specified in a subset of UML (composed of a UML class diagram and a supporting natural language description) presents the abstract syntax. The formal language (Object Constraint language) and natural language (English) is used to define the well-formedness rules while, to articulate the semantics natural language is used, but some supplementary notation can be added, relying on the component of the model that is described.

The specification of UML is composed of infrastructure and superstructure specifications. Infrastructure defines the UML core metamodel. While the superstructure defines UML elements (diagram etc). The intricacy of UML is gripped by codifying it into logical packages. These packages combine meta-classes having strong cohesion among the meta-classes within the package and loose coupling with meta-classes in other packages. Figure 2.2 shows the top-level decomposition of meta-model into packages.

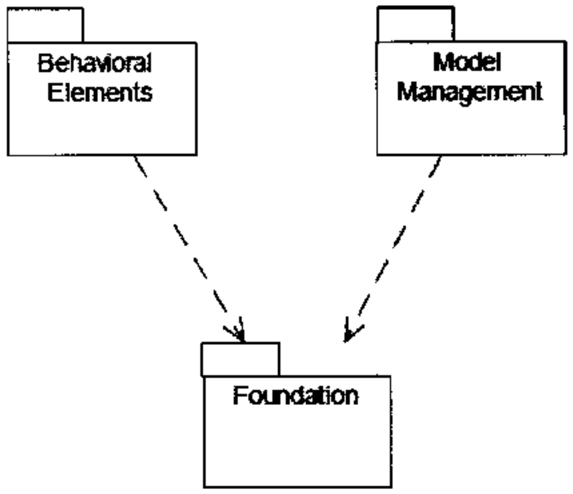


Figure 2.2 UML Package Structure [22]

Foundation package specifies the static structure of models. It is the language infrastructure. It is further decomposed into Core, Data Types and Extension Mechanisms sub packages. The Data Type package specifies basic data structures for the language. The package Extension Mechanisms states the way in which model elements can be customized and extended through new semantics. The Core package defines the fundamental concepts needed for a basic metamodel and defines architectural spine for adding further language constructs like Meta-class, Meta-attributes, etc. Figure 2.3 shows the sub packages of UML foundation package:

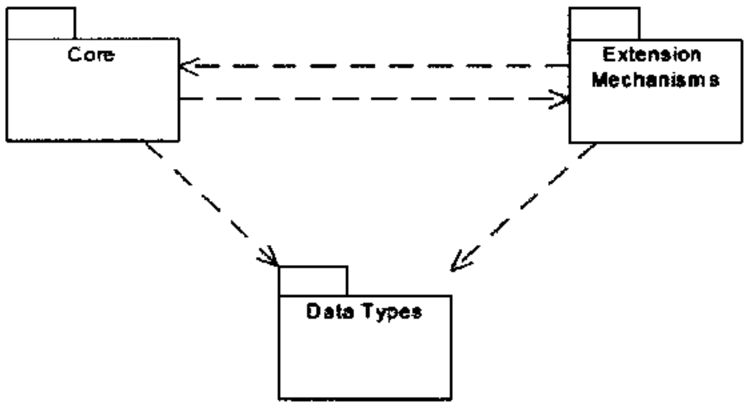


Figure 2.3 UML Foundation Package [22]

The behavioral elements package is divided into sub packages i.e. common behavior, use cases, collaborations, state machines and activity graphs. The key concepts for model elements are described by the package Common Behavior while a behavioral context for using model elements to achieve a certain task is described by the package Collaborations. The package use case uses actors and use cases to describe the behavior elements. The stat machine package describes behavior via state machines while the activity graphs package is used to model processes. Figure 2.4 illustrates sub packages of UML Behavioral Package:

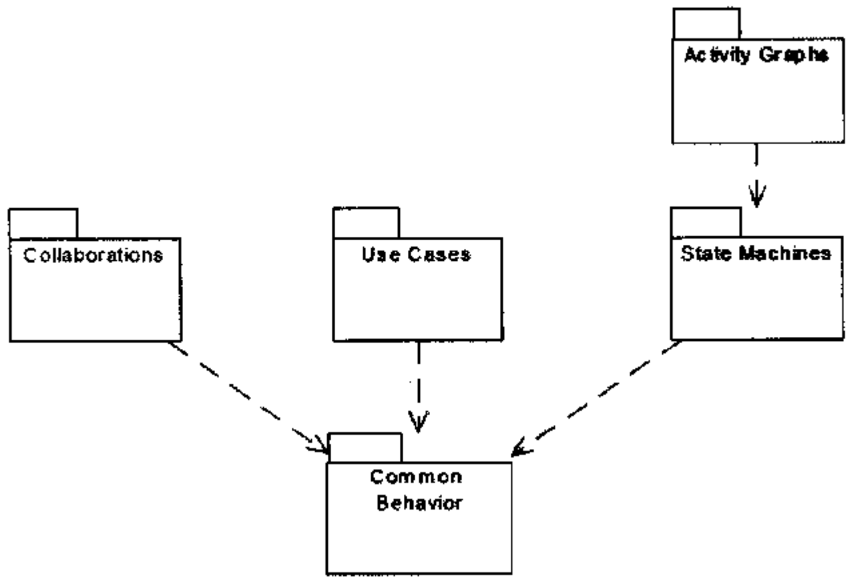


Figure 2.4 UML Behavioral Element Package [22]

The UML uses its extension mechanisms to permit variations to be expressed. The UML offers the subsequent facilities, i.e.

- Semantics and notation to tackle a spacious range of existing modeling issues in a straight and economical way.
- Semantics to deal with certain probable future modeling problems specially linked to distributed computing, component technology, excitability and frameworks.
- Extensibility mechanisms to broaden the meta-model concepts for users own use in an economical way.
- Extensibility mechanisms to develop future modeling approaches on top of the UML.
- Semantics to simplify model interchange between a various tools.
- Semantics for the stipulation of the interface to repositories for the sharing and storage of model artifacts.

A UML can be used to define each platform-independent model and platform-specific model, as one's require. The MDE uses both model types. In Model Driven Engineering, every standard or application is based on PIM which embody its functionality and behavior without of technical details. MDE tools than produce one or several platform specific models from this PIM, for one or more chosen target platforms, also in UML. This generation step is automatic. The PSM composes of similar knowledge in the form of a UML model not in executable code as an implementation. Subsequently, executable code from the PSM is generated by the code generator, with other essential documents consisting interface definition files if required, configuration files etc. The tool runs the make-files to generate deployable software, after making necessary changes in the produced executable code.

2.3.3 Meta-models

In Greek, word "Meta" means "After". Meta related themes have fascinated people through the centuries. In computer science, the term is used heavily and with several different meanings, e.g., in database, metadata means "data about data" and refers to data dictionaries, repositories, etc. In programming languages, meta-interpreter is an interpreter of a program (interpreter) [23]. In conceptual modeling, metamodel is a model of a data model, e.g., an ER model of the relational model, or an ER model of the ER model. Another dimension of metamodeling is "The world is modeled by a story; the story is modeled by a metastory".

Meta-models can be deemed as comprising part of MDE. Meta-models are formal definitions of well-structured models [24] or meta-models comprise the languages, which can portray a given reality in some abstract sense. It is a concise means of specifying the set of models for a specific domain. It defines ontology of concepts for a specific domain and vocabulary and grammatical rules of a modeling language. Modelers use meta-models mainly for domain specific language construction, model validation i.e. checking models against meta-models, for describing model-to-model transformations and model-to code transformations, and for tool integration.

2.3.4 Meta-modeling Languages

Meta modeling is the activity to develop meta-models that describe modeling language, and offer modeling tool to support this modeling language. It is one of the most significant aspects of MDE; the purpose is to construct a modeling language that describes models correctly.

To define meta-model, different meta-modeling languages exists. MDE gives three ways to choose meta-modeling language: use a standardized language, use lightweight language extensions, and use heavy weight language extensions.

MOF (A Meta-modeling Language): An organization that employs a Model driven engineering approach, uses existing meta-models or defined their own meta-models. MOF is an OMG's standard language for defining meta-models. MOF itself is defined as model so it often called as meta-meta-model. OMG standardizes MOF as a language for specifying meta-models to assist interoperability between tools. Existing tools are interoperable as MOF standardizes a means for presenting models with an additional OMG standard, XMI (i.e. XML meta-data interchange), a language for loading, storing and exchanging models [25].

The MOF and meta-model hierarchy exercised in a combination. It is designed to be at the top level of OMG's meta-model hierarchy. OMG defined a four-layer meta-model hierarchy, i.e.

- Meta-metamodel layer (M3)
- Metamodel layer (M2)
- Model layer (M1)
- Runtime layer (M0)

M3 layer specifies the language used to define meta-models. This layer comprises of one meta-model, i.e., MOF. MOF based on itself. M2 is modelling language specification layer, which is more specific with respect to M3 but still abstract. These languages used to define models. This layer can have multiple meta-models based on which models for the same domain can be defined. The M1 layer is a model layer (user specification) layer where the user can define the model confirming to model at M2 layer. This layer contains actual definition of data. The M0 layer is the instance layer that contains instances of models initialized at runtime. Figure 2.5 illustrates where MOF lies in OMG's meta-model hierarchy.

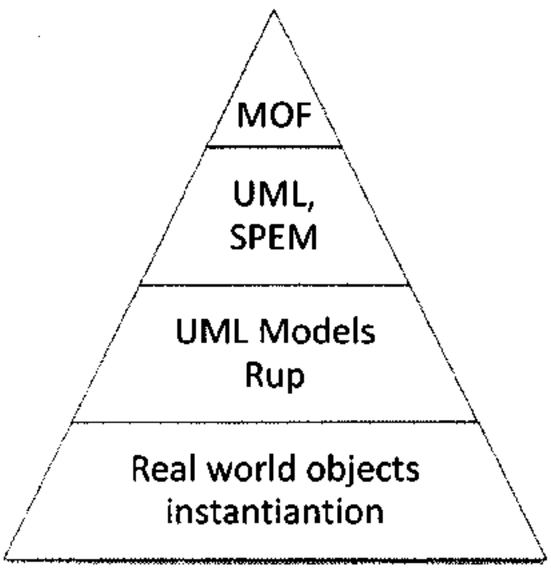


Figure 2.5 MOF positioning in OMG's meta-model Hierarchy [25]

MOF has a modular architecture. MOF uses a concept of packages to group constructs. The concept of nested packages allows creating a hierarchy out of packages. The outer most packages are MOF and UML infrastructure library. Twofold dependency exists between MOF and UML infrastructure library. 1) A reuse relationship, and 2) UML is an instance of MOF.

The UML infrastructure library defines common meta-language elements, so that to define other meta-models designers can use it. The purpose of reusing the same infrastructure amongst meta-models is to acquire architecturally aligned meta-models. MOF is a self-describing meta-model. It based on its own constructs. The interfaces and behavior of MOF meta-model can be defined by applying MOF IDL mappings to MOF itself.

2.3.5 Model Transformations

One of the main concepts, in model driven software engineering is model transformations. It provides mechanisms to automate model manipulation. Model transformations can be defined as "a program that mutates one model into another". OMG defines model transformations as "the process of converting model into another model of the same system". Model transformations can also be defined as the automation of the transition from business models to implementation models.

Kleppe et al. defines model transformations as "the automatic generation of target model from source model, according to transformation description"[12]. Mens et al. added the concept of several input or several output models to this definition and defined model transformations as "the automatic generation of one or more target models from one or more source models, according to a transformation description" [26]. Figure 2.6 shows the components of model transformations.

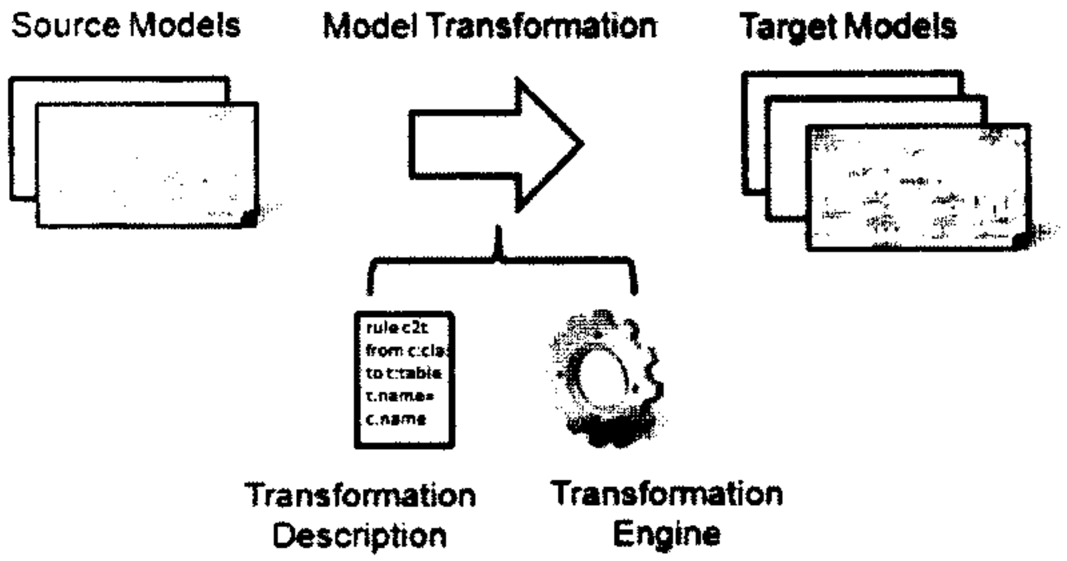


Figure 2.6 Model Transformation Components

The major challenge to model driven engineering is in the transformation of platform independent models into platform specific models that can be further utilize to generate models of the implementation level. Based on some transformation rules, the Transformation of one model to the other model implies the source model is converted into target model. Transformation rules specified relationships between two or more models. Thus, these transformation rules are the means to automate the conversion between models and from models to text [27].

Different methods can be used for specifying model transformations rules. A transformation Rule comprises of two parts: left hand side and right hand side. The left hand side selects the source model while right hand side of the rule defines parts of the target meta-model generated by the transformations from source model elements.

MDA community identified four types of transformations, i.e., PIM-to-PIM transformations, PIM-to-PSM transformations, PSM-to-PSM transformations and PSM-to-PIM transformations [19]. The more generic classification is:

- **Model-to-Model Transformations:** Model-to-Model transformations are used to generate one or more target models from a given source model (s).
- **Model-to-Text Transformations:** Model-to-Text Transformations are used to generate textual artifacts from models.

In MDE, model transformations can be used to perform various tasks, i.e., making, altering, weaving, merging or filtering models. The common feature of the above mention tasks is the reuse of captured information in models. Instead of generating new models from the scratch model transformations reuse the information captured once as a model and generate artifacts based on it.

Transformations can be used for synthesis as refinements that append information to the model. The Source and target models of transformations might have same or different meta-models. In both cases, the detail level is raised gradually and it probably includes shifting from higher abstraction level to lower abstraction level. Probably, a model-to-model transformation chain exists and the chain of model-to-code transformations does exist too. This type of code generation process, where source code is generated from models of higher abstraction level, is a particular type of synthesis transformations. The Synthesis transformations might improve a model in different manners, i.e., higher level idea's decomposition, algorithm's choice, abstraction specialization for certain usage context. Figure 2.7 illustrates the Meta-Levels of Model Transformations:

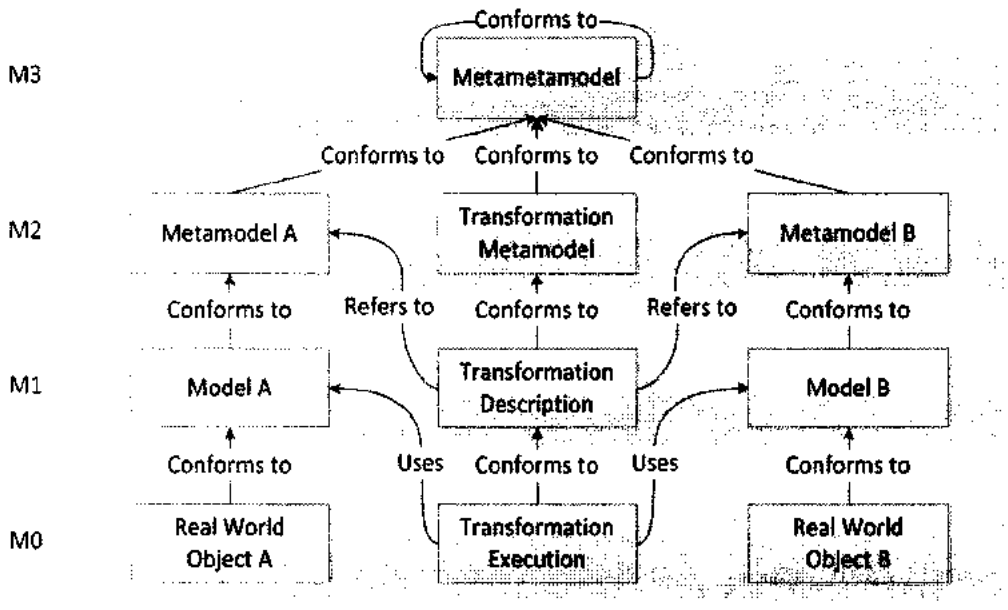


Figure 2.7 Meta-Levels of Model Transformation [26]

2.3.6 Model Transformation Approaches

Model transformation languages are built based on model transformation approaches. Model Transformation approach is a collection of design principles. Many model transformation approaches have been proposed by researcher's community, e.g., imperative, declarative, functional, relational and operational approaches.

2.3.7 Model Transformation Languages

A model transformation language comprises a well-defined vocabulary, grammar and semantics for carrying out model transformations. Language is founded on some model transformation approach.

2.3.8 Model Transformation Description

A Model Transformation Description articulates how a source model is transformed to a target model. Model Transformation description is specified using some model transformation language. If the transformation description is a collection of transformation rules then it illustrates the language is rule-based. Other names used to refer a Model Transformation description are, Model Transformation Definition, Model Transformation Code or Model Transformation Program.

2.3.9 Model Transformation Rule

Model Transformation Rule is the smallest part of Model Transformation Description. It specifies the conversion from source model scrap to the target model scrap. A rule has a source pattern and a target pattern. For each element of the source model, a target element is created in the target model. In Graph Transformations, the source pattern is also referred as Left Hand Side (LHS) and the target pattern as Right Hand Side (RHS).

2.3.10 Model Transformation Engine

A Model Transformation Engine runs a Model Transformation Description. It takes the source model as input and generates target model by applying Model Transformation Description. They are also referred as Re-write Engines. ATL, MOdTransF, ModMorf and smartQVT are the examples of Model Transformation Engines. Typical step that a Model Transformation Engine Perform while executing Model Transformation:

- Source model elements identification for transformation purpose.
- Generating related target elements for each identified source model element.
- Generate trace information, which relate source element and target element influenced by executed rule.

Source Model: A source model is a model that is passed as input to the model transformations. This model would conform to the source meta-model. The input to Model Transformation can be one or more source models.

Target Model: A target model is a model generated by Model Transformations as output. The target model would conform to the target meta-model. The target models of Model Transformations can be one or more. The term target model is used in the context of Model-to-Model transformations. In Model-to-Text transformations, the source code is generated instead of target model.

Transformations model: The description of transformation can be symbolized as model which could make it ultimately input and or output to/of some other transformations. The transformation model conforms to the transformation language meta-model in which transformations are specified. Like other models, Transformation Model can be instantiated, altered by Higher Order Transformations.

2.3.11 Higher Order Transformation (HOT)

A model transformation Description that accepts one or more transformation models as input and/or generate one or more Transformation models as output are referred to as Higher Order Transformations (HOT). Figure 2.8 gives a generic overview of how HOT works.

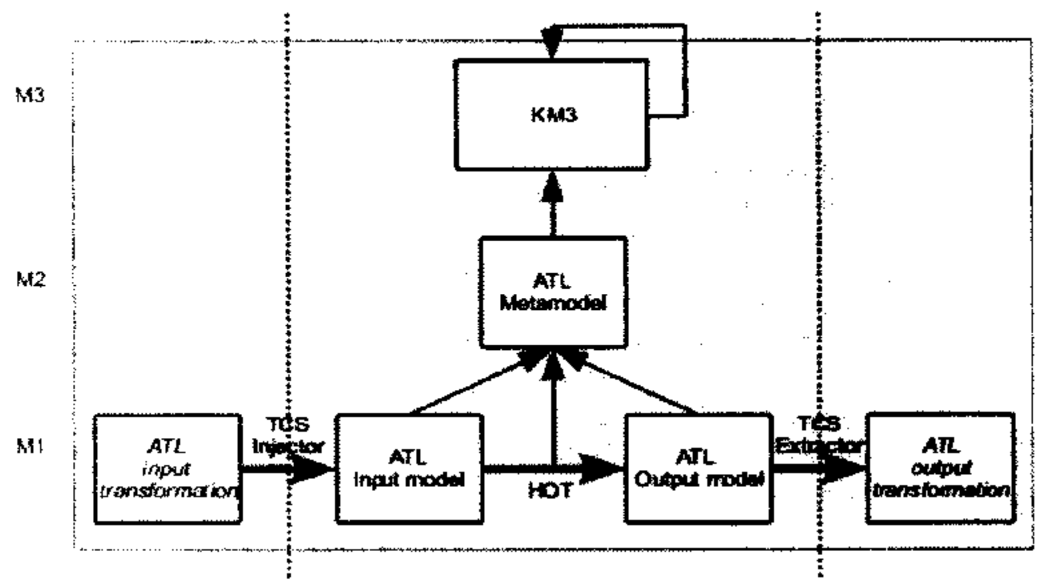


Figure 2.8 Generic view of Higher Order Transformations [12]

Unfortunately, the definition of Higher Order Transformations is not as easy as seems in the figure above. The fact that transformation language meta-model is relatively huge and transformation models seems similar to abstract syntax trees, with various containment levels made the development of HOT difficult and error prone.

2.3.12 ATL: Atlas Transformations language

Atlas transformation language is a model transformation language defined both as meta-model and textual concrete syntax [28, 29]. It is a mixture of declarative and imperative constructs, thus simple as well as complex mappings can be handled through ATL. Simple mappings are handled using declarative constructs while complex mappings are handled via imperative constructs. The recommended way of programming is declarative.

The ATL transformation program is typically composed of rules. Rules specify how source model elements are mapped and navigated to create and initialize target model elements. ATL supports QVT query, view and transformations. Even so, there are some trivial variations. ATL program carry out queries and transformations. As views are models and ATL program take model as input or create model as output, views are particular type of the input or else output of ATL programs.

Two kinds of ATL program exists, modules and queries. Modules are used to transform model into model while queries are used to transform model into text. ATL supports libraries, which allow defining repetitive sections of code in one place. ATL is founded on Object constraint language (OCL). It has type model analogous to OCL. OCL is used to expression constraints on rules.

The ATL program is composed of four sections i.e. header section, import section, rules and helpers.

- The header section defines module name and declares the input model and output model. The module name and ATL file name must be identical.
- The import section specifies the definitions used by the program from other modules or libraries.

- Transformation rules describe the mapping of a source model element on a target model element by relating meta-models. Each rule has a unique name.
- A rule is defined by a key word "rule" followed by rule's name. There are three types of rules. They are termed as lazy rules, rules and called rules.
- The part of rule called "source pattern" is use to state which element type of source model is transformed.
- The target pattern is use to state which target model element/elements the source pattern is transformed. Example of a transformation rule is given below:

```
ruleClassToClass
{
from inputClass : ClassModel!Class
to javaClass : JavaModel!JavaClass
(
name <-inputClass.getJavaClassName(),
 description <-'Java class ' + inputClass.name,
variables <-inputClass.attributes,
methods <-javaClassConstructor
),

javaClassConstructor : JavaModel!Method
(
name <-inputClass.name,
body <-' // Default constructor'
),
secondClass : JavaModel!JavaClass
(
name <-inputClass.name + '_Another',description <-'Another class ...' + inputClass.name
)
}
```

Helpers can be defined as a variable or function. Helper functions are defined as OCL expressions. The helper functions can call each other and can be called by rule as well. Helpers are use to define repetitive pieces of code at one place.

- A Helper is defined by a key word "helper"
- A name is introduced with key word def
- It finishes with a semi colon
- A context variable is defined with the help of ATL path expression metamodel!element and is accessible via a self variable. If the context is not defined, the module itself is taken as context. Example of helper definition is given below:

```
helper context Class!Class def: checkAttributeCount():Boolean=
 self.attributes->size()>2;
helper context Class!Class def:isFAttribute():Boolean=
self.name.substring(1,1).toLower()='f'
```

2.3.13 Traceability Management

Traceability is a crucial system attribute as it facilitates system evolution, validation and software management. Traceability is an essential part of the change impact analysis too. Traceability management assists in understanding, capturing, tracing and verifying software artifacts and their links and dependency relationship with other artifacts during the entire development life cycle [30]. Apart from its recognized significance, traceability management is ignored mainly due to the inherent complexity of maintaining links between different software artifacts.

In MDE, the key role of models considerably changes this landscape. Particularly, the central role of models can positively influence traceability management because the traces to preserve might be simply the links between different model elements managed during the entire development process. Besides, traces between models can be stored in other models to process them by any model processing technique e.g. model transformations, model merging, matching models etc.

2.4 MDE Principles, Standards, guidelines and Methods

Applying Model driven engineering to software development, different engineering practices and processes have been proposed. OMG's MDA is one the base line that set out guidelines for MDE. This section discusses the principles, standards, guidelines and methods.

2.4.1 Model Driven Architecture (MDA)

The OMG's Model Driven Architecture is intended at improving quality, enhancing productivity and reuse by separating the concerns and increasing the level of abstraction. One or more platform specific models are generated from a platform independent model, which is abstract model comprising sufficient information for PSM's generation. Potential Platform specific models are particular to the target platform like source code, etc., [1].

The Model Driven Architecture intends to augment portability through separating system architecture and platform architecture. The system architecture is defined through a model called platform independent model which describe the structure and function of a system. The platform architecture is defined through platform specific model, having implementation information of a specific platform. For mapping platform independent model on platform specific models, MDA has the capability to define transformations. This helps in the system engineering and makes the realization of a system implementation easy, across various platforms. Figure 2.9 shows PIM transformation to PSM:

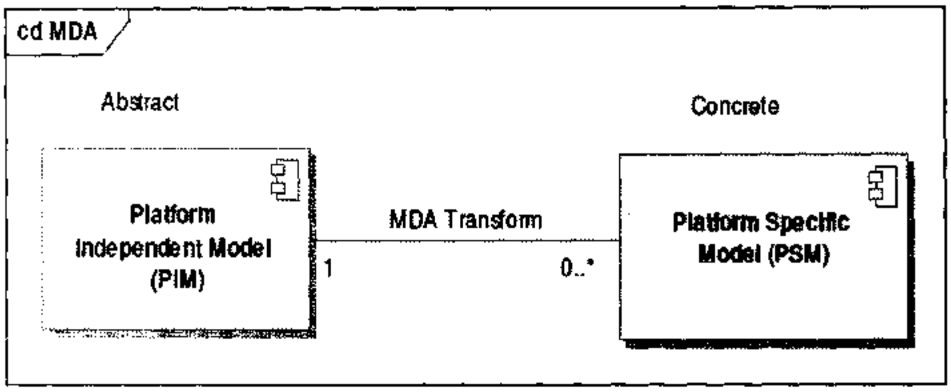

cd MDA
Abstract
Concrete
Platform Independent Model (PIM)
MDA Transform
1
0..*
Platform Specific Model (PSM)

Figure 2.9 PIM Transformations to PSM [19]

For example, a PIM is transformed to PSM, i.e., DDL. This transformation results in the creation of table elements in Data Definition Language format from a class whereas EJB entity is also produced from this class. This would result in a package that comprised the essential element required by EJB, i.e., class and interface. The tool that assists in managing such type of transformations is Enterprise Architect. It allows writing transformation rules for any language. It let the user to keep as many PSM synchronized to a single PIM. EA has a built-in support for Model transformation to C#, DDL, EJB, JAVA and XSD [18]. Figure 2.10 demonstrates the transformation of PIM to multiple PSMs:

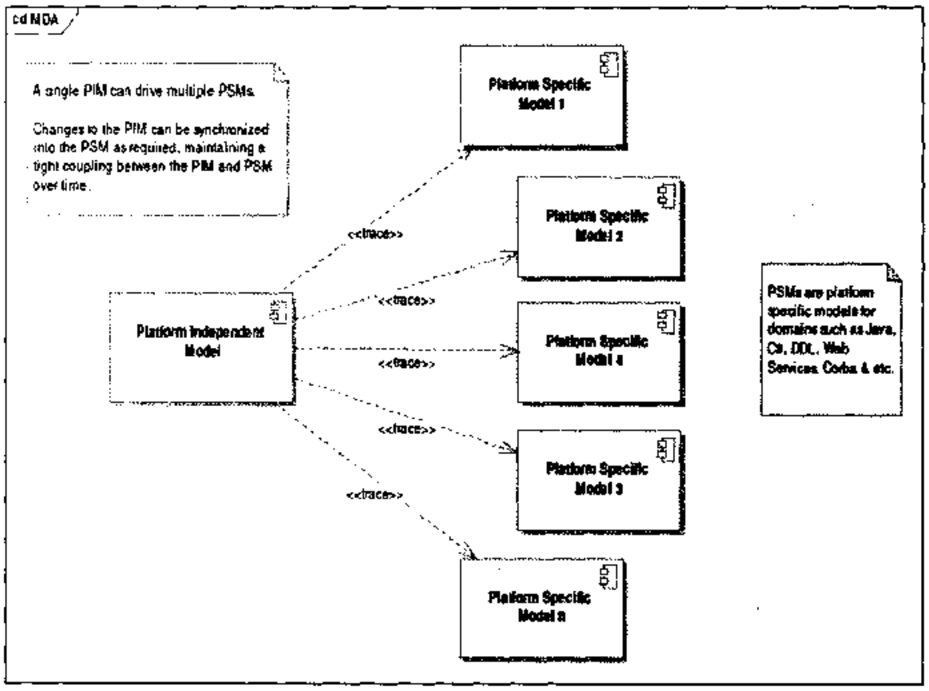

cd MDA
A single PIM can drive multiple PSMs.
Changes to the PIM can be synchronized into the PSM as required, maintaining a tight coupling between the PIM and PSM over time.
Platform Specific Model 1
<<trace>>
Platform Specific Model 2
<<trace>>
Platform Independent Model
<<trace>>
Platform Specific Model 4
<<trace>>
Platform Specific Model 3
<<trace>>
Platform Specific Model n
PSMs are platform specific models for domains such as Java, C#, DDL, Web Services, Corba & etc.

Figure 2.10 PIM transformation to Multiple PSM's [19]

Model Driven Architecture offers a conceptual framework to use models and to apply transformations to the models seeing that an ingredient of managed and well-organized software development process. Today, the main theories and aspects that govern MDA practices are:

- Models assist in understanding and communicating complex ideas.

- Depending on the context, various types of elements can be modeled. These models bring in different perspectives of the world that ought to eventually merged.
- Commonality at all levels of models in both problem analysis and proposed solution does exist.
- A well-defined development process is offered by OMG by employing the ideas of diverse types of models and its transformation between different representations. This enables the identification and reuse of general approaches.

2.4.2 MDA Principles

Model Driven Engineering is based on four principles [12]. Following are the base principles MDA relies on:

- Models articulated in a well-formed language are a basis to realizing applications for enterprise scale solutions.
- The development of applications can be structured through a set of models by enforcing chain of conversions involving models, structured into an architectural framework constituting transformations and layers.
- Meta-models, a formalized basis to express models, assist meaningful incorporation and conversion between models, and are the origin to automation through tools.
- For the broader acceptance and adoption of the proposed approach, which is model-based, industry standards need to be developed to give openness to customers, and support contest between sellers.

The Object Management Group has defined an explicit set of layers and transformations to support these principles, which offer a conceptual framework and vocabulary for MDA. Particularly, four categories of models are identified by OMG: CIM, i.e., Computation Independent Model, PIM, i.e., Platform Independent Model, PSM, i.e., Platform Specific Model and an Implementation Specific Model. Figure 2.11 gives a generic view of MDA key concept:

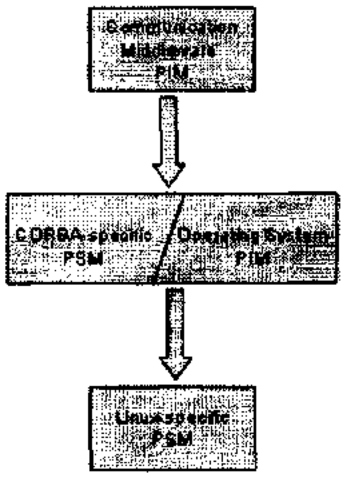


Figure 2.11 A generic view of MDA concept [12]

This means, an MDE tool may support transformation of models in a number of steps, i.e. from analysis model to running code. For example, tool like pattern facility carries multi-transformation development whereas IBM Rational Rose produce running code from UML models at once.

2.4.3 Model Driven Architecture Standards

The OMG lay down standards for MDA, as parts of the guidelines for Model-Driven Engineering. The standards are allocated to one of the four layer of OMG's Four Layer Meta-modeling Architecture.

2.4.3.1 Meta-Model Hierarchy:

The four-layer architecture defines an infrastructure for specifying models, meta-models, modeling languages and activities and provides foundation for future meta-modeling language extensions. This architecture supplies framework for meta-models exchange among different meta-modeling environments. This framework is crucial for tool interoperability, because such interoperability relies on a precise specification of the structure of the language.

Each layer in Four-Layer Meta-model Architecture represents level of model abstraction members of one layer conforms to the member of the above layer. Figure 2.12 illustrates each layer of meta-model Hierarchy:

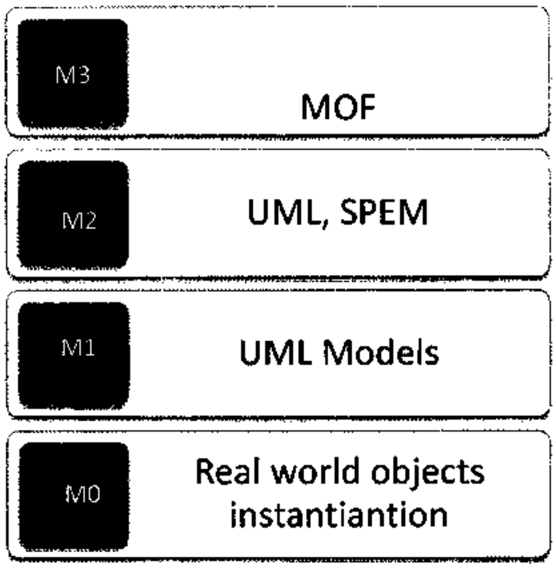


Figure 2.12 OMG's Meta-model Hierarchy

M3 Layer: At layer M3 the meta-metamodel lies that defines languages for describing meta-models. MOF is an example of meta-metamodeling language. The MOF meta-model is used to instantiate meta-models that becomes a part of layer M2. The UML meta-model conforms to MOF.

M2 Layer: The M2 contains the modeling language meta-model for example UML, meta-model is at layer M2. UML meta-model is conformant to MOF meta-model. The models at layer M1 confirms to the meta-model at layer M2. For example, class diagram defined for some business application would conform to UML meta-model.

M1 Layer: The M1 defines models for the concepts in M0. The Objects at M0 conforms to the models defines at layer M1. For example, the concept customer might be defined as a class with attributes at M1. In other words, the Layer M1 contains the models that is defined using some modeling language, e.g., UML. This model would be defined using a modeling language that lies at layer M2.

M0 Layer: The base layer of this pyramid, i.e., M0 represents the domain, i.e., a real world model instance, i.e., user instance. It is the information layer that comprises the actual objects/instances, thus data. For example, if some business entities are modeled this layer is used to represents items of the business like customer or invoice, etc. If modeling a software application the M0 layer describes the software representation of such items. Below is the Table 2-1 that illustrates each level and its description:

Layer	Description
Meta-metamodel	This is the basic infrastructure for metamodeling architecture. It specifies the language for metamodel definition.
Meta-Model	This is the instance of meta-metamodel which defines a language for model specification.
Model	This is an instance of meta-model which defines [illegible]
User Object	This is an instance of model which comprises the actual data.

Table 2-1 Meta-level Hierarchy description

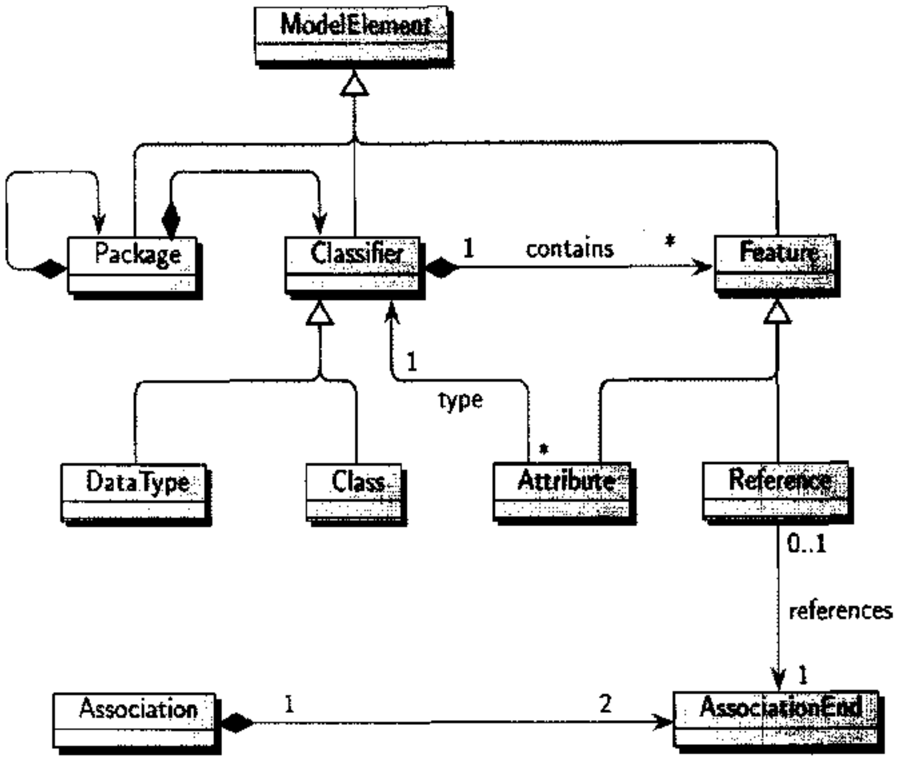


Figure 2.13 A simplified view of MOF [25]

2.4.4 Methods for Model Driven Engineering (MDE)

Various methods for MDE exist today. In this section, the most advanced method of MDE will be discussed, i.e., Domain specific modeling, Model-Centric software engineering and service oriented software engineering. Model driven engineering practitioners defined these methods. The industry (practicing MDE) used these methods repeatedly for solving problems. The methods differ in the degree to which they follow the guidelines commenced by the MDA.

2.4.4.1 Model-Centric Software Development

The Model-Centric Software Engineering is a software Engineering approach that uses concise and expressive models throughout the development process. It utilizes the models to articulate correlated notions of every domain hence they grow to be translucent and can be employed in other related domains. Although for software development, model-centric approaches have been used and employed for several years. However, Model-Centric software engineering is the particular area of model driven engineering that deals with the creation of running code from model of implementation level, which have enthused special attention over several years. Model-Centric Software Development nonetheless has a wide scope and fields like business process modeling, architectural models or enterprise wide federated repositories.

In general, it might proposes an enormous opportunity to empower individual intellectual property in software engineering and in particular, to realize domain driven development pledge of technology or business alignment when utilized appropriately. However, it can lead a project to the failure if the significant level of additional intricacy it brings in the technical and organizational level, is ignored.

2.4.4.2 Domain Specific Modeling

A particular type of model-based software engineering approach to design and develop a system. It entails systematic utilization of domain specific languages to symbolize different system's aspects under development. It also includes the concept of code generation that is automating the generation of executable source code directly from the DSL models. It presents a set of principles, practices and guidelines for developing systems by employing MDE. . DSM is based on the direct transformation of domain models into code. The motivation behind the idea of DSM is the comparison of the huge gains in productivity made when third generation programming languages were employed instead of assembler.

Domain Specific modeling can notably improve developer's efficiency because it is free from manual creation and maintenance of source code. The automatic generation of source code as compare to manual coding will lower the number of errors in resulting program therefore enhancing quality of the developed system.

The major components of DSM are Domain Specific Modeling Language, Code Generator and Framework Code. These are designed to firmly fulfill the requirements of a single organization. Below is the brief description of the above mentioned components:

- **Domain Specific Modeling Language:** The system's requirements, structure and behavior in the considered domain is formalized by its type system. DSML are specified using meta-models that define the concepts and relationships between these concepts specific to the domain. It precisely specifies the semantics and constraints related to these domain specific concepts. Developers utilize domain specific languages to develop application systems by employing elements of the type system captured by meta-models and articulate design goal declaratively rather than imperatively.
- **Code Generators:** customized code generators are employed to generate source code from models directly. Code can be generated in any programming language and for any development paradigm. It does not require any later manual changes to code. This automated generation of code is possible just because of domain specificity. During the generation process, existing platforms, libraries, components, legacy code can be used efficiently.
- **Framework Code:** It puts in a nutshell the common areas of all applications lying within that specific domain.

Domain specific modeling puts forward the huge advancement in software development productivity since the major shift from assembler to 3GLs. Industrial experiences shows consistently the efficiency of DSM than the current practices, including UML-based implementation of MDA. Nevertheless, without the right tool support building a domain specific modeling solution might be a huge task, probably preventing an organization from adapting DSM.

2.4.5 MDE Tools

Today, after many years the field of MDE is mature enough that powerful tools and language support for many MDE activities are available. This section provides brief description of MDE tools that are well suited and mature and would be used in this research. Eclipse Modeling Framework, Kermeta, ATL and Rational Software Architect are discussed in detail in the subsequent section.

Eclipse

Eclipse offers an open platform for application development tools. It runs on a variety of operating system. The Eclipse is intended to use for building Integrated Development Environments (IDE's) and arbitrary tools and applications. Applications and tools built on the top of Eclipse can easily be integrated with applications and tools built by means of Eclipse Platform. The java development components like JDT are added to the Eclipse platform. And it is turned in Java Integrated Development Environment. Likewise, C/C++ development components like the CDT are added to the Eclipse and it is turned into C/C++ development environment too. Eclipse integrates individual applications and tools in one product offering a prosperous and reliable experience for its users. One of its prominent and distinguishing features is its extensibility. Its Graphical User interface is based on Standard Widget Toolkit (SWT). Eclipse project is divided into three sub projects, i.e.

- The Eclipse Platform
- The Java Development Toolkit (JDT)
- The Plug –in Development Environment

Architecture

The Eclipse platform comprises of workbench, workspace, help and team components, in addition to the small platform runtime kernel. The tools based on Eclipse plug-in to this foundation framework to build a handy application. Figure 2.14 gives a view of Eclipse Architecture:

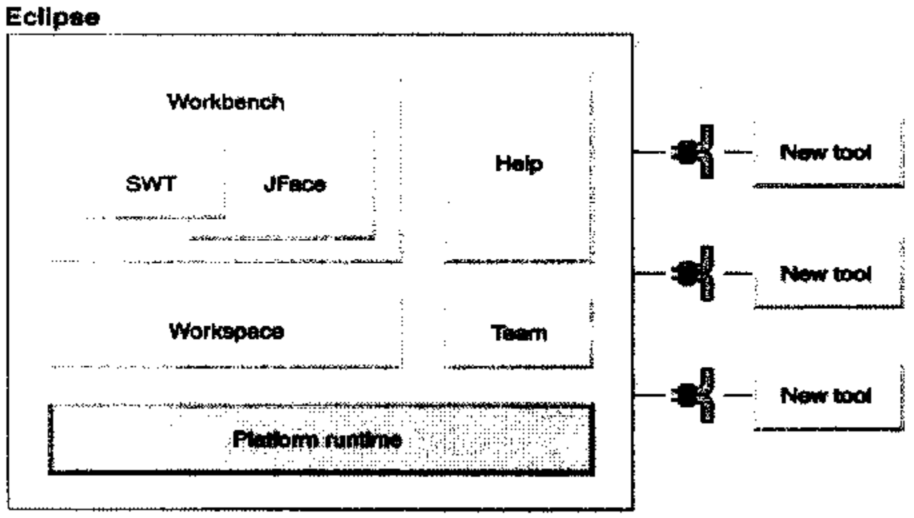


Figure 2.14 Eclipse Architecture [I]

Eclipse Modeling Framework

EMF is a java framework. It provides code generation facility for constructing applications and tools established on a structured model. It offers the facility to build data model from various sources for example UML, annotated java, XML/XSD etc. It serves as a basic building block for constructing modeling tools. It provides the facility to build both visual and programmatic manipulation tools and editors.

Eclipse Modeling Framework was initially founded on Object management Group's MOF 1.0 specification. Later on, it has influenced MOF 2.0 specifications. The EMF 2.0 claims support

Essential MOF (EMOF). EMF constitutes of an Ecore Model, Core Model, GenModel, JET and JMerge. The Ecore is a meta-model used to create a model. It is based on itself so it is a meta meta-model as well. An EMF basically consists of:

- ***EMF:*** It is the core framework which is composed of a model, i.e., Ecore which describes models as well as it offers runtime support for models. The runtime support may include change notifications and persistence support (which can be with default XMI serialization). To manipulate EMF objects generically, it provides an API.
- ***EMF.Edit:*** It consists of reusable classes that are generic in nature for constructing editors designed for manipulating EMF models. It includes a
  - Command framework, which comprises a set of command implementation classes (generic in nature) for constructing editors, which can completely supports automatic undo and redo.
  - Property source support, classes of content and label provider, and classes for other convenience that enable models of EMF to be presented by property sheets and standard desktop viewers.
- ***EMF.Codegen:*** It generates all the required things to develop a full editor for editing EMF models. It consists of a GUI. It is use to specify generation option and to invoke generators. The Java Development Tooling (JDT) component of Eclipse is leveraged by the EMF.Codegen generation feature.

The prominent facilities EMF offers are the generation of an efficient, correct and easily customizable implementation code from model definition; converts user's models to Ecore; tooling support in Eclipse Framework, reflective and dynamic invocation and supports XML/XMI serialization and de-serialization of instances of model, etc. Figure 2.15 shows an Ecore meta-model. A simple Ecore model is composed of:

- ***EClass:*** It represents a modeled class. It has a name and is composed of zero or more than one reference and attributes.
- ***EReference:*** It characterizes an end of an association that exists between classes. This reference has a name, a containment property, and type.
- ***EAttribute:*** It characterizes a modeled attribute, which must have some name and is of some type.
- ***EDataType:*** It defines the type of an attribute. It can be a primitive type e.g. an integer or Boolean or object type, e.g., java.util.date.

EMF lets the users to specify their own meta-models in Ecore, the meta-modeling language and the MOF implementation of EMF. It provides two types of editors to create a meta-model i.e. graphical and tree-based editor. Figure 2.16 illustrates a tree-based view and Figure 2.17 a graphical view for petri-net meta-model.

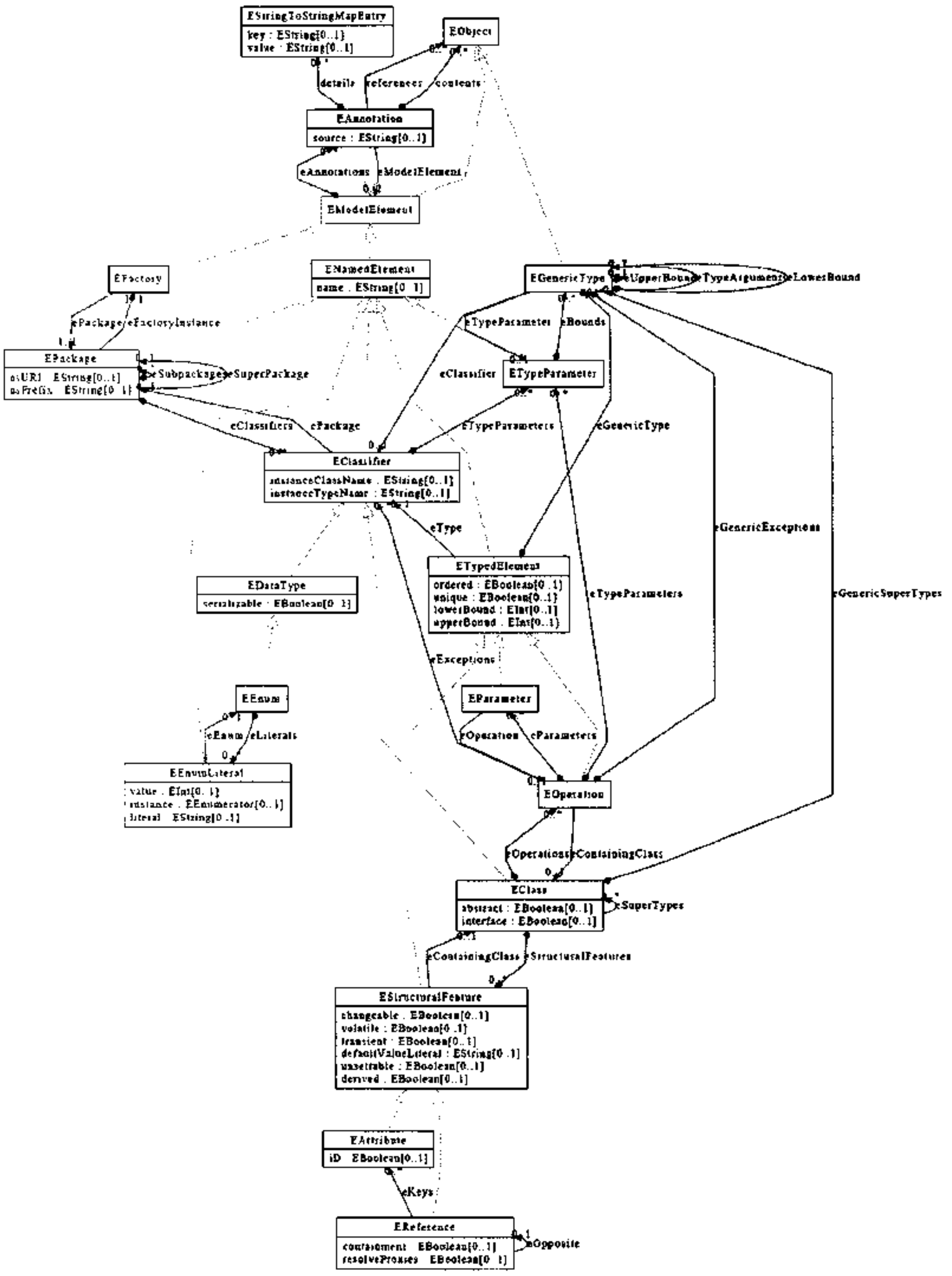


Figure 2.15 An Ecore Meta-model

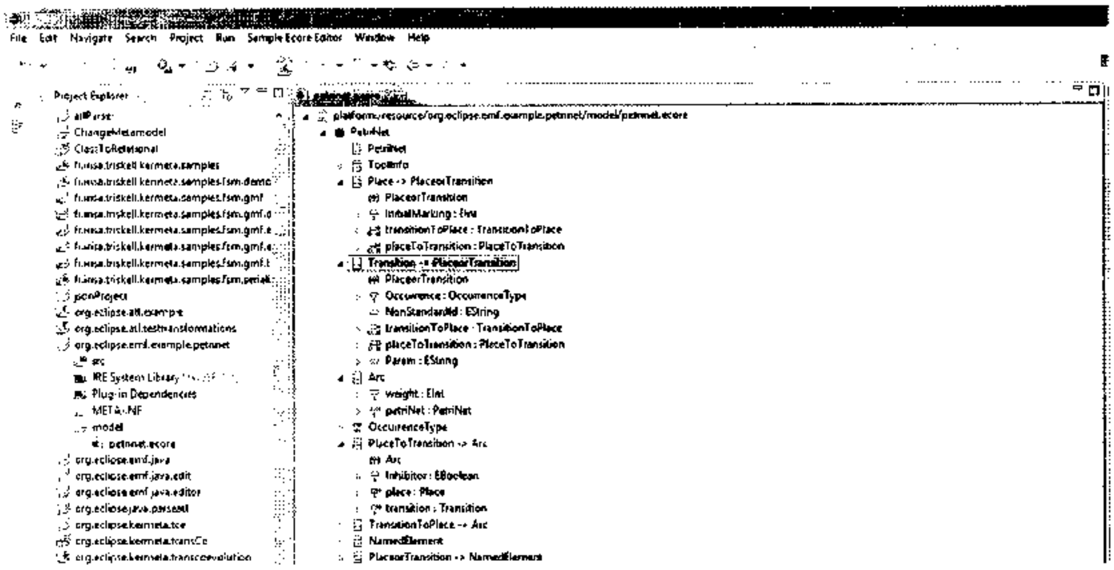

File Edit Navigate Search Project Run Sample Ecore Editor Window Help
Project Explorer
platform:/resource/org.eclipse.emf.example.petrinet/model/petrinet.ecore
PetriNet
Place -> PlaceorTransition
Transition -> PlaceorTransition
Arc
weight : EInt
OccurenceType
PlaceToTransition -> Arc
TransitionToPlace -> Arc
NamedElement
PlaceorTransition -> NamedElement

Figure 2.16 A Tree-Based Editor for Meta-model

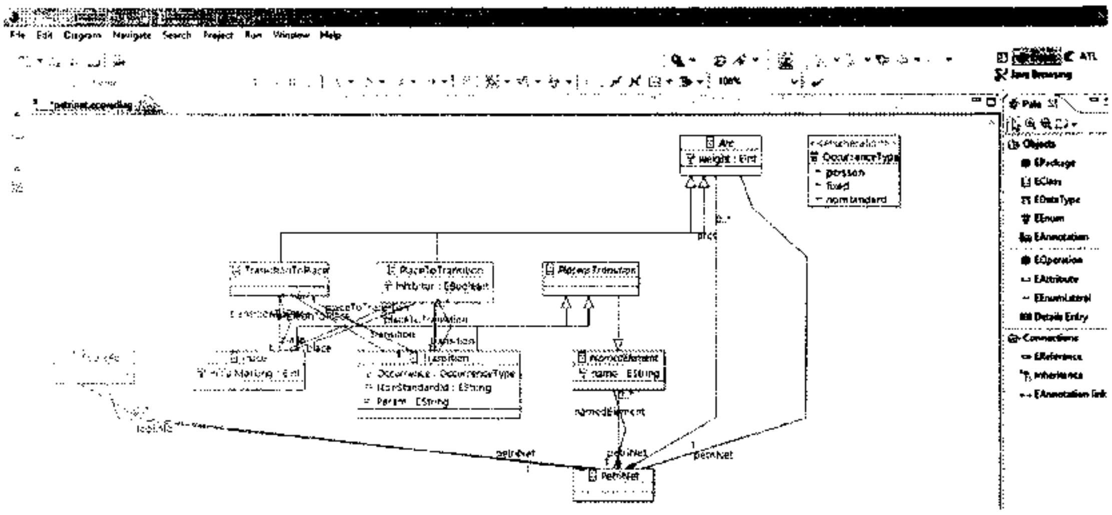


Figure 2.17 A Graphical Editor for Meta-model

The editors illustrated in Figure 2.16 and Figure 2.17 are used to manipulate the same meta-models but via using different syntaxes. A change made to the meta-model using one editor can be automatically propagated to the other editors. Furthermore, EMF can generate editors for models that conform to the underlying meta-model from which the editor is generated. For example, a simple Petri Net meta-model specified in figure was used to generate the code for the model editor shown in figure. EMF incorporates the mechanisms for loading and saving models in the model editors it generates. EMF normally generates codes that stores models via XMI, an XML dialect optimized for model interchange.

Kermeta

Kermeta is a domain specific meta-modeling language, which is designed to define both the structure and behavior of meta-models. It is compliant with EMOF and Ecoe. EMOF is a part of the MOF 2.0 specification. For defining behavior of models, it offers an action language. Kermeta is designed to be exercised as the core language of a model-oriented platform. It is

intended to be a common basis to implemented constraints languages meta-data Languages, action languages and transformation language.
The main purpose of Kermeta is to provide an action language for MOF models. The key concept is to initiate from MOF, which provides the structure of a language, and to add an action model which describes the static semantic and dynamic semantic of model. It includes most of the classical object-oriented mechanisms, i.e., operation redefinition, operation specialization and operation overloading. MOF specifies the structures and the operational semantics consequent to MOF concepts has to be specified using Kermeta.

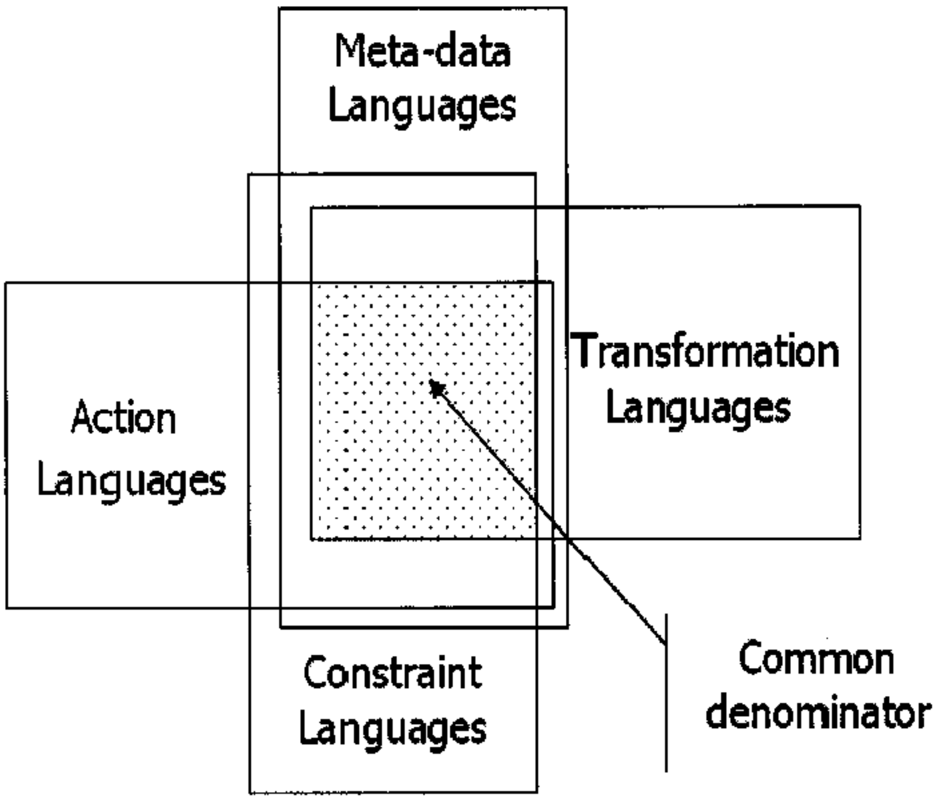


Figure 2.18 Kermeta Overview

Architecture

Kermeta is intended to be fully compliant with the EMOF (an OMG's standard meta-data language). Kermeta is composed of two key packages, i.e., structure package which corresponds to essential MOF and behavior package which corresponds to the semantics/actions.

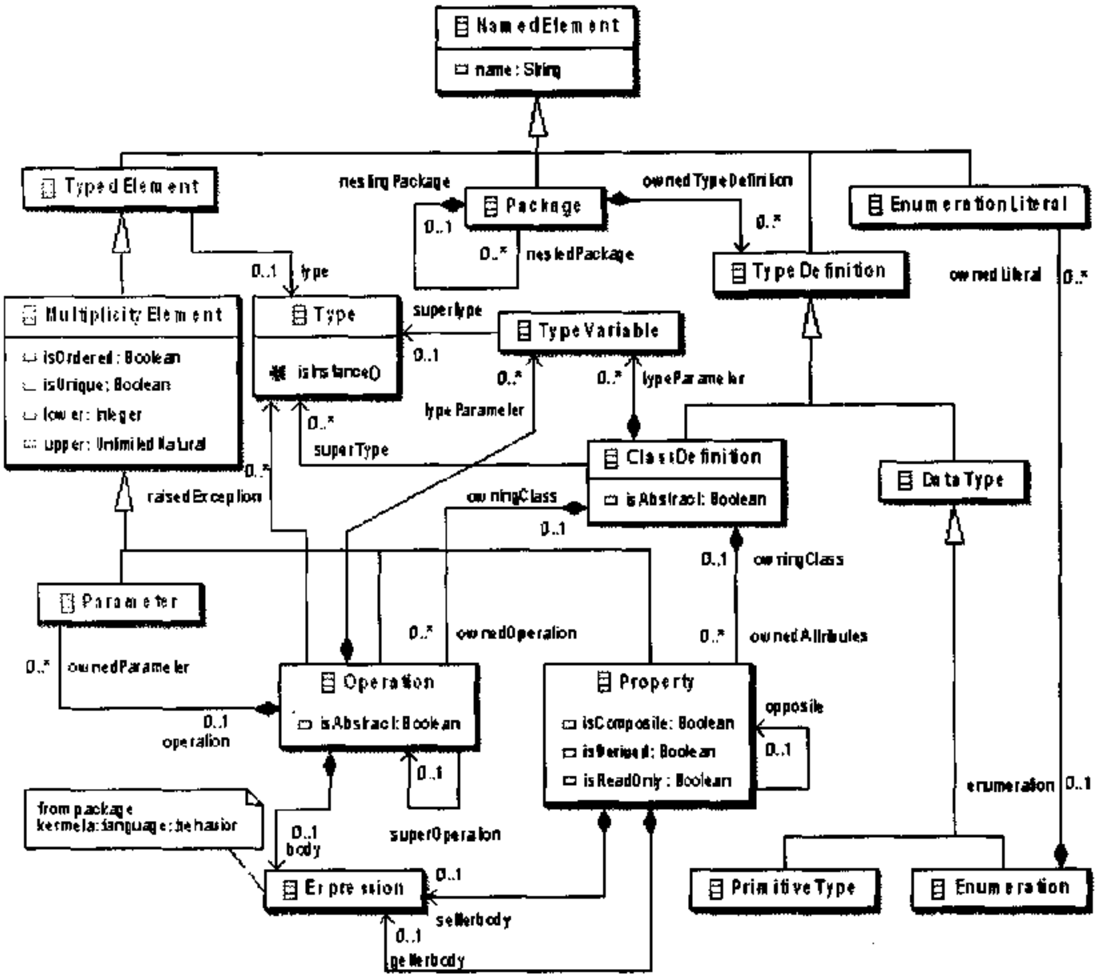


Figure 2.19 Kermeta's Structure Package

Figure 2.19 describes the structure package of Kermeta. The main classes included in structure package are presented here which defines the static structure of kermeta. The base concepts are taken from EMOF and several modifications were made since EMOF was not initially designed to be executable. For this purpose, an action language is added Kermeta's Behavior package. Figure 2.20 illustrates Behavior package.

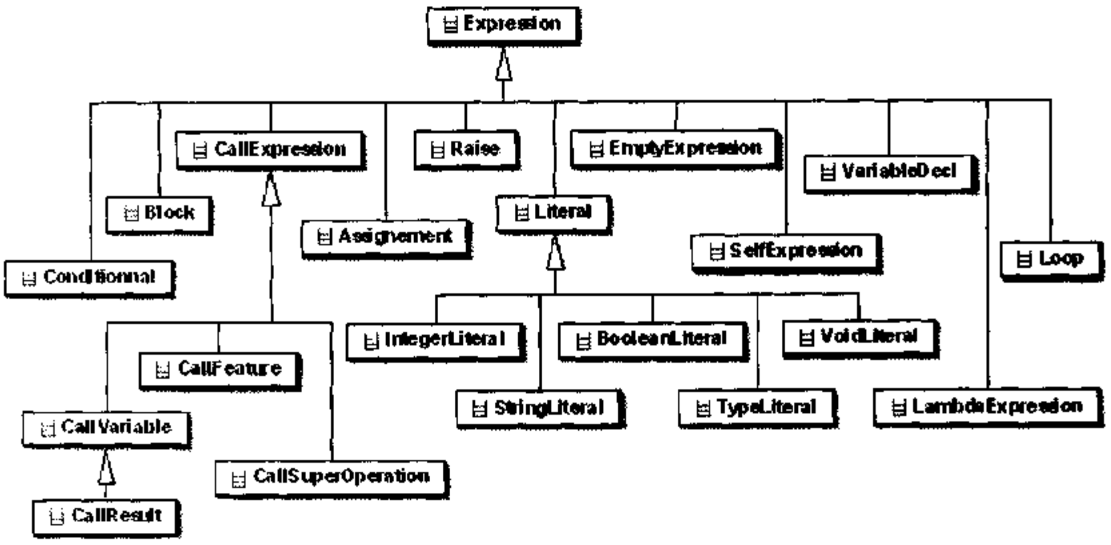


Figure 2.20 Kermeta's Behavior Package

Kermeta provide two ways to define a meta-model, i.e., using Kermeta textual syntax or creating an Ecore file via tools built by EMF. Kermeta textual syntax allows creating a .kmt file and building a meta-model programmatically. Kermeta provides the facility to convert the Ecore file in kmt format and vice versa. Below is the textual representation of sample class diagram meta-model.

```
@uri "simpleuml.com"
package SimpleUML;

require "kermeta"
require "http://www.eclipse.org/emf/2002/Ecore"
abstract class UMLModelElement
{
        attribute kind : PrimitiveTypes::String[1..1]

        attribute name : ecore::EString[1..1]

}
class Attribute inherits UMLModelElement
{
        reference owner : Class[1..1]#~attribute

        reference type : Classifier[1..1]#typeOpposite

}
class Package inherits UMLModelElement
{
        attribute elements : set PackageElement[0..*]#namespace

}
abstract class PackageElement inherits UMLModelElement
```

```
{
        reference namespace : Package[1..1]#elements

}
abstract class Classifier inherits PackageElement
{
        reference typeOpposite : set Attribute[0..*]#type

}
class Class inherits Classifier
{
        attribute ~attribute : set Attribute[0..*]#owner

        reference general : set Class[0..*]#generalOpposite

        reference generalOpposite : set Class[0..*]#general

        reference reverse : set Association[0..*]#source

        reference forward : set Association[0..*]#destination

}
class PrimitiveDataType inherits Classifier
{
}
class Association inherits PackageElement
{
        reference source : Class[1..1]#reverse

        reference destination : Class[1..1]#forward

}
@uri "platform:/resource/org.eclipse.atl.testtransformations/metamodel/SimpleUml.ecore"
package PrimitiveTypes;

require "kermeta"
require "http://www.eclipse.org/emf/2002/Ecore"
alias String : kermeta::language::structure::Object;
```

2.5 Chapter Summary

This chapter discussed the idea of model driven engineering. The basic terminologies, guidelines, methods, activities and tools utilized in Model Driven Engineering process are introduced. Two prominent methods that utilized MDE were reviewed. System evolution in the context of model driven engineering was discussed and the key issues and challenges related to evolution were highlighted in this chapter.

Traditional software engineering approaches do not consider modeling artifacts like models and meta-models as its key artifacts and they are presented in an unstructured way, if at all.

MDE comprised generating, utilizing and managing changes to modeling artifacts thus these modeling artifacts are presented in a structured way. This chapter showed that MDE tools for example EMF give structures and processes for generating and manipulating modeling artifacts but does not provide any support for managing evolutionary changes. Chapter 3 explores and investigates ways to manage evolution of modeling artifacts.

Chapter. 3 Related Work

3.1 Introduction

In model driven engineering, large number of models and transformations on different level of abstraction needs to be taken into consideration. A huge number of models engaged, hail from a layered modeling architecture, i.e., meta-models as well as from refinements, i.e., transformations from generic to implementation-centric model representations [1, 15]. Meta-models lie at the heart of Model Driven engineering as it defines structure and semantics for a set of models and transformations are specified upon it. Meta-models are subject to evolutionary pressures and hence artifacts based on it might become invalid with respect to the new version of meta-model [21].

This chapter reviews existing approaches proposed to address the co-evolution of meta-model and its related artifacts. Particularly, this chapter investigates the mechanisms used to identify changes introduced to the meta-model and manage the co-evolution process.Section 3.2 reviews co-evolution approaches. Section 3.3 reviews traceability approaches in model driven engineering. Section 3.4 explores classification scheme employed to classify meta-model changes. Section 3.5 investigates difference representation mechanisms utilized to represent difference between two versions of meta-models. Section 3.6 discusses migration strategies proposed and employed to solve the co-evolution of meta-model\model and meta-model\transformations. Section 3.7 discusses issues and challenges related to the co-evolution approaches. Section 3.8 sums up the entire chapter.

3.2 Co-evolution approaches

Meta-models are subject to change and evolve over time. With the evolution of meta-model, the artifacts, which are specified based on it, need to be co-evolved. The researchers have proposed a number of approaches to address the co-evolution of meta-model and its related artifacts. Next sections describe the research related to the co-evolution of meta-model and its related artifacts.

3.2.1 Meta-model\model co-evolution approaches

Since Meta-models evolve over time, models that conform to it possibly will not conform to the new version of meta-model anymore. The researchers proposed various approaches to model migration during the previous several years. Cicchetti et al. proposed a mechanism for meta-model evolution management [24]. They presented changes in model based manner. They employed a difference meta-model and the change representation mechanism together to record the changes introduced to the meta-model. They utilized difference model, conforming to the difference meta-model, to correctly classify meta-model modifications and to accordingly implement required countermeasures to co-evolve the existing models. Higher order transformations (HOT) are defined for the co-evolution of models, which takes the difference model as input and produce the intended model migration. However, the proposed approach only deals with breaking and resolvable changes. Human assistance is required to resolve breaking and unresolvable changes. Furthermore, empirical evaluation of the approach is required.

Antonio Cicchetti et al. designed a mechanism to manage the evolution of meta-model in a model driven engineering environment [31]. They used a model based technique to represent meta-model changes. They used difference models to classify meta-model operations accurately and to implement subsequent countermeasures to co-evolve existing instances accordingly. They then classified a meta-model evolution based on their impact on models and managed each type separately.

Moritz Eyshildt presented an approach for managing changes between different versions of EMF Ecore based meta-models and consequent evolving model instances [32]. They introduced EPatch and Metapatch formats, which are capable of specifying the amendments and routines supporting the migrations. Both formats compose textual domain specific language, each one specified by its own XText grammar. They offer a handy integrated development environment, for example code completion. EPatch is a declarative, self-reliant, meta-model agnostic. As EPatch is a meta-model agnostic it does not need the to-be-patched models to be instances of a definite meta-model. The difference between models is obtained by comparison. A comparison is made on the top of EMFCompare. EPatch has the ability to describe changes occurring due to the refactoring, construction or destruction of meta-model elements. The Patcher Tool EPatch and create a copy of the model thus preventing a source model from modification and creates a mapping between source and target model. The MetaPatch utilizes the EPatch in two distinct manners. Initially, the MteaPatch format extends the EPtach format through inheriting grammar. Additionally, it adds instructions in java or Xtend to customize the model migration algorithm and bounds the EPatch to Meta-models, i.e., EMF ecore models. Finally, the The MetaPatcherMigrator takes the mappings of the meta-model elements as input and migrates the models from one meta-model to another. The EPatch format became a part of EMF Comparer and Metapatch format will be a part of Eclipse Edapt project. They briefly described the overall process of migration. However, the resolution of complex and arbitrary changes have been ignored.

Graces et al. proposed an approach for the adaption of model after meta-model evolution [5]. This approach automatically generates adaptation transformation for migrating models to make them consistent with the evolved meta-model. Both versions of meta-model are translated into the KM3 notation. A matching strategy is defined that identifies equivalences and differences between both versions of the meta-model. A matching strategy discovers the similarity and differences by implementing a set of heuristics. Equivalence and difference are represented as matching model. The matching model conforms to the matching meta-model. The matching model is described graphically as well as textually. The main concept of matching meta-model is Equal that specifies similarity between two versions of meta-model. It has a similarity value between 0 and 1 that signifies the apparent validity of the correspondence. Higher Order transformations are defined that takes matching model as input and migrate instances of the old version of meta-model to the new version of meta-model.

Boris and Grusko Dimitrios S. Kolovos proposed a five step solution to the problem of synchronizing models with evolving meta-models that is 1) Change Detection 2) Classification 3) User Input Gathering 4) Algorithm Determination and 5) Migration [33]. In change Detection step, changes are detected either by direct comparison between versions of meta-model or by tracing and recording the changes introduced to the old version of meta-model. These differences are stored in delta model. In the next step, changes are classified under the categories mentioned as non-breaking, breaking and resolvable and breaking and non-resolvable. User input is taken for the changes that are not automatically resolvable. An appropriate migration algorithm for model migration is determined and finally models are migrated to re-establish conformance relationship of models with the new version of meta-model.

Demuth et al. proposed a generic approach that identifies the co-evolution failures and suggests model migrations to co-evolve models correctly [34]. Particularly, the approach is used to find out whether the adapted models are conformant to the evolved meta-model. The approach is composed of two phases, i.e., co-evolution failure detection and derive options for the corrections of failures. In first phase, locations where co-evolution is not performed are detected while in phase two different options for the correct meta-model change propagation to the effected models are derived. For constraints management, they used an incremental constraint management approach to update constraints after meta-model evolution in order to make it certain that the models are valid with respect to the new constraints imposed by the meta-model. Model consistency is checked via an appropriate consistency checking mechanism. To find appropriate adaptation, the approach utilizes a reasoning mechanism, which takes into consideration all design constraints present in model. However, the approach only resolves simple and atomic changes. Furthermore, the prototype implemented requires empirical evaluation.

The work in [35] specified a catalogue of operators to co-evolve meta-model and model. The defined coupled operators are able to evolve meta-models and to automatically co-evolve the corresponding meta-model instances, i.e., models in response to meta-model evolution. The catalogue covers operators from the literature and the case study they performed. The catalogue is founded on a broadly used EMOF (Extended Meta-Object Facility), a meta-modeling formalism that was refined that is the model elements that are not capable to be instantiated in the models are removed. Every time, when a meta-modeling formalism is extended by adding new model elements, then the catalogue must be must be updated by adding new operators, i.e., to add, delete or change a construct primitive operators are added while to perform more obscure evolutions involving constructs complex operators are added. This catalogue provides foundation for operator based tools and difference based tools. However, this approach has a major drawback of selecting an operator when multiple operators exist for one specific scenario.

Ludovico Iovino et al. presented an approach to better comprehend the impact significance of the changes drive on a domain metamodel [4]. This is deemed handy for a number of reasons, particularly, to understand formal documentation to be passed on to implementers responsible of grasping the desirable adaptation; and to predict the diffusion of its impact through the many artifacts and bear out the efforts required to restore the validity of the compromised components. They recognized the usual relations between a domain meta-model and its associated modeling artifacts, and symbolized them with a mega-model. Mega-models have been demonstrated to be helpful and useful in specifying those infrastructures where numerous artifacts are involved and linked together through some mechanisms, like model transformation and/or model weaving. Consequently, a general architecture is presented to automatically specify the relation, called Dependency, between modeling artifact and domain metamodel. Dependencies can be derived via the Relation characterization between the meta-metamodel and the artifact metamodel. This lets to emphasize those dependencies, which have been exaggerated by the domain meta-model evolution and to have a general overview of the change propagation. An appropriate mega-model specifies the whole ecosystem and weaving models formalizes the relationships between the artifacts and the domain meta-model. The technique is general offers tools independently from the chosen adaptation approach.

Di Ruscio et al. provided an overview of the coupled evolution methods and tools used to solve the coupled evolution problem [36]. They first defined the problem of coupled evolution and explained it through examples. Then they introduced the important elements of the coupled evolution approaches. Based on these elements, tools available for handling the problem of coupled evolution are compared. These tools compared were EMFMigrate, Flock, Cope, GMFEvolution and Levendovszky. These tools were compared based on differencing, adaptations, order, and transformation type, and focus, lack of information management, modularity/reuse, paradigm concrete syntax, refinement, model navigation, directed environment and reference platform. This paper helps in selecting tool selection for solving coupled evolution problem.

A process model and classification scheme is introduced by Becker et al to perform coupled evolution of meta-models and models semi-automatically [37]. The objective of this approach was to reduce the manual effort necessary to migrate old meta-model instances to the new version of meta-model to make them valid. In change detection step, two versions of the meta-model are taken as input and changes are traced. Once the difference is computed, it is stored as a difference model conforming to the difference meta-model. In migration algorithm construction step, changes are categorized according to the proposed classification scheme. As the non-breaking change category does not influence models in any way, no action is required to take in order to resolve this class of changes. Breaking and resolvable changes are handled automatically by generating migration rules to migrate the existing model instances, while breaking and not resolvable changes needs user assistance in order to resolve them. Once the user input is gathered, an appropriate algorithm for model migration is determined. In migration algorithm execution step, the algorithm is executed to migrate all the M1 instances into the M2 instances. However, it is unclear when the migration algorithm is executed either at time when M1 models are instantiated or subsequent to M2 changes. Furthermore, the approach cannot be applied to M3 model changes as the use of M3 model is fixed.

The work in [38] proposed a language called Flock for defining and executing model migration strategies. Flock is a rule based transformation domain specific transformation language. A transformation specification language is a mixture of declarative and imperative syntax. The hybrid M2M transformation languages stir it. Flock handles the relationship between source and target meta-model by utilizing a conservative copying algorithm, which minimizes the need for plainly doubling or un-positioning model elements. Flock utilizes a model-connectivity framework for decoupling model migration and model representation. It has a compatibility with various modeling frameworks. Flock maps each model construct of the old version of model on the corresponding construct of the migrated model by employing a novel conservative copying algorithm and migration rules defined by user. Flock has been applied on various co-evolutions of meta-model\model situations. However, efforts are still needed to develop some indigenous mechanisms for facilitating the reuse of available migrations, and to facilitate user's intervention when required.

Guido Wachsmuth presented a transformational approach to support meta-model evolution by means of stepwise adaptation [39]. For the adaptation of MOF compliant meta-models a transformation library is presented. The transformation detaches the semantic preservation properties and makes distinguish three types of transformations, i.e., transformations for semantic preserving refactoring, introduction and increase in transformation supports meta-model construction and decrease and eliminate transformations permits for meta-model destruction. The transformations are specified as QVT relations and its graphical notations are used. Likewise, transformations for model co-adaptations are also defined that are called as co-transformations. A co-transformation relies on it triggering meta-model adaptation transformations. The co-transformations are described as transformation pattern. The transformation pattern is a QVT relation that takes parameter for meta-model elements. A meta-model transformation initializes a transformation pattern, to obtain a consequent co-transformation.

Schoenboeck et al. presented an approach "CARE" to re-establish an ontological conformance relationship between meta-model and its corresponding instances based on conformance constraints facilitated via logic programming [40]. The relationship between meta-models and its instances are formalized as constraints. CARE relies on the core concepts of meta-modeling, i.e., class, attribute, reference which is a representative subset of Ecore concerning to the structure of meta-model. The approach has three step, i.e., detection of constraints violation, repairing violations and ranking and selection of best possible solution. In detection phase, constraints violations are detected based on the ontological conformance relationship formalized in dedicated constraints form. ASP rules are utilized for this purpose. In repairing phase, violations are repaired via repair actions (ASP repair rules) resulting into the generation of multiple solutions. In the last phase, the quality criteria are applied to rank and select the best possible solution to re-establish the ontological conformance relationship. The quality criteria comprised the structural and semantic knowledge about the prospective solution space. CARE is capable of re-establishing an ontological conformance relationship either the violations are caused by changes in the meta-models or models. CARE is independent of the specific tool via offering a standalone framework as it is not deeply woven into any existing modeling environment. It also minimized human intervention by using pre-defined repair actions. However, models, meta-models and the corresponding constraints are required to be transformed into ASP. Furthermore, it is applicable to the Ecore based meta-models. To make it applicable to other meta-models implementation of the appropriate transformations, modification of the conformance constraints are required.

The work in [41] proposed an approach to automatically adapt existing models after meta-model evolution. They defined meta-model relations to provide foundation for the semantic and instance preservation. The approach handles resolvable changes via QVT transformations, which are classified as refactorings, construction and destruction transformations accordingly. However, it is not clear if co-adaptation rules are automatically generated. In addition, meta-model revisions are represented as QVT transformations, which could be not always exact. Lastly, there is no support to non-resolvable change resolution.

Markus et al. provided a tool support for the coupled evolution of meta-models and models so that the intention behind adaptation cannot be lost. COPE is built on top of Eclipse Modeling Framework (EMF) [42]. The COPE trace and record meta-model evolution as a series of coupled operations. These coupled operations are stored in an explicit history model. The coupled operations are structured in a library, which is extendable. Every coupled operation is composed of meta-model adaptation and model migration. To further minimize, the effort of constructing a model migration, frequent/chronic operations are reused. The existing model instances are adapted automatically by employing history model. Besides, COPE facilitates to examine, recover and refactor the coupled evolution whenever the history model is not recorded and requires to be recovered from the old and new version of meta-models. Although it is applied to several case studies, still the applicability needs to be investigated with a larger number of operators.

They extended an operation-based approach through a way to make certain the semantic preservation of meta-modeling languages [43]. The features, meta-model adaptation, model migration and semantic preservation are combined in an operation-based approach. This approach ensures semantic preservation by specifying couples of semantic adaptation and model migration. The approach is applied to a running Petri Net Example in order to prove its applicability. However, it is applied manually and its integration to the COPE is not addressed yet. Similarly, custom-coupled operators require manual specification of model migration and semantic preservation.

The applicability of COPE is demonstrated by migrating UML Activity Diagrams from 1.5 to 2.0 [44]. Its applicability is demonstrated by reverse engineering the coupled evolution of UML Activity models. The process of reverse engineer the coupled evolution is complex and requires much effort to choose proper coupled operators. To cope with this problem, they proposed a function that based on difference model defines a coupled operator.

Bart Meyers et al. proposed an approach for model migration after meta-model evolution [45]. The approach is based on in place migrations of models. Based on the difference operates, captured during meta-model evolution, in place transformations for model migration are semi automatically derived. Based on the meta-model difference operations, in place transformations are generated for migrating models automatically. Higher Order Transformations are defined for the step-wise evolution of meta-model. Migration transformation is created from difference operation and template where the template defines the migration behavior. The in place transformation migration can then be executed for migrating models. After synthetic adaptation, semantic adaptation is done, which require user intervention. User intervention is restricted in order to ensure model conformance of the migrated model to the evolved meta-model. However, this approach relies on the assumption that difference model is constructed before. This approach does not offer support for the evolution of meta-model. This approach is applicable only when the source and target meta-models are same.

Markus et al. analyzed the practicability of the concept of coupled evolution of meta-models and models [46]. They analyzed and classified the meta-model changes based on their level of potential automation. In addition, a list of suggestions for the implementation of an effective tool for the meta-model and model co-evolution is provided. The change is called a coupled change. The automation of coupled change depends on the level on which it depend i.e. the higher the level, the more it could be reused and thus automated. The coupled changes are classified as 1) Coupled Change Specific to Model: which can be applied only to a subset of models of a meta-model 2) Model Independent, Meta-model Specific Change: can be employed to all possible meta-model instances as it is domain specific that's why it is called meta-model specific change. This type of coupled change offers automation for all model instances of a meta-model and 3) Coupled Change Independent of Meta-model: This type of change can be employed to all meta-models and its corresponding instances. The potential of coupled evolution automation in practice was then assessed by applying the above mentioned classification on the history of two industrial meta-models. This study found that most of the meta-model changes required migration of its corresponding instances. Moreover, to derive migrations automatically snapshots of different meta-model versions are not enough. This study concluded that there exists a great possibility of the reuse of coupled operations of evolution, which could help in minimizing maintenance effort and reduce the chances of error occurrence. Meta-models evolved in a series of meta-model independent as well as meta-model specific changes. Thus, the approaches for automated migration of models ought to maintain the reuse of existing coupled changes also to define new meta-model specific changes. Furthermore, automated coupled evolution requirements are defined here on the basis of results of this study which includes requirements like, reuse of model migration, expressiveness, modularity and history.

M.Wimmer et al. addressed the coupled evolution of meta-models and models by employing existing in-place transformation languages [47]. They merged the initial and final version of meta-model to obtain one meta-model to make in-place transformations applicable. To eliminate model elements that are not the part of evolved meta-model anymore, a check out transformation is used. The approach is demonstrated via employing atlas transformation language for merging meta-models and for taking in the check-out transformations.

Sander D. Vermolen et al. addressed the problem of detecting complex meta-model evolution. They proposed an approach for reconstructing complex meta-model evolution traces from difference model automatically [48]. They formalized the base concepts, which comprise meta-model, difference models and evolution traces. First, changes from difference model are mapped on primitive operators in evolution traces. Then preconditions are defined to solve dependency for all primitive operators. Then the dependency relation between operators is defined based on those pre-conditions, which add sequence to the operators on dependency and to build valid primitive evolution traces from difference model. The primitive traces are then re-ordered and patterns for mapping chain of primitive operators to complex operators are defined. Complex traces are detected by reordering primitive traces to different normal forms. A prototype is implemented in Acoda, A tool of data model evolution for WebDSL. While constructing meta-model formalism concepts like enumerations, packages annotations, operations and derived features are omitted making the approach validity vague. Furthermore, only defined primitive types are supported by the approach. User defined types are not handled for attributes.

Anantha Narayanan et al. proposed a Model Change Language [49]. It is declarative and a high level graphical language that supports a set of co-evolution idioms. The MCL is a model migration language, which is intended to specify model migration rules in order to migrate models (conforming to the old meta-model) to models (that conforms to new meta-model). The MCL comprise UML class diagrams to specify both meta-models and migration rules. MCL is defined via MOF-complaint meta-model. The MCL provides the facility to specify patterns that capture the intention behind the evolution. The basic pattern composes of a LHS element from old meta-model a RHS element from new meta-model and a MapsTo relationship between elements of both sides. However, MCL requires the change specifications. It can be applied only if the change specifications are available. The idioms only cover the syntactical co-evolution not the semantic co-evolution. Furthermore, the meta-model of the change language and its applicability is not demonstrated.

The work in [50] addressed the meta-model changes that prevent automatic model migration. The author presented several techniques to deal with model-specific changes. In addition, they presented a framework to characterize model-specific changes formally in terms of effort required for migrating old models to the new models. They discussed several techniques to deal with model-specific coupled evolution like effort analysis, interactive migration and implicit information. When all models are known and language designers and consumers are relatively close to each other, the language developers can evaluate the effort required for manual migration after meta-model adaptation. Another possible way to deal with model specific coupled evolution is interactive migration, i.e., to make users engage in model migration process. All models are migrated automatically by employing model migration algorithm to extent it is possible and wherever possible it asks for the assistance of user. The user would supply the supplementary information the algorithm is required. It can also provide several alternatives from which the user can choose one. In situation where users make the use of implicit information before making this information explicit through meta-model, the language users can make the use of this implicit information introduced by the language developers.

Markus et al. proposed a generic operation recorder to capture the sequence of operations performed on model via model editor. The recorder is implemented using Eclipse Modeling Framework [51]. The operation recorder supplies operation meta-model, functions on operations and operation recording facility. The operation meta-model express all operations on model and is independent of the meta-model of model. It explicitly defines the types of operations. It is fed with methods defining the semantics of meta-model constructs. The operation performed on model when meta-model is evolved is recorded as model, which conforms to operation meta-model. Two basic types of operations are defined by the operation meta-model i.e. the primitive operations and the composite operations. It permits to persist operations on models, provide support for different model referencing schemes and can be extended by composite and high level operations. Based on operation meta-model, functions are defined to apply and revert operations on models. The operation recording capability can be customized with different patterns to identify composite operations. The operation recorder utilizes the Observer mechanism provided by EMF to pay attention to the changes of evolving meta-model. It records the sequence of primitive operations. Composite operations can be enclosed programmatically or detected automatically. The detection mechanism group a subsequence of operations into a composite operation.

In [52] Antonio Cicchetti et al. improved the adoption of the transformational approach proposed in [53]. This approach is based on the difference model breakdown to differentiate breaking resolvable and un-resolvable changes, which is made to the meta-model. This classification is employed to state the criteria of resolution that offer the break down and the accurate scheduling of amendments. Furthermore, they depicted that how the reliance are rooted by the properties that are specified in the meta-metamodel (they considered KM3 here). This illustrates generality and independence of the results from the metamodel and its semantics. However, the approach is not validated by applying on large set of meta-models. Moreover, power model presented here is not implemented.

Antonio Cicchetti et al. proposed solution to the identified issues and challenges related to the concurrent versioning of modeling artifacts such as meta-models and models [54]. Model based representation is adapted to store the difference between the new and old language revisions. For this purpose, a difference language is obtained by simply extending the meta-model to which models conforms-to. The approach is dependent on the concept of a union meta-model that facilitates the existing representation of manipulations coming from both old and new meta-model versions and consequently their management in terms of change commits, conflict detection and resolution. Furthermore, splitting of difference models can be achieved by the annotations present in the difference meta-model. However, the migration process specified here could result in loss of information. Furthermore, the approach is not validated by applying on some industrial case study.

In [55] Antonio Cicchetti et al. improved the approach proposed in [54] and presented a better solution for the management of concurrent versions of meta-models and models exclusive of threatening abstract and concrete syntax of models and local saving formats a well. The approach relies on computing meta-model differences between meta-model versions and tracking compatibility links between meta-model and model versions. They showed the issues come up while solving the problem of concurrent versioning of meta-models and models through a scenario driven description. Then they proposed a solution for each scenario. The approach has been validated against different examples. However, the complexity of the process requires a systematic validation against industrial scenarios.

In [56] Louis M.Rose et al. analyzed the existing approaches for meta-model evolution and model co-evolution either automatically or semi automatically. They then categorized the existing approaches to co-evolution of models into manual specification, operator based and difference based approaches. They then discussed the prominent features and benefits of each approach and highlighted their limitations too.

In [33] the authors presented an analogous idea in differentiating between group of meta-model adaptations. However, the representation of evolution is not defined and it only handles non-resolvable changes, which need full specification for their resolution.

In [31] proposed an approach for automating model adaptation process every time the consequent meta-model is subject to evolution, that is for example, too complicated, random, and probably *non-monotonic* adaptations. Until now, the already proposed approaches only handled the atomic changes that are assumed to happen in separation and that can then be done automatically in a very simple and easy manner. Complex changes that can be employed with subjective multiplicity and intricacy, pose rigorous complicatedness because these changes can illustrate interdependencies, which compromise the automatablity of the adaptation process.

In [20] the author presented concepts for constructing a formal approach to meta-model evolution. The proposed approach is founded on the Diagram Predicate Framework that formalizes meta-modeling and model transformations based on category theory and graph theory. In DPF perspective, modifications of meta-model can be specified via a set of constraint-aware transformation rules application including the matches and the sequence of the rules applications. These rules are well-grounded in with respect to the modifications. The analysis of meta-model evolution problem is founded on categorical constructions like pullout and pullback. Besides, to guarantee a conformance preserving model migration additional properties are specified.

These approaches attempted to migrating models either automatically or semi-automatically. All these approaches can be classified according to the categories discussed below:

- Manual Specification Approaches defines and uses transformation languages to manually specify model migration [38], [49].
- Meta-model Matching Techniques used to generate model transformations from the difference models illustrating the changes between subsequent versions of the same meta-model [45], [53], [54]. And
- Operator Based Approaches define and record coupled operators that allow to specify meta-model changes along with the corresponding migrations to be applied on the existing models [43], [35], [42].

Apart from the fact, several approaches exist for model migration it is not possible to decide which approach is best suited for the coupled evolution of meta-models and models. Since each approach has pros and cons and depends on particular situation at venture/stake, e.g., frequent and incremental coupled evolution, minimal user intervention and inaccessibility or unavailability of meta-model change history. Next section discusses the approaches proposed to tackle the meta-model/transformations co-evolution.

3.2.2 Meta-model\transformations co-evolution approaches

Since changes introduced to a given meta-model results in transformation inconsistencies, transformations specified on the old meta-model no longer valid with respect to the new version of the meta-model. Transformations need migration/co-evolution to make them consistent and valid with respect to the new version of meta-model.

Surprisingly, metamodel/transformations co-evolution problem have been paid less attention as compare to the meta-model/model coupled evolution problem. This is because it is intrinsically more difficult as compared to meta-model/model coupled evolution problem and because the problem of coupled evolution of meta-model and model has been regarded first by natural choice.

Jokin Garcia et al. proposed a two steps solution for transformations co-evolution after meta-model evolution [2]. At detection stage, simple changes are detected by comparing original meta-model with the evolved meta-model using EMFCompre. The output of this comparison is a Difference model. Then complex changes are detected by considering complex change as predicate over simple change. Complex changes are obtained by extended difference meta-model. From implementation point of view this task is recognized as transformation that takes difference model as an input and generate DiffExtended model comprising of both simple and complex changes. The similarity analysis can be done between source and target metamodels is done using AML (AtlanMod Matching Language). The matches are saved into a weaving model". This weaving model acts as a high-level specification of adaptation transformation. The similarity analysis step is optional. At co-evolution Stage, Higher Order Transformations

(HOT) is defined that takes transformations as input and return modified transformations. It takes the changes, detected at detection stage, as input and defines correspondence that maps original transformations on evolved transformations. Higher Order Transformations (HOT) are realized as ATL rules. The transformation rules R are captured as tuple rule, i.e., (id, source, target, filters, and mappings), where "source" and "target" refers to classes of the input and output meta-model correspondingly; "filters" comprised related predicates, in the form of set, over the source elements, i.e., the rule will only be activated if the predicate is fulfilled, "mappings" refer to bindings, in the form of set, to settle the attributes of the intended target model element.

David Mendez et al. specified transformation consistency by analyzing the relationship between meta-models and transformations, called as domain conformance and outlines necessary steps to reinstate the consistency after meta-model evolution [9]. The steps they proposed for transformation migration are the impact detection, the impact analysis and the last step is transformation migration. In impact detection step, transformations inconsistencies induced by meta-model evolution are identified. Every transformation that is violating domain conformance is identified. In the next step, transformation updates required to re-establish the domain conformance relationship is gotten through possible human assistance. During the final step, all the required updates identified during impact analysis are applied. Furthermore, some useful steps to perform impact analysis are mentioned. While the steps of impact analysis are applied on some useful examples, no demonstration for the entire idea is performed.

Davide De Ruscio et al. presented a procedural approach to the co-evolution of meta-model along with ATL Transformations [8]. This approach comprises various activities that include description of meta-model changes, the assessment of their impact on the related artifacts, the maintainability of the stimulated adaptations and migration of the impacted artifacts. The coupled evolution is performed via 1) defining dependencies between source meta-model and ATL Transformations by establishing a relation of each meta-model element to its corresponding transformation. Which is later on weaved in a weaving model automatically. 2) Analyzing the impact of changes by detecting which element of transformation is impacted by a change. 4) Assessing the cost of transformation adaptation by taking into account the previously identified affected elements. 3) Determining its sustainability by ultimately reviewing certain decisions and in final step 4) if the evaluation has positive result then the impacted transformation is adapted. The supporting techniques for each activity that can be used are mentioned. Furthermore, cost evaluation of each adaptation is described in details and its applicability is shown.

Steffen Kruse presented a set of operators to adapt transformations after meta-model evolution to make them consistent with the new version of meta-model [57]. These operators assist in the evolution of meta-model along with the automatic or semi-automatic adaptation of transformations depending on the type of change introduced to the meta-model. Operators are categorized as atomic and complex operators. Complex operators formed of the combination of atomic operators. The operators are tested on Copy Transformations specified in ATL and targeted only the declarative part of ATL. They analyzed the impact of change on ATL transformation rules and highlighted the parts of the rule, which can be impacted by meta-model changes either directly or indirectly. The parts of rules like source pattern, condition imposed on source pattern and bindings can be impacted by source meta-model evolution while the target pattern can be impacted by target meta-model evolution. The evaluation demonstrated that operator-based approach could be useful if applied to the transformations co-evolution. It guaranteed the synthetic correctness of transformations. Nevertheless, semantics of transformations is still difficult to handle. Because the semantic relationship between meta-model and transformations is still not well-defined and is ambiguous. Furthermore, the impact of meta-model evolution on transformations not only depends on the type of change but also on the usage of elements in transformations. It might be possible that one transformation can be automatically adapted to the change in meta-model while other transformation using the same meta-model might require human assistance. This makes the idea of automatic evolution of transformations a bit more difficult and needs further investigation.

3.2.3 Meta-model\ Editors co-evolution approaches

As discussed formerly, meta-models support the development of broad range of relevant artifacts together with the models and transformations. For example, meta-models play a major role in the definition of domain specific modeling languages (DSMLs), since they define the abstract syntax of the language being proposed. In addition, a number of other related artifacts are developed to specify the concrete syntax as well as additional aspects associated to the semantics or requirements of a particular domain specific modeling language tool. In such scenarios, specific techniques are essential ingredient to transmit any kind of change related to abstract syntax to the reliant artifact for instance, graphical or concrete syntax models.

Different approaches have been proposed to specify concrete syntax of modeling languages, in the Eclipse Modeling Framework (EMF) [58]. For instance, for developing graphical editors GMF [12] is defined, for generating textual editors EMFText [59], TCS [60], and XText [61] are defined. These approaches are generative in nature capable of generating working editors starting from source specifications that at different level are associated to the abstract syntax of the considered DSML. The relation between the metamodel of the language and the editor models is fragile as compare to the relationship amongst the meta-model and model and metamodel and transformations. Consequently, even the detection of the inconsistencies between the new version of the metamodel and the editor's models is too difficult to be identified. Moreover, even if one could identify such inconsistencies, re-establishing consistency needs a profound knowledge of the used editor technology to appropriately proliferate the metamodel changes to the editor models, and in some scenarios to the generated code.

In [62] the author proposed an approach to adapt the GMF editors. The approach is dependent on three adapters capable of automating the propagation of domain model changes i.e. meta-model modifications to the GMFTool, EMFGen and GMFMap models needed to generate the graphical editor by GMF. Particularly, EMFGen is a model used by the EMF generator to generate Java code required for the management of models conforming to the metamodel of the modeling language. GMFTool specifies toolbars and additional periphery of the editor to assist the diagram content management. GMFMap connects jointly all the other GMF models. The mentioned approach is the only attempt made to deal with the coupled evolution problem of GMF. Further efforts are required to deal with the problem particularly to attain the full coverage of Ecore and to gain complete understanding of the semantics of model dependencies of tools and GMF.

3.2.4 Uniform approaches to Meta-model and related artifacts co-evolution

Hoisl et al. proposed an approach for solving not only the coupled evolution problem of meta-models and model but also of meta-models and transformations [36]. The approach is based on 1) creating bi-directional transformations between modeling artifacts, which ensures tracing, coupling and synchronization of all modeling artifacts and 2) employing Higher Order Transformations (HOT) on first class model representations of transformation specifications. This enables the propagation of changes in both directions, i.e., from source model to target model and target model to source model. Furthermore, transformations are presented as models and changes are propagated into horizontal and vertical transformations, i.e., transformation between models on the same as well as different abstraction level. However, the applicability of approach is not demonstrated by applying on examples. Furthermore, no prototype is developed.

Di Ruscio et al. aimed at providing a uniform and comprehensive support to the coupled evolution in a way that it is not restricted to specific types of artifacts [63]. They defined the relationships of meta-models with their related artifacts that lies in a common meta-modeling ecosystem and discovered the commanlities to define a systematic adaptation process. Based on these commonalities, they proposed EMFMigrate to manage meta-model evolution in a comprehensive and uniform way. They utilized migration programs to co-evolve artifacts uniformly without through ecosystem. EMFMigrate comprises a domain specific language aimed to provide modelers with constructs to 1) define migration libraries to substantiate and allow the reuse of repeated/perennial artifacts adaptations, 2) customization of migrations available in library 3) management of migrations not fully automated and requires human assistance. The migration rules are executed following meta-model changes. Meta-model changes can be specified manually or introduced automatically to the meta-model. Tool support is provided to implement the presented approach. The approach is demonstrated by applying on different scenarios of coupled evolution.

3.3 Model Driven Engineering and Traceability

Any of the software artifacts hardly evolves in isolation. Changes introduced to the one artifact might cause changes to the other related artifacts or it may be because of some changes to the other artifacts. For instance, source code and documentation are modified due to change in requirements. Thus, traceability is strongly related to software evolution and is used to support software management, software evolution and validation [64].

Traceability is an essential property of system. Traceability mechanisms assist in understanding, capturing, tracking and verifying software components the dependencies and relationships among different software components during the entire software development life cycle[65]. Typically, traceability is the subfield of Requirements Engineering. Nonetheless, it is now increasingly employed for other artifacts likewise. Since, Model Driven Engineering offers new opportunities to create and utilize traceability information. The next subsequent sections discuss the traceability approaches in model driven engineering.

3.3.1 Requirements Management Approaches

Traceability is defined as "The capability of defining and following the life of requirements in forward as well as in backward specification, to its following deployment and usage, and during the stages of ongoing refinement and iteration in any of these phases" in Requirements

Engineering [66]. Forward and backward requirements traceability assists the engineers and stakeholders to know in depth the semantics of requirements.

Almeida et al. designed a framework particularly tracing requirements, model transformation specification quality assessment and the quality assessment of meta-models, models and realization as well [67]. They intended to simplify relationship management among requirements and several design artifacts. The framework proposed here let the designer to associate requirements to several artifacts of the design process of MDE at the start of the development. They utilized traceability cross-tables to represent relationships among software system models and requirements, contemplating several granularities of the models and the description of conformant transformation descriptions. Furthermore, they proposed a concept of conformance among models throughout the various abstraction levels in order to trace requirements. They identified the analysis of the impact of the requirements change on these models as future work.

Cleland-Huang et al. proposed an approach to manage effect of change on the nonfunctional requirements of a system [70]. This approach is goal-centric. To model non-functional requirements and their dependencies, the Goal Centric Traceability uses a Softgoal Interdependency Graph (SIG). GCT facilitates developers in understanding and assessing the effect of modifications in functional requirements on non-functional requirements to preserve the software system quality. The goal modeling, the impact detection, the analysis of goal, and decision making are the four main stages of this process for the assessment and up-dation of the dependent artifacts with modifications. During the first phase i.e. goal modeling, all goals are broken down into sub-goals to imitate the reality that general inter-relationships happen among different non-functional requirements and is modeled by sub-goals. Besides, to comprehend the trade-offs between non-functional requirements, the sub-goals are further broken down into operations that gives contestant resolutions for each goal. During impact detection stage, once the changes to the non-functional requirements are occurred, the algorithm (a probabilistic retrieval algorithm) dynamically returns connections between elements in the SIG and affected classes. During goal analysis; the user changes the contributions, from the goal elements that are affected to their parents. For all affected elements, modifications are transmitted during the SIG to detect goals, which are affected strongly. During the last stage, i.e., decision making, decision is made about to carry on with which planned modification. Stakeholders then assess the effect of the planned modification on goals of the non-functional requirements and deal with threats.

Cleland-Huang et al. used Event Based Traceability for the management of meta-model evolutionary modification. Event-Based Traceability (EBT) is a technique to automate the generation and maintenance of the trace links [68]. In EBT, the requirements and related traceable artifacts, for instance design models, linked through publish-subscribe relationships and are not directly related any more. The proposed mechanism is built based on the Observer design pattern [69]. Links are created via an event service, rather than the direct establishment and tightly coupled relationships among requirements and other related artifacts. First, subscriber manager registered all the considered artifacts with the event server. Then, for the management of the requirements document modifications and to bring out these modifications as event to the event server, the requirements manager employed its event recognition algorithm. The event server handled some links among the requirement and its reliant artifacts by means of several information retrieval algorithms.

3.3.2 Model Consistency Management Approaches

The model driven approach is centering models as its key artifacts. Model Driven Engineering identifies the need to have several model types to represent system at different development stages, i.e., from requirements analysis through final implementation. The intended models might characterize various system characteristics that are structural or behavioral or the models might model application at different abstraction levels that is analysis models or design models. Contradictions between models, which represents diverse aspects of a software application or amongst descriptions at varying abstraction levels, may arise at some stage in any phase or between development phases, lifting up the issue of how to deal with inconsistencies between models and among models and code[71]. Grundy et al reviewed tool, approaches to manage inconsistencies, and identified the need to develop mechanisms to identify inconsistencies and warn users about inconsistencies and help in monitoring and resolving inconsistencies [72]. Desfray identified traceability as an essential ingredients if the solution to manage inconsistencies [73]. [68, 74, 75] utilized traceability as the basis for the detection of inconsistencies and to inform concerned stakeholders about inconsistencies raised at any point in any phase during development.

3.3.3 Models Synchronization approaches

Development artifacts modifications may probably require the synchronization of interrelated artifacts e.g. models, code, documentation etc. Traceability links (which record the relationships among different software artifacts) make the synchronization of interrelated artifacts possible. This section debates the ways in which change is propagated to the dependent artifacts by the researchers, which usually comprises employing an incremental style of transformation.

Several approaches to model synchronization extend the existing model-to-model transformation languages. Transformation languages with declarative syntax are appropriate for the specification of bi-directional transformations and incremental transformations, a model transformation method that supports incremental modification of the target models. In fact, considerable part of the research related to model synchronization focuses on incremental transformation. [76] [77-80]

Automated Model Synchronization: Besides live transformation, the researchers proposed several techniques to record trace links between models. For this purpose, a model-to-model transformation has been supplemented with traceability information via a generic higher-order transformation [81]. The generic higher-order transformation takes model transformation as input and adds transformation rules that produce a traceability model. In contrast to the generosity of the above mentioned approach, Drivalos et al. offered domain specific traceability meta-models for more productive and deeper traceability link semantics [82]. Additional research is needed to evaluate and identify automated model synchronization tools requirements and to decide an appropriate traceability approach for their implementation.

Impact analysis is the activity carries out to reason about the effects of a modification to a development artifact. Besides alleviating change propagation, impact analysis might assist in predicting the cost and complexity of changes[83]. Impact analysis demands solutions to several sub-problems including change detection, change impact analysis, and effective presentation of the analysis.

Impact analysis can be done for UML models by making comparison of the original and evolved versions of the same model to generate a report of evolved elements of model that are impacted by the modifications made to the original model elements [84]. To provide support for impact analysis, Briand et al. described change patterns that consist of a trigger for change detection and an impact rule for marking model elements affected by this change.

Only event-based approaches attempted to automate impact analysis activity. Winkler et al. proposed an event based approach for automating impact analysis [64]. Due to the employment of patterns for change detection and to specify resulting actions, event-based impact analysis is analogous to differencing approaches used for schema evolution for instance the approach proposed in [85]. If more than one trigger is applied, event-based impact analysis approaches should offer mechanisms for the selection of pattern between available applicable patterns. For selection between applicable patterns the selection policy employed by Briand et al. is implicit, user cannot change it and does not support the selection of particular applicable pattern between the available applicable patterns.

To end with, model synchronization tools may employ techniques, which are used in automated synchronization tools for traditional development environments. For instance, the refactoring operation of the Eclipse JDT transmits modifications between classes via a cache of the workspace to enhance scalability and performance [86].

Model-to-Text Synchronization: Until now, this section focused on model-to-model synchronization supported by traceability. Traceability is of crucial importance for other software evolution activities in a model-driven development environment for example, models-to-text synchronization and between models and trace links and these activities are now discussed below in detail.

While most of the researchers focused on model-to model synchronization, some researchers focused on synchronization between models and text or models and trace links. For change synchronization in requirements documents with models, there is loads of work in the requirements engineering field, where the requirement for traceability was first identified and taken into account. For the synchronization of models with generated text (for example, throughout code generation), Epsilon Generation Language (EGL) proposed by Rose et al [87]. EGL is a model-to-text language, which creates traceability links between code generation templates and generated files. Segments of code can be checked off protected, and are not overwritten by consequent calls of the code generation template. The MOFScript is a model-to-text language, like EGL, which provides protected sections. But contrasting to EGL, MOFScript records and saves traceability links in a structured way, facilitating impact analysis, model coverage (so as to highlight which areas of the model is contributing to the generated code) and orphan analysis (so as to detect invalid traceability links)[88].

Model-to-Trace Links Synchronization: Trace links might also be affected when modification is made to the development artifacts. Synchronization tools depend on accurate trace links and thus the maintenance of trace links is crucial. Winkler & Pilgrim proposed that trace versioning should be used to deal with the challenges of trace link maintenance [64], which consist the accidental inclusion of unplanned dependencies as well as the exclusion of essential dependencies. Besides, they noted that, though versioning traces has been investigated in many specialized areas, e.g., hypermedia[89], there is no holistic approach for versioning traces.

3.4 Change classification

In [48] a classification scheme has been proposed for meta-model changes in the Eclipse Modeling Framework Ecore meta-metamodel. They categorized all possible change operations in three categories i.e. non-breaking changes, breaking and resolvable changes and breaking and non-resolvable changes.

In [47] this classification scheme is adapted for classifying MOF-based meta-model changes. In addition to this classification scheme, they proposed change meta-model to represent changes introduced to the MOF based meta-model. The change meta-model is capable to express evolution process of MOF-based meta-model employing MOF itself as a change description language. So far, this is the only proposed classification scheme employed to classify change operations while addressing co-evolution problem, particularly co-evolution of meta-model\model.

3.5 Difference Representation Mechanisms

Meta-models are subject to evolutionary pressures which ineluctably/certainly affects its related artifacts, i.e., models and transformations, etc. These artifacts are no longer consistent to the new version of meta-model and need to be migrated.

To cope up with this issue, different co-evolution approaches have been proposed during last several years. These co-evolution approaches heavily relied on differencing techniques. Differencing techniques are employed to find out difference between the old and new version of meta-models. These differences are then utilized to derive migration strategies.

In [3] the difference between the two versions of meta-model are symbolized as model, whose meta-model is derived from KM3 meta-model automatically. To represent deletion, addition and modification of each meta-class MC AddedMC, deletedMC and changedMC are produced. All types of changes either breaking and resolvable or breaking and not resolvable are formalized as difference model. The difference model is capable to capture the breaking and resolvable and breaking and non-resolvable changes altogether that is then decomposed into two disjoint models i.e. ΔR and Δ¬R, denoting breaking and resolvable and breaking and non-resolvable, respectively. Furthermore, to resolve the parallel dependencies between breaking and resolvable and breaking and non-resolvable changes, the analysis of the dependency and criteria for resolution are presented to break and plan the modifications.

A Meta-model to present delta (difference) between models has been proposed in [57] by David Hearnden et al. the Delta model is founded on MOF and is capable to describe model deltas by means of identity maps. However, an element's addition, deletion and modifications made to properties are specified systematically, the delta model misses the descriptive methods for the modifications made to the model structure e.g. containment or generalization. This work focused on meta-model transformations, for which evolutionary aspects are characterized by means of the TefKat language. The change meta-model proposed here is fixed, as the meta-model is the MOF model, and the change semantics can be defined more explicitly, concerning meta-model changes effects to existing instances.

In [40] a meta-model independent approach to derive the difference model, which also relied on single change operations is presented. The difference between two models is represented as model, which conforms to the difference meta-model derived from the former meta-model. Moreover, the DiffModel provoked transformations that can be applied to one of the delta models to get the other model.

In [4] the changes between two versions of meta-model are represented as EPatch format. An EPatch comprised a textual domain specific language defined by XText grammer. EPatch is obtained from two available versions of meta-model. The Patcher tool applies an EPatch format to upgrade an old version of meta-model to the new version automatically. The EPatch is capable to define changes occurred due to constructing, destructing or refactoring elements of a meta-model. The EPatch relies on difference computed via EMFCompare. Since EMFCompare DiffMOdel holds a hard, reference to the compared models and thus can't be employed to just one model.

3.6 Model/transformations Migration strategies

The approaches proposed to meta-model\model co-evolution attempted to migrating models either automatically or semi-automatically. All these approaches can be classified according to the categories discussed below:

- Manual Specification Approaches, define and use transformation languages, which manually specify model migrations However, manual specification of migration is monotonous and error prone and the developers need to learn more languages
- Meta-model Matching Techniques use to generate model transformations from the difference models illustrating the changes between subsequent versions of the same meta-model Nonetheless, matching techniques have the capability to completely automate the migration building process, and it might lead to incorrect migrations. Furthermore, they can suggest more than one strategy for migration. and
- Operator Based Approaches define and record coupled operators that allow specifying meta-model changes along with the corresponding migrations to be applied on the existing models. Though operator based approaches, overcome issues of the manual as well as matching techniques. Nonetheless, it requires seamless integration into meta-model editor. Of the proposed approaches to model migration, several consider only atomic changes as in [15], and it is not clear that the approach can be employed in general case.

Apart from the fact, several approaches exist for model migration it is not possible to decide, which approach is best suited for the coupled evolution of meta-models and models; since, each tool has pros and cons and depends on particular situation at venture/stake, e.g., frequent and incremental coupled evolution, minimal user intervention and inaccessibility or unavailability of meta-model change history.

Up till now, researchers addressing the meta-model/transformations co-evolution problem utilized the migration strategies proposed and employed to cope up with co-evolution of meta-model\model tuning out the fact that the relationship between meta-model and models and meta-models and transformations is different. Hence, transformation migration requires distinct approaches to be employed to address meta-model/transformation co-evolution. Furthermore, a well- defined relationship might help in defining migration approaches to address co-evolution problem. To this end, the relationship between meta-model and transformation is formalized in [19] and further refined in [15]. An appropriate approach that can re-establish the formalized relationship between meta-model and transformation after meta-model evolution is needed to be developed.

3.7 Issues and challenges related to the co-evolution approaches

Model transformation holds a significant part in model driven engineering [30]. Since transformations are defined based on meta-models, evolution of meta-model causes inconsistencies in the existing transformations with respect to the evolved meta-model. To make transformations consistent with respect to the evolved version of meta-model only few approaches have been proposed, so far. This section sums up the literature review by highlighting the potential research directions, discussing the issues of existing approaches and summing up the chapter with a problem statement.

3.7.1 Issues of change classification scheme

Meta-models might evolve in several ways: some changes may be additive and autonomous in nature, thus demanding no or little action. However, most of the changes to meta-model lead to inconsistencies and incompatibilities that cannot be easily resolved. Because, the coupled modifications, essential to adapt modeling artifacts in response to the modifications made to the corresponding meta-model, depending on the relation that mutually couples the modeling artifact and the meta-model. As the literature (e.g., [5], [15], [19] and [49, 50] suggests, the relations involved in the co-evolution problem are

- ConformsTo: a relation between meta-model and model
- DomainConnformsTo: a relation between meta-model and transformations
- DependsOn: a generic and more complex type of relation that exists between meta-model and other modeling artifacts which don't have a direct and well-defined dependence on meta-model

Since meta-models have different type of relationship with its related artifacts, changes introduced to the meta-model might influence its relationship with different artifacts in different manner. Meta-model changes may affect severely its relationship with one specific artifact and may have no impact on other. Therefore, changes should be classify with respect to its impact on considered artifact, i.e., models, transformations and\or other related artifacts. As discussed previously, the classification scheme proposed for meta-model\model co-evolution problem classified meta-model changes with respect to its impact on models. Same classification scheme is employed for transformations co-evolution tuning out the fact the relationship type vary depending upon the considered artifact. Tough, meta-model\transformations co-evolution requires different type of change classification schemes; currently no change classification scheme exists for meta-model\transformation co-evolution problem.

3.7.2 Issues of Existing Difference Representation Mechanisms

Previously proposed difference representation mechanisms for co-evolution of meta-models\models are all meta-model dependent as in [3], [53]. The approach in [40] claimed that the difference representation is independent of meta-model however, the Diff meta-model is derived from the old version of meta-model making the approach meta-model dependent. A change meta-model proposed in [47] is based on MOF and represent a difference only between meta-models which are based on MOF.

The dependency of difference mechanisms on meta-models restricts their applicability to only specific situations and meta-models. Furthermore, the existing mechanisms lack the capability of representing the complexity as well as severity of change impact. The categorization of difference representation is formless, making it difficult and quit confusing to place a change under a certain category.

3.7.3 Issues of Model Migration strategies

Migrations strategies proposed for meta-model\model co-evolution fall in three basic categories, i.e., manual specification approaches, meta-model matching techniques and operator based approaches. The conspicuous issues of these approaches are:

- Manual specification of migration is tedious and error prone and the developers must have to learn new languages to specify migrations.
- Matching techniques can lead to incorrect migrations and can propose more than one workable strategy for migration and

- Operator-based approaches require seamless integration into model editors to identify changes.

Existing approaches typically wind up with obscuring the gap between impact analysis and adaptation semantics. Consequently, it is of crucial importance to illuminate the type of the relation, which exists between the meta-model and the other artifacts. Certainly, dependencies come out at different stages throughout the life-cycle of meta-model, and with various levels of causality relying on the nature of the artifact under study. Besides, some of approaches and tools work on the assumption the traces of meta-model changes are available as in [6], [16], which might not always be feasible.

Furthermore, as discussed in [52] and [15] it is unfeasible to identify the best tool to support the meta-models\models co-evolution. Every tool has some strong and weak points and relying on the particular situation of interest (e.g., recurrent, and incremental co-evolution, minimal user intervention, and missing meta-model change history) some approaches can be preferred with respect to others.

3.7.4 Issues of Transformations migration strategies

Meta-model evolution may affect transformations and cause inconsistencies to the transformations specified on it, hence, transformations might no longer satisfy the "DomainConformsTo" relationship. To re-establish the domain conformance relationship only few approaches have been proposed by the researchers during last several years. Most of the researchers employed the migration strategies proposed to cope up with meta-model\model co-evolution problem overlooking the fact that co-evolution of meta-model\model and meta-model\transformations differs in nature and, therefore, requires a different and dedicated solution.

Moreover, these approaches guaranteed the syntactic correctness but semantics of transformations is still difficult to handle. Because, the semantic relationship between meta-model and transformations is still not well-defined and is ambiguous. Furthermore, the impact of meta-model evolution on transformations not only depends on the type of change but also on the usage of elements in transformations. It might be possible that one transformation can be automatically adapted to the change in meta-model while other transformation using the same meta-model might require human assistance. This makes the idea of automatic evolution of transformations a bit more difficult than expected or assumed and need further investigation.

3.8 Chapter Summary

This chapter has assessed and reviewed the research related to the evolution of meta-model. Three types of approaches addressing the co-evolution of meta-model\model were identified and explored. Manual specifications approaches, specify and employ transformation languages that manually defines migration strategies. However, manual specification of migration is difficult and the error prone and the developers need to learn more languages. Matching techniques generate model transformations from delta models illustrating the difference between the subsequent versions of meta-model. Nonetheless, matching techniques have the capability to completely automate the migration building process; it might lead to incorrect migrations. Furthermore, they can suggest more than one strategy for migration. Operator-Based Approaches define and record coupled operators that allow specifying meta-model changes along with the corresponding migrations to be applied on the existing models. Though operator based approaches, overcome issues of the manual as well as matching techniques.

Nonetheless, it requires seamless integration into meta-model editor. Of the proposed approaches to model migration, several consider only atomic changes as in [15], and it is not clear that the approach can be employed in general case.

Further review of the literature explored the employment of operator-based approach to co-evolution of meta-model\transformation. Though, meta-model have different type relationship with models and transformations. This aspect has been totally overlooked while addressing co-evolution of meta-model\transformation issue and employing it to both issues i.e. meta-model\model co-evolution and meta-model\transformation co-evolution. Both issues must be handled in different way and different techniques, focusing on the relationship type, needed to be developed.

Chapter. 4 Problem Definition

4.1 Introduction

Model transformation performs a significant role in the context of MDSE (model driven software engineering). Transformations are defined based on meta-models. Therefore, meta-model evolution causes inconsistencies in the existing transformations. So far, the proposed approaches to solve transformations co-evolution problem focuses only on some specific aspects of transformation co-evolution problem leaving the others.

4.2 Problem Statement

Meta-model may evolve due to several reasons such as design refinements and requirements change. The evolution causes inconsistencies in artifacts that are relying on meta-models and need to co-evolve to re-establish the relationship between meta-model and related artifacts. The problem of co-evolution of models is extensively investigated during last several years. Several approaches manual, semi-automated or automated have been proposed to address models co-evolution problem.

Although, transformations are fundamental ingredients of Model Driven Engineering (MDE) and are specified upon source and target meta-models. Like any other artifact, the evolution of meta-model also cause inconsistencies in transformations defined over the meta-model. Transformations need to co-evolve with the evolution of meta-model to re-establish its consistency. However, transformations co-evolution has been paid less attention. Only few approaches have been proposed to address the transformations co-evolution problem. However, many of these approaches are acquired from model co-evolution approaches. The fact that co-evolution of model and transformations differs in nature, is totally overlooked.

Being particular, the change classification scheme proposed for model co-evolution is employed for transformation co-evolution problem. This classification scheme classifies meta-model changes based on its impact on models. The impact of meta-model change on model depends only on the type of change, while the impact of meta-model evolution on transformations not only depends on the type of change but also on the usage of elements in transformations. Besides, models and transformations differs in nature and both have different type of relationship with meta-model, therefore it is inappropriate to employ or devise same classification scheme for both.

In addition, difference between meta-model versions needs to be represented in some formal form. Current approaches do not employ any difference mechanism to compute difference between different versions of considered meta-model. Difference representation mechanism assists in identifying and recording the difference between old and new version of meta-model. Based on this difference, migration strategies can be devised.

Furthermore, there is no support to establish dependency relationship between meta-model and transformations. In [10] the author created implicit traces between meta-model elements and transformations by defining bi-directional transformations between modeling artifacts. However, no explicit traces are defined and stored which can be used later on to detect the impact of meta-model changes on transformations.

4.3 Research Gap

Moreover, the approaches proposed previously guaranteed the synthetic correctness but semantics of transformations is still complicated to handle. Because the semantic relationship, between meta-model and transformations, is still not well-defined and is ambiguous. Therefore, Meta-model\transformation co-evolution problem requires a novel and dedicated solution to overcome the following limitations:

- Current approaches lack the capability to provide change classification Scheme particular to the Transformation co-evolution problem that categorize meta-model changes according to its impact on transformations and is generic enough to employ in any kind of situation of transformation co-evolution problem.
- Existing approaches lack the capability to devise difference mechanism, which is meta-model independent and generic enough to be applicable in any situation of transformation co-evolution problem.
- Transformation migration strategy, are meta-model and transformations specific. Currently, there is no generic transformation migration strategy, which is meta-model and transformation language independent is independent.
- Existing difference mechanisms lack the capability of estimating the complexity and severity of change.
- The proposed approaches guaranteed the syntactic correctness but semantics of transformations is still difficult to handle.
- The existing approaches overlooked the fact that the impact of meta-model evolution on transformations not only depends on the type of change but also on the usage of elements in transformations.

Chapter. 5 Proposed Approach

5.1 Introduction

Meta-model evolution affects its related artifacts drastically. The impact of meta-model evolution varies from artifact to artifact. This is because it depends on the type of meta-model change as well as the type of relationship that meta-model hold with its related artifacts. In previous chapters, we have analyzed various approaches that have been proposed during previous decades targeting meta-model evolution and co-evolution problem. The thorough analysis of the existing approaches has identified several issues with the existing approaches.

To reduce the effort required to evolve meta-model and to co-evolve transformations and to overcome the limitations of the existing approaches identified and highlighted in the previous chapters, we propose a systematic and automatable approach to transformations co-evolution. Our approach is remarkably eccentric from existing approaches to transformations co-evolution. Our approach employs traceability meta-model and change meta-model to automate the entire strategy of co-evolution of meta-model and transformations. Until now, none of the existing approaches used the concept of traceability for the impact analysis and change propagation to evolve meta-model and co-evolve transformations.

This chapter illuminates the systematic automatable approach to meta-model and transformations co-evolution. The chapter begins by presenting some important concepts significant to our proposed approach and proceeds by defining and founding the basic idea of "traceability" and "change" that would be employed by the approach for change identification and propagation and impact analysis. Section 5.5 gives an overview of the supported algorithm and 5.6 summarizes the entire chapter.

5.2 Pre-Requisites

This section introduces some essential and basic concepts and terms required to support and explain the proposed approach. Section 5.2.1 presents the running example, which is used to explain different steps of the approach. Section 5.2.8 discusses the meta meta-models. Section 5.2.9 describes the input source meta-models. Section 5.2.10 discusses the input transformations.

5.2.1 Transformations

The term transformation T denotes the mapping between source models and target models. Transformation can be queries, transformation rules or helpers depending on the type of transformation language used to specify mapping between source and target models. For example, the transformation rule defined to map a UML class on relational Table using an ATL language is transformation T.

5.2.2 Model Element

The term model element E is used to represent the element of the source or target meta-model element employed by the transformations T either implicitly or explicitly. A model element can be a type, class, an attribute or reference. For instance, a class "Attribute" in Class meta-model is model element.

5.2.3 Trace

A trace TR is the mapping between a model element E and Transformations T. These traces can be either implicit or explicit depending on the type of usage of model element E by transformation T. For instance, A TR "ClasstoCT" is the trace that mapped the model element class in Class meta-model on the transformation ClassToTable in ATL Transformation file.

5.2.4 Change

The concept "Change" C represents the change that is introduced to some model element E. For example, renaming a model element "Attribute" to "Property" denotes a change C in the name of model element "Attribute", which would be now identified as "Property", I.e.,

$$C \rightarrow \Delta E \text{ where E is model Element}$$

5.2.5 Co-change

The co-change Cc depicts the resultant changes of the model element change that need to be made to the transformations T in order to reflect the model element change C in transformations T and make them consistent with the new version of underlying input source meta-model. For example, C that is introduced into model element "Attribute" requires incorporating the Cc to the source of the rule "AttributetoColumn", i.e.

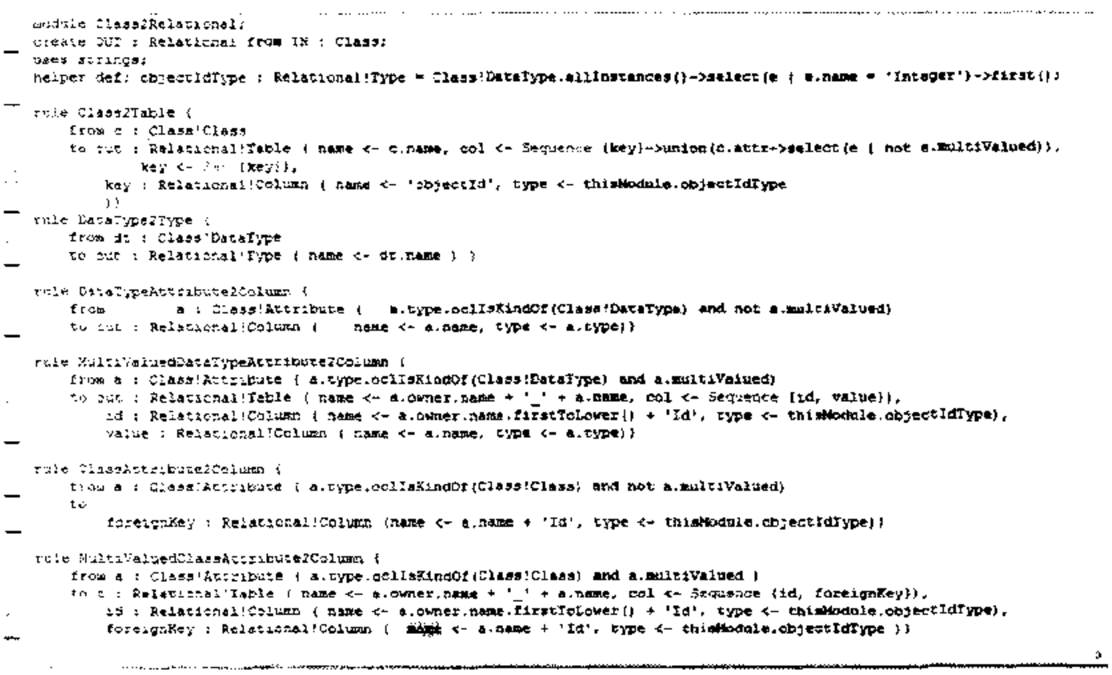
$$Cc \rightarrow \Delta T \text{ where T is transformations and Cc is the resultant change of C}$$

5.2.6 Transformation Example

"Class to Relational" transformation example obtained from ATL Zoo (link) is used to demonstrate the proposed approach. The "ClassToRelational" example specifies the simplified model transformations of a class model to a data base model. This transformation executes the case study applied in the Workshop Model Transformations in Practice collocated with the MoDELS Conference, October, 2005. The description of the case study can be found at the workshop web site: http://sosym.dcs.kcl.ac.uk/events/mtip05/.

5.2.7 Model Transformations

Model transformations play a vital role in Model Driven Engineering since they are capable to produce target models from source models. Transformation definition is based on source and target meta-models. Figure 5.1 illustrates transformations capable to generate tables conforming to relational meta-model from class models conforming to simple UMl class diagram meta-model and is part of the "ClassToRelational" transformation example.



```
module Class2Relational;
create OUT : Relational from IN : Class;
uses strings;
helper def: objectIdType : Relational!Type = Class!DataType.allInstances()->select(e | e.name = 'Integer')->first();

rule Class2Table {
    from c : Class!Class
    to out : Relational!Table ( name <- c.name, col <- Sequence {key}->union(c.attr->select(e | not e.multiValued)),
            key <- Set {key}),
        key : Relational!Column ( name <- 'objectId', type <- thisModule.objectIdType
        ))
rule DataType2Type {
    from dt : Class!DataType
    to out : Relational!Type ( name <- dt.name ) }

rule DataTypeAttribute2Column {
    from        a : Class!Attribute (   a.type.oclIsKindOf(Class!DataType) and not a.multiValued)
    to out : Relational!Column (    name <- a.name, type <- a.type)}

rule MultiValuedDataTypeAttribute2Column {
    from a : Class!Attribute ( a.type.oclIsKindOf(Class!DataType) and a.multiValued)
    to out : Relational!Table ( name <- a.owner.name + '_' + a.name, col <- Sequence {id, value}),
        id : Relational!Column ( name <- a.owner.name.firstToLower() + 'Id', type <- thisModule.objectIdType),
        value : Relational!Column ( name <- a.name, type <- a.type)}

rule ClassAttribute2Column {
    from a : Class!Attribute ( a.type.oclIsKindOf(Class!Class) and not a.multiValued)
    to
        foreignKey : Relational!Column (name <- a.name + 'Id', type <- thisModule.objectIdType)}

rule MultiValuedClassAttribute2Column {
    from a : Class!Attribute ( a.type.oclIsKindOf(Class!Class) and a.multiValued )
    to t : Relational!Table ( name <- a.owner.name + '_' + a.name, col <- Sequence {id, foreignKey}),
        id : Relational!Column ( name <- a.owner.name.firstToLower() + 'Id', type <- thisModule.objectIdType),
        foreignKey : Relational!Column (  name <- a.name + 'Id', type <- thisModule.objectIdType )}
```

Figure 5.1 Sample ATL Transformations

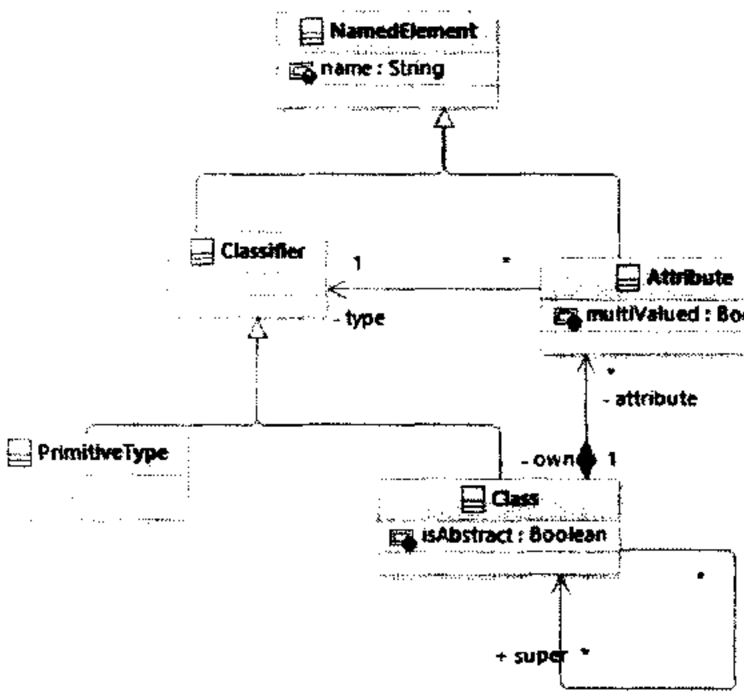


Figure 5.2 Original Class meta-model

The simple UML class diagram meta-model UML comprises classes that must have some name. These classes inherit their name from the class "NamedElement" which is abstract. The main class is class "Class" which is composed of a set of attributes. The attributes are of type "Attribute". The class "DataType" represents primitive types. The class "Datatype" and "Class" inherit from "Classifier" which is used to declare the type of attributes. Figure 5.2 shows part of the original UML meta-model comprising all the mandatory concepts discussed above.

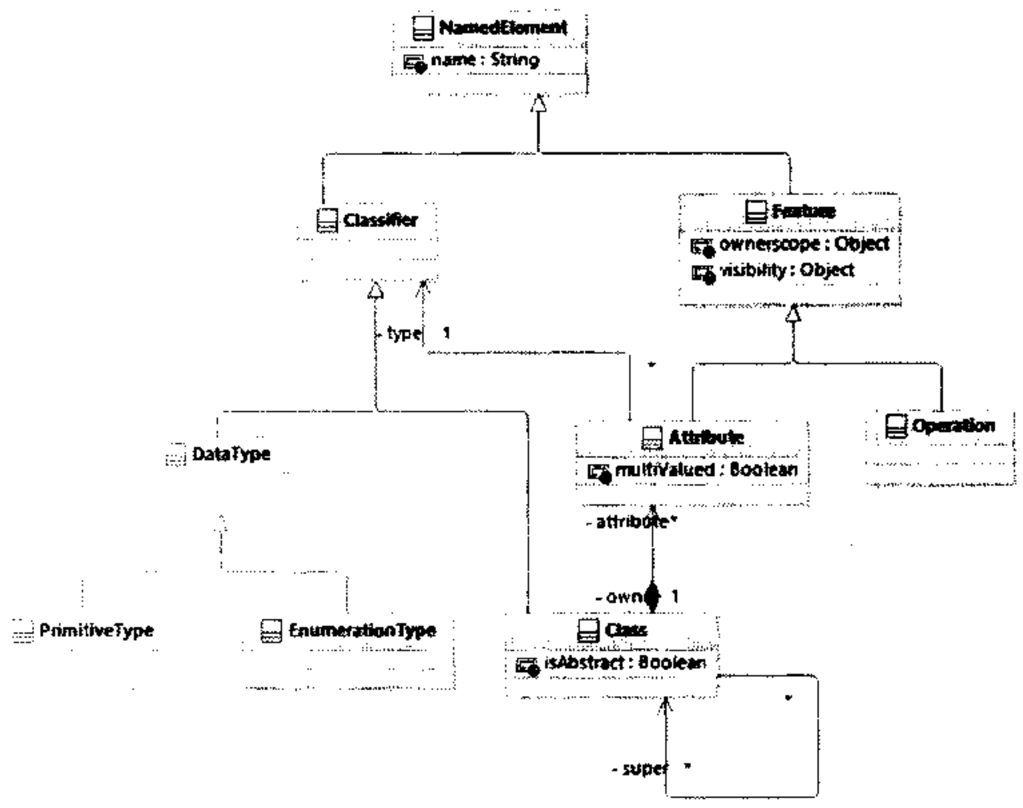
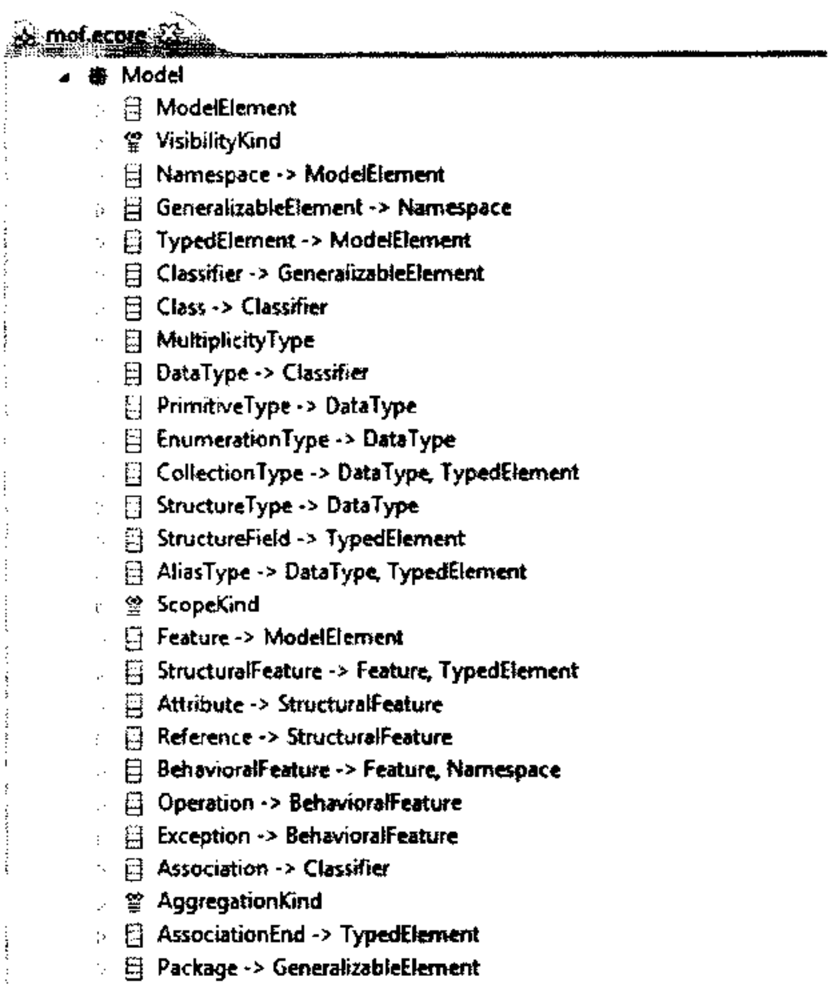


Figure 5.3 Evolved version of meta-model

The evolved Class meta-model consists of the main concept class "Class" which comprised "Feature" either structural or behavioral. As the class "PrimitiveType" models primitive type, here in the evolved version it is renamed as "DataType" and two classes "PrimitiveType" and "EnumerationType" are added as subtypes. Figure 5.3 demonstrates the evolved concepts in new version of sample Class meta-model.

5.2.8 The Meta Meta-models

Our approach relies on the meta meta-model(s). To find the difference between the old and new version of the underlying meta-model, we need to give an old and new version of source meta-model as input to the approach. The meta-model is defined in ecore format supported by eclipse. To make the approach generic both Emf/ecore and MOF meta meta-models are defined. This is because any meta-model either MOF-based or emf-ecore based can be given as input to the approach and thus make the approach generic. Figure 5.4 illustrates MOF meta meta-model representation.


mof.ecore

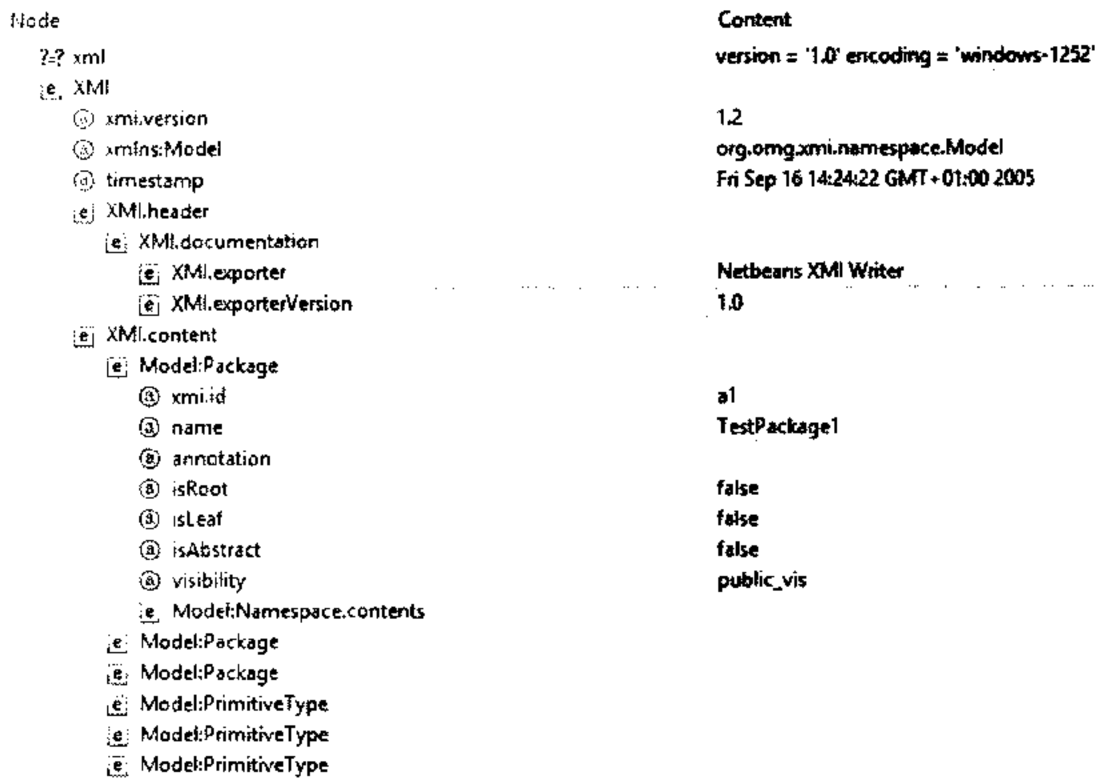
- Model
  - ModelElement
  - VisibilityKind
  - Namespace -> ModelElement
  - GeneralizableElement -> Namespace
  - TypedElement -> ModelElement
  - Classifier -> GeneralizableElement
  - Class -> Classifier
  - MultiplicityType
  - DataType -> Classifier
  - PrimitiveType -> DataType
  - EnumerationType -> DataType
  - CollectionType -> DataType, TypedElement
  - StructureType -> DataType
  - StructureField -> TypedElement
  - AliasType -> DataType, TypedElement
  - ScopeKind
  - Feature -> ModelElement
  - StructuralFeature -> Feature, TypedElement
  - Attribute -> StructuralFeature
  - Reference -> StructuralFeature
  - BehavioralFeature -> Feature, Namespace
  - Operation -> BehavioralFeature
  - Exception -> BehavioralFeature
  - Association -> Classifier
  - AggregationKind
  - AssociationEnd -> TypedElement
  - Package -> GeneralizableElement

Figure 5.4 Sample MOF meta meta-model representation

5.2.9 The Input meta-models

Our approach will take as input the class diagram of the original (old version) and evolved version of the source meta-model. This source meta-model can be either be an EMF/Ecore based or MOF –based model. Our approach is generic enough to take any type of meta-model as input. The input source meta-model would be treated as model that would conform to a certain meta meta-model e.g. the UML meta-model conforms to the MOF Meta meta-model discussed in section 5.2.8. The input source model will be presented in xmi format. Since meta-model is the model itself conforming to a meta meta-model, therefore, for simplification purpose we will consider and refer the meta-model as model and a meta meta-model as meta-model in further discussion.

The basic concepts that our approach requires for mapping are: the class, its attributes and relationships. The input source model will be traversed along with the hierarchy. Below, Figure 5.5 shows the sample format of input source meta-model.



Node	Content
?-? xml	version = '1.0' encoding = 'windows-1252'
XMI	
xmi.version	1.2
xmlns:Model	org.omg.xmi.namespace.Model
timestamp	Fri Sep 16 14:24:22 GMT+01:00 2005
XMI.header	
XMI.documentation	
XMI.exporter	Netbeans XMI Writer
XMI.exporterVersion	1.0
XMI.content	
Model:Package	
xmi.id	a1
name	TestPackage1
annotation	
isRoot	false
isLeaf	false
isAbstract	false
visibility	public_vis
Model:Namespace.contents	
Model:Package	
Model:Package	
Model:PrimitiveType	
Model:PrimitiveType	
Model:PrimitiveType	

Figure 5.5 Sample input model

5.2.10 Input transformations

The other important file that our approach will take as input is the transformations that need to be co-evolved with the evolution of meta-model. Our approach is generic enough that can take transformations written in any dedicated transformation language either declarative or imperative e.g. Kermeta, ATL or QVT. Transformations written in generic programming languages e.g. java is not supported by our approach. The format of the input transformations depends on the type of language the transformations written in. For example, sample QVT transformations are given in Figure 5.6.

```
transformation Book2Publication(in bookModel:BOOKS,out
pubModel:PUB);

main() {
  bookModel.objects()[Book]->map book_to_publication();
  }

mapping Book::book_to_publication () : Publication {
  title := self.title;
  nbPages := self.chapters->nbPages->sum();
}
```

Figure 5.6 Sample QVT transformations

5.3 Definitions

This section defines the Traceability and change meta-models and introduces the transformation and change classification schemes. In section 5.3.1 the concept of relationship based change propagation is introduced. Section 5.3.2 defines and explains the trace meta-model, which would be employed to carry out the relationship based change propagation activity. In section 5.3.3 the concept of difference representation and visualization is introduced and explained. While in section 5.3.4 the change meta-model, which would assist in representation and visualization of change. Section 5.3.5 and 5.3.6 introduce the change classification scheme and transformation classification scheme correspondingly.

5.3.1 Relationship based Change Propagation

For change propagation, we would use relationship based change propagation mechanism. For this purpose, we first need to establish a "Consistency" relationship between meta-model elements and transformations, and then make these relationships explicit; so, that any inconsistency in the "Consistency" relationship introduced by model element change can be identified and make it consistent by propagating model element change to the transformations. Following the MDE principles, we formalized a meta-model based mechanism to establish the "Consistency" relationship and make it explicit for propagating the change and analyzing its impact. This mechanism is based-on trace meta-model. The "Consistency" relationship is captured as model conforming to the Trace meta-model. The trace model is capable to capture each "Consistency" relationship either explicit or implicit and provides support to capture the forward as well as backward traceability. Next section describes briefly the trace meta-model.

5.3.2 Trace meta-model

The problem of co-evolution is intrinsically complex and requires specialized methods and notations to cope up with it, particularly to determine its impact propagation. Currently there is no support to establish and formalize the dependency relationship between meta-model and transformations which the researchers called "consistency" relationship. We formalized this consistency relationship to make it explicit and well-defined so that it becomes easy to find the impact of meta-model change on transformations and identify the usage of meta-model elements in transformations. Since, the impact of meta-model change on transformations not only depends on the type of change; but also depends on the usage of meta-model element by transformations, so, the formalization of the relationship between meta-model and transformation is of crucial importance.

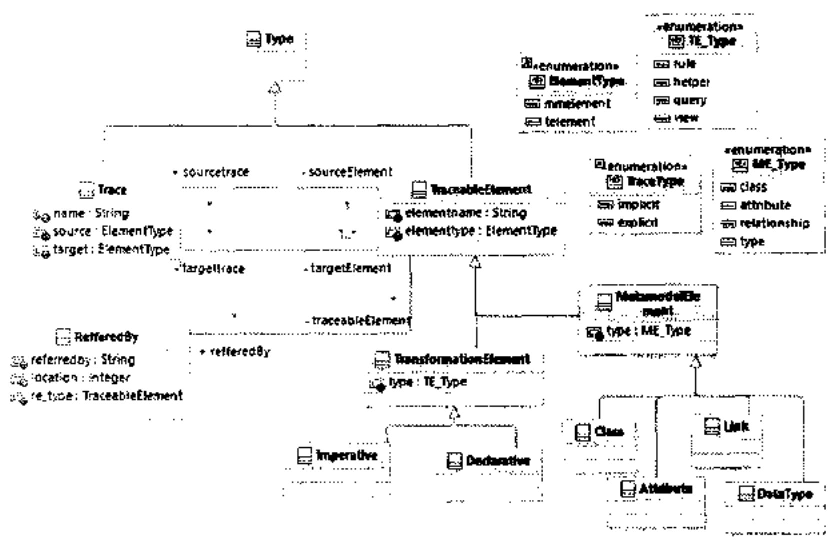


Figure 5.7 Trace meta-model

To formalize the dependency relationship between model elements "E" and transformations "T" we proposed a trace meta-model. The trace meta-model is composed of traces, i.e. "TR" capable to capture the relationship between model element "E" and transformation "T". The trace "TR" would comprise all the required information including the source and target element of the trace. For identification and unification of traces "TR", the trace would have assigned some unique name. In addition, it would capture the type information of model element "E" as well as transformation "T". This would make it easy to map model element changes, i.e., C on traces, i.e., T at the time of impact analysis. It would also assist in searching and locating transformation element T during impact detection. Moreover, it provides support for both forward and backward traceability.

The main concept, that Trace meta-model comprised, is "Trace". The concept "Trace" holds the information about the source element and the target element of the "Trace". Each "Trace" would have some unique name. A "Trace" would compose of mainly only one source element. The "Trace" can have 0 or more target elements because it is possible that some meta-model elements are not employed by the transformations. However, these elements are part of the meta-model because they have mandatory relationship with the elements employed by transformations.

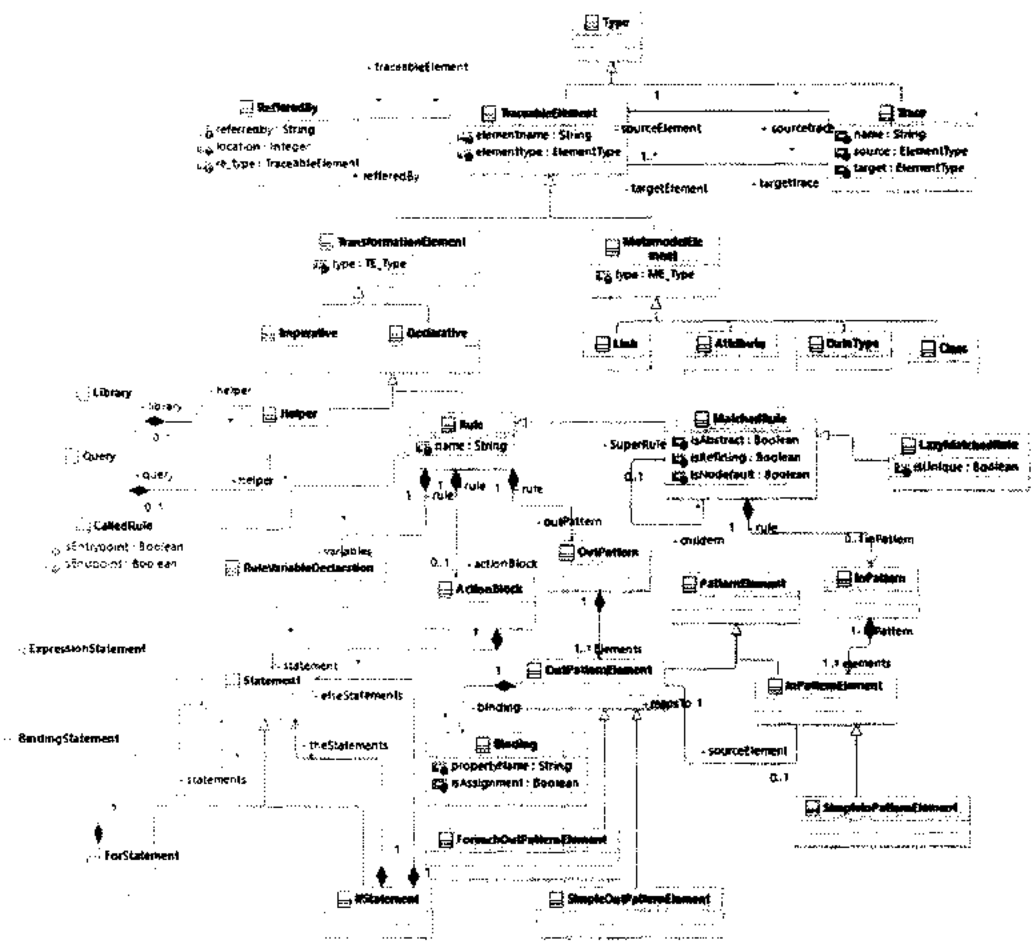


Figure 5.8 Extended Trace meta-model for UML2JAVA ATL Transformations

Another important concept of the trace meta-model is "TraceableElement". The "TraceableElement" are the source element and target element of the "Trace". The "TraceableElement" can either be a "Meta-modelElement" or a "TransformationElement" depending on the type of traceability (Forwar or backward). To store the call/references to the transformations by other transformations, the concept of "RefferedBy" is used. A transformation can be called by "0" or more transformations. The Trace meta-model is capable to capture all the calls. This information is necessary in case of the deletion of meta-model and ultimately for the transformation employing the deleted meta-model element.

A "MetamodelElement" can be a class, an attribute, a type or a relationship. While a transformation element can, be either declarative or imperative depending on the type of transformation language used for model transformations. To make the Trace meta-model generic enough the meta-model is designed in a way that it can be easily extended by appending the transformation language elements. For the running example discussed in section 5.2.1 the trace meta-model is extended with the ATL meta-model elements. The basic Trace meta-model is enriched with the ATL meta-elements to make it capable of capturing traces of ATL transformations. Figure 5.7 shows the trace meta-model, while Figure 5.8 demonstrates it extended version, which is enriched with the ATL language constructs. The Figure 5.9 below shows how the traces are represented as trace model.

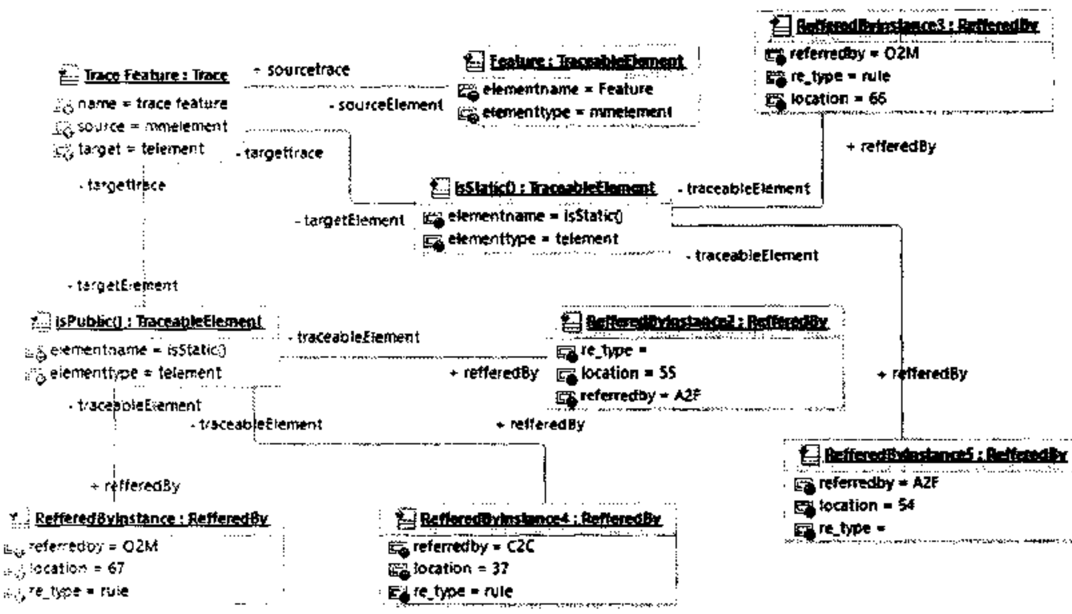


Figure 5.9 Trace of UML meta-class "Feature" and ATL transformations

5.3.3 Change representation and visualization

To visualize and represent difference between two versions of the underlying meta-model, a well-defined mechanism is required. Following the MDE principles, we proposed and formulated a meta-model based mechanism. The difference between two versions of underlying meta-model would be captured as model conforming to the change meta-model. The meta-model changes/difference can be visualize easily as model whenever required.

Difference between meta-model are specified and presented as a model in [40]. However, difference visualization is well supported by the mechanism proposed in [40] but it captures limited information, insufficient to propagate meta-model changes to transformations. In contrast, our difference mechanism is capable to not only capture differences but visualize meta-model differences as well. The aim to define difference mechanism is to provide a means to represent version differences, capture changes appropriately and in a structured way and to propagate change impact in well-defined way. Our difference visualization and representation mechanism not only represent and visualize differences between different versions of meta-models, but, also assists in identifying and classifying meta-model changes based on its impact on transformations.

The change meta-model is capable of capturing each change introduced to the meta-model along with all the required information including change type, element type and affected element. The change model is capable to capture each minor and major meta-model change. This would assist in identifying meta-model changes impact on transformations together with trace model. In addition, it provides a means to identify co-changes that need to introduce to the transformations to reflect meta-model changes. Next section describes the change meta-model briefly.

5.3.4 Change meta-model

To identify, which types of changes are introduced to the meta-model, we proposed a change meta-model. The change would be presented as model conforming to the change meta-model.

Either each change model would capture a single change or multiple changes at once, depending on the type of change introduced to the meta-model.

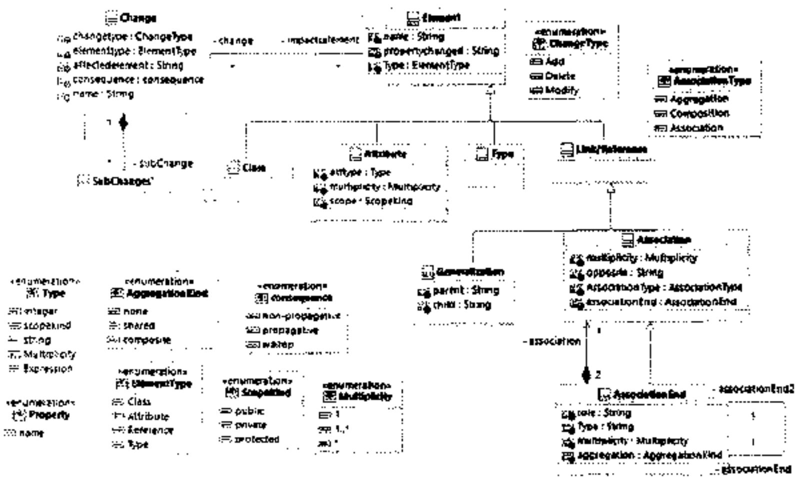


Figure 5.10 Change meta-model

A change meta-model is mainly composed of the two concepts "Change" and "ImpactedElement". The meta-model elements can be impacted by the "Change" introduced to that "Element". The important information about the change, i.e., change's type, consequence and name (for unique identification) would be captured and stored. A "Change" can be additive, subtractive or up-dative. The Enumeration "ChangeType" is added to represent the type of change. The "Consequence" of "Change" is presented using an enumeration "Consequence" having three literals i.e. non-propagative, propagative and wallop. These literal illustrate the transformation classification categories proposed in this research.

A "change" is either composed of 0 or more "Subchange(s)". The meta-class "ImpactedElement" captures the name of the impacted element its type and the impacted part. If any of the property of the meta-model elements is changed, for example, "name", the meta-class "PropertyChange" captures the "name" and "type" of the property, its "old value" and "new value". An "Attribute" change, might be a change in its "type", "multiplicity" or "scope". The change meta-model is capable to capture all these changes.

The meta-model "Element" can be a "Type", an "Attribute", a "Class", or a "Reference". The "reference" includes "Association(s)" and "Generalization(s)". Since the child, class contains information about its parent and parent class does not have any information about its children. So, for "Generalization" relationship information about the change in parent is mandatory to capture as it would have an impact on transformations. The meta-class "Association" in change meta-model captures the information about the change in association. The essential information about any type of change in association is captured by this meta-class including the change in "multiplicity", "opposite", "associationtype" and "associationend". The meta-class "AssociationEnd" has attributes role, type, multiplicity and aggregation. These attributes represent the properties of the association end that can be change and it might have an impact on transformations. Figure 5.10 above illustrates the change meta-model. The Figure 5.11 below illustrates how change "pull meta-property" is captured by the change model. Next section, describes the change classification scheme.

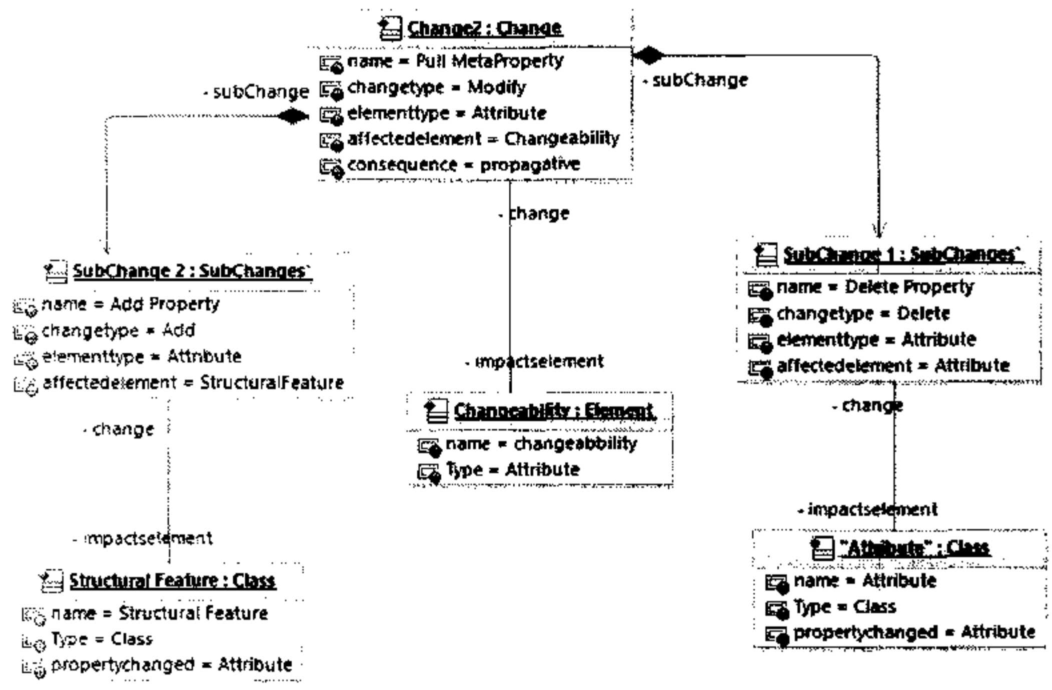


Figure 5.11 change model of the change "pull meta-property"

5.3.5 Change classification scheme

Previously, meta-model changes are classified with respect to its impact on models [48]. Since it is cleared, that meta-model evolution affects its related artifacts in different way depending on the type of relationship that it hold with its related artifacts. Furthermore, the impact of meta-model change on transformations not only depends on type of change but also on the usage of elements in transformations. Therefore, the change classification formulated for model co-evolution cannot be employed for transformation co-evolution. The change classification proposed in [38] for transformation co-evolution classifies meta-model changes from adaptation automation point of view. This classification scheme does not reflect the impact of changes on transformations. Moreover, change classification scheme should be generic enough, so that it can be applicable in almost all co-evolution scenarios. Keeping the above factors in mind, a generic change classification scheme is proposed here:

- **Non-propagative changes:** Changes that do not require propagation to the transformations, to reflect meta-model changes in transformations, will be classified as non-propagative changes.
- **Propagative changes:** Changes that require propagation to transformations, to reflect meta-model changes in transformations, will be classified as propagative changes.
- **Wallop changes**: Changes, which hit multiple transformations simultaneously and demand propagation at multiple places, to reflect meta-model changes in transformations, would be wallop changes.

Our approach will employ this change classification scheme to classify meta-model changes. The change classification together with transformations classification and trace links would assist in formulating co-evolution strategy.

5.3.6 Transformations classification scheme

Transformation classification is essential for determining the required steps to co-evolve transformations. This would help in devising the necessary transformations migration strategy. Transformations can be classified as:

- Use as it is, i.e., required no action
- Add transformations
- Delete transformations
- Modify transformations

5.4 Approach's Overview

We proposed a systematic and automatable approach for transformations co-evolution with meta-model evolution. The approach is capable to detect, visualize and represent model element changes, classify these changes according to its impact on transformations, and establish a dependency relationship between model elements and transformations, classify transformations according to the required co-changes and finally co-evolving transformations. The proposed approach is automatable. The principle steps of the proposed approach are:

- Mapping model elements of the source meta-model on transformations by establishing an explicit dependency relationship between model elements and transformations.
- Identify model element changes, capture them as change model, and classify model elements changes according to the proposed classification scheme.
- Detect the impact of model element changes on transformations.
- Classify and highlight the impacted transformations and finally
- Co-evolve the impacted transformations while copying the remaining transformations, which are not impacted by the model element changes.

The steps of the proposed approach are explained via applying on the running example described in section 5.2.1. The change model captured for the change "add meta-class" and the trace "Trace " is kept under consideration. This is because the property "Changeability" is pulled from meta-concept class "Attribute" to the meta-class "StructuralFeature" and the trace pull meta-property trace is tracing this meta-element "Changeability", which is changed and the change is captured and stored in change model. This would assist in the demonstration and validation of the proposed approach. Figure 5.12 demonstrates how the proposed approach works.

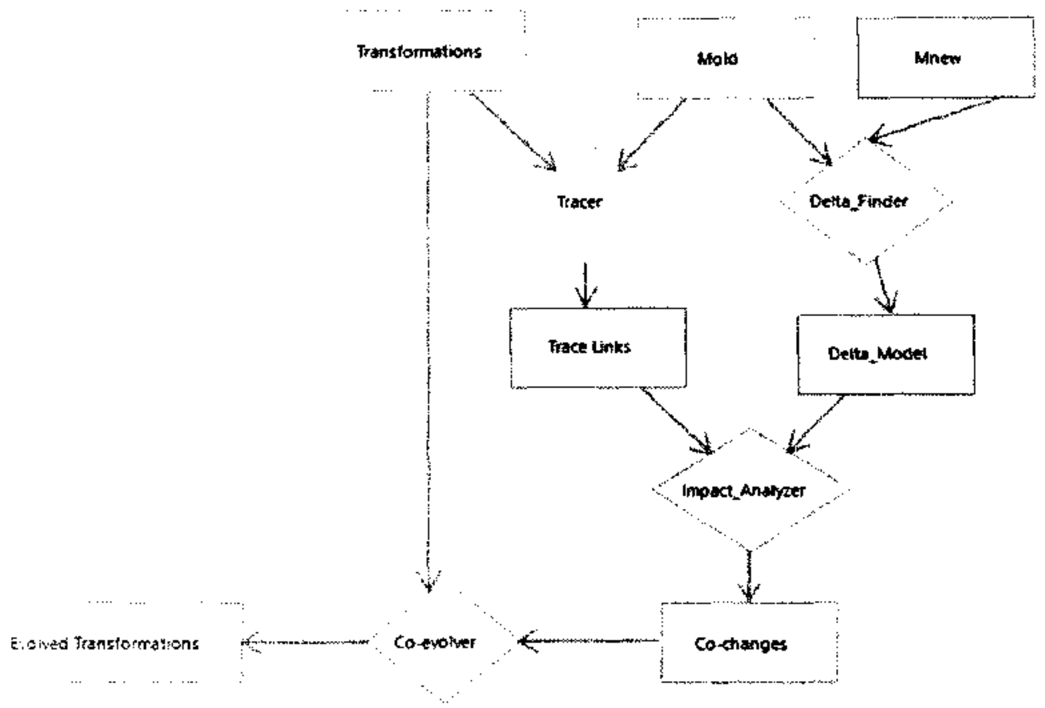


Figure 5.12 Flow Chart of the approach

5.4.1 Mapping Elements

In first step, a relationship between model elements and transformations is formalized through establishing trace links that would conform to "Trace" meta-model (discussed in section 5.3.2).

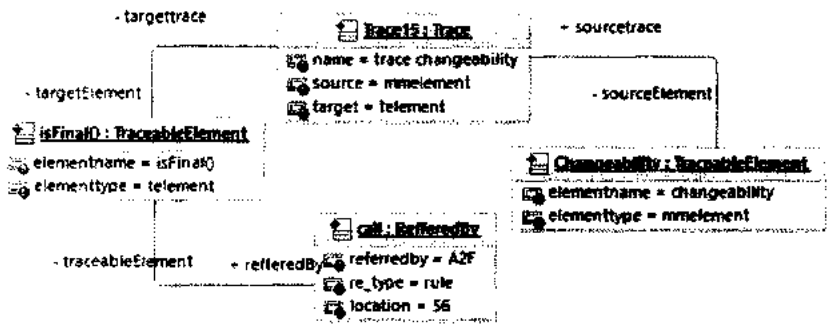


Figure 5.13 trace model for meta-attribute "changeability"

Consider the example given above in section 5.2.6. The meta-attribute "changeability" in M_{old} is selected and the transformations are searched for any mapping that is using it. The transformations, i.e., "isFinal()" is identified. In the next step, it would search for calls made to the "isFinal(). Once all the transformations and calls are collected, the trace is captured and stored. Remember, if there were no call made to the transformations then only transformations element would be the part of trace, not the calls. All the traces are captured this way. These trace links would assist in identifying meta-model elements usage in transformations and would be use during impact analysis. As discussed before, the meta-model change impact on transformations not only depends on the type of change but also on the usage of meta-model elements in transformations. Thus capturing trace links is essential ingredient to identify and analyze the impact of meta-model changes on transformations. Figure 5.13 illustrates the trace captured for the meta-attribute "changeability" of UML meta-model.

5.4.2 Detect and classify Changes

When meta-model evolves, some changes are introduced to the new version of meta-model. These changes can be additive, subtractive or up-dative. To identify the type of changes introduces to the meta-model, meta-model changes are captured as change model conforming to the change meta-model (dicussed in section 5.3.4). The change model represents the meta-model changes. For instance, meta-attribute "changeability" is pulled to the super class "StructuralFeature".

Figure 5.14 illustrates the change model representing the change "pull meta-property".

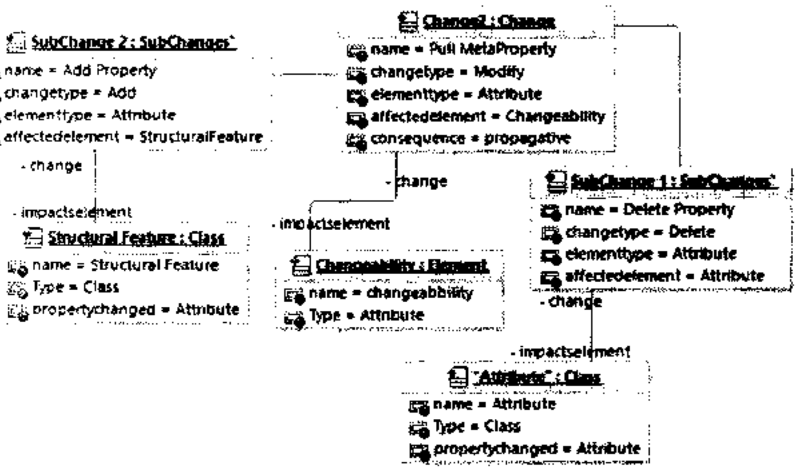


Figure 5.14 change model for change "pull meta-property"

All the meta-model changes are captured this way. Once the changes are identified, these changes are then classified according to the proposed classification scheme (Discussed in section 5.3.5). The pull meta-property change is classified as propagative change as it requires modification in the context of helper "isFinal()".

5.4.3 Impact Detection and Analysis

In this step, impact of changes is identified by mapping changes on trace links. The impact analyzer takes the change model and traces as input and returns the transformations that are impacted by the meta-model changes. The impact analyzer classifies the transformation according to the co-changes required to make to the transformations and transformations are then treated accordingly.

The impact analyzer takes the trace links and change model as input. It selects the elements that are changed and search for traces exist for that specific meta-element. If traces exist, it navigates through it and tries to find the impacted transformations. If there exist some transformation for that considered model element, the analyzer tries to find, which part of that transformation element is impacted. It then classified transformations based on the type of co-change required to make to the transformations. The analyzer is capable of copying the transformations that are not impacted by model element changes.

For instance, the change pull meta-property is selected. It is cleared from the change, that the element affected by the change is the "Changeability" itself, "Attribute" and

"StructuralFeature". The affected element names, i.e., "Changeability and its type, i.e., attribute is extracted from the change model and the corresponding transformation, which is employing this model element, is searched for the trace that is impacted by this change. The trace "changeability" is found and by its target links, the impacted transformations are extracted. These transformations are returned to the user. The users then identified the co-changes required to propagate the meta-model change into transformations. Figure 5.15 shows the impact of renaming class feature on transformations

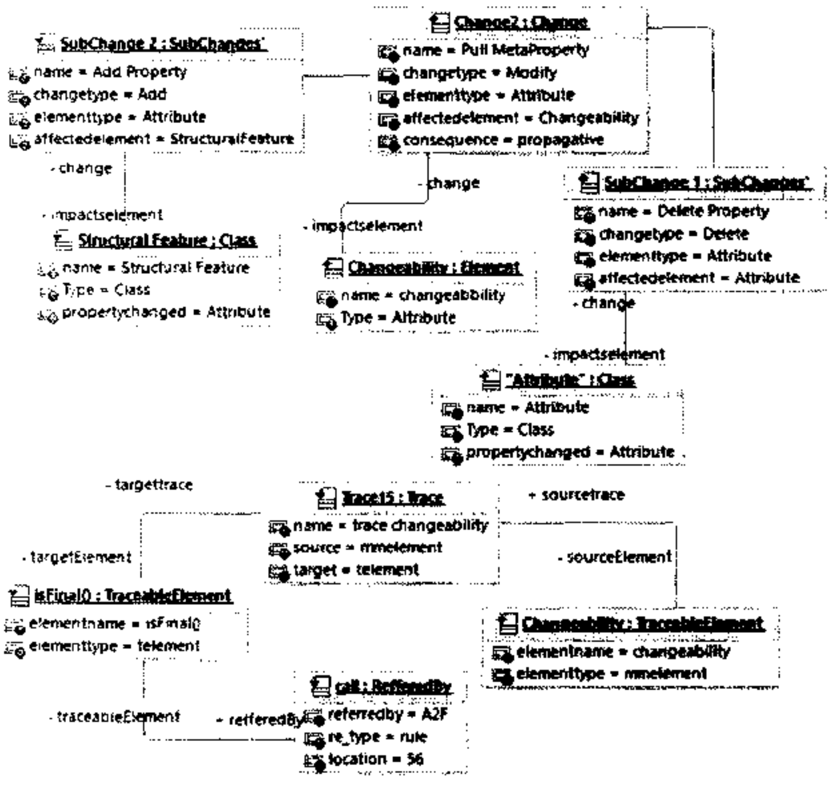


Figure 5.15 Detecting the impact

5.4.4 Transformations co-evolution

After, the impacted transformations are identified. The next step, is to co-evolve impacted transformations. An algorithm is designed for this purpose, which takes the impacted transformations and asks for the desired changes along with the location of change. For instance, the rename class "Feature" impacted two transformations i.e. "isPublic()" and "isStatic()". The particular part that needs to be changed, in both transformations, is the context of these transformations. The user would specify the context along with the co-change, i.e., replace "UML!Feature" by "UMLStructuralFeature". The algorithm changes the context and returns the co-evolved transformations. Below are the old and evolved transformations:

Original Transformations:

```
Helper context UML!Attribute def: isFinal():Boolean=
Self.changeability=#ck_frozen;
```

Evolved Transformations:

```
Helper context UML!StructuralFeature def: isFinal():Boolean=
Self.changeability=#ck_frozen;
```

5.5 Co-Evo: An Algorithm

An algorithm for transformation co-evolution abbrivated as "Co-Evo", is designed to support the steps defined and explained above. The "Co-Evo" algorithm will take the old and new version of meta-model and transformations as input. First, consistency relationship between the elements of old version of meta-model and transformations will be made explicit by establishing traces between meta-model elements and transformations. Traces are established for each meta-model element, captured and stored; as model conforming to the trace meta-model.

Next, it will compare the old and new version of meta-model to find, what changes are introduced in the new version. Each change is captured as model conforming to the change meta-model. The comparison starts from the root element, i.e., the root class of the old version of meta-model and compare it with the elements in the new version to check either it is same or not. Each property of the selected meta-element is compared one by one to the element in the old version. If the properties are same, it is ignored else the difference is captured as change model and stored. All the differences are captured and stored this way.

Once, traces are established and changes are identified the next step is to identify the impacted transformations. For this purpose, these the algorithm select a change from the change model extract change name, type and the impacted element name and type and try to search the repository for the trace with the same model element name and type. Once, it gets the element in the trace with the same name and type, it navigates through the trace and extracts the impacted part of the transformations. The next step is to classify the impacted transformations based on the type of change introduced to the meta-model. After extracting and classifying transformations, the classified impacted transformations are returned by the algorithm along with the co-change suggestion if poosible to predict. The user then analyze if the co-change is correct or some other actions need to take to modify transformations. In the last step, transformations would be tested by running on the instances of the new version of meta-model. Below is the proposed algorithm:

```
Algorithm   Co-Evo($mm_1$, T, $mm_2$)
  Input:    $mm_1$: Old version of meta-model, T: transformations, $mm_2$: new version of
            meta-model
  Output:   impacted transformations, co-change
   1. Begin
   2. for each e in $mm_1$
   3. establish trace tr=e→t where t∈T
   4. save tr
   5. for each $e_i$ in $mm_2$ where $e_i$ is the element selected from $mm_2$
        a. select $e_i$.name && $e_i$.type
             i. if $e_i$.name==$e_j$.name where $e_j$ is the element of $mm_1$
                  1. if $e_j$.type==$e_j$.type
                  2. do nothing
```

```
                    3. else capture ΔC where c is the change in the selected element
                    4. end if
                ii. endif
6. increment i
7. end for
8. for each ΔC
9. select e.name && e.type && c.type from ΔC where c.type is the change type
    a. if e.name==tr.ename and e.type==tr.etype
    b. navigate tr to the target trace
    c. get t∈T
    d. for each t in T
    e. classify t as t.type=c.type
    f. retun t.type
    g. end for
10. increment j
    a. if t.type==add transformations
    b. generate transformation template
    c. elseif t.type==delete transformations
    d. delete tr
    e. elseif t.type==update transformations
    f. predict co-change
11. return classified and updated transformations
12. end for
13. end
```

5.6 Chapter's Summary

The proposed approach for managing co-evolution of meta-model and transformation has been presented in this chapter. The way, meta-model change impact models, is different; from the way it affects other related artifacts. In addition, the meta-model change impact not only depends on the type of change but also on the usage of meta-model elements in transformations. Therefore, the problem of co-evolution of meta-model and transformations must be handled differently than the problem of co-evolution of meta-model and models. The proposed approach solves these issues by devising a relationship-based mechanism along with the change meta-model. Finally, an algorithm is designed to carry out the entire process of co-evolution of meta-model and transformations automatically with minimal user intervention.

The relationship-based mechanism consists of traceability meta-model, which makes the consistency relationship explicit by establishing traces between meta-model elements and transformations, represent, and store these traces as model. The trace model conforms to the trace meta-model. These traces assist in impact analysis and accurate change propagation. The trace meta-model is defined as EMF/Ecore model in Eclipse and its instances are stored and manipulated as xmi files.

For detecting, representing and visualizing changes introduced in the new version of meta-model, a change meta-model is proposed. Changes introduced to the meta-model are captured as change model. This mechanism not only captures the changes but also visualizes and represents these changes as model whenever required. This assists in getting a better understanding of which type of changes are introduced to the meta-model and propagating these change changes accurately to transformations.

Meta-model evolution affects its related artifacts in a different way depending on the type of relationship it holds with its related artifacts. The impact of meta-model change on transformation not only depends on the type of change but also on the usage of elements in transformations. Therefore, change classification is designed specifically for transformations co-evolution as the classification scheme proposed for co-evolution of models does not suit well to the transformations co-evolution problem. According to the proposed classification scheme, Changes can be classified as non-propagative, propagative and wallop change.

Transformations classification scheme is proposed, to classify transformations. This would assists in taking action in the co-evolution/migration strategy selection process. Transformation classification scheme is composed of use as it is, add transformation, delete transformation and modify transformation. Next chapter applies the approach on three different examples.

Chapter. 6 Considered Cases

6.1 Introduction

This chapter presents the underlying cases and considered modifications, which we would use to demonstrate the applicability of our approach in practice. We used to model the co-evolution of two existing meta-models. In this chapter, we would present three major cases that we considered to evaluate and validate our approach. Because of the availability and the application of wide range of transformation languages in industry, we would apply the approach on three different transformation cases having the transformations written in three different transformation languages. Originally, the transformations were defined using ATL language. We have re-written the transformations using QVT for specifying PetriNet2PNML transformation example. While for Class2Relational transformation example, we have re-written the transformations using Kermeta. This would assist in the validation of the approach's generality. We have selected three Cases from ATL ZOO, i.e., UML2Java, UML2MOF and Petri Net transformation example.

In section 6.2, ATL transformation is discussed in detail. Section 6.3 explains the Kermeta transformation example. Section 6.4 outlines QVT transformation example. Section 6.5 presents the changes introduced to the meta-model.

6.2 UML2JAVA: an ATL transformations Example

This section is screened off into four subsections. The first subsection provides an overview of the UML 2JAVA transformations specifications. The second subsection briefly describes the functionality of the Transformations. Transformation defined using ATL, are depicted in third subsection. The original and evolved version of the input source meta-model is described in subsection fourth and fifth. In subsection sixth, the target meta-model of ATL transformations is discussed.

6.2.1 Transformations

The main function of the ATL transformations is to transform a UML model to a simplified Java models. The Java model carries the information for the creation of java classes especially what concerns the structure of these classes, namely the package, reference, the attributes and the methods.

6.2.2 Rule Specifications

To write transformations for transforming UML model to a java model first all the essential rules, conditions and circumstances are specified. Below, the specifications of the rules to transform a UML model into Java model are explained:

- A java package instance has to be created for each UML package instance.
    - Their name must have to keep up a correspondence. However, java package contains the complete path information as compare to UML packages, which hold plain names. The path separation is a point "."
- A java class instance has to be created for UML class instance.
    - Their names have to equate.
    - Package reference must have to keep up correspondence.
    - The modifiers must have to correlate.
- A java primitive type instance must have to be created for each UML data type instance.

    - Their names must match.
    - The package reference has to correlate.
- A java field instance has to be produced for each UML Attribute instance.
    - Their name must equate.
    - Their type must correlate.
    - Their modifiers must match.
    - The classes have to keep up correspondence.
- A java method instance has to be generated for each UML Operation instance.
    - Their name must equate.
    - Their type must correlate.
    - Their modifiers must match.
    - The classes have to keep up correspondence.

6.2.3 The ATL Code

The ATL code that transforms a UML model to a simplified java model comprised various functions and rules. It is of crucial importance to declare the "getExtendedName" function as it explores recursively to hook up a complete path name. The remaining functions like isPublic(), isStatic() and isFinla() set the accessibility, scope and finalizing the implementation for the java classes methods and attributes.

Regarding the rules, there are crucial remarks related to the rule O2M, which generate java method instance for UML operation instance. This rule demonstrates how to access sets employing OCL expressions. For simplification of implementation, the return type of the java method is the first parameter of an UML operation. Concerning the remaining rules, rule P2P generates java package from UML package calling the getExtendedname function, which discovers repeatedly to concatenate a full path name for java package. The rule C2C generates java class instance for each UML class instance importing namespace, scope and name from UML class instance. Rule D2P produces java primitive type for UML Data Type. Some trivial details like modifiers are not yet completely implemented. Below is the ATL code to transform UML model to a java model:

```
module UML2JAVA;
create OUT : JAVA from IN : UML;

helper context UML!Feature 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())}

rule C2C {from e : UML!Class
        to out : JAVA!JavaClass (name <- e.name,isAbstract <- e.isAbstract,isPublic <-
e.isPublic(),package <- e.namespace)}

rule D2P {from e : UML!DataType
        to out : JAVA!PrimitiveType (name <- e.name,package <- e.namespace)}

rule A2F {from e : UML!Attribute
        to out : JAVA!Field (name <- e.name,isStatic <- e.isStatic(),isPublic <- e.isPublic(),
                isFinal <- e.isFinal(),owner <- e.owner,type <- e.type)}

rule O2M {from e : UML!Operation
        to out : JAVA!Method (name <- e.name,isStatic <- e.isStatic(),isPublic <-
e.isPublic(),owner <- e.owner,
                type <- e.parameter->select(x|x.kind=#pdk_return)->asSequence()-
>first().type,
                parameters <- e.parameter->select(x|x.kind<>#pdk_return)->asSequence())}

rule P2F {from e : UML!Parameter (e.kind <> #pdk_return)
        to out : JAVA!FeatureParameter (name <- e.name,type <- e.type)}
```

6.2.4 Source Meta-model: Simplified UML model

The simple UML meta-model composes of classes having name which they inherits from the abstract class "ModelElement". The main class is class "Class" comprised a set of attributes of type "Attribute". The class "DataType" models primitive types. The class "Datatype" and "Class" inherit from "Classifier" which serves to declare the type of attributes. Figure 6.1 shows part of the original UML meta-model comprising all the mandatory concepts, we have discussed before.

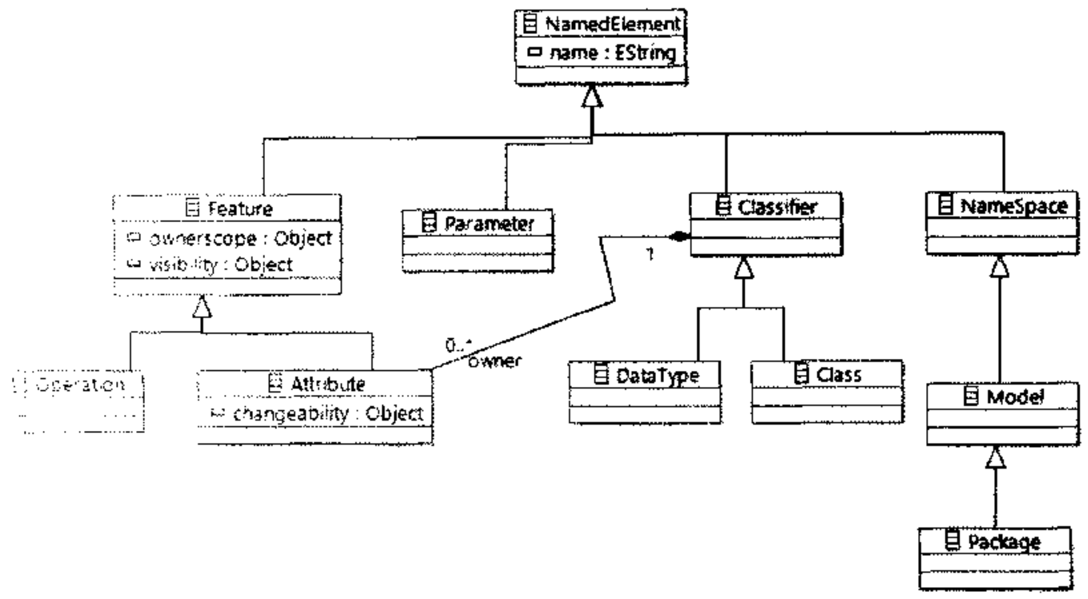


Figure 6.1 Part of UML meta-model

6.2.5 Evolved Class Meta-model

The evolved version of class meta-model consists of an additional meta-class i.e. "StructuralFeature", which classifies the Attribute a structural feature of class. The association end "owner" is moved from meta-class "Attribute" to newly added meta-class "StructuralFeature". The attribute "changeablity" is pulled up to the meta-class "Structural Feature". A new attribute "isAbstract" is added to the classifier of type "Boolean" that represents, if the class is abstract or not. The meta-class "DataType" is renamed as "PrimitiveType". The generalization hierarchy of meta-class "Attribute" is moved from meta-class "Feature" to "StructuralFeature".

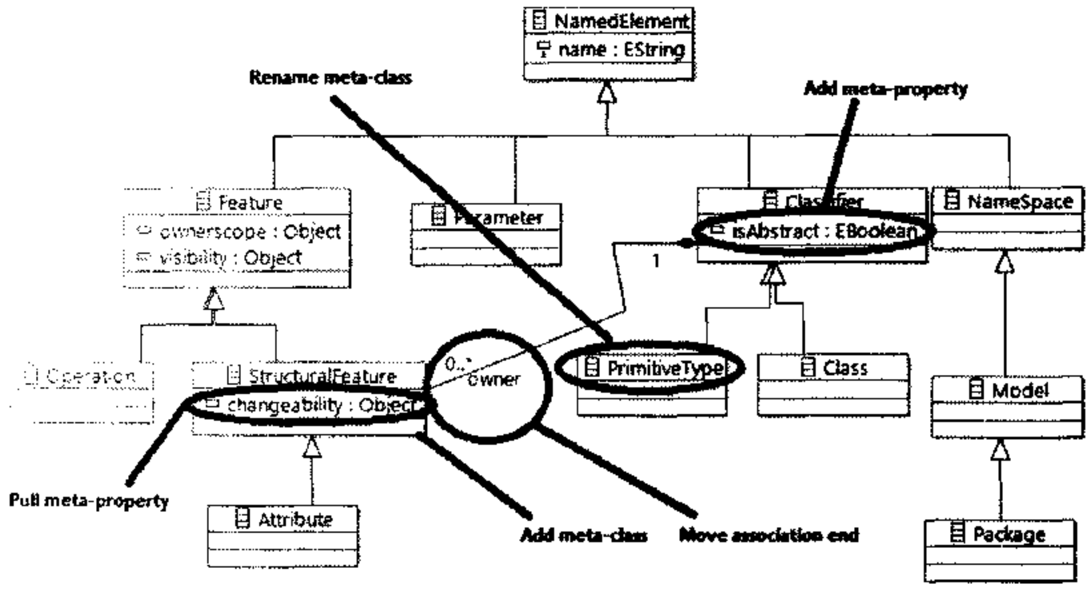


Figure 6.2 Evolved version of UML Class meta-model

6.2.6 Target Meta-model: Simplified Java model

A simplified target java meta-model principally comprises Java Elements which all have some unique name. A Java Class is possibly composed of some methods and fields and is a part of some package. Classes, fields and methods are some part of modifiers and therefore signify them as public, static or final. Java Classes and Methods have "isAbstract" attribute to indicate whether they are abstract or not. The types in java meta-model are classes and primitive types. A java field must be of some type. Java methods have parameters of some types and they must have some return types. Below is the target java meta-model that illustrates main concepts of the Java Language.

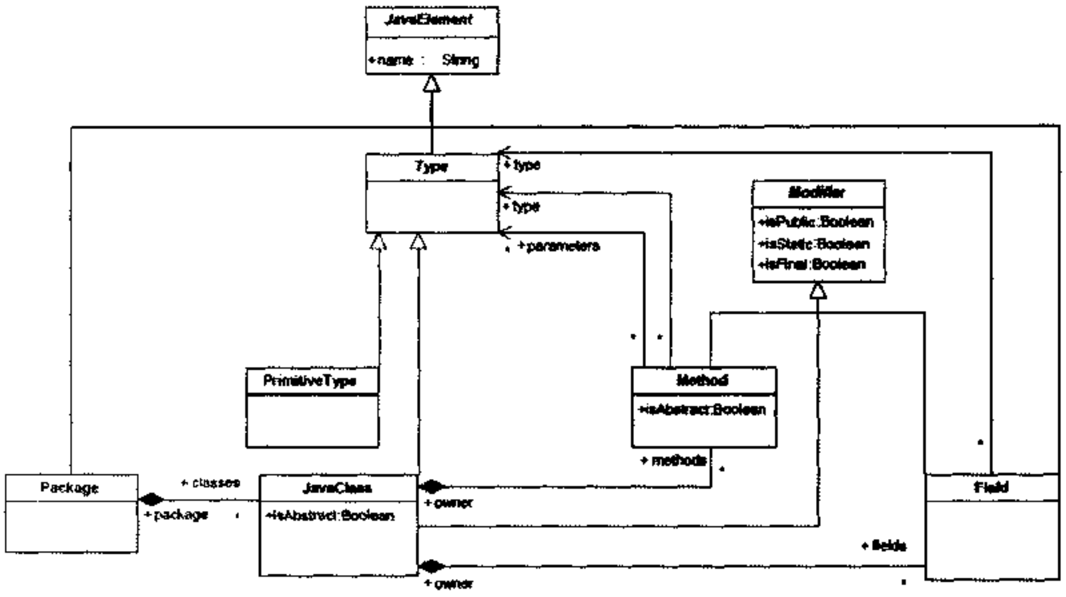


Figure 6.3 A simplified Java Meta-model

6.3 PetriNet2PNML: Qvt Core Transformation Example

The PetriNet transformation example is chose from ATL Zoo. The transformations are originally specified using ATL language. But for demonstrating the applicability of our approach, the transformations are redesigned and specified using Qvt transformation language. This section is screened off into four subsections. The first subsection provides an overview of the PetriNet2PNML transformations specifications. The second subsection briefly describes the functionality of the Transformations. Transformation defined using kermeta are depicted in third subsection. The input source meta-model of PetriNet is described in subsection fourth. In subsection fifth, the target meta-model of Kermeta Transformations is discussed.

6.3.1 Transformations

The main function of the PetriNet to PNML transformations is to transform a PetriNet models conforming to the PetriNet meta-model into a Petri net Markup language model models conforming to the Petri Net Mark-up language meta-model. The PNML meta-model comprises the mandatory information about PNML document.

6.3.2 Rule Specifications

To write transformations for transforming PetriNet models into PNML models first all the essential rules, conditions and circumstances are specified. Below, the specifications of the rules to transform a PetriNet model into PNML model are explained:

- PNMLDocument instance of PNML has to be created for every PertiNet Instance of PetriNet model.
    - Their name must have to correspond.
    - Since the PNML document name is composed of a PNML Label its value must be initialized by PetriNet name.
    - The PNML Document set of contents must have to correspond to the union of the PetriNetElement and Arcs.
- For every instance of the Net in PetriNet model, a corresponding NetElement instance of PNML model has to be created.
    - Their name of PNML Net must be copied to create a PNML Label which value is initialized by the Net Instance name of The PetriNet model.
    - Their location must have to correspond.
    - Their type must have to correspond.
- For each Place instance in PetriNet model, a corresponding Place instance of PNML model has to be created.
    - Their name of PNML Place must be copied to create a PNML Label which value is initialized by the Place Instance name of The PetriNet model.
    - Their location must have to correspond.
    - The PNML Place Id must be created by copying the Name of PetriNet Place Element instance.
- For each instance of Transition in PetriNet model, a corresponding Transition instance of PNML model has to be created.
    - Their name of PNML Transition must be copied to create a PNML Label which value is initialized by the Transition Intance name of The PetriNet model.
- For each Arc instance in PetriNet model, a corresponding PNML Arc instance has to be created.
    - Their name of PNML Arc must be copied to create a PNML Label which value is initialized by the Arc Instance name of The PetriNet model.
    - The source and target references of the PNML arc must correspond to the "from and to references" of the Arc element instance in the PetriNet model.

6.3.3 QVT Code

The Qvt code that generates a PNML document from a simple Petrinet model is composed of few mappings. The Qvt transformation PetriNet_to_PNML is first defined that generates PNMLDocument. The mapping Net() generates a NetElement instance of PNML model corresponding to the net instance of PetriNet model. It does not only generate the PNML document but the net elements too on which the PNML document is composed of. Its set of contents corresponds to the union of PetriNet elements and arcs. The mapping place generates a PNML Place elment from a PetriNet Place. It creates a PNML name comprised of a PNML Label. The value of the label is initialized by the name of Place element of PetriNet. The mapping Transition produces a PNML Transtion correlating to the PetriNet Transition. The name of the PNML Transition is copied from the input PetriNet Transition. It is composed of PNML name comprised of PNML Label. The PetriNet Transition initializes the label's value. The mapping Arc produces PNML Arc correlates to the Arc element of PetriNet model. Its name, source and target referecnes are copied from the input PetriNet arc and correspond to the "to" and "from" references respectively.

```
Transformation Petrinet_to_PNML(in netmodel:PetriNet, out pnml:PNML);
main()
{
Netmodel.object()[PetriNet]→map Petrinet_to_PNML();
}
mapping Net::net():Net
{
{
xlmns:=self.uri;
Location:=self.location.name;
Nets:=net;
}
Uri:=PNML.uri (value:='http://www.informatik.hu-berlin.de/top/pnml/ptNetb');
Net:=PNML!NetElement
{
Name:=name;
Location:=self.location;
Id:=self.location;
Type:-type_uri;
Contents:=self.places.union(self.transition)
}
Name:=PNML!Name(label:=label);
Label:=PNML!Label( text:=self.name);
Type_uri:=PNML!URI(value:=' http://www.informatik.hu-berlin.de/top/pnml/ptNetb')
}
 mapping Place::place2place():Place
{
{
Name:=self.name;
Id:=self.name;
Location:=self.location
Source:=self."src";
}
Name:=PNML!name(label:=label);
```

```
Label:=PNML!Label( text:=self.name);
}
mapping Transition::transition():Transition
{
{
Name:=name;
Id:=self.name;
Location:=self.location;
Source:=self."src";
Target:=self."des";
}
Name:=PNML!Name(label:=label);
Label:=PNML!Label( text:=self.name);
}
 mapping createarcsrc(
```

6.3.4 Source Meta-model: PetriNet model

The petri nets also called as place/transition net or P/T net was first defined by call adam petri. They extend the state machines with the notion of concurrency. The petri net is the graphical and mathematical representation of distributed systems.

A simplified PetriNet model in the selected example mainly comprises places and transitions, which are enclosed in the net element. The class "Net" is the root element of PetriNet mode,l which represents the PetriNet. It mainly consists of places and transitions.

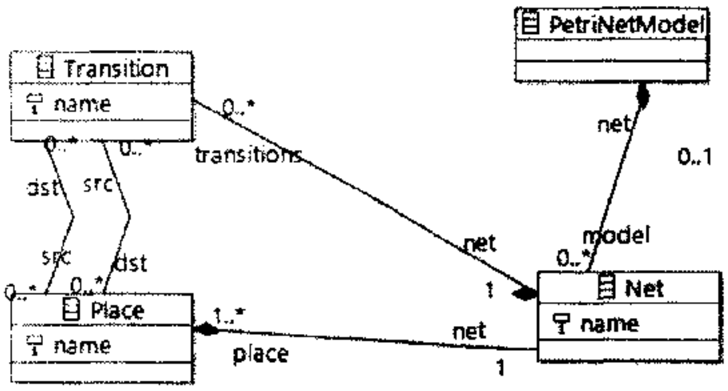


Figure 6.4 A simplified PetriNet model

6.3.5 Evolved PetriNet Meta-model

The Evolved version of PetriNet meta-model consists of additional meta-classes that is "Arc", "PlaceToTransition" and "TransitionToPlace". The references "places" and "transition" have been merged into the new elements. Furthermore, the class "Net" is renamed as "PetriNet".

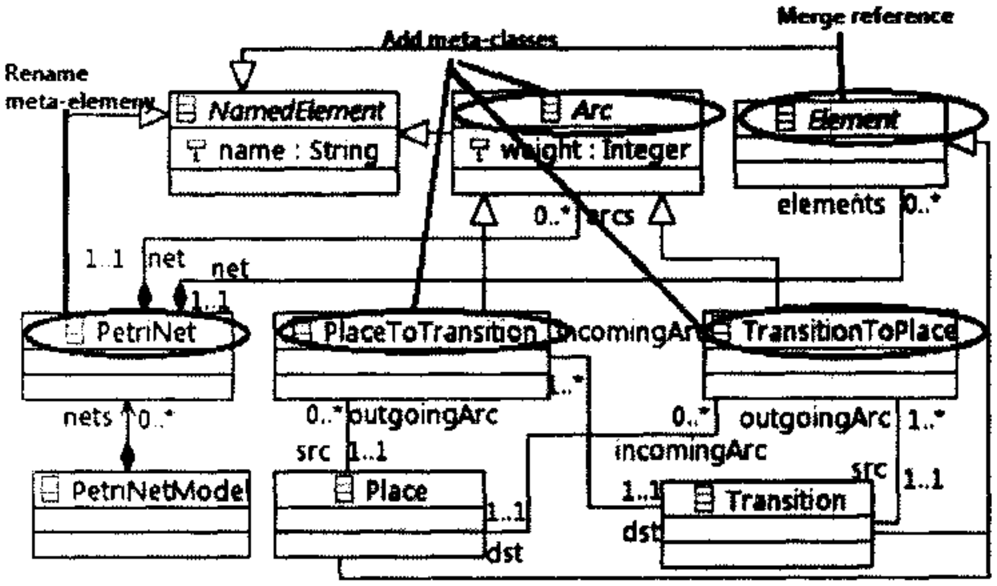


Figure 6.5 Evolved version of PetriNet Meta-model

6.3.6 Target Meta-mode: PNML model

The target PNML model composes of the root element PNMLDocument class. This class consists of Petrinets defined via NetElement instances. A PetriNet is comprised of NetContent elements, which are differentiated into arc, place and transition. Net elements and net contents may have a name which is a labled elemnt comprised of Labels.

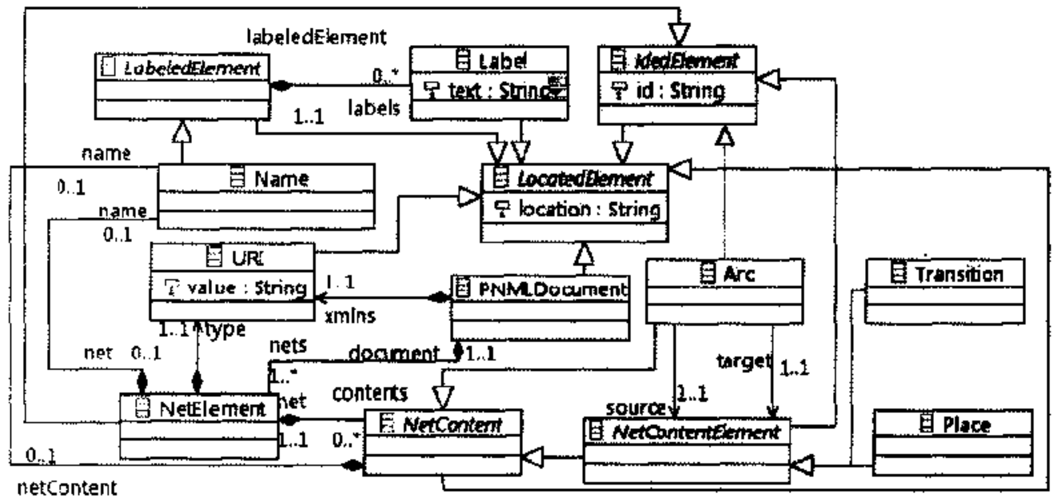


Figure 6.6 Target PNML model

6.4 Class2Relational: kermeta Transformation Example:

This section is screened off into four subsections. The subsection 6.4.1 provides an overview of the Class2Relational transformation example. The subsection 6.4.2 briefly describes the functionality of the Transformations. Transformation defined using kermeta are depicted in subsection 6.4.3. The input source meta-model of Class is described in subsection 6.4.4. In subsection 6.4.5, the evolved Class meta-model is defined and in subsection 6.4.6, the target meta-model i.e. relational meta-model of Kermeta Transformations is discussed.

6.4.1 Transformations

The Class2Relational transformations specifies a simple transformation from class schema model into a relational database model.

6.4.2 Rule Specifications

Below is the description of rules to generate a relational model from a class model:

- A Table instance has to be generated from a Class instance.
    - Name of the Class and Table have to correspond.
    - The col reference set has to comprise all Columns that have been created for single valued attribute and also the key described in the following
    - An Attribute instance has to be created as key
        - Its name has set be as "objetcId"
        - Its type reference has to reference a Type with the name Integer which -if not yet existing-has to be generated.
- A Type instance has to be generated, for each DataType instance.
    - Names and types of the Type and DataType have to be similar.
- A Table instance has to be generated for each multivalued Attribute instance of type DataType.
    - The Table's name is the name of the Attribute's Class concatenated with an underscore and the name of the Attribute.
    - The col reference set has to reference the two columns described in the following.
    - An identifier column instance has to be created.
        - Its name has to be set to the Attribute's class name concatenated with 'Id'
        - Its type reference has to reference a Type with the name Integer which –if not yet existing- has to be created.
    - A Column instance has to be created to contain the values of the Attribute.
        - The name and their types have to correspond.
- A new column has to be generated, for each single-valued Attribute of type Class.
    - Its name has to be set to the attribute's name concatenated with 'Id'.
    - Its type reference has to reference a Type with the name Integer which –if not yet existing- has to be created.
- A new Table has to be generated, for each multivalued Attribute of the type Class.
    - The Table's name is the name of the Attribute's Class concatenated with an underscore and the name of the Attribute.
    - The col reference set has to reference the two Columns described in the following.
    - An identifier Column instance has to be created.
        - Its name has to be set to the Attribute's class name concatenated with 'Id'.

        - Its type reference has to reference a Type with the name Integer which - if not yet existing - has to be created.
    - A foreign key Column instance has to be created.
        - Its name has to be set to the Attribute's name concatenated with 'Id'.
        - Its type reference has to reference a Type with the name Integer which - if not yet existing - has to be created.

6.4.3 Kermeta Code

Below is the Kermeta code for generating Table model from Class model. The transformations were originally written in ATL. It was re-written using Kermeta to demonstrate the proposed approach in this dissertation.

```
/** The t r a c e o f t h e t r an s fo rm a t i on */

r e f e r e n c e c l a s s 2 t a b l e : Trace<C l a s s , Tab le>

/** S e t o f k e ys o f t h e ou tpu t model */

class Class2Relational{

reference fkeys: Collection <FKey>

operation transform(inputModel: ClassModel): RelationalModel is do

//Initialize the trace

Class2table:=Trace<Class, Table>.new class2table.create

Fkeys:=Set<FKey>.new

Result:=RelationalModel.new

//create tables

getAllClasses(inputModel).select{c|c.ispersistent}.each{c| var table:Table init Table.new

table.name:=c.name

class2table.storeTrace(c, table)

result.table.add(table)

}

//Create columns

getAllClasses(inputModel).select{c|c.ispersistent}.each{c|createColumns(class2table.getTarg
etElem©, c, "")

}
//create foreign keys
Fkeys.each{k|k.createKeyColumns}
end
}
operation createColumns(table : Table, cls : Class, prefix : String) is do
// add all attributes
getAllAttributes(cls).each{ att | createColumnsForAttribute(table, att, prefix)
}
// add all associations
getAllAssociation(cls).each{ asso | createColumnsForAssociation(table, asso, prefix)
}
end
operation createColumnsForAttribute (table : Table, att : Attribute, prefix : String) is do
// The type is primitive : create a simple column
if PrimitiveDataType.isInstance(att.type) then var c : Column init Column.new
c.name := prefix + att.name
c.type := att.type.name
table.cols.add(c)
if att.is_primary then table.pkey.add(c) end
else
var type : Class type ?= att.type
// The type is persitant
if isPersistentClass(type) then
// Create a FKey
var fk : FKey init FKey.new
fk.prefix := prefix + att.name
table.fkeys.add(fk)
```

```
fk.references:=class2table.getTargetElem(getPersistentClass(type))
fkeys.add(fk)
else
// Recusively add all attributes and associations of the non-persistent table
createColumns(table, type, prefix + att.name)
end
end
end
}

class FKey
{
reference references : Table
reference cols : Column[1..*]
/**
* prefix for the name of the columns
* used by the createFKeyColumns method
*/
attribute prefix : String
/**
* Create the FKey columns in the table
*/
operation createFKeyColumns() is do
var src_table : Table
src_table ?= container
// add columns
references.pkey.each{ k |
var c : Column init Column.new

c.name := prefix + k.name

c.type := k.type

self.cols.add(c)

src_table.cols.add(c)

}

end

}
```

6.4.4 Source Meta-model: Class Meta-model

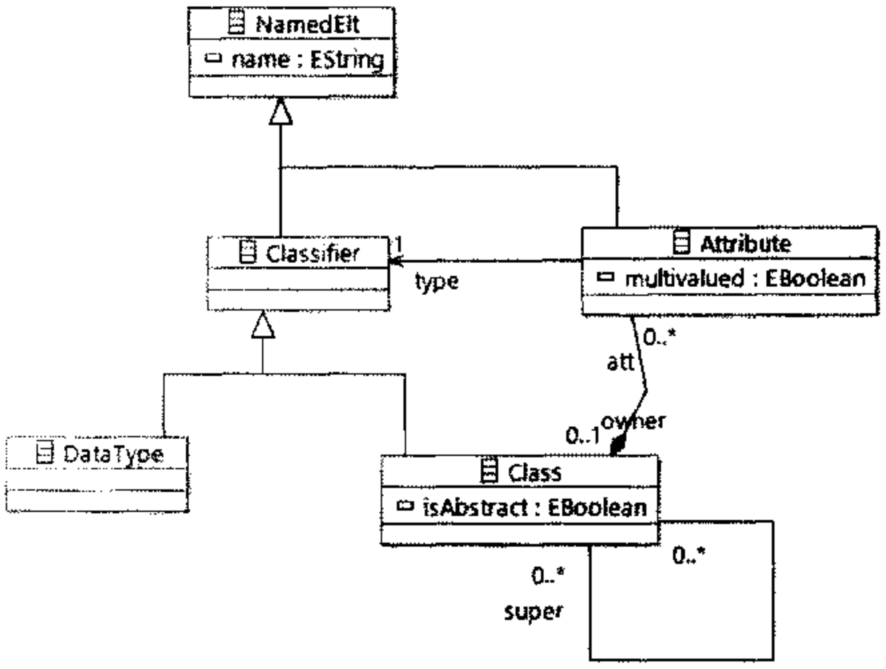


Figure 6.7 Original Class meta-model

The simple class meta-model mainly composed of classes. Every class have some name, which they inherit from the abstract class "NamedElt". The main concept is the class "Class" which encompasses a set of attributes of type "Attribute" and has the super references pointing to super classes for modeling inheritance trees. The Class "DatyeType" represents primitive data types. "Class" and "DataType" inherits from Class "Classifier" which serves to declare the type of Attributes. Attributes can be multivalued, which has an important impact on the transformation.

6.4.5 Evolved Class Meta-model

The evolved version of "Class" meta-model consists of additional class "Feature" with two attribute i.e. "ownerscope" and "visibility". The meta-class "Feature" classify the "Attribute" as feature of class and defines the scope and visibility of the attribute. The meta-class "Reference" is added to the new version of "Class" meta-model. The meta-class "Class" may composed of attributes and references. The reference is the composition of associations having '2' or '3' association ends. Figure 6.8 illustrates the evolved version of class meta-model.

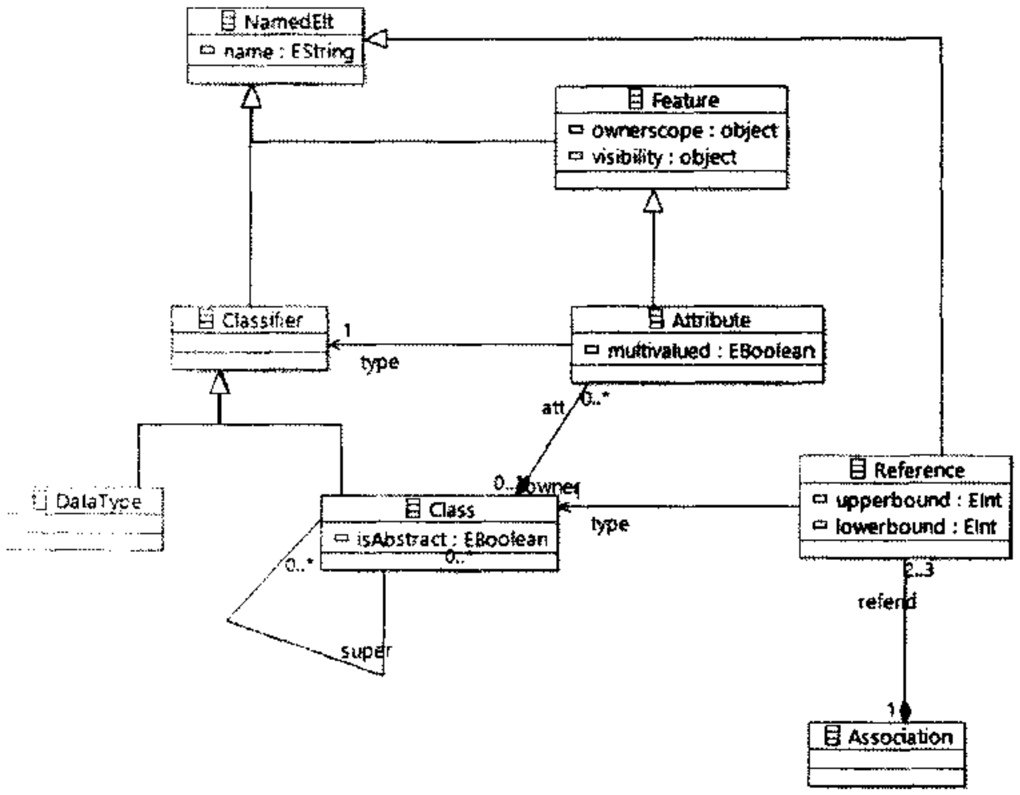


Figure 6.8 Evolved Class Meta-model

6.4.6 Target Relational Meta-model

The Relational meta-model encompasses classes having some name, which they inherit from the class "Named" that is abstract. The main class "Table" encompasses a set of "Columns" and has a reference to its keys. The class "Column" has the references 'owner' and 'keyOf' pointing to the "Table" it belongs to and of which it is part of the key (only in case it is a key). In addition, "Column" has reference to Type.

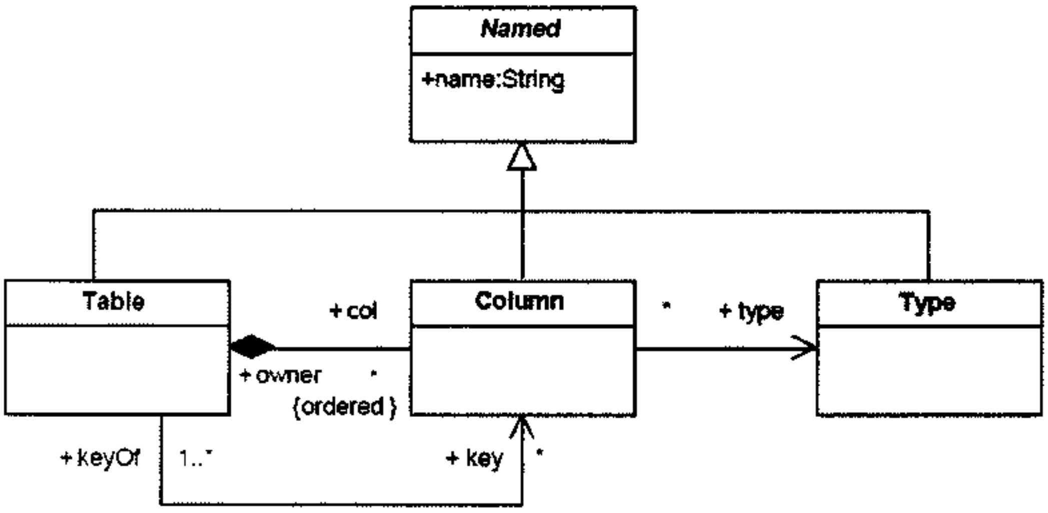


Figure 6.9 Target Relational Meta-model

6.5 Considered Changes and Its Types:

The whole consequences of transformations co-evolution can be handled according to the possible changes made to the underlying meta-model. These changes can be divided into different categories that are *additive*, *subtractive*, and *up-dative* changes. Specifically, additive changes can be referred to as additions of model elements to the meta-model. These additive changes can be further categorized as follows:

- ***Add meta-class***: The type of change in which a new meta-class is introduced to the meta-model. It is an ordinary change in meta-model evolution which brings extensions into meta-model. The addition of a new meta-class causes co-evolution problems only, in case, if the added model elements are compulsory according to the specified cardinality. In the above mentioned case, the instances of the newly added meta-class must be initialized accordingly in the existing models and therefore the must be handled by the specified transformations.
- ***Generalize meta-property***: If it type or multiplicity of the meta-property are relaxed. It is commonly referred to as the generalization of a meta-property. No action to co-evolve models and transformations is needed to be taken as the existing model instances still it is conformant to the new version of the meta-class and the instances are still handled by the transformations.
- ***Add meta-property***: The type of change in which a new meta-property is introduced to the underlying meta-model. This case is similar to the above mentioned case. A newly defined meta-property can be or cannot be compulsory according to the specified cardinality. Therefore, the already defined models have to maintain the "Conformance" relationship with the underlying meta-model and the transformations must handle it if the addition is made to some abstract meta-class without having sub-classes; in rest of the scenario, some human assistance is essential to define a value for the newly introduced meta-property in all the involved elements of model.
- ***Extract super-class***: If a new meta-class is defined in a hierarchy as super-class and some repeated and common meta-properties are moved to that super-class. This type of change is referred to as Extract super-class. If the super-class is abstract, the existing

model instances are conserved; in other case the consequences are similar to that of the meta-property pulls.

- *Pull meta-property*: when a property p is moved to a super-class Z from a sub-class A and the old one is removed from a subclass A, this is called pulling a meta-property. This change requires the modification of the instances of the meta-class Z through inheriting the value of p from the instances of the meta-class A and therefore the transformation specified to handle this considered meta-property need to be modified to enable it to handle the pulled up meta-property.

The Subtractive changes comprised the removal of the existing model elements. These changes are termed in the following:

- *Eliminate metaclass*: a metaclass is eliminated from meta-model to make it either simple or to exclude some obsolete concepts. In common, such changes prompt the deletion of the meta-class instances in the corresponding models. So the transformation defined to handle such meta-class is no longer consistent with the new version of meta-model and need to be eliminated. Besides, if the targeted meta-class has some other classes as sub-classes or it is called by other meta-class, the elimination brings terrible side effects.
- *Push metaproperty*: A meta-property is pushed to subclasses. It indicates that this meta-property is eliminated from its original super-class. Let suppose X and then replicated in every sub-class of let suppose Y of X. Now let suppose if super-class X is abstract in nature then this type of change does not require any step to co-evolve model and ultimately transformation co-evolution. In other cases, the existing instances of X and its sub-classes require to be modified accordingly.
- *Eliminate meta-property*: The impact of the elimination of the meta-property is the same as the elimination of meta-class.
- *Restrict meta-property*: If the type or multiplicity of a meta-property is imposed this type of change is referred as Restrict meta-property. This is a complicated situation in which existing instances require to be restricted or co-evolved and the transformations need to be modified to handle the imposed restriction. If the restriction is imposed on the upper bound of the multiplicity then it requires definite slots to be removed. Growing the lower bound of the multiplicity needs new values to be added and therefore need transformations modification accordingly. As for as type restriction of a property is concerned it needs the conversion of type for each value.
- *Flatten hierarchy*: The removal of a super-class and pushing down all its properties into its subclasses is referred a Flatten Hierarchy.

Lastly, a new version of the model might compose of some modifications of the existing model elements causing up-dative changes in transformations that can be categorized as follows:

- *Move metaproperty*: it is a type of change in which a property p is moved from a metaclass A to a metaclass B. This is a propagative change and the existing transformation can be co-evolved simply by updating the context of the transformation.
- *Rename metaelement*: A complex type of modification in which the change requires to be propagated to existing transformations and can be propagated in an automatic way or require user assistance depending on the type of transformation language used to define transformation.
- *Extract/inline metaclass*: A meta-class is said to be extracted if a new meta-class is created and all the related attributes are pulled from the old class into the newly created class. All the attributes of the considered meta-class are moved into newly added meta-class and eliminate the previous meta-class in order to inline a meta-class. These

mentioned meta-model re-factorings stimulate either automated or semi-automated transformations co-evolutions.

The categorization mentioned above until now, put together verification of the primary role of evolution representation. In first glimpse, it gives the impression that the categorization does not include *references,* which are associations between meta-classes. Nonetheless, references can be regarded as properties of meta-classes at the same level of attributes.

Chapter. 7 Evaluation

7.1 Introduction

This chapter is dedicated to explicate, explore and assess the results of the experiments, we have executed to validate our proposed approach for meta-model and transformations co-evolution. We will employ the proposed approach on the examples, discussed in the previous chapter to validate our proposed approach, in this chapter.

7.2 UML2Java transformations

This section explains the set-up we made for performing an experiment, discuss and assess the results of our proposed approach by applying it on UML2Java transformation example. The transformations are written in ATL and the meta-model is MOF based. The changes introduced were discussed in previous chapter (see section 6.5).

7.2.1 Case-1 Set-up

This section sums up the main steps we performed to execute our experiment below:

- In the first step, the trace meta-model is extended by adding model element of ATL meta-model. Since, the example is based on ATL transformations. All the essential ATL concepts either declarative or imperative are added to the trace meta-model. This empowers the Trace meta-model to capture the exact impacted part of transformations. Then, we established traces between considered UML model and underlying transformations. Total thirteen traces were captured between the old version of sample UML model and the implemented transformations in the underlying UML2JAVA example. The traces are recorded as trace models conforming to the trace meta-model. Each trace model captured the source and target of the trace. Below is the Table 7-1 trace models summary that illustrates the detail related to traces:

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	Tr_feature	Feature/class	isStatic()/helper	O2M/Rule & A2F/Rule
			isPublic()/helper	O2M & C2C & A2F (Rules)
[illegible]	Tr_melement	ModelElement/class	ε	ε
[illegible]	Tr_param	Parameter/class	P2F/rule	ε
			O2M/rule	ε
[illegible]	Tr_classifier	Classifier/class	ε	ε
[illegible]	Tr_namespace	NameSpace/class	C2C/rule	ε
			D2P/rule	ε
			getExtendedName()/helper	P2P/rule
[illegible]	Tr_op	Operation/class	O2M	ε
[illegible]	Tr_attr	Attribute/class	isFinal()/helper	A2F/rule
[illegible]	Tr_dt	DataType/class	D2P/rule	ε
[illegible]	Tr_model	Model/class	ε	ε

[illegible]	Tr_package	Package/class	P2P/rule	ε
[illegible]	Tr_name	Name/attribute	ε	ε
[illegible]	Tr_scope	Ownerscope/attribute	isStatic()/helper	O2M/rule & A2F/rule
[illegible]	Tr_visibility	Visibility/attribute	isPublic()/helper	O2M & C2C & A2F (Rules)

Table 7-1 trace models summary

- In second step, we introduced some changes to the sample UML model. Total nine well known and most common changes reported by the industry as well as academia were introduced to the sample UML model. Below, Table 7-2 shows the type of changes introduced to the sample UML model.

[illegible]		
• Add meta-class	• Eliminate meta-class	• Extract meta-class
• Add meta-property	• Eliminate meta-property	• Rename meta-property
• Generalize meta-property	• Push meta-property	• Move meta-property
• Pull meta-property	• Flatten hierarchy	• Inline meta-class
• Extract super-class	• Restrict meta-property	

Table 7-2 Type of changes

- In the third step, the change introduced to the sample UML model are captured as change model. The change model captured each simple change individually. Each change is identified and then classified according to the proposed criteria for change classification based on its impact on transformations. Classification is shown in Table 7-2.
- In the fourth step, the change models employed together with the trace model to analyze the impact of meta-model evolution on transformations.
- In fifth step, the changes are propagated to the transformations. The co-changes are incorporated to the impacted transformations and the un-impacted transformations are copied unchanged.

To scrutinize the correctness of the co-evolution of transformations the co-evolved transformations are applied to the evolved sample UML model. This guarantees the synthetic and semantic correctness of co-evolved transformations.

7.2.2 Results and discussions

Table 7-3 illustrates the type of changes introduced to the sample UML model and its impact on transformations. We introduced total nine changes to the meta-modei and assessed its impact on transformations. Table 7-3 depicts the important outcomes of the different steps of approach. This illuminates that if the trace for the impacted element does exist or not. This make it obvious if model element like that were changed, it would not have any impact on transformations. Our approach is capable to detect such kind of changes and classify them as non-propagative change. Table 7-3 also demonstrates that none of the relationship transform by the transformations are mapped and captured in traces by trace model. It might be due to the limitation of our approach or due to the implicit way of transforming the relationships.

The trace meta-model captured traces for each element either its corresponding transformation does exist or not. This indicates that if the model element is not used in transformations, yet the trace is captured. If that element changed, it would have no impact on transformations and those changes would automatically go in the non-propagative change class. Traces for model elements like "Feature" captured not only multiple targets for the trace against that model element but also the calls made to those targets. The information about the calls made to the transformation is useful in case of eliminating a model element that requires the deletion of the corresponding transformation as well as calls made to those transformations. Since mapping, elements and change detection and identification are not dependent on each other. Both activities can carried out in parallel. This improves the performance of our approach.

Changes introduced to the metamodel saved and represented as change model conforming to the change meta-model. The change meta-model captured all the information related to the change and require to detect the impact of changes on transformations. For instance, for the change, rename meta-class "DataType" as "PrimitiveType", the change model captured the name of the impacted element i.e. meta-class "DataType", the property changed i.e. "name", its old value i.e. "DataType" and newvalue i.e. "PrimitiveType". All this information would captured by the meta-model in any case renaming a class, an attribute, either a relationship or Type. Capturing the type of change assists in determining the type of action needs to take to co-evolve transformations. For instance, if the change type is "Modify" it implies that the corresponding transformations need some modification. This helps in classifying transformations as well as determines the type of co-change for transformations modification.

Meta-model changes might not affect transformations only if the element experiencing the change is not employed by the transformations. Moreover, the main reason behind classifying changes with the minor impact on transformations as propagative is that, a meta-element change, which requires co-change in transformations make transformations inconsistent and transformation file does not run at all and generate an error. The change "Extract superclass" were not propagated accurately. The approach generated only transformation for the added super class. While it requires not only the addition of transformation but also the reference to that superclass by the corresponding transformation of child classes.

The changes identified were then mapped on the traces to find the impacted transformations. The impacted element name and type was taken from the change model and was mapped on the trace. Apart from that the type of change is also captured. Capturing the type of change assists in determining the type of action needs to be taken to co-evolve transformations. It also assists in classifying the transformations according to the proposed classification scheme according to the proposed criteria.

Since the trace meta-model is able to capture implicit as well as explicit transformations as demonstrated in the previous chapter by employing it on running example. Therefore, the possibility of the implicit way of transforming references is more obvious.

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	•	•	Non-Propagative	Add	Generate transformation
[illegible]	•	•	Non-Propagative	Add	Generate transformation
[illegible]	Trace 15	isFinal(): context	Propagative	Modify	Replace context

[illegible]	Trace 9	Rule D2P	Propagative	Modify	Modify source pattern
[illegible]	Trace 14	isPublic(): context	Propagative	Modify	Modify context
[illegible]	•	•	propagative	Add	Generate transformation
[illegible]	Trace 8	C2C	Wallop	Delete	Delete rule C2C and C2C();
[illegible]	Trace 15	isFinal()	Wallop	Delete	Delete isFinal() and calls
[illegible]			Wallop	Modify and add	Modify and generate
[illegible]			Wallop	Delete and add	Delete and add
[illegible]	Trace 5	getChangeability(); , getUMLChangeability();	Wallop	Add and modify	Add and modify
[illegible]	Trace 6	Attribute name[0-1]	Non-propagative	Use as it is	Copy
[illegible]	Trace 3	Attribute visibility [0-1]	Non-propagative	Use as it is	Copy
[illegible]	Trace 16		propagative	Modify	Modify context

Table 7-3 summary of the approach's application on UML2JAVA example

[illegible]		
• Generalize meta-property	• Pull meta-property	• Eliminate meta-class
• Restrict meta-property	• Rename meta-property	• Eliminate meta-property
• Add meta-class	• Move meta-property	• Push meta-property
• Add meta-property	• Extract super-class	• Extract meta-class
	• Inline meta-class	• Flatten hierarchy

Table 7-4 change classification

7.3 QVT transformations

This section explains the set-up we made for performing an experiment, discusses, and assesses the results of our proposed approach for PetriNet transformations written in Query view transformation language.

7.3.1 Case-2 Set-up

This section summarizes the set-up we prepared to execute an experiment for validating the proposed approach

- In the first step, we extended the trace meta-model by adding model element of QVT meta-model. Since the example is based on QVT transformations. We added the QVT concepts particularly, the concepts related to the core and the operational part of QVT to the trace meta-model. This enables the Trace meta-model to capture the exact impacted part of transformations. Then, we established traces between the underlying PetriNet model and QVT transformations. Total seven traces were captured between sample PetriNet model and the implemented transformations in the underlying PetriNet2PNML example. The traces are recorded as trace models conforming to the trace meta-model. Each trace model captured the source and target of the trace. The Table 7-5 given below illustrates the details related to traces:

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
1	Tr_Net	Net/class	Net()/mapping	ε
2	Tr_place	Place/class	place2place()/mapping	ε
3	Tr_Trans	Transition/class	transition()/mapping	ε
4	Tr_name	name/attribute	Place2place()/mapping	ε
5	Tr_trname	Name/attribute	Transition()/mapping	ε
6	Tr_netname	Name/attribute	Net()/mapping	ε
7	Tr_link	Src-dest/link	Place2place()/mapping	ε
			Transition()/mapping	ε

Table 7-5 summary of the captured traces

- In second step, some changes were introduced to the sample petri net meta-model. The PetrNet meta-model was refined by introducing four changes to obtain the new version of meta-model i.e.
    - o TransitionToPlace and PlaceToTransition meta-classes have been added.
    - o The class Arc has been added as super class of the meta-classes TransitionToPlace and PlaceToTransition.
    - o The class Net has been renamed as PetriNet.
    - o The references place and transition of the Net class have been merged in the elements reference of the PetriNet class.
- In the third step, the change introduced to the sample PetriNet model are captured as change model. The change model captured each simple change individually. Each change is identified and then classified according to the proposed criteria for change classification based on its impact on transformations.
- In the fourth step, the change models are employed together with the trace model to analyze the impact of meta-model evolution on transformations.
- In fifth step, the change propagation is done via employing higher order transformations.

7.3.2 Results and discussions

During the investigation process of evaluating the applicability of the approach, the approach applied to PetriNet transformation example. In this example, the transformations were defined using Qvt core mappings. Total four mapping does happen between PetriNet and PNML. Total seven traces were identified. Total six changes were introduced to the PetriNet meta-model. It is clear from Table 7-5 that the trace meta-model is capable to capture the traces of links as well but the links must be transformed by the transformations explicitly. The trace model is unable to capture the implicit transformation of the links.

Table 7-6 depicts the summary of the approach's applicability. Table 7-6 clearly illustrates that the change "merge references" is a complex change that requires multiple actions not only at model level but also at transformation level to propagate it accurately. The change "merged reference"requires the deletion of mapping content from the mapping object Net(). The approach captured this change as the deletion of reference place and transition from model element Net and as the addition of new reference to the model element PetriNet class. This shows that the approach is capable to capture simple modification change while complex modifications like "merged reference" is handled as simple additions and deletions. This results in the context loss while propagating change to the transformations.

Table 7-7 shows the change classification. If we compare the changes introduced to the Case-II and I. It clearly shows that change classification may vary from example to example and language to language. Since the impact of change depends not only the type of change but also on the usage of model element by transformations and the relationship that transformations hold with the meta-model. That is why change that might not have an impact and classify as non-propagative in one case might have a drastic impact in the other underlying case and fall in a wallop class.

We have not observed any of the non-propagative change in Case-II. Four out of six changes were propagative while two were classified as wallop changes. The change "Merged References" were a complex change and it was not identified as "merge references". It was identified as the deletion of reference "place" and "transition" and the addition of new element "Element", addition of reference "element" and "net" between class "Element" and "Net". The deletion of reference "place", "transition", and addition of class "Element" were handled accurately. However, the addition of reference "net" and "element" were not handled accurately as these two reference required the modification the previously added transformation and the approach simply generated transformation for both of references.

The change add reference "net" and "element" were erroneously classified as non-propagative by the approach. Complex changes like "merged references" are not handled by our approach and while handling such changes the actual context of the change is lost and hence, the changes were propagated erroneously. This shows the limitation of our approach that does not handled references and complex modifications accurately. It requires further investigation. We observed in previous case that implicit traces between references and transformations were not captured. This requires further improvement to handle the references as well as complex modifications.

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	Tr_Net	Net()	Propagative	Up-dative	Modify source Type
[illegible]	ε	ε	Propagative	Add transformation	Add transformations
[illegible]	ε	ε	Propagative	Add transformation	Add
[illegible]	Tr_Net	Net()	Propagative	delete the mapping contents in mapping body	Delete content statement
[illegible]	Tr_Net	Net()	Propagative	Delete the mapping content	Delete statement contents

[illegible]	ε	ε	Non-propagative	Add transformation	Generate transformation
[illegible]	ε	ε	Non-propagative	Add statement	Modify transformation

Table 7-6 summary of the application of proposed approach

[illegible]	[illegible]	[illegible]
ε	Rename meta-element Add super class Add class Eliminate meta-property	Merged references Pull meta-property

Table 7-7 change classification

7.4 Kermeta transformations

The third case we considered to evaluate our approach is the Kermeta transformation example i.e. Class2Relational. This section elucidates the set-up we made to evaluate the applicability and validity of our proposed approach by applying it on Kermeta transformation example.

7.4.1 Case-III Set-up

This section sums up the set-up we prepared to execute an experiment for validating the proposed approach on Kermeta transformation example.

- In the first step, we extended the trace meta-model by adding model element of Kermeta meta-model. Since the example is based on Kermeta transformations. We added the essential Kermeta concepts particularly, the concepts used to specify transformations to the trace meta-model. This enables the Trace meta-model to capture the exact impacted part of transformations. Then, we established traces between the underlying "Class" model elements and Kermeta transformations. The traces are recorded as trace models conforming to the trace meta-model. Each trace model captured the source and target of the trace. The Table 7-8 given below illustrates the details related to traces:

Trace#	[illegible]	[illegible]	[illegible]	[illegible]
1	Tr_Class	Class/ Class	Class2table/Trace	Transform/Operation
			createColumns/Operation	ε
2	Tr_name	Name/Attribute	creatColumnsForAttribute/ Operation	createColumns/ Operation
			createFKeyColumns/ Operation	ε
			Transform/Operation	ε
3	Tr_Attribute	Attribute/Class	createColumnsForAttribute/ Operation	createColumns()/ Operation
			createColumns/Operation	ε
4	Tr_Classifier	Classifier	ε	ε
5	Tr_Datatype	DataTye/Class	createColumnsForAttribute/ Operation	createColumns()/ Operation

6	Tr_isAbstract	isAbstract/ Attribute	ɛ	ɛ
7	Tr_multi-valued	Multivalued/Attribute	ɛ	ɛ
8	Tr_namedElt	NamedElt/Class		
9	Tr_association	Association/Reference	createColumns/Operation	ɛ

Table 7-8 Traces Captured for Kermeta Transformations

- In second step, some changes were introduced to the sample considered "Class" meta-model. The Class meta-model was extended by introducing some changes to obtain the new version of meta-model i.e.
    - Addition of class "Feature" that classify the attribute of the class as its feature.
    - The Addition of attributes "ownerscope" and "visibility" to the class "Feature" that defines the visibility and the scope of the attribute.
    - The addition of "Reference" meta-class that have upper and lower bound.
    - The class is composed of references and attributes. Each reference composed of associations having '2' association ends.

Add class	Add Attribute	Add composition
	Add association end	Add Associations

Table 7-9 Change Classification

- In the third step, the change introduced to the Class model are captured as change model. The change model captured each simple change individually. Each change is identified and then classified according to the proposed criteria for change classification based on its impact on transformations.
- In the fourth step, the change models are employed together with the trace model to analyze the impact of meta-model evolution on transformations.
- In fifth step, the change propagation is done via incorporating co-changes to the transformations.

7.4.2 Results and Discussions

The proposed approach is applied on the considered Kermeta example. The possible changes added to the class meta-model was addition of classes, attributes, references of type composition and aggregation and its effect were analysed. The nature of Kermeta transformations and the structure of transformation definition is quite different from the other considered transformation languages i.e. ATL and QVT. The transformations are defined as operations and the processing is done like general-purpose languages. Therefore, the impact of evolution were observed to be quite different and illustrated different results.

The approach captured nine traces as trace model successfully and identified the exact location where the model elements are employed by the transformations. Trace for the attributes "isAbstract" and "multivalued" showed that there is no target element for those two model elements. This indicates that if these model elements are modified they will not impact any transformations and therefore the corresponding transformations are copied automatically. The rest of the traces captured the transformations as well the calls to that transformations, which would be helpful in propagating corresponding changes to the transformations.

The addition of meta-class "Feature" did not affected any transformation and it does not need to be propagated since, this class is defined as parent class of the meta-class "Attribute". The purpose of this addition is the extension of meta-model and to define the scope and visibility of the attribute. Therefore, the generation of transformation for meta-class "Feature" would be less significant. However, the meta-attributes "ownerscope" and "visibility" of meta-class "Feature" would be propagated. The reason of their classification as propagative is that one can now define scope and visibility of the attributes and the transformations must need to take into account its visibility and scope.

The addition of association end were classified by our approach as propagative. However, the association ends are handled already by the transformations and it cannot effect transformations. The approach classified it as propagative and the corresponding action taken for propagating the change to transformations make a change that was ambiguous and created duplicate statements that caused an error.

The addition of new reference types i.e. "aggregation" and "composition" were not evaluated in the previous example. Both the changes require to define operation and require to create tables while generating relational model. This requires a highly intelligent mechanism to generate operation that handle the added relationships. This showed that co-evolution of transformation can be hard in case of additions as well.

7.5 Simplified View of Classified Changes and Corresponding Action for Case-I

After applying the approach on three different example, we observed the pattern of classification that is explained in the Table given below:

Change Class	Change Type	Change	[illegible]
Non-propagative	Additive	Extract Super-class	Generate a new rule if the super class is mandatory
	Additive	Generalize Meta-property	No action is required
	Subtractive	Push Meta-Property	No action is required
	Subtractive	Flatten Hierarchy	Deletion of the rule along with the references is required
	Additive	Pull Meta-property	The context requires modification
Propagative	Subtractive	Eliminate Meta-Class	Rule requires deletion along with the calls
	Up-dative	Rename Meta-Element	Synchronization with the new name is required

	Subtractive	Eliminate Meta-property	Deletion of the rule is required along with the references made to it and values initialized for that particular property
	Subtractive	Restrict Meta-property	Modification is required to limit the bounds
Wallop	Additive	Add Meta-property	New conditions and transformations require to be added along with the bindings
	Additive	Add Meta-class	New conditions, transformations and bindings is required to be added
	Up-dative	Extract Meta-class	Add transformations and modify context for the properties moved to the that class
	Up-dative	Inline Meta-class	To modify the context of the transformations defined the properties moved and the deletion of the transformation
	Up-dative	Move Meta-property	The transformations context require modification

Table 7-10 **Classification of model element changes**

7.6 Comparison

The exiting approaches for transformations co-evolution use the strategies and schemes devised for model co-evolution problem. Therefore, in addition to the inherent limitations, these approaches have some additional issues that are particular to transformations co-evolution problem. Our approach employs dedicated mechanisms we formulated specifically for transformation co-evolution problem keeping in mind:

- The difference between the nature of models and transformations.
- The difference in the nature of relationship between meta-model/transformations and meta-model/models. and
- In case of transformations co-evolution the impact of change not only depends on the type of change but also on the usage of model element by transformations and the intelligence the transformations use to generate model elements.

This section compares our approach with the existing approaches by considering whether they support the essential steps that one must carry out while performing the process of co-evolution. We also performed the comparison based on the essence of the approaches by setting criteria. Below is Table 7-11 comparison with the existing approaches that summarizes the comparison of the approaches.

Co-evolution Approach	*Change Detection*	*Change Classification*	*Impact Analysis*	*Change Propagation/co-Evolution*
Operator Based Approach	Change history is already available	[illegible]	[illegible]	[illegible]
Difference Based Approach	Use any of the existing tool	Uses classification scheme of model co-evolution	ε	Uses higher order transformations
Trans-Cos	Defines and uses a dedicated mechanism to identify ~~tnm~~ changes	Defines dedicated scheme	[illegible]	[illegible]

Table 7-11 comparison with the existing approaches

7.6.1 Difference detection and identification mechanisms

The existing transformation co-evolution approaches suppose either meta-model change history is already available (which is not feasible in every case) or use the available tools dedicated to difference detection (which requires the integration of tools). However, our approach is significantly different from existing approaches regarding the change detection and identification phase. Our approach proposes and employs a dedicated mechanism to identify changes between two versions of meta-model. Therefore, our approach do not require tool integration neither the availability of the meta-model change history. It is capable to identify changes if the two versions of the meta-model are available. It addresses all the issues and cover up the limitations of existing approaches.

7.6.2 Change Classification Scheme

The existing approaches use the classification scheme proposed for model co-evolution process ignoring the difference in the essence of models and transformations and the type of relationship that they hold with meta-model. Our approach have a classification scheme that considers the difference in both the nature of models and transformations and the difference in the type of relationship that meta-model holds with transformations and models. This makes our approach significantly different from the existing approaches and is appropriate for solving the problem of transformations co-evolution.

7.6.3 Impact Analysis

The existing co-evolution approaches does not provide any support for performing impact analysis. Impact analysis is an important step; need to perform to identify the impacted parts of transformations. Therefore, that meta-model change can be accurately propagated to transformations to co-evolve transformations and make transformations consistent to the evolved version of meta-model. The Operator based approach defines coupled operators for the co-evolution of meta-model and ATL transformations they analyzed the transformations manually to identify that model element change can influenced which parts of the transformation. They defined operators for propagating changes excluding the impact analysis part. While our approach identify the impacted transformations during performing, the co-evolution process and we used a meta-model based strategy to identify the impacted transformations, which enables the approach applicable on a variety of transformations written in any language.

7.6.4 Migration Strategy

For transformations co-evolution, we used a relationship based change propagation mechanism while the existing approaches used either Higher Order Transformations (HOT) or operators. The availability of multiple operators creates misperception and the selection of incorrect operator can lead to incorrect migrations. Similarly, the transformations migration in difference-based approach depends on the difference captured by the comparer. Our approach used relationship-based approach, which employs the relationship to locate inconsistencies in transformations and therefore, lead to correct change propagation.

7.7 Assessment

This section emphasizes mainly the key advantages and the limitations of our proposed approach. We observed numerous benefits of the proposed approach and some limitations we will discuss in the next subsections.

7.7.1 Benefits

During the execution of the designed experiments, various unique features and advantages of the proposed approach are observed. The key benefits that our approach offers over the existing approaches are discussed in this section.

Generic Approach: our approach is generic approach, which is proficient enough to co-evolve transformations written in any dedicated transformation language. The approach is generic enough to be applicable on transformations written in any dedicated transformations language.

Systematic Approach: Our approach is systematic i.e. it supports the entire process starting with the detection of changes between two versions of meta-model, establishing a relationship between model elements and transformations, analyzing the impact of change and propagating the change to the transformations.

Implementable Approach: our proposed approach is implementable. We provided a co-evolution algorithm, which can be implemented to provide a tool support that can co-evolve transformations automatically.

Meta-model Independent Approach: It is independent of the meta-model as well. The approach is applicable on any type of meta-model either EMF/Ecore based or MOF based. This makes our approach distinct and better from the existing approaches for transformations co-evolution.

Transformation Language Independent Approach: it is independent of the transformation language i.e. this approach is applicable on transformations written in any dedicated transformations language. This makes our approach generic and diverse and marks as distinct and better from the existing transformations co-evolution approaches.

Synthetic Correctness: our approach ensures the synthetic correctness of evolved transformations by employing relationship based approach to propagate model element changes.

Support source and target meta-model evolution: Our proposed approach is capable to support the evolution of both source and/or target meta-model and co-evolution of transformations in response to source/target meta-model. The trace meta-model as well as the difference meta-model is independent of the meta-model type and its usage in transformations. It can capture the traces for meta-model either it I used on the left hand side of transformations or at the right hand side of transformations.

Relationship based change propagation: Our approach is capable to propagate changes based on the relationship between model elements and transformations. This make the approach capable to capture the usage of model elements by transformations, which was ignored by the previous transformations co-evolution approaches, and therefore making those approaches incapable to propagate change changes appropriately.

Difference Representation and Visualization: Our approach provide a means to represent and visualize changes introduced in the new version of meta-model. The aim to define difference mechanism is to provide a means to represent version differences, capture changes appropriately and in a structured way and to propagate change impact in well-defined way. Our difference mechanism not only represent and visualize differences between different versions of meta-models but assists in identifying and classifying meta-model changes based on its impact on transformations.

Minimal User Intervention: our proposed approach required minimal user intervention. The user needs only to validate either the transformations are co-evolved correctly by executing co-evolved transformations on the evolved meta-model instances.

7.7.2 Limitations

Nonetheless, our approach provides numerous features that are distinct and better from the existing approaches. Yet some limitations exist that confines the applicability of the approach and that requires attention and in-depth study and analysis to overcome. Below are the major limitations of our proposed approach observed during the conduction of case study.

Capturing traces for implicitly transformed model element: Our study results illustrated that this approach is not capable to capture the traces for the links/references that transformed implicitly by the transformations. This means that the model elements that are explicitly used by the transformations is captured by the proposed approach but the model elements that are utilized by the transformations implicitly cannot be traced by the transformations.

Capturing complex modifications: Complex changes e.g. modifications that requires multiple additions and deletions are identified and handled by the approach as simple additions and deletions of the model elements, which results the loss of the context of change while propagating the change to transformations.

Application to Diverse examples: we applied our approach on three different examples. To demonstrate and validate its applicability on a variety of transformations, it should be applied to additional diverse and large examples.

Chapter. 8 Conclusion and Future Work

8.1 Introduction

This chapter is dedicated to highlight the momentous and noteworthy findings from this research. The entire research and its significant findings are summarized as general conclusion in section 8.2. As a final point, section 0 sums up some future key research directions identified by this search.

8.2 Conclusion

This research has an impact and can be viewed as an endowment to the area of co-evolution of meta-model and its related artifacts particularly to the transformations, which is still an open research area to be studied and researched intensely. The area of meta-model evolution and the co-evolution of it related artifacts has been enriched and supported by various co-evolution approaches, languages and tools by the industry and academia for the meta-model and model co-evolution. These efforts make it convenient to perform the co-evolution process of meta-model and models semi-automatically. Because of these efforts, the process of co-evolution of meta-models and models is quite mature and plausible. The study of the existing research in the area of co-evolution of meta-model and its related artifacts proved that meta-model model co-evolution has been largely investigated while the problem of meta-model and transformations co-evolution has been not sufficiently investigated and researched. This makes it obvious that the meta-model and transformations co-evolution research is in its formative years and the key issues need to be dealt with. Only few efforts has been made to cope up with the problem of meta-model and transformations so far.

Existing research related to transformations co-evolution mostly employed the model co-evolution strategies ignoring the fact that there is significant difference between the nature of these artifacts and their relationship to the meta-model. The relationship between meta-model and transformations does not only be influenced by the type of relationship but also the usage of model element by transformations and the intelligence the transformations employed to transform source elements into target elements. It makes the employment of model co-evolution strategies on transformations impractical and the results are not viable and practicable. The supposition that meta-model change history is available, further confines the applicability of the existing co-evolution approaches to transformations co-evolution. The co-evolution approaches particular to the transformations co-evolution does not cover the entire co-evolution process. Some of the approaches define only the co-evolution strategy while assuming that meta-model change history is already available while few of them only concentrate on the recovery of meta-model change history.

A broad and thorough study of the existing research discloses the fact that current approaches are not systematic and holistic in nature, which could cover the entire process of transformations co-evolution. As discussed above, some of the approaches are concerned with devising strategies for transformations co-evolution neglecting the fact of model element change identification and detection. While others focused on employing the model migration strategies leaving the fact of change in the nature of models and transformations and the relationship that does exist between meta-models and models and meta-models and transformations, behind.

This study also reveals that transformations languages employed to specify transformations also vary in nature as well as the way they specify transformations. It makes it clear that the usage of model element by transformations will also differ and ultimately the impact of same model element change might influence transformations of different transformation languages entirely in different way. It might be possible that one transformation written in transformation language x can be automatically adapted to the change in model element while the transformation defined in language y consuming the same model element might require human assistance. This makes the idea of automatic evolution of transformations a bit more difficult than expected or assumed and needs further investigation. Our research originates from these interpretations and elucidations to believe and consider the systematic and automated approach as one holistic solution to resolve the issues identified by this research study and to answer the research challenges of transformations co-evolution process via providing traceability and change propagation support.

In this research study, we have devised a systematic and automatable approach to transformation co-evolution by employing the concepts of relationship based change propagation, i.e., tracing model elements through transformations and difference representation and visualization. The algorithm is designed to support and automate the co-evolution process. First, model elements of input source model are traced to transformations to identify model element usage by transformations and to utilize it later on for relationship based change propagation and impact analysis. Second, the changes incorporated in the new version of meta-model are identified and captured as change models. The change model captures the change along with every detail essential for analyzing its impact on transformations and for propagating change to the transformations. The change model either is capable to capture each change individually or as sequence, depends on the type of strategy employed. Third, for analyzing the impact of model element on transformations or its propagation, the approach used the change model and trace model. The recorded change and the model element and change type is selected from the change model and traces are searched for the selected model element. Once the trace having the underlying model element is identified, it is navigated to capture the particular part where the model element is used. Fourth, the impacted transformations are classified according to the proposed classification scheme. The next step is to search the impacted part in the actual transformation file and to incorporate the co-change identified during impact analysis.

An algorithm referred as transformations evolver abbreviated as Co-Evo supplements the proposed approach. We executed the algorithm successfully on the underlying examples. It takes the old and new version of meta-model and a transformation file as input. The algorithm comprised three basic parts i.e. the change detection and classification, tracing model elements and transformation and analyzing change impact and propagating this change to the transformation file. The approach is generic enough to be applicable on transformation specified in any transformation language as well as to any type of meta-model. The experimental results illustrated the applicability and generality of the approach. However, the only pre-requisite is that the trace meta-model must be extended and enriched by the transformation language elements of the considered transformations.

An in-depth and thorough analysis of the experimental results bring out that our approach is capable to propagate model element change keeping the synthetic correctness preserved. It re-establish the consistency relationship between model elements and transformations successfully and in a systematic way. The approach is capable to copy the transformations that are not impacted by the model element change. Changes can be represented and visualize as model conforming to the change meta-model.

The comparison of our approach to the existing transformations co-evolution approaches showed significant difference. Our approach support to carry out the impact analysis activity for transformations co-evolution process. While none of the existing approaches provide support to carry out impact analysis for co-evolution process. To carry out impact analysis activity to co-evolve transformations, showed promising results. It assisted not only in identifying the impacted parts of transformations but also the model element usage by transformations. The comparison also illustrated that none of the existing approach is using dedicated change classification scheme for transformations co-evolution process. We proposed and used dedicated change classification scheme that classifies changes according to its impact on transformations. Furthermore, the employment of relationship based change propagation mechanism showed momentous results. This mechanism employed a relationship between model element and transformations to propagate change. The exact part of the transformation, which uses the model element, is identified by establishing traces between model element and transformations during mapping element step. This relationship along with the change model is then used to identify the impacted part and part and incorporating co-change to transformations. This ensures the correct change propagation to transformations.

However, the proposed approach have also some limitations. The trace meta-model is not capable to capture traces for model elements that are transformed implicitly by the transformations. Furthermore, the change model could not capture complex modifications e.g. "merged references" accurately. This change was captures as sequence of individual changes and migration action was taken accordingly, which resulted in the loss of the context of change and therefore the change is propagated erroneously.

8.3 Future Work

This section discusses and highlights the potential research areas that need to be further investigated and explored.

8.3.1 Tool support

The proposed approach is automatable and need to be implemented. The supported algorithm and its successful execution and demonstration on the three underlying examples proved its applicability and practicability. The best tools and language that will assist in its implementation are the Eclipse Modeling Frame work (EMF), OCL, Ecore, JAVA and Kermeta.

8.3.2 Design Validation strategy

We performed the validation of co-evolved transformations manually by employing co-evolved transformations on the instances of the new version of meta-model. A suitable validation mechanism is required to be designed to make it possible to automate the validation process of the co-evolved transformations.

8.3.3 Defining Semantic Relationship and ensuring Semantic correctness

The proposed approach guarantees the synthetic correctness of the evolved transformations but the semantic correctness still needs to be investigated as the semantic relationship between meta-model and transformations is still not well-defined and is ambiguous. The semantic relationship between meta-model and transformations need to be first explored and defined and established appropriately. After defining the semantic relationship, it would become easy to explore and investigate methods and mechanisms to ensure it.

8.3.4 Investigate reverse transformations co-evolution

We have validated the applicability of the proposed approach by applying it on the source meta-model of transformations. As this presented approach is devised keeping in mind the target meta-model as well. In addition, we claim that it is capable to handle the co-evolution of target meta-model and transformations, therefore co-evolution of target meta-model and transformations need to be investigated in future.

8.3.5 Unified Solution for Co-evolution of models and transformations

The researchers investigated the problem of co-evolution of models and transformations separately. A possible research idea is to compare both problems and the existing approaches to address these problems to identify the commonality and variability. Based on the identified commonality and variability, a unified solution can be designed and implemented.

8.3.6 Co-evolution management Processes and structures

As the co-evolution is continues and step by step process, therefore it needs to be handled in a structured and systematic way. Hence, appropriate structures and process for managing the process of co-evolution of meta-model and its related artifacts needs to be investigated and defined.

8.3.7 Transformation migration notations

An open research challenge is the identification of appropriate notations for defining and specifying transformations migration strategy. Existing approaches employed either general-purpose programming languages or higher order transformation languages. The migration strategy is a specialization of model transformations languages, therefore to define specialized languages for defining transformation migration might be more appropriate.

8.3.8 Application to diverse transformations languages

Presently, we have demonstrated the diverse and generic nature of our approach by applying it on transformation example specified using three different transformations languages i.e. QVT, ATL and Kermeta. To investigate its generality it should be applied to the transformation example specified using other transformation languages in future.

8.3.9 Application to industrial case studies

We have demonstrated the applicability of the proposed approach by applying it on small and medium sized transformations examples. Nonetheless, to prove its effectiveness its applicability to large industrial case study should be investigated in future.

References

[1] D. C. Schmidt, "Model-Driven Engineering," *Computer,* vol. 39(2), pp. 25-31, 2006.

[2] J. García, O. Diaz, and M. Azanza, "Model transformation co-evolution: A semi-automatic approach," in *Software Language Engineering*, ed: Springer, 2013, pp. 144-163.

[3] J.-M. Favre, "Meta-model and model co-evolution within the 3D software space," in *ELISA: Workshop on Evolution of Large-scale Industrial Software Applications*, 2003, pp. 98-109.

[4] L. Iovino, A. Pierantonio, and I. Malavolta, "On the Impact Significance of Metamodel Evolution in MDE," *Journal of Object Technology,* vol. 11, pp. 3:1-33, 2012.

[5] K. Garces, F. Jouault, P. Cointe, and J. Bézivin, "Adaptation of models to evolving metamodels," 2008.

[6] M. W. T. Mens, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," *IWPSE,,* pp. 13-22, 2005.

[7] D. S. K. L. M. Rose, R. F. Paige, and F. A. C. Polack, "Enhanced automation for managing model and metamodel inconsistency," *ASE,* pp. 545-549, 2009.

[8] D. Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio, "A methodological approach for the coupled evolution of metamodels and atl transformations," *Theory and Practice of Model Transformations. Springer Berlin Heidelberg,* pp. 60-75, 2013.

[9] D. Mendez, et al., "Towards transformation migration after metamodel evolution.," *Model and Evolution Workshop. 2010,* 2010.

[10] B. Boehm, "A view of 20th and 21st century software engineering," 2006, pp. 12-29 %@ 1595933751.

[11] J. Gray, J.-P. Tolvanen, S. Kelly, A. Gokhale, S. Neema, and J. Sprinkle, "Domain-specific modeling," *Handbook of Dynamic System Modeling,* pp. 7-1, 2007.

[12] A. Kleppe, J. Warmer, and W. Bast, "MDA Explained. The Practice and Promise of The Model Driven Architecture," ed: Addison Wesley Reading, 2003.

[13] D. Frankel and J. Parodi, "Using model-driven architecture to develop web services," *IONA Technologies white paper,* 2002.

[14] J.-M. Favre, "Foundations of model (driven)(reverse) engineering: Models," 2004.

[15] S. Kent, "Model driven engineering," 2002, pp. 286-298 %@ 3540437037.

[16] D. C. Schmidt, "Model-driven engineering," *COMPUTER-IEEE COMPUTER SOCIETY-,* vol. 39, pp. 25 %@ 0018-9162, 2006.

[17] M. Blaha and J. Rumbaugh, *Object-oriented modeling and design with UML*: Pearson Education Upper Saddle River, 2005.

[18] B. Selic, "The pragmatics of model-driven development," *IEEE software,* vol. 20, pp. 19-25 %@ 0740-7459, 2003.

[19] F. Truyen, "The Fast Guide to Model Driven Architecture The Basics of Model Driven Architecture," *URL: http://www. omg. org/mda/presentations. htm, January,* 2006.

[20] J. Bézivin, "On the unification power of models," *Software & Systems Modeling,* vol. 4, pp. 171-188 %@ 1619-1366, 2005.

[21] C. Atkinson and T. Kuhne, "Model-driven development: a metamodeling foundation," *Software, IEEE,* vol. 20, pp. 36-41 %@ 0740-7459, 2003.

[22] O. M. G. Uml, "2.0 Superstructure Specification," *OMG, Needham,* 2004.

[23] T. Kühne, "Matters of (meta-) modeling," *Software & Systems Modeling,* vol. 5, pp. 369-385 %@ 1619-1366, 2006.

[24] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Meta-model differences for supporting model co-evolution," in *proceedings of the 2nd Workshop on Model-Driven Software Evolution, MoDSE'2008,* 2008.

[25] Q. Omg, "Meta object facility (mof) 2.0 query/view/transformation specification," *Final Adopted Specification (November 2005),* 2008.

[26] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electronic Notes in Theoretical Computer Science,* vol. 152, pp. 125-142 %@ 1571-0661, 2006.

[27] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," 2003, pp. 1-17.

[28] F. Allilaire, J. Bézivin, F. Jouault, and I. Kurtev, "ATL-eclipse support for model transformation," 2006.

[29] A. Group, "ATL User Manual Version 0.7," *LINA & INRIA,* 2006.

[30] A. G. Ismˆenia Galv˜ao, "Survey of Traceability Approaches in Model-Driven Engineering."

[31] A. Cicchetti, D. Di Ruscio, and A. Pierantonio, "Managing dependent changes in coupled evolution," in *Theory and Practice of Model Transformations,* ed: Springer, 2009, pp. 35-51.

[32] M. Eysholdt, S. Frey, and W. Hasselbring, "EMF Ecore based meta model evolution and model co-evolution," *Softwaretechnik-Trends,* vol. 29, pp. 20-21, 2009.

[33] B. Gruschko, D. Kolovos, and R. Paige, "Towards synchronizing models with evolving metamodels," in *Proceedings of the International Workshop on Model-Driven Software Evolution,* 2007.

[34] A. Demuth, R. E. Lopez-Herrejon, and A. Egyed, "Co-evolution of Metamodels and Models through Consistent Change Propagation," in *ME 2013–Models and Evolution Workshop Proceedings,* 2013, p. 14.

[35] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth, "An extensive catalog of operators for the coupled evolution of metamodels and models," in *Software Language Engineering,* ed: Springer, 2011, pp. 163-182.

[36] B. Hoisl, Z. Hu, and S. Hidaka, "Towards Co-Evolution in Model-driven Development via Bidirectional Higher-Order Transformation."

[37] S. Becker, B. Gruschko, T. Goldschmidt, and H. Koziolek, "A process model and classification scheme for semi-automatic meta-model evolution," in *1st*

Workshop MDD, SOA und IT-Management (MSI), GI, GiTO-Verlag, 2007, pp. 35-46.

[38] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. Polack, "Model migration with epsilon flock," in *Theory and Practice of Model Transformations*, ed: Springer, 2010, pp. 184-198.

[39] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *ECOOP 2007–Object-Oriented Programming*, ed: Springer, 2007, pp. 600-624.

[40] J. Schoenboeck, A. Kusel, J. Etzlstorfer, E. Kapsammer, W. Schwinger, M. Wimmer, and M. Wischenbart, "CARE–A Constraint-Based Approach for Re-Establishing Conformance-Relationships," *Information Technology (CRPIT)*, vol. 20, p. 23, 2014.

[41] S. Roser and B. Bauer, "Automatic generation and evolution of model transformations using ontology engineering space," in *Journal on Data Semantics XI*, ed: Springer, 2008, pp. 32-64 %@ 3540921478.

[42] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "COPE-automating coupled evolution of metamodels and models," in *ECOOP 2009–Object-Oriented Programming*, ed: Springer, 2009, pp. 52-76.

[43] M. Herrmannsdoerfer and M. Koegel, "Towards semantics-preserving model migration," in *International Workshop on Models and Evolution*, 2010.

[44] M. Herrmannsdoerfer, "Migrating UML activity models with COPE," *Transformation Tool Contest 2010 1-2 July 2010, Malaga, Spain*, p. 72, 2010.

[45] B. Meyers, M. Wimmer, A. Cicchetti, and J. Sprinkle, "A generic in-place transformation-based approach to structured model co-evolution," *Electronic Communications of the EASST*, vol. 42, 2012.

[46] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "Automatability of coupled evolution of metamodels and models in practice," in *Model Driven Engineering Languages and Systems*, ed: Springer, 2008, pp. 645-659.

[47] M. Wimmer, A. Kusel, J. Schönböck, W. Retschitzegger, W. Schwinger, and G. Kappel, "On using inplace transformations for model co-evolution," in *Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL)@ TOOLS*, 2010, pp. 65-78.

[48] S. D. Vermolen, G. Wachsmuth, and E. Visser, "Reconstructing complex metamodel evolution," in *Software Language Engineering*, ed: Springer, 2012, pp. 201-221.

[49] A. Narayanan, T. Levendovszky, D. Balasubramanian, and G. Karsai, "Automatic domain model migration to manage metamodel evolution," in *Model Driven Engineering Languages and Systems*, ed: Springer, 2009, pp. 706-711.

[50] M. Herrmannsdoerfer and D. Ratiu, "Limitations of automating model migration in response to metamodel adaptation," in *Models in Software Engineering*, ed: Springer, 2010, pp. 205-219.

[51] M. Herrmannsdoerfer and M. Koegel, "Towards a generic operation recorder for model evolution," in *Proceedings of the 1st International Workshop on Model Comparison in Practice*, 2010, pp. 76-81.

[52] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *Enterprise Distributed Object*

Computing Conference, 2008. EDOC'08. 12th International IEEE, 2008, pp. 222-231.

[53] A. Cicchetti, D. Di Ruscio, and A. Pierantonio, "A Metamodel Independent Approach to Difference Representation," *Journal of Object Technology,* vol. 6, pp. 165-185, 2007.

[54] A. Cicchetti, F. Ciccozzi, T. Leveque, and A. Pierantonio, "On the concurrent Versioning of Metamodels and Models: Challenges and possible Solutions," in *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, 2011, pp. 16-25.

[55] A. Cicchetti, F. Ciccozzi, and T. Leveque, "A Solution for Concurrent Versioning of Metamodels and Models," *Journal of Object Technology,* vol. 11, pp. 1:1-32, 2012.

[56] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack, "An analysis of approaches to model migration," in *Proc. Joint MoDSE-MCCM Workshop,* 2009, pp. 6-15.

[57] S. Kruse, "On the use of operators for the co-evolution of metamodels and transformations," in *International Workshop on Models and Evolution*, 2011.

[58] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*: Pearson Education, 2008.

[59] "Emftext project: concrete syntax mapper available at http://www.reuseware.org/index.php/EMFText.."

[60] F. Jouault, Bézivin, J., Kurtev, I., "TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering.," *In: Procs of the 5th Int. Conf. on Generative programming and Component Engineering (GPCE '06),* vol. ACM(2006), pp. 249-254.

[61] S. Efftinge and M. Völter, "oAW xText: A framework for textual DSLs," 2006, p. 118.

[62] D. Di Ruscio, R. Lämmel, and A. Pierantonio, "Automated co-evolution of GMF editor models," in *Software Language Engineering,* ed: Springer, 2011, pp. 143-162 %@ 3642194397.

[63] D. Di Ruscio, L. Iovino, and A. Pierantonio, "What is needed for managing co-evolution in MDE?," in *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, 2011, pp. 30-38.

[64] S. Winkler and J. Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Software and Systems Modeling (SoSyM),* vol. 9, pp. 529-565 %@ 1619-1366, 2010.

[65] T. A. A. L. Naslavsky, D. J. Richardson, H. Ziv, "Using Scenarios to Support Traceability," *In TEFSE '05:Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering,* pp. 25–30, 2005.

[66] A. C. W. F. O. C. Z. Gotel, "An Analysis of the Requirements Traceability Problem," *IEEE Computer Science Press,* vol. In Proceedings of the International Conference on Requirements Engineering., 1994.

[67] P. v. E. J. P. Almeida, M.-E. Iacob, "Requirements Traceability and Transformation Conformance in Model-Driven Development," *Tenth IEEE*

International Enterprise Distributed Object Computing Conference (EDOC'06), pp. 355-366, 2006.

[68] C. K. C. J. Cleland-Huang, M. Christensen, "Event-Based Traceability for Managing Evolutionary Change," *IEEE Transactions on Software Engineering,* pp. 796–810, 2003.

[69] R. H. E. Gamma, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," *Addison-Wesley Professional,* 1995.

[70] R. S. J. Cleland-Huang, O. BenKhadra, E. Berezhanskaya, S. Christina, "Goal-centric Traceability for Managing Non-functional Requirements," *In Proceedings of the 27th International Conference on Software Engineering (ICSE'05),* pp. 362–371, 2005.

[71] B. T. N. N. Aizenbud-Reshef, J. Rubin, Y. Shaham-Gafni, "Model Traceability," *IBM SYSTEMS JOURNAL,* vol. 45, 2006.

[72] J. H. J. Grundy, W. B. Mugridge, "Inconsistency Management for Multiple-View Software Development Environments," *IEEE Transactions on Software Engineering,* vol. 11, pp. 960–981, 1998.

[73] P. Desfray, "MDA—When a Major Software Industry Trend Meets Our Toolset," *Softeam,* 2001.

[74] J. G. T. Olsson, "Supporting Traceability and Inconsistency Management Between Software Artifacts," *Proceedings of the IASTED International Conference on Software Engineering and Applications, Boston,* 2002.

[75] R. F. P. N. Aizenbud-Reshef, J. Rubin, Y. Shaham-Gafni, D. S. Kolovos, "Operational Semantics for Traceability," *ECMDA Traceability Workshop, Nuremberg, Germany,* pp. 7–14, 2005.

[76] M. L. D. Hearnden, K. Raymond, "Incremental model transformation for the evolution of model-driven systems," *International Conference on Model Driven Engineering Languages and Systems (MoDELS),* vol. 4199, pp. 321-335, 2006.

[77] G. B. I. Rath, A. Okros, and D. Varro., "model transformations driven by incremental pattern matching," *International Conference on the Theory and Practice of Model Transformations (ICMT),* vol. 5063, pp. 107-121, 2008.

[78] L. Tratt, "A change propagating model transformation language," *Journal of Object Technology,* pp. 107-124, 2008.

[79] R. F. P. D.S. Kolovos, F.A.Polack, "The Epsilon Object Language (EOL)," *European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA),* vol. 4066, pp. 128-142, 2006.

[80] J. J. M. Fritzsche, S. Zschaler, A. Zherebtsov, A. Terekhov, "Application of tracing techniques in model-driven performance engineering," *Proc.Traceability Workshop, co-located with the European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA),* pp. 111-120, 2008.

[81] F. Jouault, "Loosely coupled traceability for ATL," *In Proceeding Workshop on Traceability co-located with the European Conference on Model-Driven Architecture (ECMDA),* 2005.

[82] R. F. P. N. Drivalos, K.J. Fernandes, D.S. Kolovos, "Towards rigorously defined model-to-model traceability," *In Proceeding Workshop onTraceability, co-*

located with the European Conference on Model Driven Architecture (ECMDA),, 2008.

[83] S. A. Bohner, "Software change impacts - an evolving perspective," *In Proceeding International Conference on Software Maintenance (ICSM)*, pp. 263-272, 2002.

[84] Y. L. L.C. Briand, L. O'Sullivan, "Impact analysis and change management of uml models," *In Proc. International Conference on Software Maintenance (ICSM)*, pp. 256-265, 2003.

[85] B. S. Lerner, "A model for compound type changes encountered in schema evolution," *ACM Transactions on Database Systems*, pp. 83-127, 2000.

[86] A. K. R.M. Fuhrer, M. Keller, "Refactoring in the Eclipse JDT: Past, present, and future," *In Proc. Workshop on Refactoring Tools (WRT), co-located with European Conference on Object-Oriented Programming (ECOOP)*, 2007.

[87] R. F. P. L.M. Rose, D.S. Kolovos, F.A.C. Polack, "The Epsilon Generation Language," *In Proceedings European Conference on Model Driven Architecture -Foundations and Applications*, vol. 5095, pp. 1-16, 2008.

[88] J. O. G.K. Olsen, "Scenarios of traceability in model to text transformations," *In Proceeding European Conference on Model-Driven Architecture - Foundations and Applications (ECMDA-FA)*, vol. 4530, pp. 144-156, 2007.

[89] C. T. T.N. Nguyen, E.V. Munson, "On product versioning for hypertexts," *In Proc. International Workshop on Software Conguration Management (SCM)*, pp. 113-132, 2005.

[90] http://www.eclipse.org/atl/atlTransformations/.