MULTI-LAYER SYNTACTICAL MODEL TRANSFORMATION FOR MODEL BASED SYSTEMS ENGINEERING

A Dissertation
Presented to
The Academic Faculty

by

Ky-Sang Kwon

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
H. Milton Stewart School of Industrial and Systems Engineering

Georgia Institute of Technology
December 2011

MULTI-LAYER SYNTACTICAL MODEL TRANSFORMATION

FOR MODEL BASED SYSTEMS ENGINEERING

Approved by:

Dr. Leon F. McGinnis, Advisor
School of Industrial and Systems Engineering
Georgia Institute of Technology

Dr. Marc Goetschalckx,
School of Industrial and Systems Engineering
Georgia Institute of Technology

Dr. Charles Eastman,
Colleges of Architecture and Computing
Georgia Institute of Technology

Dr. Joel Sokol,
School of Industrial and Systems Engineering
Georgia Institute of Technology

Dr. Chris Paredis,
School of Mechanical Engineering
Georgia Institute of Technology

Date Approved: November 1, 2011

DEDICATION

To Lord who I will praise forever, I trust in your unfailing love for ever and ever.

And to my lovely wife, Sangmin, for her love, faith, prayer, and patience ….

ACKNOWLEDGEMENTS

First of all, I would like to thank my wife, Sang Min Lee. Without her, I could not have got through my long journey toward Ph.D. Throughout the journey, her prayer was the source of strength; her love was the source of joy; and her presence was the source of peace.

I also deeply appreciate the endless supports from my parents, parents-in-law, my brother, and brothers-in-law.

I would like to express my best gratitude to my advisor, Dr. Leon McGinnis. His precious insight, abundant knowledge, and right attitude to research inspired and enabled me to overcome all intellectual challenges. Whenever I struggled with my research, he has always encouraged me to keep my study, and guided me to the right direction with incredible patience. I sincerely appreciate all his supports. I am really honored and pleased to have Dr. McGinnis as my advisor.

I also want to give my special thanks to my dissertation committee members, Dr. Paredis, Dr. Goetschalckx, Dr. Eastman, and Dr. Sokol. Their valuable comments increased the quality of my dissertation. I learned a lot from their sharp intuition and research experience. Especially, I would like to appreciate the numerous valuable advices from Dr. Paredis who I have worked with for a long time.

Above all, I sincerely thank God, who has always guided my every step into His righteousness. In addition, I am also really grateful to all church members in SKBC. It was so wonderful to have such a great community in faith for God.

.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Systems engineering supports interdisciplinary decision making in design processes for complex systems by allowing engineers from different disciplines to have an integrated view of the target system. Unlike traditional document based systems engineering, model-based systems engineering (MBSE) uses various types of models to facilitate formal communication among the disciplines, which is essential for system level decisions.

In this dissertation, we propose a new model transformation approach to deal with this variety of models, which plays an essential role in MBSE. Up to now, model transformation has primarily been used to support development processes of software systems in the context of model driven engineering (MDE); applying the model transformation to MBSE, which deals with general systems, gives rise to a number of new problems. We indentify and focus on two key problems: instance data integration for virtually evaluating target systems, and syntactical inconsistencies among commercial engineering tools.

In order to address these two issues, we propose multi-layer syntactical model transformation by extending the standard model transformation methods and tools. We intuitively present the key concepts and the practical benefits of the new model transformation. This intuitive description is supported by theoretical discussion based on graph grammar theory. Finally, we demonstrate the new approach by implementing it in the optimization domain, which is a primary analysis domain of industrial engineering.

This dissertation is organized as follows.

Chapter 1 highlights the importance, and the motivations of this research. It also provides the detailed outline of the dissertation. Chapter 2 explains necessary background knowledge on formal modeling and model transformation. In chapter 3, we propose the new model transformation, the key ideas and concepts.

Chapter 4 provides the theoretic foundation of our multi layer model transformation. We find one necessary condition under which the new transformation is viable. The theoretical discussion is based on the mathematical formalism of model transformation, called triple graph grammar (TGG). We prove two key properties of

TGG; these properties play important roles in establishing the theoretical foundation of the multi layer model transformation, and implementing it in the following chapters.

The following two chapters demonstrate the proposed approach. For the demonstration, we use meta-modeling frameworks, and model transformation tools which are built upon the Eclipse development environment. In order to practically support the model transformation, we improve some parts of the tools; and we also propose new ways to use the existing functions.

Finally, we end this dissertation with the summary, the contributions, and the future research topics.

CHAPTER 1

INTRODUCTION

"A model is a simplification of reality" [1]. Human beings have limited ability to understand things in the world; models help them to interpret those things by formally defining aspects important for understanding and leaving out unimportant ones. People use various models to explain and verify their understating of various fields, ranging from economics to business and science.

Engineering is one of the fields that heavily use models. Engineers work with models in order to design and analyze a complex engineering system before they actually build the system. Mechanical engineers use mechanical drawings to design a car. Aerospace engineers test a miniature model of an airplane in a wind tunnel to evaluate the aerodynamic performance of the aircraft. Industrial engineers develop a number of mathematical models or simulation models to evaluate the operational performance of a plant.

Although there are a wide variety of models across engineering fields, contemporary engineering models have one thing in common; computers plays a growing role in their creation and use. Since computers allow engineers to manipulate and analyze models at low cost, increasing numbers of engineers develop their models using computers. As computing power has grown, the range of application and the level of fidelity of computer models also has dramatically increased; mechanical engineers are able to handle a full 3D CAD model of a car, and industrial engineers are able to run a full scale simulation model of a wafer fab with tens of thousands of entities.

The increasing importance of computer models has led to a lot of research aimed at developing modeling languages that help engineers build and use their models . This research has various streams: ontology, language formalism, domain specific language,

etc. Ontology engineering identifies a set of common and primitive constructs with which to represent knowledge of a target domain [2-4]. Language formalism is a linguistic foundation to formalize modeling languages so that both a human and a computer can unambiguously interpret models expressed in the languages. Domain specific languages allow domain experts to capture their knowledge so that it can be commonly used to describe and analyze systems in a specific domain; it enables modelers to accurately and efficiently develop the models of the systems in the domain.

Software engineering leads the way in formal modeling languages and associated methodology. Model Driven Engineering (MDE) has been suggested as a comprehensive modeling approach to develop software systems in heterogeneous platform environments [5]. Two key parts of MDE are the Unified Modeling Language (UML) and model transformation. UML allows software developers to build a model of a target software system that is independent from platforms on which the software system will be deployed. In MDE, the software system model is called a Platform Independent Model (PIM). Although the implementation independence of PIM allows the developers to have greater focus on the problem itself, a PIM is not an actual software program, which is the ultimate deliverable of software development. Model transformation generates executable software programs by converting a PIM to multiple Platform Specific Models (PSMs) that are compatible with the technical specifications of the software development platforms. The approach using PIM and model transformation increases productivity of software development by reusing the PIM across multiple platforms; promoting communication among development teams; and ensuring interoperability among their programs.

In contrast, other engineering domains have approached this topic in ad-hoc ways. Each engineering domain develops its own modeling concept. Even within the same domain, each tool for a given problem may have a different syntax to implement the modeling concept for various reasons: technical difficulties, or establishing an entry

barrier to competitors. The lack of common modeling language and the diversity of syntax hinder interoperability between engineering models and tools; engineers have developed ad hoc integration solutions using general purpose program languages, like Java, C++, etc. However, this is not a generic approach; that is, the integration programs can work with the specific combination of the models that they target. The possible number of the programs that are needed to integrate n tools would be O(n×(n-1)), if we need to support every combination of the tools.

There is a formal approach to the engineering tool interoperability. ModelCenter is a comprehensive tool where an engineering analysis process is captured in a formal flow chart-like model, and the engineering tools involved in the process are integrated through a standard interface, such as built-in interfaces with COTS tools, and wrapper interfaces for customer developed programs [6]. In spite of the formality, the standard interface does not support full integration in terms of model transformation of MDE. It is unable to generate a new model from an existing model of a different domain; it just enables existing models to exchange data. This means that models themselves are separately developed across different engineering tools. Moreover, ModelCenter does not have a central PIM; hence, the possible number of interfaces between n tools would be still O(n×(n-1)). ModelCenter lacks one of the important benefits of MDE, model generation from a PIM.

There have been efforts to tackle this problem by incorporating key features of MDE. MDE's language formalism can be a good base for a common modeling language. Also, model transformation can play the role of generic integration tool, substituting for the ad hoc integration interface. Moreover, the model transformation approach based on PIM can reduce the number of integrations. PIM is a central model from which multiple PSMs are generated. Since the conversions establish linkages with the PIM, the PSMs can be integrated via the central PIM without direct connections between PSMs. This

allows us to reduce the number of connections between n PSMs to O(n) because each PSM is linked to only the PIM [7].

In particular, the systems engineering community has made significant progress. Model Based Systems Engineering (MBSE), which is a new systems engineering approach based on models, plays a central role in conveying the key MDE concepts to systems engineering [8]; e.g., Cloutier has explored the applicability of MDA (Model Driven Architecture, OMG's version of MDE) to MBSE [9]. These efforts identified the need to extend the two key parts of MDE because systems engineering deals with general systems beyond software systems.

With regard to modeling languages, there is a significant achievement; OMG developed the System Modeling Language (SysML) by reusing some parts of UML2 and extending UML2 through the profile mechanism [10]. SysML has three new diagram types: requirement diagram, internal block diagram, and parametric diagram. Combined with the existing UML diagrams, these new diagrams allow engineers to describe, design, analyze, and verify various components throughout the systems engineering process.

There have been a few efforts to extend model transformation methodology for MBSE. For instance, [11] used graph model transformation to address consistency issue between the multiple models and views, one major problem in complex systems design. However, research on model transformation still approaches the topic largely from a software development perspective; there are few efforts addressing the application of model transformation to MBSE.

1.1 Motivations for This Research

The objective of this dissertation is to explore the extension of model transformation, which has drawn relatively less research attention from the systems engineering community, to Model Based Systems Engineering. In order to indentify model transformation issues that arise in systems engineering, we need to understand

how differently models (UML model for MDE, and SysML model for MBSE) and model transformations are used between software engineering (MDE) and model based systems engineering (MBSE).

In software engineering, both the UML models and the target systems, which are executable software programs, exist as software objects. As pointed out, model transformations in MDE link them; the model transformations convert the UML models into the executable software programs. Once the programs are generated, developers carry out processes for their verification . They do not need to use other analysis models because the target system itself is an executable software object. Therefore, the primary purpose of the UML models is to provide the necessary input to the process that will generate the target software systems.

In contrast, the development targets for systems engineering are systems designs. It is really expensive to make and test real objects compared to software systems; e.g., in order to make a real car, engineers need to build up production facilities and make production plans. Therefore, before actually building the target system, engineers have to evaluate and analyze the target system, the car, to make sure their design will meet requirements. For the evaluation and analysis, they use a wide variety of engineering analysis models; most of the engineering models are computer models that run in various engineering design and analysis tools. Productivity of the system development process, therefore, largely depends on the efficiency of generating and managing the computational analysis models. In this sense, SysML models in MBSE are used to generate the engineering models for subsequent evaluation and analysis, as opposed to UML models in MDE, which are used to generate executable software code.

In this dissertation, we indentify the two critical model transformation issues that this difference poses: instance data integration and syntactic inconsistency.

Instance Data Integration

Analysis and evaluation with engineering models require specific information about the particular system that a design team is developing; that is, engineers need to incorporate instance data into their engineering analysis models. For example, in order to decide how many machines a factory should have, the engineers should know future demand of the products the factory will produce and the performance specifications of the machines. If we use simulation to support the decision, the simulation model will represent machine elements with performance specifications like average process time or mean time to failure; i.e., the generic machine defined in factory domain libraries should be populated with the information specific to the factory being designed.

In order to support this requirement, a model transformation in MBSE has to deal with not only the generation of models, but also the integration of instance data for the generated model; e.g. for optimization analysis, a model transformation needs to not only generate a mathematical model as the general description of the problem, but also feed instance data into the generated model to specify the problem.

Syntactic Inconsistency

The computational evaluation of engineering models with instance data is usually performed using COTS (commercial off the shelf) tools. Since development of engineering analysis tools requires deep knowledge of execution methodologies and a high level of computer programming skill, it is common that the tools have been commercially developed by experts; e.g. CPLEX in optimization solvers, Arena™ in discrete event simulation, Nastran™ in FEM (Finite Element Model) solver, etc.

Since engineering communities develop COTS tools for their own purpose, the syntax of the various tools, for either models or instance data, are quite diverse. However, the traditional meta-modeling framework, in which model transformations are built, lacks the capability of coping directly with this syntactic diversity. This is because the model

transformation frameworks usually expect the permanent representations of the source and target models in the form of XMI (XML Metadata Interchange) [12]. Few contemporary engineering tools are able to read and write the XMI files syntactically. For this reason, many implementations of model transformation use injectors or extractors for pre processes or post processes, which sort out the syntactical incompatibility between the XMI and the tools. The injectors and extractors are usually developed in ad-hoc ways using a general purpose programming languages, like Java, or C++.

For more elaborate discussion of these two issues, we formally define them in terms of layered language formalism in chapter 2.

1.2 Dissertation Outline

Chapter 2 provides basic concepts of layered language formalism and the formal descriptions of the two key problems. The background knowledge about the layered language formalism is essential to understanding the modeling and model transformation issues throughout the rest of this thesis. Section 2.1 gives an explanation of the key concepts: linguistic meaning of layer, linguistic relation between the layers, and practical purpose of the layers. In Section 2.2, we discuss the key idea of model transformation in the context of the layered language framework. Furthermore, we formally define a model transformation and its execution using mathematical notation for concise discussion throughout the rest of this dissertation. In Section 2.3, the key problems are formally described in terms of the layered language formalism and the notation so that we can have clear and common understanding of the problems.

In Chapter 3, we suggest Multi Layer Syntactic Model Transformation as a comprehensive approach to solve the two key problems. This consists of two key parts: multi layer model transformation, and separate syntactical model transformation. We briefly and intuitively explain these two parts.

Section 3.1 covers the separate syntactical model transformation to solve syntactical inconsistency. We introduce a domain semantic model, which semantically captures the common concepts of a modeling domain (e.g., optimization, simulation, etc) without any consideration of syntactical representation. The separate syntactical model transformation converts this generic semantic model into concrete syntactical models of various COTS tools (e.g. AMPL, GAMS, LINDO, etc for optimization [13-15]) based on the common concepts.

Section 3.2 describes the key idea of multi layer model transformation and how it addresses the problem of instance data integration. A normal model transformation is able to handle only models without instance data. In contrast, the suggested multi layer model transformation incorporates the integration of the instance data by generating model transformation rules for the conversion of the instance data syntax. For the generation, we do not develop any special program. Instead, we use an existing tool of MDE, a model transformation, in a novel way; it is a special type of model transformation in that it has to deal with the transformation rules themselves as the target model unlike a normal model transformation. Higher-order model transformation (HOT) was introduced for this purpose [16]. We apply the HOT to our context.

We suggest two alternatives: a direct approach and a correspondence model-based approach. They use HOT in two different ways to generate the model transformation rules. We compare pros and cons of the two approaches.

In section 3.3, we discuss how well our approach incorporates a model transformation for engineering design by indentifying its practical benefits: reducing complexity of model transformation, and sharing domain specific concepts between different tools within the modeling domain.

In chapter 4, we prove multi layer model transformation is theoretically valid using graph grammar theory, the mathematical formalism of model transformation. In section 4.1, we introduce key definitions and theories of graph grammar related to model

transformation. In section 4.2 and 4.3, we focus on TGG (Triple Graph Grammar) since it provides the theoretical foundation for many model transformation tools [17] . In section 4.4, and 4.5 we prove two key properties of TGG to show theoretical viability of the multi layer model transformation, and we discuss under which condition we can use this new approach.

In chapter 5 and chapter 6, we suggest practical implementations of our approach. In order to implement solutions, we identify technical issues that arise with existing tools. We solve the issues by extending the existing tools or suggesting new ways to use the tools. Throughout the dissertation, we use two example modeling domains: MS-ACCESS^{TM} (a relation database (RDB) tool), and AMPL (an optimization modeling language). We take these two examples because of their importance in industrial engineering; RDB plays an important role as a data source; and optimization is one of the most frequently used analysis methodologies in industrial engineering.

Chapter 5 covers the syntactical model transformation that addresses the syntactic inconsistency problem. In section 5.1, we introduce the scenario in which we implement our demonstration with RDB and optimization domain. In section 5.2, we analyze the target modeling domains in terms of the layered language formalism so that readers can have common understanding of the linguistic characteristics of the domains. In the following section, we review and compare the existing approaches for the syntactical problem. Among the approaches, we take hybrid modeling approach, where a semantic model and a syntactic grammar are formally defined together, as our basic approach since it can support bidirectional integration between a semantic model and a textual representation, which is essential to engineering tool integration. We explain elaborately how the hybrid approach solves the syntactic problems. As one tool for the hybrid approach, we introduce Xtext, an EBNF (Extended Backus–Naur Form) based meta-modeling tool [18]; we use the tool for the optimization domain. In addition, we use a different approach based on XSD (XML Schema Definition) for the RDB domain. We

view the XSD based approach as a special case of the Xtext based approach. We discuss the advantages of the XSD based approach and the condition under which we can use the approach.

In chapter 6, we implement multi layer model transformation, specifically the correspondence model-based approach. The key idea of the correspondence model-based approach is to generate model transformation rules for instance data from the result of another model transformation through HOT. For the implementation, we first develop the meta-model of the correspondence model independently of model transformation tools so that we can reuse the meta-model. Second, we specify the syntactical representation of the correspondence model in the syntax of ATL (ATLAS Transformation Language) [19]. The HOT is defined between them in a way that generates an executable ATL script that integrate the instance data.

In chapter 7, we conclude the dissertation. In Section 7.1, we make a summary of our research. In Section 7.2, we discuss future research topics.

CHAPTER 2

BACKGROUND AND PROBLEM DEFINITION

In this chapter, we introduce essential background knowledge for the research. First of all, we explain the OMG four layer language formalism because it is the fundamental basis of formal modeling framework on which this research is developed. Second, we introduce key ideas of a traditional model transformation: how to define transformation rules and how they are executed. Finally, we formally define the two key issues identified in chapter 1, in term of the layered language structure.

2.1 Layered Language Formalism

The layered language formalism describes a model in accordance with semantics of a meta-model. The meta-model is a model of a model; that is, it is a model that defines semantics that is used to describe another model. A language layer is defined by the relation between the meta-model and the model described by the semantics of the meta-model; the upper layer is the meta-model, and the lower layer is the model.

The relation is the same as that between class and object in object-oriented modeling. One calls the relations between the layers ‘instance of’ or ‘instantiation.’ In the language formalism, the relations are renamed ‘conform to’ or ‘conformance.’ These new names highlight the linguistic perspective; the models of any layer can be said to be described in terms of the languages defined in the next higher layer.We use the two terms – ‘instance of’ and ‘conform to’ - interchangeably.

Modeling language designers can recursively use the relation as many times as they want. OMG (Object Management Group) proposes a four-layer architecture to support MDA (Model Driven Architecture), OMG’s version of MDE [20]. The four layer architecture is widely accepted by other modeling domains. We also discuss our topic in

the context of the four-layer architecture. The following section gives brief explanation of the four-layer architecture.

Four-layer architecture

Figure 1 depicts OMG's four layer architecture. The top layer, M3, is a meta-meta-model, the Meta Object Facility (MOF) [21]. It is interesting that MOF has self-descriptive capability; that is, MOF can describe itself. This is why MOF can be a starting point of language formalism. Indeed, all of OMG's modeling standards (UML, SysML, CWM, etc) are specified in terms of MOF [10, 21, 22]. This common linguistic foundation enables the modeling standards to be compatible with one another [23].

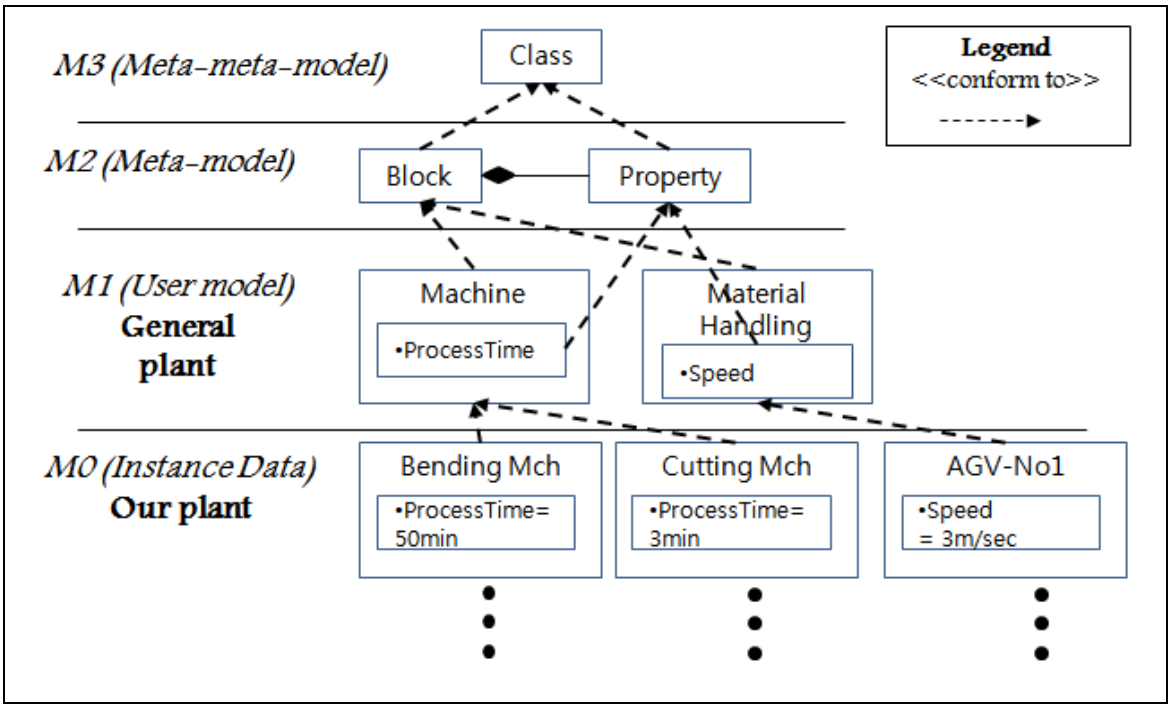

Figure 1. Four-layer Architecture of OMG

The primary purpose of the M2 layer (meta-model) is to define formal languages that support modeling standards (e.g., UML, CWM, RDBM, and SysML). Each modeling standard has its own modeling tools according to the primary purpose of the methodology. For example, the primary purpose of MDE is software development. Therefore, UML, which is a modeling language of MDE, has a number of diagrams that allow users to

specify software systems. For instance, the sequence diagram of UML enables software engineers to describe the logical flow of a software program. SysML, a modeling language for MBSE, has an internal block diagram (IBD) which allows system engineers to specify the internal structure of a block by describing physical or logical connectors between blocks via ports [10].

In the M1 layer, domain experts develop user models in order to capture various aspects of systems in their domain of interest using the formal language of the modeling standards defined in the M2 layer; they identify and describe the system's structure and behavior in the user model. Note that the user model captures general knowledge applicable to any system in the domain; that is, it does not contain any information specific to a particular system so that others can reuse the models for designing similar systems in the same domain. For example, user models can express that workstations of a factory have a number of machines. However they should not specify the number of machines in the workstations.

The system specific information is defined at the M0 level, which is the instance data layer. In this layer, engineers and designers of a particular system put together the domain specific modeling concepts, captured in M1, and specify them to describe the system. For example, the instance data represents a specific factory, not a generic factory; i.e., M1 describes that a factory has a number of machines, whereas M0 specifies how many machines the specific factory has and the performance of each machine.

Throughout this dissertation, we use M0, M1, M2, and M3 to refer to the four OMG language layers.

2.2 Model transformation

There are a number of standards and implementations of model transformation [24-26]. In spite of the diversity, they have a common way of defining and executing the model transformation. Figure 2 depicts this; a model transformation is defined by

mapping rules between source meta-model (MM_S) and target meta-model (MM_T); and it generates target models (M_T), which conform to MM_T, from source models (M_S) conforming to MM_S. We hereafter use the figure as a visual representation of a model transformation.

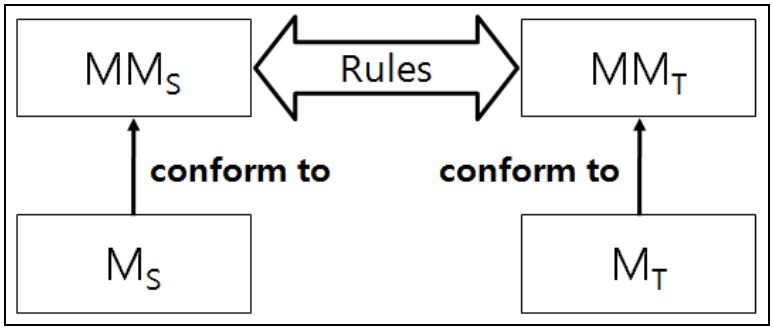

Figure 2. Model Transformation

We define a formal notation for the model transformation model, which defines the rules of the model transformation using mapping rules between source meta-models and target meta-models.

Definition 1. ***(Model Transformation Model)***

A model transformation model is a 3-tuple $MTM<MM_S, MM_T, MR>$, where MM_S is a set of source meta-models, MM_T is a set of target meta-models, and MR is a set of mapping rules between MM_S, and MM_T. In addition, MR is semantically compatible with MM_S, and MM_T.

We do not define the semantic compatibility of MR. We informally say that MR is compatible with MM_S, and MM_T, if all mapping rules in MR are described using the components of the meta-models in MM_S, and MM_T. We will formally define the compatibility in chapter 4, in which we will discuss the theory of model transformation.

Model transformation uses the model transformation model to convert source models to target models. Figure 3 and Definition 2 show the visual representation and the formal notation of the model transformation (MT), respectively.

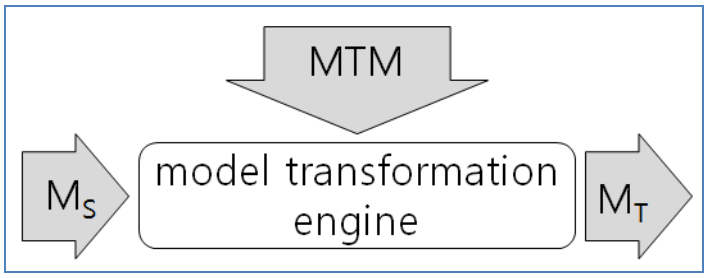

Figure 3. Execution of Model Transformation (MT)

Definition 2. ***(Model Transformation)***

A model transformation is a combination of a 2-tuple and a set of models, *MT(<MTM*, M_S> → M_T) where *MTM* is a model transformation model defined by Definition 1, M_S is a set of source models, each conforming to some meta-model $MM_x \in MM_S$, and M_T is a set of target models, which are generated by the execution of *MT*, and each of which conforms to some meta-model $MM_y \in MM_T$.

Relative meta-model vs absolute meta-model (M2)

We use two different concepts of meta-model: an upper layer model in a conforms-to relation between two models; and the M2 layer in the OMG four-layer architecture. The former concept is relative in that the conform-to relation may be recursively applied to all layers. Any layer can be a meta-model of the one-level down layer; i.e., M3 is the relative meta-model of M2, M2 is the relative meta-model of M1, and so on. In contrast, the latter concept is an absolute layer, which is the third layer (M2) from the bottom layer, M0, in the four-layer architecture. In order to prevent the confusion between them, we introduce the following two notations.

Definition 3. ***(Relative Meta-model)***

A relative meta-model of a model *M* is *RMM(M)*; *M* conforms to *RMM(M)*. Inverse notation RMM^{-1} is also defined to refer to the reverse direction.

Definition 4. ***(Absolute Layer Function)***

Function *AL* returns the absolute layer of a meta-modeling component – model, meta-model, model transformation model, and model transformation; i.e., if a model *M* is in the *k* th layer from the bottom layer (M0), *AL*(M) = *k*-1.

By the definition, we can easily say the following:

Proposition 1. If M_x = RMM(M_y), then *AL*(M_x) = *AL*(M_y) + 1.

Proposition 2. If a model transformation, $MT(<MTM, M_S> \rightarrow M_T)$, is valid, then *AL*(MT) = *AL*(M_S) = *AL* (M_T) = *AL*(MTM) -1.

Proposition 2can be restated that in a valid transformation, two "layer" relationships must be satisfied—(1) the source and target models are on the same layer, and (2) the transformation model is one layer up from the source and target models.

2.3 Formal definition of the key issues

In this section, we formally define the two key issues –instance data integration, and syntactical inconsistency - in terms of the layered language formalism. Figure 4 depicts the issues between PIM (Platform Independent Model) and an engineering tool (Tool-1).

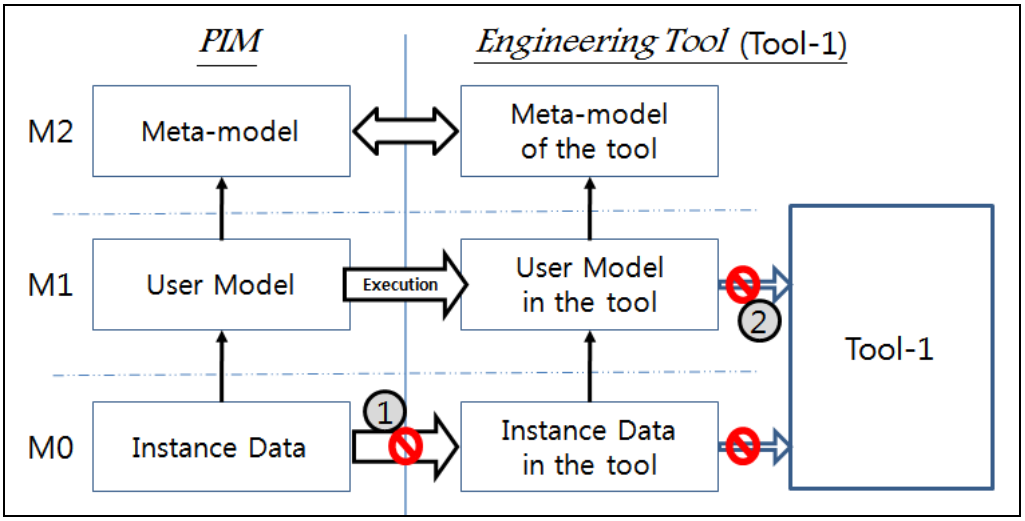

Figure 4. Key Issues

The first issue is that a normal transformation cannot directly deal with conversion of instance data in M0. As discussed in the previous section, a model transformation works between two language layers. The traditional way to use a model transformation is to define the rules in M2 and execute them for M1; $MT(<MTM, M_S> \rightarrow M_T)$, where AL(MTM) = 2 and AL(MT) = 1. Since the instance data is in M0, the model transformation cannot be applied to the instance data; the transformation model is defined in M2 but the data model is in M0, violating Proposition 2.

The second issue is that the target engineering tool cannot read the generated models and instance data because of the syntactic inconsistency between the transformation engine and the engineering tool. In Figure 4, the user model and the instance data conform to their relative meta-models respectively. However, this can assure only the semantic conformance because a normal meta-modeling framework offers means of defining only semantics of the meta-models [21, 27]. The syntax of the models and the instance data are not explicitly defined. The meta-modeling framework by default stores them in the form of XMI, which is OMG's XML standard for storing and exchanging models. Unfortunately, few engineering tools support XMI.

In order to discuss the syntactical issue, we need to elaborate the definition of a model. We introduce the following extended definition to explicitly show the syntax of a model.

Definition 5. ***(Model with modeling domain and syntax)***
A model is a 2-tuple *M(d, syn)*, where *d* is a modeling domain, *syn* is a syntax in which the model is written. If *syn = XMI*, it can be omitted; i.e., *M(d)*

According to Definition 5, 'user model in the tool' in Figure 4 can be expressed as *M*(Tool-1, XMI) or *M*(Tool-1).

2.4 Conclusion

We reviewed language layer as a key concept in the meta-modeling framework. The language layer plays important roles in theory and practice of model transformation;

that is, model transformation is defined and executed over two consecutive language layers. the instance data layer, which we focus on, is also formally defined in terms of OMG's four-layer language formalism. Throughout this dissertation, the language layer is essential to understanding our key discussions.

In addition, we introduced a number of formal notations for the key meta-modeling concepts that we use in this dissertation. These notations clarify the definitions of the concepts, which are ambiguously used in the research literature. The notations introduced here are unique in two senses. First, they make distinction between concepts that bring about confusion: model transformation model versus model transformation; and relative layer versus absolute layer. Second, they add modeling domain and syntax to the definition of model. This extended definition plays an important role in defining a new conform-to relation in the next chapter.

The language layer concept and those notations allow us to formally describe the two key issues of this dissertation. In particular, Proposition 1 and Proposition 2 show that the traditional model transformation works well between two consecutive layers. This prevents us from using the model transformation for instance data integration because the instance data (M0) is two levels down from the meta-model (M2) where a model transformation model is usually defined.

Nevertheless, relative meta-model, Definition 3, casts light on a solution to model transformation that can be used across more than two layers. According to the definition, we can apply model transformation to any language layer; i.e., M1, or M0. If two model transformations in two different language layers work together in a way that the upper model transformation manages the lower one, we could develop the model transformation that works over three layers. This idea underlies multi layer syntactical model transformation we will propose in the following chapter.

CHAPTER 3

MULTI LAYER SYNTACTIC MODEL TRANSFORMATION

In this chapter, we suggest an overall framework to extend standard model transformation in two directions: a horizontal extension for the syntactical inconsistency issue, and a vertical extension for the instance data integration issue. We discuss the concepts behind these two extensions and what practical benefits they provide in terms of managing engineering models.

3.1 Syntactic model transformation based on domain semantic model

Figure 5 shows the overall picture of the syntactical issue with an engineering tool domain. An engineering tool domain is a set of tools which have common purpose and modeling concepts. For example, the optimization domain is a set of tools that find optimal solutions of an engineering decision problem based on set-oriented modeling. The RDB domain is a set of database tools that store engineering data using table-oriented modeling. However, object-oriented simulation tools and process-oriented simulation tools are not in one engineering tool domain because those simulation tools have different modeling concepts in spite of their common purpose.

The domain semantic model plays a central role; it describes the target system in terms of the common modeling concepts for the engineering tool domain. The common modeling concepts are the basic conceptual building blocks with which the target systems are described. By the definition of engineering tool domain, the engineering tools of the domain are developed on the basis of these concepts, the tools have semantically similar structure regardless of their syntactical representations; e.g., optimization modeling languages - AMPL, GAMS, and LINDO [13-15]- have similar set-oriented concepts in spite of differences in syntactical details. The domain semantic model is intended to isolate the common semantic representation from tool dependent syntax.

Table 1 shows the notation that we use for clear discussion throughout this chapter.

Table 1. Notations

Notations	Description
TD	Target engineering tool domain
Tools(TD)	A set of tools in TD
T_i	A tool of TD; $T_i \in$ Tools(TD)
Syn(T_i)	The syntax of tool T_i
Syn(T_i,*l*)	The syntax of tool T_i in layer *l*
DSM(TD)	Domain semantic model of TD

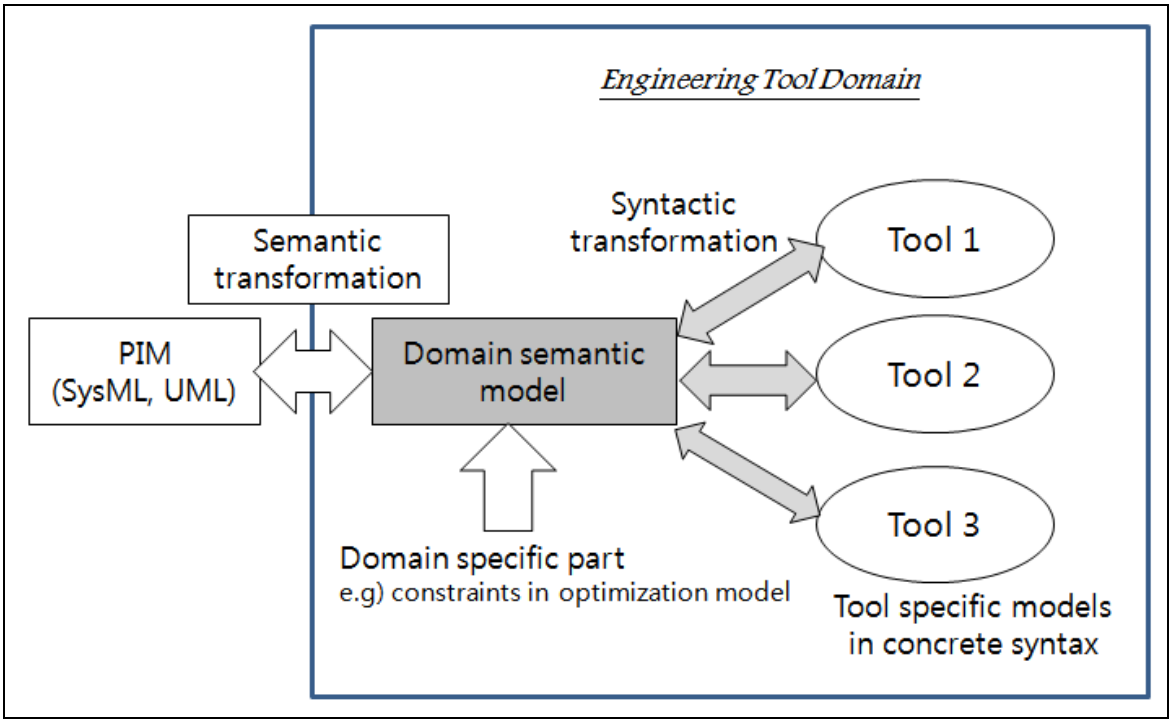

Figure 5. Key Idea of Horizontal Extension

As shown in Figure 5, the domain semantic model has two roles: a repository for modeling domain specific components and a coupling between semantic model transformation and syntactic model transformation.

First, as a repository, the domain semantic model contains the tool domain specific semantic components. For example, mathematical constraints are a component of the optimization modeling domain. If we put the constraint semantics into the PIM, PIM

will become a large and complicated union of semantic components of both the application domain and all possible engineering tool domains. We suggest putting the mathematical constraints into optimization semantic model, *DSM*(Optimization), instead. This enables us to keep PIM as concise as possible; that is, PIM can capture only common components that are used across all domains. In spite of the separation, we do not lose the primary benefit of MDE, integration through the PIM, in that *DSM*(Optimization) is shared by all tools in the optimization domain.

Second, the domain semantic model allows the transformation process to be divided into two parts—semantic and syntactical. The first part, semantic model transformation, converts PIM to *DSM*(*TD*), where *TD* is the target engineering tool domain. Both PIM and *DSM*(*TD*) have their own modeling concepts; i.e., UML domain describes an object as 'class', whereas optimization domain uses 'set' to represent the object. This semantic model transformation deals with the conceptual difference between the two modeling domains. It is not concerned with any syntactic aspects; thus, a normal model transformation approach can be used for semantic model transformation.

Syntactic model transformations, however, are a different matter. Syntactic model transformations bridge between an analysis domain semantic model and the concrete syntactic models of specific tools within the analysis domain; that is, these transformations associate concrete syntax with the semantic concepts. This is the key part of the horizontal extension to handle diverse syntactic representations.

Prior research has approached this topic in different ways. We will survey the approaches in chapter 5 in detail. In particular, [28] introduced a comprehensive tool interoperability framework based on AtlanMod Model Management Architecture (AMMA) [29]. The key part of this approach is an intermediate model that contains the common concepts and features shared by the target tools. This is a similar concept to our domain semantic model.

In spite of the similarity, our approach is differentiated from the approach in [28] in that we take into account multiple layers: the user model layer and the instance data layer. In our approach, the syntactical model transformations are defined in a way that handles syntactical specification in both layers.

Proposed syntactical model transformation

We propose an approach of syntactical model transformation based on a meta-model which specifies models not only semantically but also syntactically. In order to make the definition of the complete meta-model clear, we introduce a concept of meta-syntax. The meta-syntax is analogous to the meta-model of the meta-modeling framework. Remember that a meta-model is a model of model. Likewise, a meta-syntax is a syntax of syntax; the meta-syntax is a syntax that defines another syntax. We call the relation between the meta-syntax and the syntax, defined in the meta-syntax, syntactical conform-to. Note that the meta-syntax is also a relative concept like the meta-model.

Definition 6. ***(Meta-syntax)***

A meta-syntax of a syntax *S* is *MS(S)*; *S* syntactically conforms to *MS(S)*.

The complete meta-model is a combination of the usual semantic meta-model and the meta-syntax; it defines the semantics of a model through the meta-model, and the syntax that describes the model through the meta-syntax. The EBNF-based hybrid approach, which we will discuss in chapter 5, enables us to define the complete meta-model by incorporating the meta-syntax into the meta-model by means of a modified EBNF-language.

Definition 7. ***(Complete meta-model and complete conform-to relation)***

A complete meta-model of model M(d, *Syn*(T_i)) is 2-tuple *CMM*(M(*d*, *Syn*(T_i)) = (*RMM*(M(d)) , *MS*(*Syn*(T_i))), where *d* is a engineering tool domain, and T_i is a tool of *d* (i.e., $T_i \in$ Tools(d)). We say that M(d, *Syn*(T_i)) completely conform to *CMM*(M(*d*, *Syn*(T_i)).

Figure 6 depicts the visual representation of the complete meta-model and the complete conform-to. Note that the complete meta-model is technically written in one language of the EBNF-based approach.

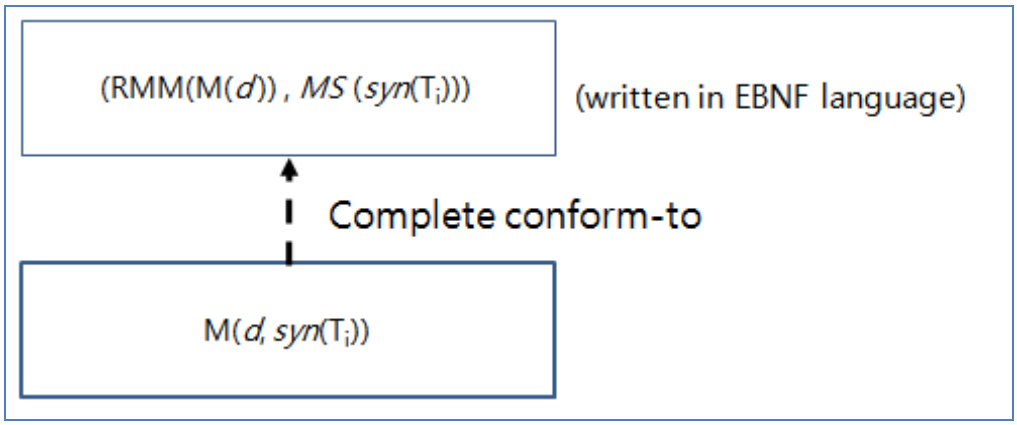

Figure 6. Complete Meta-model

Figure 7 shows how our approach deals with the syntactical model transformation over two layers. It consists of two parts, which take care of the syntactical representations of the target tool in M1 and M0, respectively.

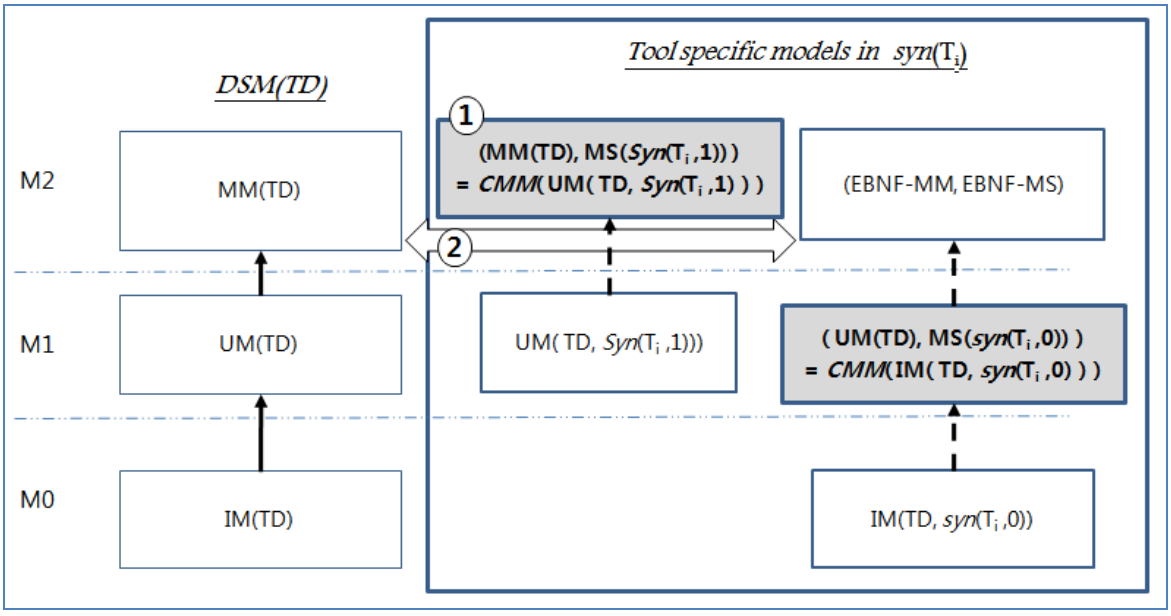

Figure 7. Syntactical Transformation over Multiple Layers

At the first step, we construct a complete meta-model of user model in the syntax of tool T_i using the EBNF approach. To do this, we add the meta-syntax of the tool syntax in M1, *MS*(*syn*(T_i, 1)), into the existing meta model of DSM(TD), MM(TD). This leads to 2-tuple (MM(TD), *MS*(*syn*(T_i, 1))).

Proposition 3. (MM(TD), *MS*(*syn*(T_i, 1))) = *CMM*(UM(TD, syn(Ti, 1))).

Proof) As shown on the side of *DSM*(TD), UM(TD) conforms to MM(TD); that is, MM(TD) = RMM(UM(TD)). This means (MM(TD), *MS*(*syn*(T_i,1))) = **(**RMM(UM(TD)) , *MS*(*syn*(T_i,1))**).** By Definition 7, **(**RMM(UM(TD)) , *MS*(*syn*(T_i,1))**) =** *CMM***(**UM(TD**,** syn(Ti, 1)**))** □

This proposition implies that an EBNF tool generates UM(TD**,** syn(Ti, 1)**)** in M1, which is the user model written in the syntax of tool T_i. This is what is required to use the target tool in M1.

The novel part of our approach is the second step. Unlike the first step, we generate the necessary complete meta-model rather than construct it. More specifically, we define a model transformation that generates the complete meta-model for the instance data written in the tool syntax, IM(TD, *syn*(T_i,0)). As shown in Figure 7, the model transformation model is defined between MM(TD) and CMM(EBNF-MM, EBNF-MS), where EBNF-MM (or EBNF-MS respectively) is the semantic meta-model (or the meta-syntax respectively) of the EBNF language. Since the target meta-model is the complete meta-model of the EBNF language itself, which is defined by the developers of the EBNF language, the target model of the model transformation is grammatically an EBNF model; the target model therefore technically can be a complete meta-model of another model.

The conversion rules of the model transformation are defined in a way that duplicates the semantic information of the source model, UM(TD) and defines the meta-syntax of the tool syntax for instance data, MS(syn(T_i ,0)); the target model of the model

transformation is (UM(TD), *MS*(*syn*(T_i, 0))). Formally, the model transformation can be expressed as follows:

MT(<*MTM*(MM_S,MM_T,*MR*), { UM(TD) }> → {UM(TD), *MS*(*syn*(T_i, 0))}),

where MM_S = {*MM*(TD)}, MM_T = { *CMM*(EBNF-MM, EBNF-MS) }.

Proposition 4. (UM(TD), *MS*(*syn*(T_i, 0))) = *CMM*(ID(TD, syn(Ti, 0))).

Proof) It is same as Proposition 3. □

This proposition implies that an EBNF tool generates ID(TD, syn(Ti, 0)), which is the ultimate goal of the syntactical model transformation.

3.2 Multi Layer Model Transformation

Figure 8 shows the traditional way to use a model transformation to address the instance data integration issue. As discussed in chapter 2, the model transformation works between the consecutive language layers. It can be applied to any two consecutive layers because of the relative characteristics of the meta-model. The natural approach to deal with instance data in M0 is to define a model transformation model in M1, which is denoted by MTM(1). In consequence, we need to separately construct MTM(2) and MTM(1), which convert the user models in M1 and the instance data in M0 respectively.

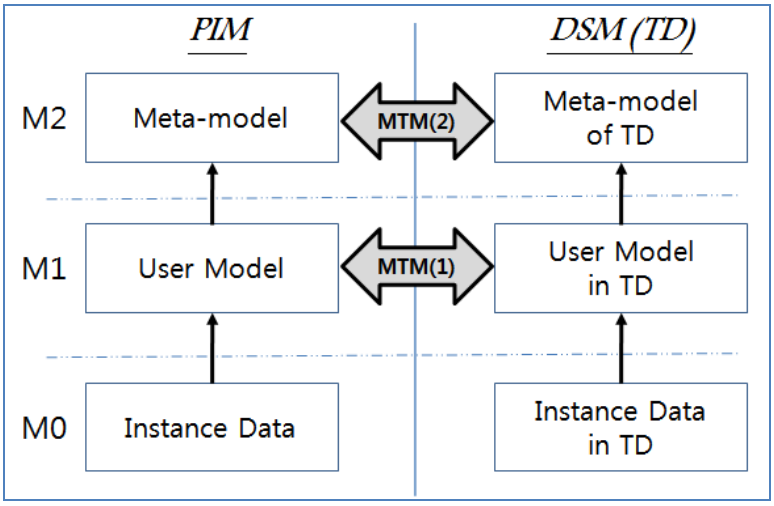

Figure 8. Traditional Approach for Instance Data

As the vertical extension, we suggest a new approach in which we generate MTM(1), instead of constructing it. More specifically, we use a special type of model transformation for the generation of MTM(1). The advantage of our approach over the traditional approach is that we can accommodate changes in user models without human effort to modify MTM(1). In the tradition approach, we need to manually revise MTM(1) for any changes in the user models because MTM(1) is manually constructed. However, our approach allows us to accommodate the changes by regenerating MTM(1) though the special model transformation.

Brief Introduction of Higher-order Model Transformation

A key aspect of our approach is the use of a special type of model transformation, which is called high-order model transformation (HOT). We briefly explain how HOT generates other model transformations. We will present more details about HOT in chapter 6.

HOT is a model transformation for a model transformation [30]. In other words, it is a special type of model transformation to handle model transformation rules as either the sources or the targets. Particularly, in our approach, HOT is used to generate MTM(1).

This concept diverges from standard model transformations, which usually have models as the source or target for the transformation; it might be expected that extensions to the normal model transformation would be required. However, HOT does not require any special extensions, but rather simply changing the perspective on model transformation. The change is to view a model transformation rule itself as a type of model.

HOT considers a model transformation rule as a model, specifically, a transformation model. It deals with the transformation rule just like a normal model using the current modeling infrastructure. It defines the meta-model of the transformation rule using a normal modeling language; the meta-model is used as either a source meta-model

or a target meta-model to define a model transformation model in a normal way; a normal model transformation engine, in turn, can read or generate the transformation rules through the meta-model.

Two HOT-based approaches

The key idea of the multi layer model transformation is to generate MTM(1) through a HOT defined in M2. In order for the generated model transformation model to be valid, we need to verify the model transformation model in two senses: rule validity and syntactic validity. The first, rule validity, assures that the generated transformation describes the transformation rules we intend it to construct in M1. The second, syntactic validity, requires that the generated transformation conforms to the syntactic requirements of the model transformation tool we work with.

We suggest two different approaches to ensure the validity of the HOT generated transformation: a direct approach, and a correspondence model based approach. We discuss how both approaches work and their pros and cons.

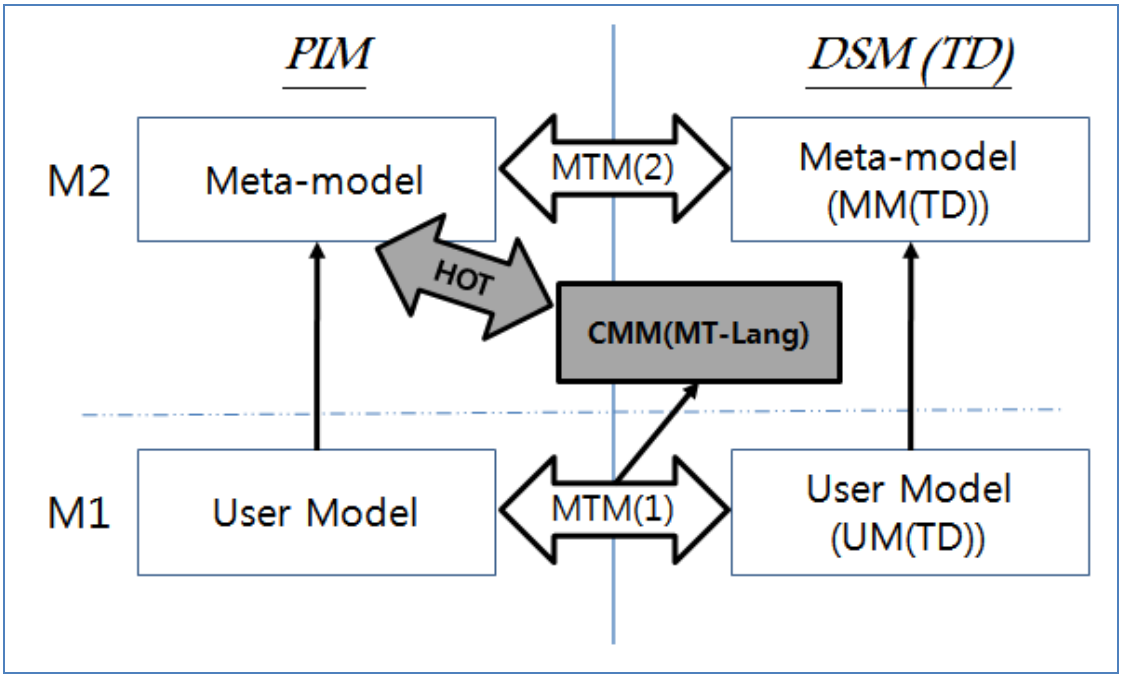

Figure 9. Direct Approach

In the direct approach, we ensure validity using HOT. Figure 9. shows the HOT, which is developed between the PIM meta-model and the complete meta-model of model transformation language, CMM(MT-Lang), with which we describe MTM(1). MT-Lang denotes the model transformation language. The complete meta-model of MT-Lang is usually defined by the developers of the transformation language. Note that the HOT is a different model transformation from MTM(2) for the user model conversion.

For rule validity, the HOT needs to use semantic information of user models in M1. Suppose that we want to construct mapping relation between component 'A' of PIM and component 'B' of the target domain in M1 layer. In order to generate the rule using the HOT, the HOT needs to know that those components exist in the user models. This means the HOT has to use semantic information of both user models: the PIM user model, and the DSM(TD) user model (UM(TD)). As shown in Figure 9. , the HOT takes the semantic information from the PIM user model, the source model of the HOT. Although it does not take UM(TD), the semantic information of UM(TD) can be generated from the PIM user model if needed. This is because UM(TD) is generated from the PIM user model through MTM(2).

For the syntactic validity, the HOT needs to write the generated rules in the valid syntax of the model transformation language. In order to do that, the HOT takes the complete meta-model of the model transformation language, CMM(MT-Lang), as the target meta-model - by the definition of HOT, it should do so. The HOT puts the semantic information together in terms of CMM(MT-Lang) in a way that describes the rules between the semantic components of the user models. We can formally express the execution of the proposed model transformation as follows:

MT(<*MTM*(MM_S,MM_T,MR), { PIM user model }> → { MTM(1)}), where MM_S = { PIM meta-model}, MM_T = { CMM(MT-Lang) }.

The direct approach has pros and cons. As pointed out, the HOT introduced by the direct approach is defined independently from MTM(2) for the user model conversion.

On one hand, the independence allows high expressive power of the model transformation model that the HOT generates. Since we can independently manipulate the conversion rules of the generated transformation using the semantic conversion of the HOT, we can have full flexibility in expressing the rules; e.g., we can make a mapping of an arbitrary pair of components, and put complicated logical expressions on the generated rule. On the other hand, the independence may give rise to a compatibility issue with the output user model that the normal model transformation generates. As shown in Figure 9. , the target meta-model of the generated model transformation model, MTM(1), is the DSM(TD) user model. The user model is generated through MTM(2), while MTM(1) refers to the user model through independently generated semantic information in the HOT. Therefore, some inconsistency may occur between them. For example, the HOT could generate a wrong name for a semantic component of the user model. A human has to ensure the compatibility. This means when one modifies one of MTM(2) and the HOT, a human has to revise the other in a way that maintains compatibility. This makes the maintenance in this approach difficult.

In order to address this shortcoming, we suggest another approach based on a correspondence model. The key part of this approach is to use the correspondence model as the conversion rules of MTM(1), which we manually define in the HOT in the previous approach.

The correspondence model represents the result of an execution of a model transformation as associations between the source models and the target models. In other words, it tells us about what component of the source models are converted to what components of the target models; it maintains traceability between the source models and the target models.

Definition 8. (Correspondence model)

Correspondence model, *CM*, consists of a set of correspondence associations, where a correspondence association is a link between components of a source model and the target components that are generated from the source.

Correspondence associations are used as the rules of MTM(1); a correspondence association is interpreted as a mapping relation in the M1 layer. In M0, the mapping relation transfers the instance data of the source components to instance data of the target components that are linked to the source components through the correspondence association.

Figure 10 illustrates how to generate the correspondence model from MTM(2). Unlike the previous approach, MTM(2) of this approach has an additional output, which is the correspondence model (CM). In order to incorporate the correspondence model in to MTM(2), we define the meta-model of the correspondence model, RMM(CM). We developed RMM(CM) in a way that is independent of the target domain (TD) and the model transformation language, MT-Lang. This independence allows us to reuse RMM(CM) for any engineering tool domain and model transformation tool. We will discuss how to design RMM(CM) in chapter 6. Definition 9 redefines MTM(2), which has one source meta-model and two target meta-models.

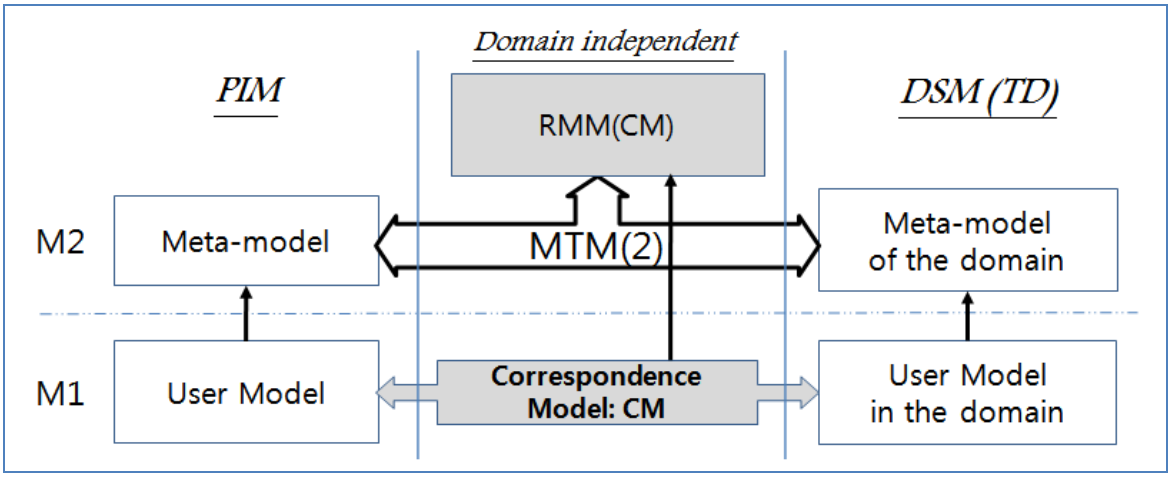

Figure 10. Generation of Correspondence Model

Definition 9. MTM(2) of the correspondence-based approach $MT<MM_S,MM_T,MR>$, where MM_S = {PIM meta-model}, MM_T = { M(2,TD) , *RMM*(CM)}

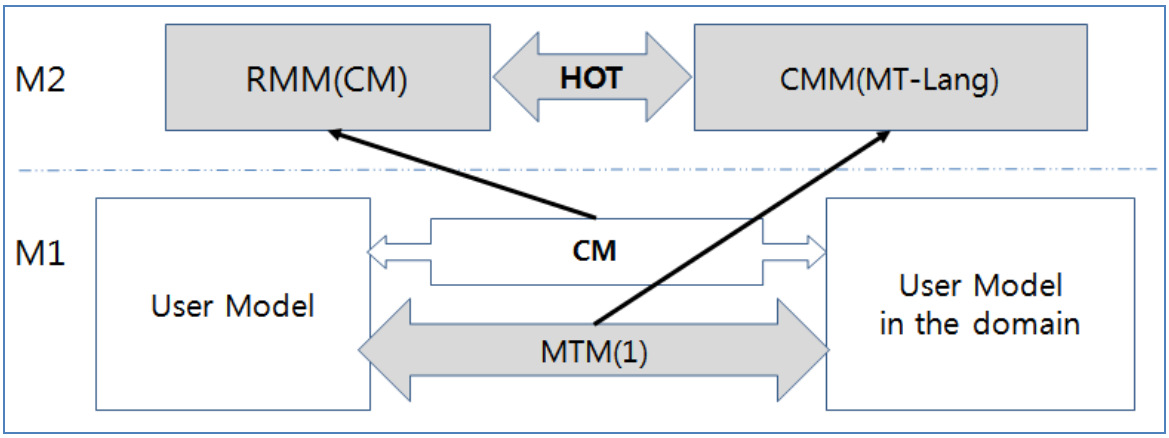

Figure 11. HOT of Correspondence Model-based Approach

This approach uses HOT to convert the correspondence model to MTM(1). The HOT takes the correspondence model as the source model (See Figure 11). Since the correspondence model is used as the conversion rules of MTM(1), all the HOT has to do is to grammatically convert the rules into a valid model transformation script. In order to create the valid script, the HOT use the complete meta-model of the transformation language, CMM(MT-Lang), as the target meta-model like the previous approach. This makes sense in that the ultimate outputs are same.

In this approach, the HOT takes care of only the grammatical compliance of MTM(1). This makes the HOT much simpler in the correspondence model approach than in the direct approach. Furthermore, this makes the HOT more generic; that is, the HOT can be reused for different engineering tool domains. Since the HOT addresses only grammatical aspects, we can use the HOT as long as the grammatical conversion remains same – the model transformation tool does not change,

In spite of the benefits, this approach has limited applicability because it has less expressive power than the direct approach. In this approach, the associations of a

correspondence model are used as mapping rules. We therefore can use this approach in cases where the rules that the correspondence model express are what we want to apply to instance data.

The example of Figure 12 illustrate how the correspondence can be used as the mapping rules of MTM(1). The example is a model transformation between object-oriented (OO) modeling domain and relational database (RDB) modeling domain. As shown in the figure, the 'machine' class is converted to the 'machine' table as the result of a model transformation in M1; therefore they have a correspondence association indicated by the grey arrow. Since the association is interpreted as a mapping relation, objects of 'machine' class (e.g., 'welding machine', and 'bending machine') are mapped to records of 'machine' table in M0. This is exactly how the instance data is integrated between the two domains.

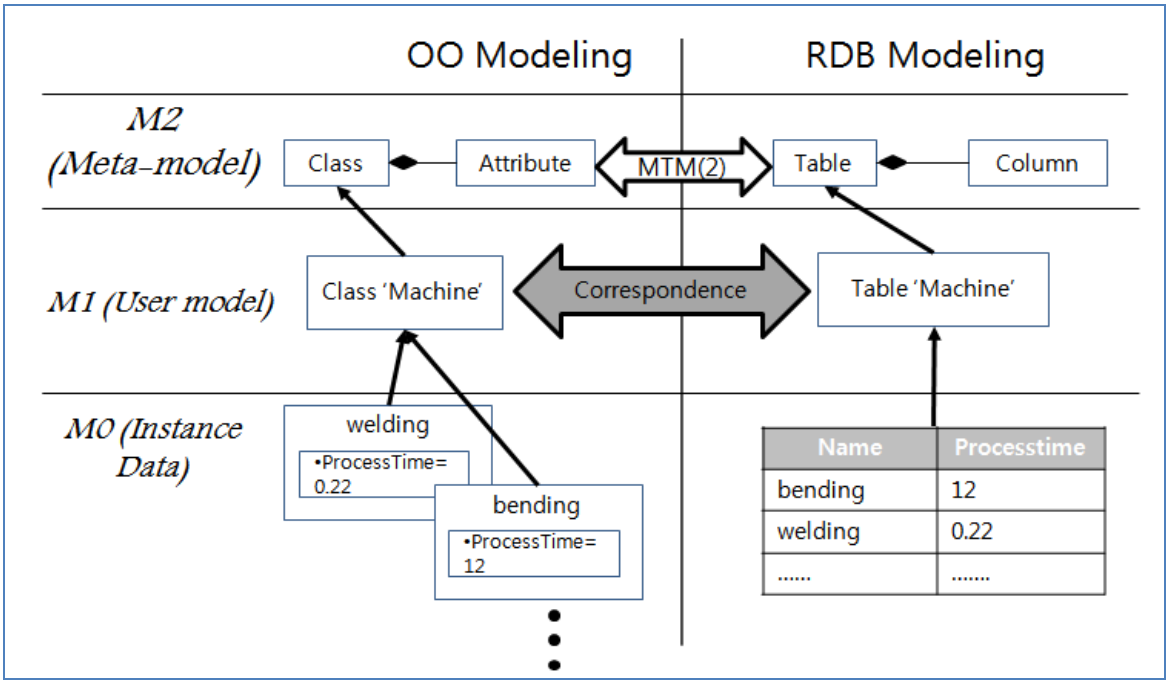

Figure 12. Example of Correspondence Model based Approach

If we want more complicated transformation rules in M1, we cannot use this approach. In this case, we have to use the direct approach, which allows us to

independently express the semantic conversion of the generated model transformation model.

The model transformation rule that the correspondence model can represent looks too simple; the usage of this approach is too limited. However, we can find many practical cases where we can apply this approach. The aforementioned RDB is the typical case. The reason the simple transformation rules work with the example is that the target domain allows us to define our own user model. As a result, we can convert the factory class diagram from the OO domain to the factory database schema of the RDB modeling domain in a way that results in the very similar semantic structure between them. This in turn leads to the simple mapping rules in MTM(1).

If there is a predefined user model in a target side and customization is not allowed, the mapping rules become too complex to be expressed by the correspondence model. Fortunately, many contemporary engineering tools allow customization of the user model, which is usually implemented as engineering libraries, because the user defined libraries are the key to productivity in engineering design or analysis. We can apply the correspondence model-based approach to these engineering tools by generating the libraries from the PIM user model.

Table 2. Comparison of the Two Approaches

Direct approach	Correspondence model based approach
• More expressive power in model transformation rule	• HOT rules are simple • HOT rules are generic; they are independent of changes in M2 • Less expressive power, but practically the usage is not too limited

3.3 Justification of Our Approach

The proposed approach has two advantages.

The first advantage is that we can reuse the developed model transformation rules independently from the user model. In the both extensions, the direct approach; and correspondence model-based approach, all model transformation models necessary for our approach are defined in M2. In the horizontal extension, the model transformation that generates the complete meta-model of tool specific instance data is defined in M2; in the vertical extension, the HOT that generates the model transformation model, MTM(1) for instance data conversion is also defined in M2 in either the direct approach or the correspondence model-based approach.

The independence from M1 layer makes our approach very productive. Whatever problem we have to solve, meta-models in M2 do not need to be reconstructed once the modeling domains are determined. Unlike the meta-models, user models in M1 depend on the type of the problem; e.g., if we want to solve a production planning problem instead of a transportation problem, we have to construct a totally new user model in M1. This means that the user models need to be reconstructed more frequently. If any approach demands that some necessary model transformation models need to be developed in association with user models of M1, the transformation must be changed whenever the type of problem is changed.

The multi layer aspect of our approach allows us to avoid the frequent manual reconstruction through the model transformations, which are defined in M2, that generate the necessary components in M1: the model transformation model, MTM(1) for instance data, and the complete meta-model for instance data. The model transformation models are not affected by the type of problem we want to solve; once the approach is set for particular modeling domains and their engineering tools, it can be reused for any type of problem within the modeling domain and the tools.

The second advantage is that the separation between semantic transformation and syntactic transformation increases productivity in developing the model transformation models. In general, the semantic aspect of the transformations has two interesting characteristics. It tends to be hard to develop because it should deal with conceptual gap between two different modeling domains; and it is independent of tools within an engineering tool domain since the tools are developed based on a common modeling concept. Therefore, the semantic conversion is the common part of the transformation across the tools. It is inefficient to put both the semantic aspects and syntactic aspects together into one model transformation model and separately apply it to every tool in the modeling domain (see Figure 13). The reason is that we need to repeatedly put complicated but common rules into each model transformation model.

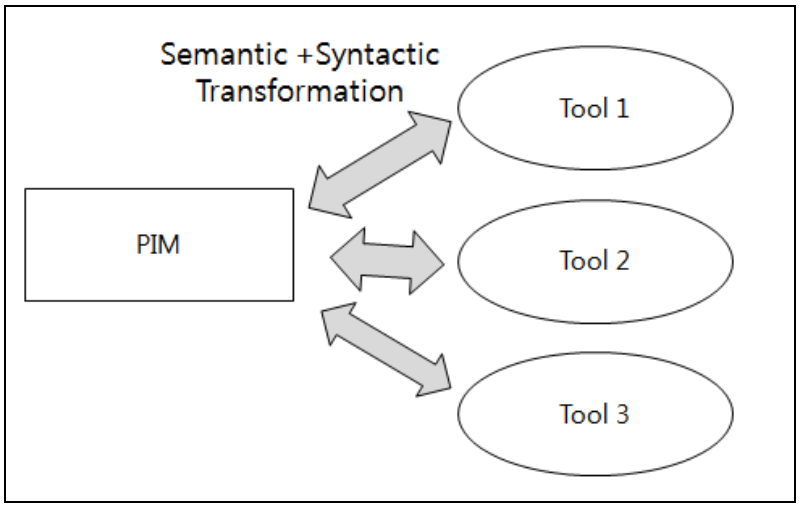

Figure 13. Traditional model transformation in MBE

The separate syntactic model transformation of our approach allows one to avoid the time consuming job, where we repeatedly incorporate the common semantic transformation rules into a model transformation model with every tool. Through the syntactic model transformation, we separate all tool specific syntax from the common semantic part. The common semantic conversion is captured in the semantic model

transformation with the domain semantic model (see Figure 5). This is developed once and shared by multiple tools via the domain semantic model.

3.4 Conclusion

We proposed multi layer syntactical model transformation as a comprehensive solution to the two key issues of this dissertation. We gave explanations of the ideas behind the proposed model transformation, and justified it by presenting a number of practical benefits.

Multi layer syntactical model transformation comprises two parts: the syntactical model transformation that copes with syntactical inconsistency, and the multi layer model transformation that handles instance data integration.

The syntactical model transformation was suggested in a way that converts a PIM model to tool specific models through sequential steps of semantic transformation and syntactical transformation. We proposed two important concepts that are essential to establishing the syntactical model transformation. First, the domain semantic model plays roles of an intermediate model coupling the two steps, and a repository model of domain specific components. Second, we introduced the notion of the ‘complete meta-model’ (Definition 7) that is used to formally define a model both semantically and syntactically; meta-syntax, which is a syntax of syntax (Definition 6), is incorporated into the meta-model introduced in the previous chapter.

The key idea of the multi layer model transformation is to uses a HOT defined in M2 to generate a model transformation model, MTM(1), which handles instance data in M0. We introduced two approaches – the direct approach and the correspondence model-based approach, which differently use the HOT. The HOT of the direct approach creates the mapping rules of MTM(1) independently of MTM(2), whereas the HOT of the correspondence mode-based approach converts the result of MTM(2), which is called correspondence model, to MTM(1).

Although the syntactical model transformation and the multi layer model transformation have been suggested for different purposes, they have something interesting in common. Both of them use model transformation to resolve the issues of model transformation; i.e., the syntactical model transformation uses model transformation to generate a complete meta-model of tool specific instance data, and the multi layer model transformation uses the special type of model transformation, HOT, to create a model transformation model for instance data. In this sense, our approach extends the standard model transformation by suggesting new ways of using existing model transformation rather than changing the fundamental structure of model transformation.

We proposed a number of novel ideas for multi layer syntactical model transformation. However, these ideas have been intuitively presented; they should be rigorously verified. In the rest of this dissertation, we will theoretically prove some of the ideas using graph grammar theories, and practically demonstrate the entire framework using a concrete example.

CHAPTER 4

THEORETICAL FOUNDATION OF MULTI LAYER MODEL TRANSFORMATION

This chapter aims to provide the theoretical foundation of the multi layer model transformation. Between the two approaches of the multi layer transformation – the direct approach and the correspondence model-based approach, we focus on the correspondence model-based approach because the viability of the approach is not obvious. In the direct approach, the HOT is intended to directly generate the model transformation model. And the analyst obviously can develop the HOT in a way that results in a valid model transformation model. In contrast, the correspondence model-based approach does not directly generate a model transformation model; rather it is suggested that a correspondence model (Definition 8) can be converted to a model transformation model. We therefore need to show that the correspondence model can be a valid model transformation model. Moreover, the correspondence model-based approach is more efficient if it is applicable (see Section 3.2); hence, it is practically important to indentify the conditions under which we can use the approach. In this chapter, we prove theorems establishing the conditions; in Chapter 6, we use these theorems to implement this approach as a demonstration.

For the theoretical discussion on the viability of the correspondence model-based approach, we need a formal definition of a model transformation model to establish the criteria for a valid model transformation model. There have been a number of efforts to establish a mathematical formalism of model transformation models, based on graph grammar theories [17, 31]. Among them, we adopt triple graph grammar (TGG), because it has been proven by successful implementations in a number of practical tools [17, 26, 32].

This chapter presents three contributions to the theory of model transformation:

i. We extend TGG in a way that deals with deletion operation. The original TGG assumes that model transformation always adds elements. The extension allows us to eliminate this limiting assumption.

ii. We prove a recursive property of individual triple productions of TGG, where a triple production is the mathematical representation of the mapping rule of model transformation model. The recursive property means that an execution of a triple production results in a graph that conforms to the definition of triple production.

iii. We prove that the extended TGG, which includes deletion operation, has the determinism property, which we define in section 4.1 to mean that a set of triple productions of a TGG results in an equivalent graph regardless of their execution order.

The last two properties of TGG allow us to conclude that a correspondence model is a valid model transformation model, if MTM(2), which generates the correspondence model (Figure 10), can be described in TGG.

The Detailed Outline of This Chapter

This chapter consists of two parts: introduction and extension to existing graph grammar theory – TGG in particular, and theoretical discussion on the viability of the correspondence model-based approach based on the graph theory.

In the first part, there is an important extension to TGG; we extend TGG so that it can deal with an important practical case, which the original TGG ignores. By definition, TGG incorporates only non-deletion transformations; this is referred as to the monotonic property of TGG. Schürr [17] omits the deletion case with the justification that model editing - deletion or modification - is not the primary purpose of model transformation. However, deletion or property value modification is commonplace in practice; i.e., once a source model is transformed to a target model, if any components are removed in the

source model, the corresponding components in the target model should be deleted as well. We show that transformation rules that can deal with deletion operation can be derived from the existing definition of TGG. This allows us to apply TGG to more general cases without extension to the original definition and theorems.

In the second part, we use TGG to prove that a correspondence model can be converted to a valid model transformation model. To do this, we propose two validity conditions that the correspondence model should have: individual rule validity, and collective execution order validity. The individual rule validity requires that individual correspondence associations should be valid mapping rules. In contrast, the collective execution order validity demands that the correspondence associations collectively results in the desired output. If the correspondence associations are dependent on one another when they are converted into mapping rules, the output depends on the order in which the rules are executed, and we need to make sure the execution order is correctly set.

We show the individual rule validity of a correspondence model by proving the recursive property of TGG. As mentioned, the recursive property assures that an execution of a triple production mathematically results in a triple production. In practice of model transformation, a mapping rule is instantiated as a correspondence association. Therefore, we can say that correspondence associations are described in triple production, mathematical representation of a valid mapping rule.

However, the correspondence model does not necessarily satisfy the second validity rule, collective execution order validity, because the correspondence model does not contain information that controls the execution order of the rules. By Definition 8, a correspondence model contains just an unordered set of correspondence associations.

Nevertheless, we can make the correspondence model valid by specifying an execution order. We address the execution order in the following two steps: i) we prove that the execution order does not affect the ultimate result of TGG, ii) we propose an algorithm to determine an efficient execution order.

For the first step, we use determinism of graph theory, which is explained in section 4.1. Determinism holds if a set of transformation rules results in an equivalent output, regardless of the order in which the rules are applied. Since the ultimate result is independent of the order, we can use any arbitrary order.

Although the determinism ensures that we can eventually obtain an equivalent result in any execution order, the number of execution steps varies depending on the order. The maximum number of steps for n rules, as we will prove in section 4.5, is $O(n^2)$. We propose an algorithm that ensures n TGG rules reach an end in O(n) steps.

4.1 Introduction to Graph Transformation

Graph transformation has been introduced in the 1970s [33-35], and applied to a wide variety of areas in computer science: formal language theory, pattern generation and recognition, complier construction, visual modeling, model transformation, etc.

The basic idea of graph transformation is to modify graphs using rules described in other graphs. This basic idea has different practical meanings depending on the application area. In this section, we discuss graph transformation in the context of model transformation. We briefly introduce the key concepts of graph transformation that are necessary to understand the mathematical background of model transformation; we go over how to mathematically define a graph transformation system (GTS), how to execute it, and how to interpret it in the context of model transformation. In addition, we introduce some advanced concepts – local Church-Rosser theorem, and local confluence - that are essential to further discussion in Section 4.5 on the determinism of triple graph grammar. Note that the definitions and the theorems of graph grammar that we introduce in this section come from [36] and [37].

Basic concepts

A graph is the basic element of graph transformation; a transformation rule is defined by a combination of two graphs, and the rule is applied to a graph. In addition, a graph makes a contribution to the various applications of graph transformation because the graph is able to naturally represent many types of systems in an abstract level; that is, it plays the role of a formal abstract model in various areas of computer science, such as UML diagrams, Petri Net, data modeling, etc.

In this sense, we start with the formal definition of a graph. In order to support the various application areas, computer scientists have introduced a number of variations of graphs: labeled graph, typed graphs, attributed graphs, etc. However, we just introduce the definition of a basic graph, because all variations satisfy the theorems and the properties that we need in this dissertation. By using the simplest one, we can avoid unnecessary complexities that the extended variations may cause.

Definition 10. Graph

A graph $G = (V, E, s, t)$ consists of a set V of vertices, a set E of edges, and two functions $s, t: E \rightarrow V$, which are the source and the target function, respectively.

A graph morphism defines mappings of nodes and edges between two graphs in a way that preserves the source and the target of each edge.

Definition 11. Graph morphism

Given two graphs G_1, G_2 with $G_i=(V_i, E_i, s_i, t_i)$ for i=1,2, a graph morphism $f : G_1 \rightarrow G_2$, $f=(f_v, f_E)$ consists of two functions $f_v: V_1 \rightarrow V_2$ and $f_E: E_1 \rightarrow E_2$ such that $f_v \circ s_1 = s_2 \circ f_E$ and $f_v \circ t_1 = t_2 \circ f_E$.

A graph transformation rule is constructed and executed based on the above two basic concepts: graph, and graph morphism. The transformation rule, which is called a graph rewriting rule in graph grammar, is defined by a graph production, p, which consists of a pair of graphs, (L, R) and a graph morphism between them.

Definition 12. Graph production

A graph production p = (L, R, M) consists of a source graph L, a target graph R, and a graph morphism M: L→ R. 2-tuple (L, R) is the short notation where the morphism is omitted. BC(p) denotes L (before condition graph), while AC(p) denotes R (after condition graph).

A graph transformation is an application of a graph production to a graph. The graph production is applied to an original graph, G, via a match m, which is technically a graph morphism. This graph transformation produces a modified graph, H. In addition, it generates a derivation, which represents the occurrence of the graph production, between G and H. The application operation is mathematically supported by pushout operation developed based on category theory [38]. Detailed discussion on the pushout operation and the category is beyond our scope. Instead, we introduce the following practical way of conducting the pushout operation without mathematical explanation:

i) keep elements that exist both in L and R.

ii) delete elements that exist only in L.

iii) add elements that exist only in R.

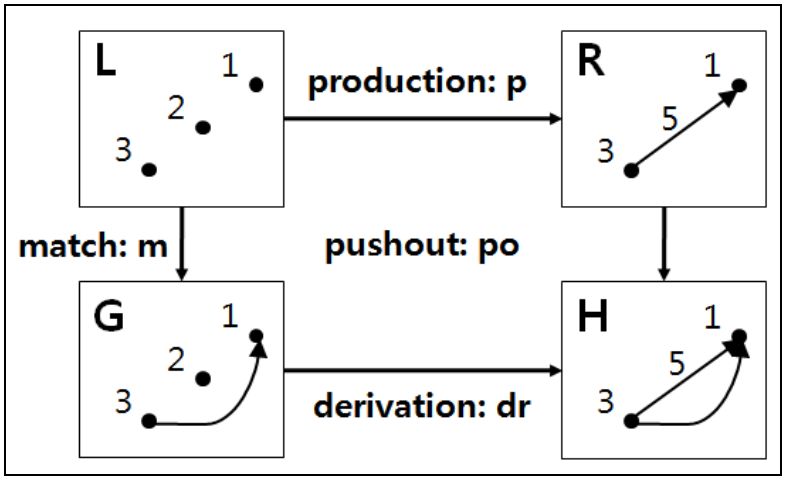

Figure 14. Definition and Execution of Graph Transformation

Figure 14 shows a graph transformation is defined by a production p, and executed by a pushout operation po. Node 1 and 3 are kept because they exist in both L

and R. Node 2 is deleted because it exists only in L, whereas edge 5 is added because it exists only in R. Definition 13 formally represents the graph transformation. Note that the relation between a graph production and its graph transformation is analogous to that between a model transformation model and its model transformation; that is, the graph production (or the model transformation model, respectively) defines the rule, while the graph transformation (or the model transformation, respectively) executes the rule. Definition 13 describes a direct graph transformation, which results from an execution of a single graph production.

Definition 13. Direct Graph Transformation

Given a graph production p = (L, R, M), a graph G,and a graph morphism m: L→ G, which is called match, a graph transformation GT = $G \overset{p,m}{\Longrightarrow} H$ generates a modified graph H by pushout operation.

In practice, a graph is rarely transformed by only one rule; the graph is modified via a sequence of rules, which are represented by graph productions in terms of graph grammar theory. Indeed, a model transformation model is defined by a set of rules. In order to deal with this aspect, graph transformation system and general graph transformation has been defined as follows.

Definition 14. Graph Transformation System and Graph Transformation

A graph transformation system is a 2-tuple GTS = (G, P), where G is the initial graph, and P is a set of graph productions. If graph G turns into H by the sequential applications of n graph productions in P, the collective transformation is denoted by $G \overset{*}{\Rightarrow} H$. TR(GTS) is a set of all possible transformations that can be obtained from P, i.e., permutation of P', where P' is any subset of P.

In the following section, we discuss advanced theories on how the sequence of applications affects the behavior of a graph transformation system.

Advanced concepts

The graph productions are usually dependent on one another. In this case, the result of the sequential applications of the productions is affected by the order of the execution. Graph grammar research community has intensively explored the effect of the execution order on the result of a graph transformation system. Among a number of important theories on this topic, we focus on theories relevant to determinism property, which is the key to theoretical discussion on multi layer model transformation in section 4.5.

The determinism property means that a graph transformation system produces an equal or isomorphic graph regardless of the execution order when a graph transformation is terminating. A graph transformation is called terminating if no more graph production is applicable to the current graph.

Confluence plays an important role in proving the determinism property. For a graph transformation system GTS = (G,P), GTS is called confluent if, for every pair of graph transformations in TR(GTS) - $G \overset{*}{\Rightarrow} H_1$ and by $G \overset{*}{\Rightarrow} H_2$, there exist a graph X, and two graph transformations, GT_1 and GT_2, such that $H_1 \overset{*}{\Rightarrow} X$ and $H_2 \overset{*}{\Rightarrow} X$. Figure 15 depicts the confluence property.

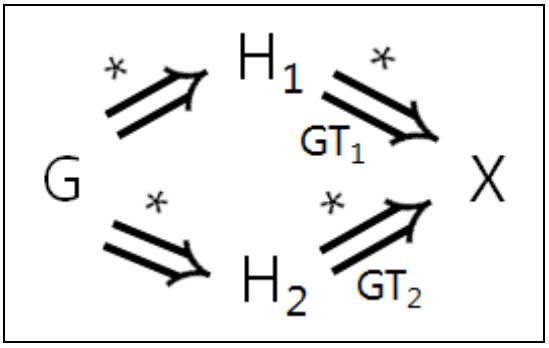

Figure 15. Global Confluence

The following lemma, proven in [36], shows that confluence assures determinism of a graph transformation system. We can also give an intuitive argument for the lemma.

Suppose that there are two terminating graph transformations $G \overset{*}{\Rightarrow} H_1$ and by $G \overset{*}{\Rightarrow} H_2$. By the definition of termination, no graph production of the graph transformation system is applicable to H_1 and H_2. If both are not equal to each other, there does not exist transformations that covert them into an equivalent graph. This is contradictory to the definition of confluence.

Lemma 1 (global determinism of a graph transformation system) Every confluent graph transformation system is deterministic. (See [36] for details)

However, the application of this lemma is practically limited because there exist too many graph transformations. If a graph transformation system has n graph productions, the number of graph transformations is $\sum_{k=1}^{n} P(n,k)$, where P(n,k) denotes permutation of picking k out of n. Fortunately, there is a weak version of confluence - local confluence – which we can use to efficiently prove determinism in a special case. Local confluence means that the confluence property holds only for pairs of direct graph transformations.

Definition 15. Local Confluence

A graph transformation system GTS = (G,P) has the local confluence property if, ∀ $G \Longrightarrow H_1$ and $G \Longrightarrow H_2$, there exist a graph X, and graph transformations such that $H_1 \overset{*}{\Rightarrow} X$ and $H_2 \overset{*}{\Rightarrow} X$.

We can use local confluence instead of global confluence if a graph transformation system is terminating; a graph transformation system is called terminating if all possible graph transformations derived from the initial graph are finite.

Lemma 2 (termination and local confluence) Every terminating and locally confluent graph transformation system is deterministic as well. (See Appendix C of [36] for details)

This lemma makes it easier for us to prove determinism of a graph transformation system; we need handle only a reasonable number of direct graph transformations instead

of all possible graph transformations, i.e., n vs. $\sum_{k=1}^{n} P(n,k)$. Moreover, the termination condition is not too restricted in the context of model integration. The purpose of model integration is to transfer information from source models to target models; non-termination of model transformation means either never-ending growth of the target models or cyclic modifications of the target models. Both cases are not desired in the practice of model integration. In Proposition 8, we mathematically prove the termination of model transformation.

We use this local confluence lemma to prove the determinism property of TGG, which plays an essential role in finding conditions under which we can use correspondence model based multi layer model transformation, in section 4.5.

4.2 Triple Graph Grammar: Mathematical Formalism of Model Transformation

Schürr introduced triple graph grammar as a mathematical formalism of model transformation in [17]. We review the key definitions and theorems of TGG and how they support theoretically model transformation.

As mentioned, model transformation handles three key parts: source model, target model, and mapping rules between them. TGG uses a special type of graph, called triple graph, to accommodate that structure. A triple graph is a combination of three graphs that represent the three key parts, respectively. It is formally defined as follows.

Definition 16. Triple graph

A triple graph is a 5-tuple (SG, CG, TG, *lm*, *rm*), where SG (source graph), CG (correspondence graph), and TG (target graph) are graphs; and *lm* (or *rm* respectively) is a graph morphism from CG to SG (or TG respectively), i.e., *lm* (or *rm* respectively) : CG → SG (or TG respectively). The 3-tuple (SG, CG, TG) is the simplified notation where the graph morphisms are omitted.

TGG extends normal graph grammar by using triple graphs instead of normal graphs as its basic building blocks. A transformation rule of TGG is defined as a triple

production, which consists of two triple graphs associated through graph morphisms. This triple production converts one triple graph to another triple graph in accordance with the rule.

Before getting into the formal definition of triple production, we need to introduce the following monotonic production. The right graph (after condition graph) includes the left graph (before condition graph); practically, it means that this production always adds something but deletes nothing.

Definition 17. Monotonic Production

Production p : (L,R) is monotonic if $L \subset R$.

A triple production consists of two triple graphs, which are associated with each other through three monotonic productions. These three productions connect the corresponding graphs between the two triple graphs respectively; i.e, the source graph of the first triple graph is linked with that of the second triple graph, and so forth. It is formally defined as follows.

Definition 18. Triple Production

A triple production is defined by two triple graphs TGL = (SL, CL, TL, sr, tr), and TGR = (SR, CR, TR, sr', tr'), where sr = sr'|CL, and tr = tr'|CL. The triple graphs are associated through the three monotonic productions: sp: (SL,SR), cp: (CL,CR), and tp: (TL,TR). We denote the triple production as follows:

((SL,SR) ← sr – (CL,CR) – tr → (TL,TR)).

The way of applying a triple production to graph transformation is analogous to the way of a normal graph production. As mentioned, a normal graph production is applied to a graph through a match morphism, and converts the graph. Likewise, a triple production is matched to a triple graph, and coverts it into another triple graph. The only difference is that there are three separate match morphisms that apply the three monotonic production of the triple production to the three parts of the triple graph

respectively, i.e., the production of the source side (sp) is matched to SG of the triple graph (SG, CG, TG), and so forth. Definition 19 formally describes the process of the application.

Definition 19. Triple graph transformation

Given a triple production ((SL,SR) ← sr – (CL,CR) – tr → (TL,TR)); a triple graph TGG = (SG, CG, TG,sc,tc); and three graph morphisms sm: SL→SG, cm: CL→CG, and tm: SL→SG, triple graph transformation generates another triple graph TGH=(SH,CH,TH,sc',tc') through the following productions:

SG $\overset{sp,sm}{\Longrightarrow}$ SH, CG $\overset{cp,cm}{\Longrightarrow}$ CH, and TG $\overset{tp,tm}{\Longrightarrow}$ TH, where sp = (SL,SR), cp = (CL,CR), and tp = (TL,TR).

Figure 16 illustrates the transformation operation of a triple production. The shaded back side represents triple production (Definition 18). This triple production is applied to a triple graph over three matching morphisms (sm, cm, tm). It transforms the original triple graph, (SG, CG, TG), to the derived triple graph, (SH, CH, TH), while generating three derivations (sd, cd, td). The front side, which consists of the original and derived triple graphs and the three derivations, represents the result of executing the triple production.

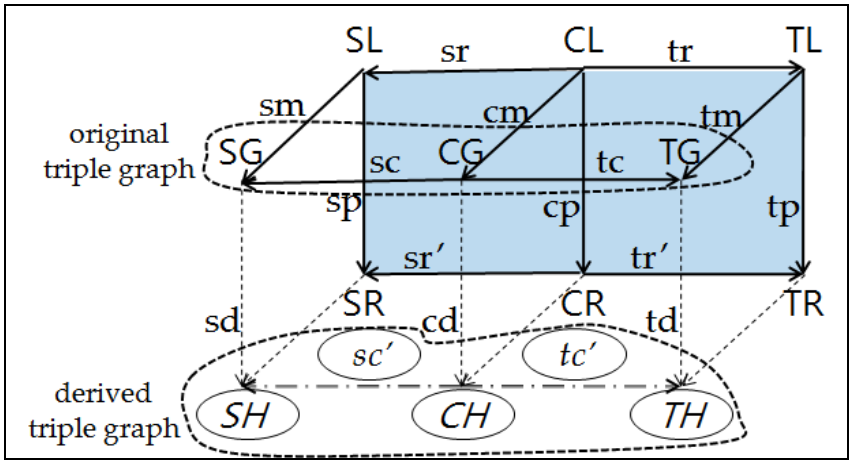

Figure 16. Triple Graph Transformation

The definition of triple graph transformation represents a simultaneous transformation of the three parts of a triple graph. However, this is not the way model transformation works in practice. The following proposition has been suggested and proven in [17] to support the actual process in which model transformation is used.

Proposition 5. Triple Production Separation Theorem

A given triple production p = ((SL,SR) ← sr – (CL,CR) – tr → (TL,TR)) can be replaced by the sequential applications of two particular triple productions:

p_L = ((SL,SR) ← ε – (Ø, Ø) – ε → (Ø, Ø)), and p_{LR} = ((Ø, Ø) ← ε – (CL,CR) – tr → (TL,TR)). The sequential applications of p_L and p_{LR} produce the equivalent triple graph to the original triple production p.

This theorem says that a triple production can be split into the two separate triple productions, i.e., the triple production for only the source graph, and the triple production for the rest of the graph. A series of applications of the split triple productions results in the equivalent output to the output of the original triple production.

This separation accounts for the way that we actually use the model transformation. In practice, the three parts of a triple graph are not transformed simultaneously. Instead, the source model is changed by humans' authoring activities, and then the change is transferred to the target model.

The first triple production p_L represents the authoring activities; the result of the production is the snapshot of the source model right after the authoring activities. The result is used to recognize the pattern that we need to transfer to the target model (See section 4.3). In other words, it is used as the application condition under which model transformation can be executed. If the authoring activity results in the pattern that meets the condition, the second triple production p_{LR} transfers the changes of the source model to the target model in accordance with the triple production; this is the production that transfers information from the source to the target, while completing the after condition

graph of the original triple production from which p_{LR} is obtained. We can therefore view the second triple production as the practical execution of the model transformation.

This separation property of triple production serves as the theoretical foundation of the derived transformation rules, which we will discuss in the following section..

4.3 Derived Transformation Rules of TGG

As discussed, TGG defines simultaneous transformation of the source model and the target model. Practically, TGG is not directly applied to model transformation; instead, it is used to derive rules that are used in the model transformation. A number of transformation rules can be derived from a single TGG rule. They have been introduced in [32]; i.e., consistency checking rule, correspondence creation rule, left-to-right transformation rule, and right-to-left transformation rule.

However, these rules do not cover operations that delete components. This is because a triple production consists of three monotonic productions, which always add elements. In order to support deletion, we propose a way to derive deletion rules from the current definition of TGG.

In this section, we introduce the existing derived transformation rules and propose the new derived transformation rules for deletion.

Derived Transformation Rules: Creation rule

Among the rules introduced in [32], we focus on left-to-right rule and right-to-left transformation rule, which are practically important. Since both rules create graph elements, we call them ‘creation rules.’

Figure 17 shows one example of a triple production. Note that we substitute ‘Left’ (or ‘Right’, respectively) for ‘Source graph’ (or ‘Target graph’, respectively) of the original definition of TGG. We use the new terms to reflect bidirectional transformation property of TGG, i.e., either side can be the source model.

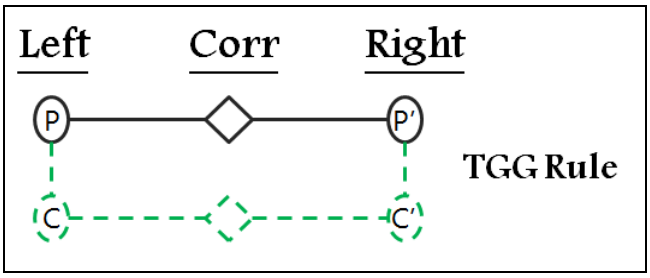

Figure 17. Example of Triple Production

As shown in the figure, the triple production can be depicted in a single graph. Although it consists of two triple graphs, we can use this compact graph because of the monotonic property. By the definition of monotonic production, the after condition graph completely includes the before condition graph; this means the before condition graph is a sub graph of the after condition graph. The before condition graph can be describe by highlighting some part of the after condition graph. In Figure 17, the continuous line represents the before condition. Hence, we can interpret this production as follows: if the continuous line pattern is found, the thr triple production completes the whole graph – the after condition - by adding the dashed pattern. In order to formally indicate the added pattern, we introduce the following notation. In monotonic production (L,R), the dashed pattern is L\R.

Definition 20. Difference Operation

Given two graphs G1 and G2, difference of G1 and G2, denoted G1\G2, is {e∈ G1| e∉ G2}.

Figure 18 shows the two creation rules: left-to-right transformation rule and right-to-left transformation rule. The left-to-right transformation transfers information from the left side to right side; i.e., if the left side model is modified, the transformation transfers the modification to the right side model. Part a) describes the operation; i.e., if component C is added to the left side, component C' and the correspondence component are created. Part b) represents the reverse direction, the right-to-left transformation. It can be symmetrically derived.

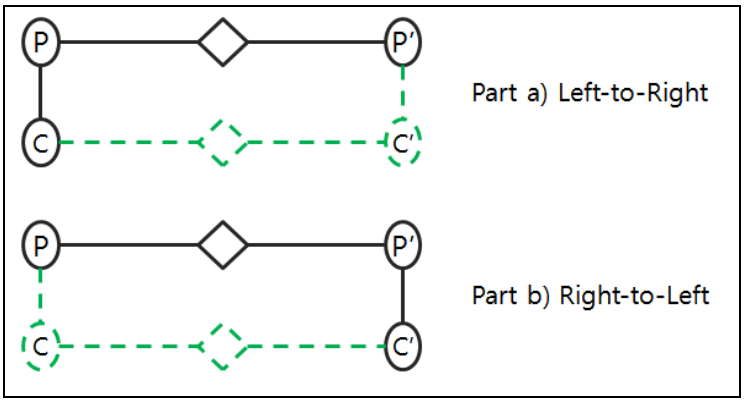

Figure 18. Derived Creation Transformation Rules

The triple production separation theorem, which is introduced in section 4.2, plays an important role in deriving the transformation rules. By the theorem, the TGG can be split into two parts: left side production, and the correspondence together with the right side production. As discussed, the left side production represents the model authoring activities. The authoring activities are done before the application of a left-to-right transformation; hence, the authoring activities are a prerequisite of the left-to-right transformation. The left side should be the part of the application condition. We can derive the application condition of the left-to-right by incorporating the left side production into the original application condition of TGG. We can derive Part b) in the symmetric way.

Definition 21. Derived Creation Rules

A given triple production p = ((SL,SR) ← sr – (CL,CR) – tr → (TL,TR)), left-to-right (or right-to-left, respectively) creation rule is defined as production p_{LR} = ((SR,CL,TL), (SR,CR,TR)) (or p_{RL} = ((SL,CL,TR), (SR,CR,TR) respectively). CR(p) is a set of the creation rules of production p.

Definition 21 shows the formal definition of the derived creation rules. Technically, these productions are the second production in Proposition 5 .

Proposed Derived Transformation Rule for Deletion: Cancellation Rule

In model transformation, deletion operations have a unique usage; i.e., they are used to cancel the existing result of the transformations that were executed in previous steps. In other words, if we delete any components of one side (e.g., left side graph) in an existing transformation result, we need to delete the corresponding components of the other side (e.g., right side graph). Figure 19 shows one example of the operation. Deleting component C in left side leads back to the original condition before the application. Hereafter, we call the operation 'cancellation rule.'

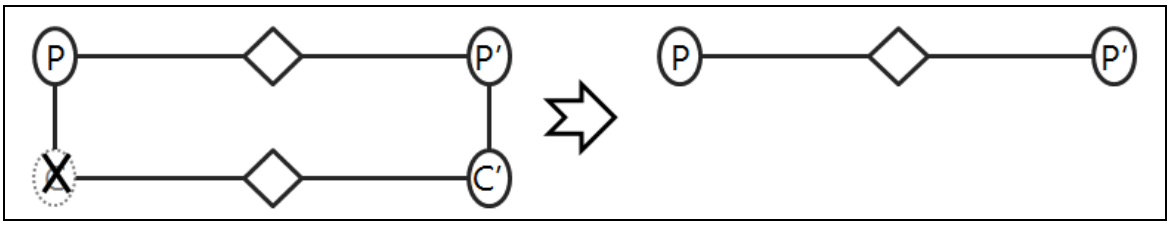

Figure 19. Deletion Operation of Model Transformation

We show that there are two types of cancellation rules: intended cancellation rule, and forced cancellation rule. The intended cancellation rule is straightforward; it is applied to components that a modeler intends to delete. Cancellation of (C, C') in Figure 19 is an example of the intended cancellation; deleting component C means that the modeler intends to also delete C', which was created from component C.

In contrast, a forced cancellation is triggered by other intended or forced cancellations; that is, the forced cancellation rule is forced to delete the existing model transformation result because other model transformations, on which the model transformation is built, are cancelled. For example, suppose we delete component P in Figure 17 with intention to cancel transformation of (P, P'); then (P, P') is the intended cancellation. However, if P and P' do not exist, C and C', which are created based on P and P', cannot exist. Therefore, we need to also cancel transformation of (C, C'), which is a forced cancellation.

In order to express the cancellation rules, we need a new type of application condition, called negative application condition. A normal application condition is tested based on existence of components; i.e., if a sub graph matches up with the pattern of the application condition, the transformation rule can be applied to the graph. However, this type of condition cannot handle absence of components. In other words, it cannot check if a pattern does not exist. In order to deal with the negative information, [39] suggested negative application condition.

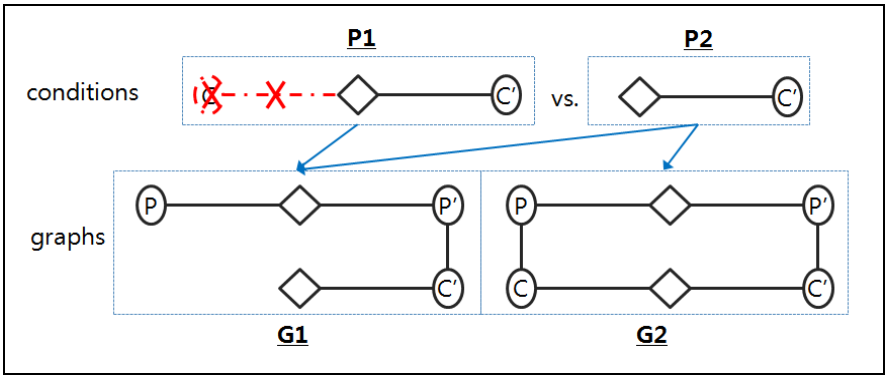

Figure 20. Negative Application Condition

Figure 20 shows how differently a negative application condition works. The left condition (P1) represents a negative application condition. The components drawn in dashed lines with 'X' mark represent a negative pattern, which must not exist. A transformation rule can be applied only if the solid components exist but the negative pattern does not; that is, P1 is applicable only to graph G1 because G2 has component C, which is prohibited by the negative pattern. In contrast, the normal application condition (P2) is applicable to both graphs G1 and G2 because the application condition does not care about the existence of component C.

Definition 22. Graph production with a negative application condition

A graph production p = ((L, R, M), NAC(X, *x*)), where *x* is a morphism from L to X, i.e., *x*: L→X, is applicable to a graph G through a match morphism m: L→G if there does not exist an injective graph morphism m': X→G such that m'∘*x* = m. For simple

notation, morphism x can be omitted when it is obvious, i.e., p = ((L, R, M), NAC(X)).

In order to deal with the negative pattern, we extend the definition of graph production (Definition 22). In the example of Figure 20, graph X in Definition 22 is triple graph (C, Corr, C'). For G1, we cannot make any injective graph morphism from X to G1 with m'$\circ x$ = m because G1 has no component that can be mapped to component C; hence, the production is applicable to G1. In contrast, (C, Corr, C') is a subgraph of G2. An injective morphism m' satisfying m'$\circ x$ = m can be developed by making one-to-one mapping between corresponding components, i.e., C→C, Corr→Corr, and C'→C'. It is not applicable to G2.

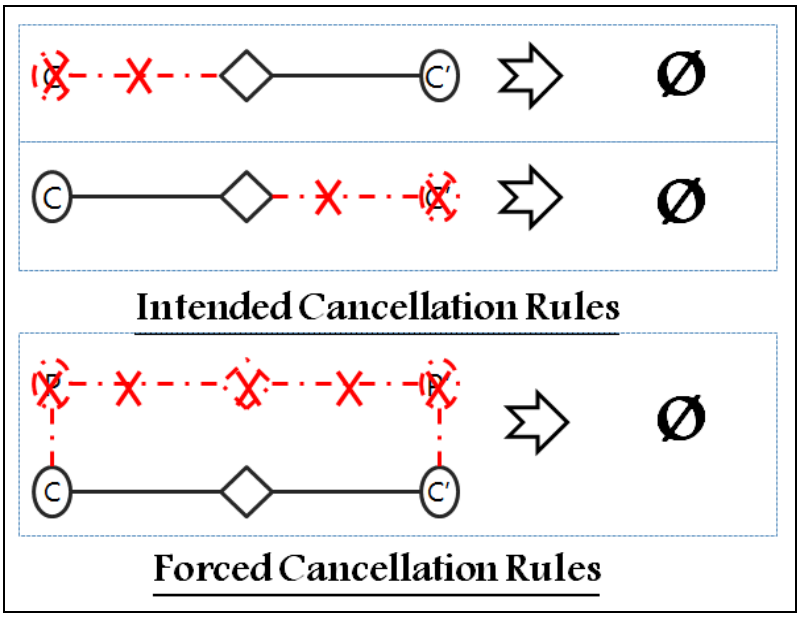

Figure 21. Derived Cancellation Rules

Figure 21 depicts the cancellation rules that are derived from the triple production described in Figure 17. The first two rules are intended rules. Since the correspondence component exists, the pattern can appear after the derived creation rules of the same triple production are applied. The negative patterns of the cancellation rules result from deleting components in the left or right side graph. This practically means that a modeler intends to cancel the existing result of the creation rule by getting rid of the left side or

the right side; that is, the modeler deletes a graph element from one side, and then the model transformation takes away the rest (i.e., the correspondence component, and the other side).

In contrast, the bottom one is a forced cancellation. Unlike the top two, which are created by the creation rule, are preserved. This means that the modeler does not explicitly intend to cancel the result. However, the negative condition shows that the before condition of the triple production should not exist. Since the result of the triple production cannot exist without the before condition graph, the result is forced to be deleted; that is, the triple production is cancelled as the result of the cancellation of another triple production.

We formally define the two classes of derived cancellation rules using as follows.

Definition 23. Derived Cancellation Rules

For a given triple production p = ((SL,SR) ← sr – (CL,CR) – tr → (TL,TR)), the two intended cancellation rules are described as follows:

((Ø,CR\CL,TR\TL), (Ø, Ø, Ø), NAC((SR\SL,CR\CL,TR\TL))) and

((SR\SL,CR\CL,Ø), (Ø,Ø,Ø), NAC((SR\SL,CR\CL,TR\TL))). IDEL(p) denotes the set of intended cancellation rules. The forced cancellation rules is

((SR\SL,CR\CL, TR\TL), (Ø,Ø,Ø), NAC((SR,CR,TR))). FDEL(p) denotes the set of the forced cancellation rules. In addition, DEL(p) = IDEL(p) ∪ FDEL(p).

In addition, we suggest the definition of a set of all derived transformation rules as follows.

Definition 24. Derived rules of a triple production

Given a triple production p, the derived transformation rules set is defined as *DRules*(p) = CR(p) ∪ DEL(p).

Cross-over Derived Transformation Rules

In the previous two subsections, we have been discussed the ways to derive transformation rules from a triple production. The derived rules of a triple production p are constructed in a way that modifies the constructs of p – BC(p) and AC(p). However, those discussions ignored a case that the constructs of other triple productions could involve in the derived rule construction of p. Schürr did not identify this case in [17] where he introduced TGG. In this section, we further explore the case. In order to deal with the case, we introduce cross-over derived transformation rules, which are derived from more than one triple production.

Mathematically, that case occurs when $(AC(p1)\backslash BC(p1) \cap AC(p2)\backslash BC(p2)) \neq \emptyset$, i.e., p1 and p2 creates common elements. If p1 creates the common elements of p2, the derived creation rules of p2 should be constructed differently depending on whether p1 is applied or not. For instance, if p1 has been applied, the derived creation rules of p2 do not need to create the common elements again because they have been already generated by p1. If p1 has not been executed yet, the derived creation rules have to create the common elements. Cancellation operations also raise the same issue. A cancellation operation of p1 must preserve the common elements if the result of p2 exists. However if p2 have been removed, the cancellation should delete the common elements so that nothing unnecessary remains.

Definition 25. Cross-over Relation

Given two triple productions p1 and p2, they are said to have a cross-over relation if $AC(p1)\backslash BC(p1) \cap AC(p2)\backslash BC(p2)) \neq \emptyset$.

In order to get cross-over derived transformation rules, we modify the existing derived cancellation rules by adding more negative application conditions (See Appendix A). These negative application conditions prevent the rules from being applied in a case that there exist common elements that have been generated by other triple productions.

Unfortunately, these additional derived transformation rules make it more difficult for us to analyze dependencies among triple productions. The following example shows a tricky situation that the additional derived transformation rules introduce.

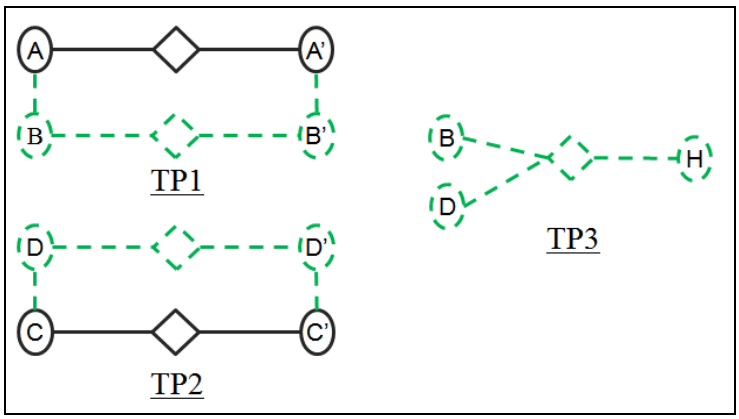

Figure 22. Triple Productions of Cross-over Example

Figure 22 shows three triple productions that illustrate the tricky situation. Component B (or Component D, respectively) is the common part between TP3 and TP1 (or TP2, respectively).

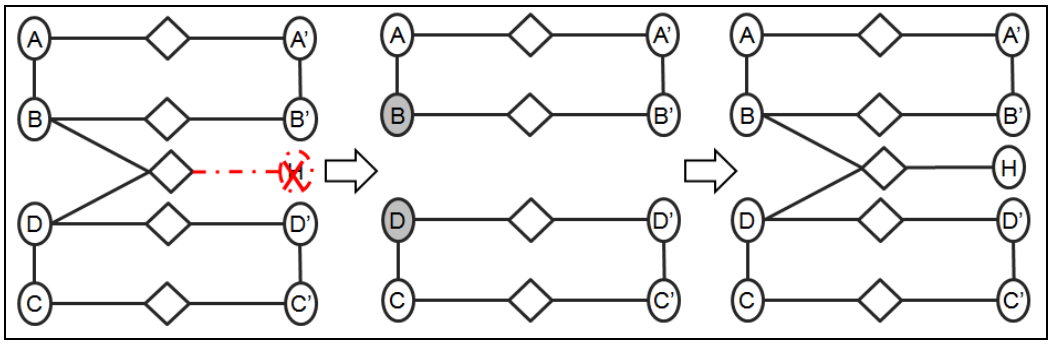

Figure 23. Cross-over Example

The left most graph in Figure 23 is the starting graph that we sequentially apply derived transformation rules to. Component H is deleted from AC(TP3); one intended cancellation rule of TP3 can be applied. However, due to the cross-over relations with TP1 and TP2, the cancellation rule does not delete B and D; that is, B and D should exist as the part of T1 and TP2. This results in the middle graph, which has the complete graph of left side of TP3. This is the application condition of the left-to-right derived creation

rule of TP3 (Definition 21); the derive transformation rule reconstructs the complete graph of TP3. Interestingly, although we delete H in order to cancel TP3, these operations end up back to the complete graph of TP3.

In section 4.5, we will discuss dependency among triple productions including this cross-over case. Fortunately, we find out a way to rule out these complicated cases from the dependency discussion using a practical assumption.

4.4 Individual Rule Validity of a Correspondence Model

To establish the individual validity rule, this section rigorously answers the following question: can the correspondence associations created by a model transformation define a valid mapping rule? Theoretically, does the result of triple production individually conform to the definition of triple production? We prove this by extending the theorems of TGG.

Before answering the question, we introduce a key proposition of TGG that is essential to the proof. The following lemma shows the existence and uniqueness of correspondence relations in the derived triple graph. The formal proof is presented in [17].

Lemma 3. In the definition of triple graph transformation (see Figure 16), the morphisms sc' and tc' always exist uniquely such that sc = sc'|CG and tc = tc'|CG.

The lemma leads to the following proposition, which is the answer to the above key question.

Proposition 6. Given a triple production ((SL,SR) ← sr – (CL,CR) – tr → (TL,TR)); and a triple graph G = (SG, CG, TG,sc,tc), a triple production is formed by combining the original triple graph G, the derived triple graph H=(SH, CH, TH,sc',tc') and the three derivations sd, cd, and td (See Figure 16 for notations).

Proof) According to the definition, there are two key conditions which a triple production must satisfy: monotonic production and constraint on morphisms between correspondence side and other two sides, respectively. Each vertical rectangle in

Figure 16 represents a normal graph production; the derivations of three sides are outcomes of pushout operations. The productions sp, cp, and tp are monotonic by the definition of a triple production. According to the execution rules of the pushout operation, all actions occurring during the transformation are adding or preserving, which means the source graphs are subsets of the derived graphs in all three sides. Therefore, they are monotonic productions. The second condition requires that sc=sc'|CG and tc=tc'|CG. This has already been proved in Lemma 3. □

This proposition shows that the execution of a triple production technically results in another triple production; the combination of the derived triple graph and the original graph through derivations that are obtained by pushout operations forms a triple production. Practically, a triple production represents a transformation rule of a model transformation model, and the result of an execution of the triple production is a correspondence association. Therefore, we can say that a correspondence association can be used as a transformation rule.

4.5 Collective Order Validity of a Correspondence Model

To establish the collective order validity, we use the local confluence theorem, which we introduced in section 4.1. The local confluence theorem has been investigated in the context of general graph grammar; but there is no theorem that is specific to the model transformation context. Although [40] discussed the local confluence theorem for a special model transformation case from Statecharts to Petri nets, they did not derive general theorems for model transformation. We find a general property of TGG by applying the local confluence theorem to TGG; i.e., we prove that the derived transformation rules of TGG are locally confluent, thus have the determinism property.

Dependency between triple productions

Before moving on to the proof of the determinism, we discuss dependency in TGG. We suggest an indirect approach to identify dependency among the derived transformation rules. Instead of directly handling dependencies between derived transformation rules, we identify the dependencies through dependencies between triple productions, from which the derived transformation rules are obtained. This indirect approach has advantages because the dependency of triple productions is easier to identify and allows us to handle the dependencies as a group.

In general, if a production (p2) depends on another production (p1), the application of p1 results in a condition under which p2 is applicable. This could be because p1 creates the components that are prerequisites for p2, or p1 deletes some components that prevent the application of p2.

The monotonic property of triple production limits the occurrence of dependency between triple productions; that is, the dependency is caused by only the former case, i.e., p2 is not applicable until p1 creates the required components. Since a triple production always adds some components, the latter case (i.e., deletion of prohibited components) cannot be a reason of dependency.

In this sense, given two triple production p1 and p2, p2 depends on p1 if the condition for application of p2 (i.e., BC(p2)) relies on the existence of p1 (i.e., AC(p1)\ BC(p1)). This statement can be mathematically expressed as follows:

Definition 26. Dependency of Triple Productions

Given a graph transformation system GTS = (G, P), where the initial graph G is a triple graph and P is a set of triple productions, for p1, p2 $\in$ P, p2 depends on p1 if

AC(p1)\ BC(p1) $\subset$ BC(p2).

This definition allows us to easily determine dependency between triple productions by finding the elements that meets the suggested condition. This is much

simpler than the dependency conditions of general graph production that are introduced in [36].

Dependency between two triple productions is easy to identify and can be used to identify dependencies among their derived transformation rules; the dependency of triple productions allows us to efficiently deal with the dependencies of the derived transformation rules. It is worthwhile to further explore the dependency of triple productions. The following proposition shows one way dependency: two triple productions do not depend on each other; that is, if triple production p2 depends on triple production p1, then p1 does not depend on p2.

Proposition 7. Given a triple graph transformation system GTS = (G, P), and p1, p2 ∈ P, if p1 and p2 are applicable, dependency between them is one-way; i.e, if p2 depends on p1, p1 does not depend on p2, and vice versa.

Proof). Suppose p1 and p2 depend on each other. By definition, the execution of p1 should be prior to that of p2, and vice versa; obviously, both cannot be executed. □

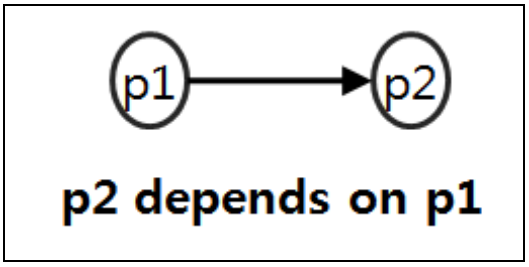

Figure 24. Dependency between Triple Productions

Figure 24 depicts the dependency between two triple production p1 and p2. It describes the dependency as a precedence relation between the triple productions, i.e., the application of p1 should precede that of p2. This graphical representation of the dependency is used to develop a graph that represents the all dependency relations in the entire set of triple productions. We call the graph a ‘dependency graph.’

Definition 27. Dependency Graph

Given a set of triple productions P, the dependency graph of P, DG(P), is defined as a graph (V,E), where vertex set V = P, and edge set E = {e | source(e) = p1 and target(e) = p2, and p2 depends on p1}

Technically, establishing the dependency graph does not require a lot of computation. Checking the condition of Definition 26 is nothing more than seeing if a graph includes some elements of other graphs. In addition, the number of triple production pairs we need to check is $n(n-1)/2$ for n triple productions. Both steps can be done within polynomial time with respect to the number of triple productions.

The dependency graph (V, E) is acyclic, i.e., it does not have any cycle. Existence of any cycle in the graph violates Proposition 7. There is a well-known algorithm that identifies a topological order of an acyclic graph. In a topological order, for every edge (node1, node2) of the graph, the order of node 1 is lower than that of node 2. See [41] for more details on the algorithm. Since the dependency graph is acyclic, we can easily obtain a topological order of the graph.

Definition 28. Topological Order of Dependency Graph

Given a set of triple productions P and the dependency graph of P, DG(P), TO(P) is a ordered set of the triple productions of P in a topological order. *toi*(p) is the index of $p \in P$ in TO(P).

This topological order plays a key role in proving determinism property of triple production, and coming up with the way to construct an efficient execution order of a triple production set.

Triggered Creation and Termination

Before we move on to the proof of determinism property, we need to discuss how the cross-over relation, which we introduced in section 4.3, affects behaviors of a graph transformation system; the cross-over relation may cause undesired behaviors such as

non-termination because it may bring about triggered creations. A triggered creation is a creation operation triggered by creation operations of other triple productions. In other words, it is a creation operation that is triggered not directly by modeler intention but by other triple productions.

A set of triggered creations could cause non-termination of a graph transformation system if their triggering sequence forms a loop; if this happened, the set of triggered creations would be repeatedly executed without terminating. This would cause infinite growth of the target graph. What is worse, such a loop might be non-terminating even without the infinite growth if the loop contains forced cancellation rules as well as triggered creation rules. Since the forced cancellation rules delete components without modeler intervention, a combination of forced cancellation rules and triggered creations may may be nonterminating because of the repeated deleting and creating of a certain set of triple productions.

For clear discussion, we introduce the formal definition of the triggered creation with the following notation.

- **GTS = (G, P)**: Triple graph transformation system
- **TP = ((SL,SR) ← sr – (CL,CR) – tr → (TL,TR))** : a triple production ∈ P
- **TP_i = ((SL_i,SR_i) ← sr_i – (CL_i,CR_i) – tr_i → (TL_i,TR_i)):** triple productions in P that have a cross-over relation with TP
- **CO(TP) = { TP_i }** : the set of all triple productions with cross-over relation with TP

Figure 25. Notations for Triggered Creation Issue

Definition 29. Triggered Creation Issue

Execution of TP may be triggered by other triple productions if there exists C⊂CO(TP) such that SR ⊂ $\bigcup_{i=1,..,n} SR_i \setminus SL_i$, or TR ⊂$\bigcup_{i=1,...,n} TR_i \setminus TL_i$.

SR is the part of the before condition graph of the left-to-right creation rule of TP (see Definition 21). $\bigcup_{i=1,..,n} SR_i \backslash SL_i$ is the union of the left side graphs of TP_i ∀ i. Since $\bigcup_{i=1,..,n} SR_i \backslash SL_i$ includes SR, a series of applications of TP_i may create the graph to which the left-to-right creation rule of TP is applicable; this can trigger an execution of the left-to-right creation rule.

Figure 22 shows an example. TP1 and TP2 together generate the complete left side graph of TP3; since it is the before condition of the left-to-right creation rule of TP3, AC(TP3)\BC(TP3) is automatically generated as the byproduct of TP1 and TP2.

Before going to the detailed discussion on the triggered creation, we introduce the following assumptions because disconnected graphs in production rules could cause unexpected results.

Assumption 1. Given triple production ((SL,SR) ← sr – (CL,CR) – tr → (TL,TR)), the after condition graphs of both sides (i.e., SR, TR) are connected graphs. A graph is connected if there exit a path between every pair of nodes.

Assumption 1 does not restrict the usage of triple production in that a disconnected triple production results in practically undesired outputs. Assumption 1 shows a triple production with a disconnected after condition graph in a typical model transformation between RDB schema and class diagram; the left graph, which consists of Table and Column, is disconnected. Since this graph condition does not specify any relation between a table and a column, it can be applied to any pair of them. Given a table *T*, the derived creation rule converts any column (even though it is irrelevant to *T*) to an attribute of the corresponding class that have been generated from *T*. In this sense, disconnection of a graph of a triple production practically means the triple production handles together modeling components that are irrelevant to one another. The connectivity assumption gets rid of this undesired case.

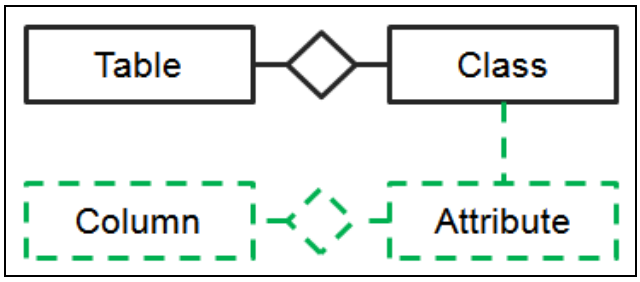

Figure 26. Triple Production with Disconnected Graphs

In the rest of this sub section, we show that triggered creations do not actually cause the non-termination problem in TGG.

Lemma 4. Given two triple productions p1 and p2 ∈ P, p2 cannot trigger p1 if there exist a dependency path from p1 and p2 in DG(P).

proof) One condition for triggering a creation rule of p1 is that the complete AC(p1) does not exist. If it exists, one creation rule of p1 has already been executed; any more creation rule cannot be applied to a target graph; we suppose that the current target graph has incomplete AC(p1).

A dependency path from p1 to p2 is a set of dependency relations that connect p1 and p2 in the dependency graph of P. Technically, if there exists a dependency path from p1 to p2, p2 depends on p1. There are two types of dependencies: direct dependency and indirect dependency.

i) Direct dependency means that p2 has a dependency relation with p1. In this case, a creation rule of p2 cannot be executed without the existence of complete AC(p1). Since the current graph does not has complete AC(p1), any creation rule of p2 is not applicable; if p2 does not create anything, it cannot trigger creation rules of other triple productions.

ii) Indirect dependency means that p2 relies on p1 through more than one dependency relations. In this case, BC(p2) does not contain any component of AC(p1); otherwise, p2 directly depends on p1. This means that any newly created component by p2 (i.e.,

e∈ AC(p2)\BC(p2) cannot have any edge with AC(p1); (AC(p2)\BC(p2)) ∪ AC(p1) is a disconnected graph. Therefore, any creation rule of p2 cannot result in complete SR or TR of p1 because both are connected graphs by Assumption 1.

In both cases, p2 cannot trigger a creation rule of p1. □

Lemma 4 shows how the dependency between triple productions affects the triggered creation between them. It says that a triple production cannot trigger a creation rule of another triple production that it depends on.

Lemma 5. A forced cancellation cannot cause a triggered creation, and vice versa**.**

proof) By Definition 23, a forced cancellation deletes some part of the target graph. As shown in Definition 29, a triple production p is produced by a triggered creation when other productions collectively create the complete graph of the left side (or right side) of p; obviously, deleting something cannot result in the complete graphs.

By Definition 21, a creation rule adds some components to the target graph. As shown in Definition 23, the application condition of a cancellation rule of a triple production p' is the incomplete (or partial) graph of BC(p') or AC(p')\BC(p'); in order for a triple production to trigger the cancellation rule, it delete some part(s) from BC(p′) or AC(p′)\BC(p′). A derived creation rule cannot do that because it does not delete but add something. □

Lemma 5 shows that forced cancellations and triggered creations cannot occur alternatively. Lemma 4 and Lemma 5 leads to the termination property of a set of triple production.

Proposition 8. A triple graph transformation system GTS = (G, P) is always terminating if P is a finite triple production set.

Proof) If a finite triple production set can be executed infinitely, there should exists a loop of triggered operations (i.e., forced derived rule, or triggered creation); since the

number of triple productions is finite, non-termination means that some production rules are executed repeatedly without modeler intervention. Without the triggered operation loop, this cannot happen. There are three types of triggered operation loops: a loop with only triggered creations, a loop with only forced cancellations and a loop with both triggered creations and forced cancellation.

i) The first type (with only triggered creation) cannot exist. This is because once a derived creation rule has been applied, it cannot be executed again until the creation is rolled back by a derived cancellation rule. A series of triggered creation cannot form a loop.

ii) The second type (with only forced cancellation rules) cannot be executed repeatedly by nature of deletion. We cannot delete something forever if there is no creation.

iii) The third type could form a loop as illustrated in the following figure.

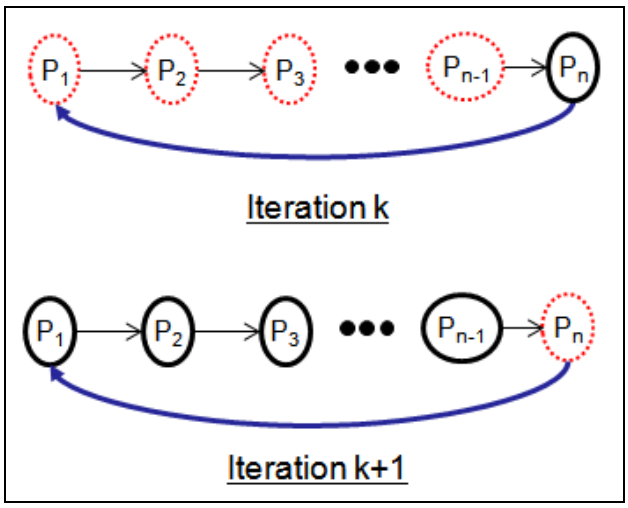

Figure 27. Mixed Triggered Operation Loops

In iteration k, forced cancellation rules are sequentially applied from P_1 to P_{n-1}. Because of Lemma 5, any triggered creations cannot occur in the middle of the iteration. The existence of P_n could trigger creation of P_1 before it is cancelled by its forced deletion triggered by P_{n-1}.

Iteration k+1 gets started as the result of the creation of P_1 triggered by P_n in iteration k. The creation of P_1 could cause a series of triggered creations all the way to P_{n-1} again. At this time, the absence of P_n could trigger the forced cancellation of P_1. This triggering could result in the same operations in iteration k again. In this way, iteration k and k+1 can be repeated infinitely.

Fortunately, this non-termination does not happen because Lemma 4 prevents the triggering from P_n to P_1 in iteration k. In the iteration, a series of forced cancellation are executed. By definition, the forced cancellation of a triple production is triggered as the result of deletions of other triple productions that it depends. This means a forced cancellation is triggered between two triple productions with a dependency relation. Therefore, $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n$ is a dependency path. By Lemma 4, P_n cannot cause a triggered creation of P_1.□

This proposition shows that a triple production set is terminating in spite of triggered creations. The termination property of TGG allows us to easily prove the determinism property of TGG in the next subsection.

Determinism of Triple Production

Thanks to the termination property of triple production, we can use local confluence theory to prove determinism property of TGG. As discussed in section 4.1, local confluence theorem allows us to prove the determinism by checking confluence property between direct graph transformations.

Lemma 6 is repeatedly used in the proof of Proposition 9 where the determinism of TGG is proven based on the local confluence theorem.

Lemma 6. Given a graph G and two productions p1 and p2, if only one of the two productions is applicable, there exists confluence graph H that satisfies the condition of local confluence in Definition 15.

Proof) Suppose only production p1 is applicable. The two direct graph transformations of p1 and p2 result in $G \overset{p1}{\Rightarrow} H_1$, and $G \overset{p2}{\Rightarrow} G$ respectively. Note that the result of p2 is the initial graph G because it is not applicable. We can easily convert H_1, and G into the confluent graph in the following way: $G \overset{p1}{\Rightarrow} H_1 \overset{\emptyset}{\Rightarrow} H1$, $G \overset{p2}{\Rightarrow} G \overset{p1}{\Rightarrow} H_1$. $\emptyset$ denotes no transformation. □

Proposition 9. A triple graph transformation system GTS = (G, P) has the determinism property.

Proof) The local confluence theorem requires us to show the confluence property between every pair of direct graph transformations. Instead of taking into account all possible combination of individual derived transformation rules, we use the classes of derived transformation rules: derived creation rule, and derived cancellation rule (see section 4.1). At most one derived transformation rule of each class is applicable to a given graph. Furthermore, all derived transformation rules of each class result in an equivalent graph; i.e., all creation rules generate complete AC(p)\BC(p), while all cancellation rules delete the existing AC(p)\BC(p). In this sense, the following four combinations between the two classes can represent all possible cases.

Let dt_1 and dt_2 denote the two direct graph transformations from the initial graph G, respectively. Suppose there are two triple production p_i and p_j where $i < j$. Index i and j are determined by the topological order of the dependency graph (see Definition 27). By the definition, $i < j$ means that p_j depends on p_i. Due to the one way dependency proven in Proposition 7, p_i and p_j can represent all possible relations between two arbitrary triple productions.

Because of Lemma 6, it is enough to show the local confluence property of dt_1 and dt_2 when both of them are applicable.

Case 1: $dt_1 \in CR(p_i)$, and $dt_2 \in CR(p_j)$

Only one of them is applicable: i) if dt_1 is applicable, dt_2 is not, because p_j depends on p_i. Any derived creation rules of p_j is not applicable until p_i is applied; ii) as the reverse direction, suppose that dt_2 is applicable. This means dt_1 has already been applied; hence, dt_1 is not applicable to the current graph anymore.

Case 2: $dt_1 \in CR(p_i)$, and $dt_2 \in DEL(p_j)$

If dt2 is applicable, one element of $CR(p_j)$ must have been applied in one of the previous steps. This means $CR(p_i)$ has also been applied; the only way of applying any creation rule of p_i is that existing p_i is cancelled and executed again by a triggered creation. Figure 23 shows the example.

dt2 does not affect the applicability of dt1 because p_i is independent of p_j; we need to see if dt1 affects the applicability of dt2 for two cases: $dt2 \in IDEL(p_j)$, and $dt2 \in FDEL(p_j)$.

i) $dt2 \in IDEL(p_j)$: This means that some part of $AC(p_j) \backslash BC(p_j)$ is missing. dt1 cannot add the missing parts because they can be created by only $CR(p_j)$. dt2 is still applicable.

ii) $dt2 \in FDEL(p_j)$. This implies $AC(p_i)$ does not exist; edges between $AC(p_i)$ and $AC(p_j) \backslash BC(p_j)$ cannot exits. The creation of pi does not reconstruct these edges; they are generated when p_j is created. By Assumption 1, dt1 cannot restore the complete graph of $AC(p_j)$; dt2 is still applicable.

Since dt1 and dt2 do not affect the applicability of each other, they have the local confluence property; i.e., $G \overset{dt_1}{\Rightarrow} H1 \overset{dt_2}{\Rightarrow} H$, and $G \overset{dt_2}{\Rightarrow} H2 \overset{dt_1}{\Rightarrow} H$.

Case 3: $dt_1 \in DEL(p_i)$, and $dt_2 \in CR(p_j)$

If both dt1 and dt2 are applicable, p_j does not directly depends on p_i. This is because if dt_1 (the cancellation of p_i) is applicable, the target graph does not have $AC(p_i)$ in the

complete form; dt2 cannot be executed because the existence of p_j directly depends on p_i.

In the case of indirect dependency, dt1 does not affect the applicability of dt2 because $BC(p_j)$ does not have any component of $AC(p_i)$. In addition, dt2 does not create the complete graph of AC(pi) because of Lemma 4. Application of dt2 does not prevent the execution of dt1. dt1 and dt2 have a local confluence property, ie., $G \overset{dt_1}{\Rightarrow} H1 \overset{dt_2}{\Rightarrow} H$, and $G \overset{dt_2}{\Rightarrow} H2 \overset{dt_1}{\Rightarrow} H$.

Case 4: $dt_1 \in DEL(p_i)$, and $dt_2 \in DEL(p_j)$

A cancellation rule does not affect the applicability of other cancellation rules because deleting components cannot result in the after conditions of a triple production, which prevent the cancellation operations of the triple production. dt1 and dt2 is independent of each other; they have a local confluence property, ie., $G \overset{dt_1}{\Rightarrow} H1 \overset{dt_2}{\Rightarrow} H$, and $G \overset{dt_2}{\Rightarrow} H2 \overset{dt_1}{\Rightarrow} H$.

In consequence, GTS is locally confluent. This conclusion and Proposition 8 together assure that a triple graph transformation system has always determinism by Lemma 2.

□

Execution order generation

We have proven the determinism of triple graph transformation system. This property allows us to make a correspondence model a completely valid model transformation model by constructing an execution order. Since any arbitrary execution order leads the model transformation to an equivalent result at the termination, it is not theoretically important to discuss how to construct the order. However, it is practically important because model transformation usually has to handle large models. It is not uncommon, for example, that a factory model has thousands entities. The following

proposition shows the worst case in terms of the number of execution steps that a model transformation takes until reaching the termination.

Proposition 10. Given a triple graph transformation system with n triple productions in its production set, the graph transformation system takes $O(n^2)$ attempts in the worst case, when reaching the termination.

Proof), A finite number of transformation rules are derived from a triple production. c denotes the number; there are c•n derived transformation rule. By definition, at least one rule should be applicable to the current graph if the graph transformation is not terminated. Suppose that only one of the derived transformation rules is applied at the last step, i.e., c•*n*-1 other derived transformation rules fail to be applied and then the last rule is applied. Since the graph is modified by the derived graph, we need to check the applicability of the derived transformation rules to the new graph. Suppose that a derived transformation rule of triple production can be executed only once; then, we can rule out the derive rule that has already been executed.

Suppose that the same thing happens in every iteration, i.e., only one derived transformation rule is applied in the last step of every iteration. In this case, we make c•*n* - *k* attempts at *k*th iteration. Therefore, the worst case is $\sum_{k=1}^{n}(c \cdot n - k)$ = $O(n^2)$

□

```
For a triple graph transformation system GTS = (TG, TP),

Step 1. generate a topologic order of TP, TO(TP)

Step 2. apply derived cancellation rules in the order of TO(TP)
for all p ∈ TP in the order of TO(TP)
        for all dr ∈ DEL(p)
                if dr is applicable, then
                        apply it to the current graph (CTG)
                        quit the for loop
                end if
        end for
end for
```

```
Step 3. apply derived creation rules in the order of TO(TP)
for all p ∈ TP in the order of TO(TP)
        for all dr ∈ CR(p)
                if dr is applicable, then
                        apply it to the current graph (CTG)
                        quit the for loop
                end if
        end for
end for
```

Figure 28. Algorithm that executes derived transformation rules

We suggest a way to construct an efficient execution order. We use the topological order of dependency graph (see Definition 27). Figure 28 shows the algorithm that executes the derived transformation rules of a triple graph transformation system.

Basically, this algorithm executes all derived cancellations rules in step 2 and then applies all derived creation rules in step 3. In the both steps, we use a topological order of the dependency graph to sort derived transformation rules. This order ensures that any derived transformation rule does not fail to be applied at the first attempt if the derived transformation rule is applicable eventually; this is the order that takes the fewest steps to the termination. The topological order works differently for step 2 and step 3.

In step 2 for derived cancellation rules, the topological order prevents failures of applications for the following reason. By definition, intended cancellation rules can be applied independently of other rules. However, a forced cancellation rule (*fc*) is affected by other triple productions; it is caused by cancellations of other triple productions. This means that if *fc* is applied prior to the cancellation of the other triple productions that it depends on, we fail to apply the forced cancellation rule. By the definition, the topological order of the dependency graph ensures that the forced derived cancellation rules that *fc* depends on precedes *fc*.

In step 3 for derived creation rules, it is more straightforward. By the definition, a derived creation rule of a triple production p can be applied when the complete before condition graph of p exists. This before condition graph is created as the results of applications of the other triple productions that p depends on. By the definition, the other triple productions that p depends on precede p in the topological order. Since we do not specify p, this is true for all triple productions of the triple graph transformation system; the topological order executes the applicable derived creation rules without failure.

In addition, executing derived cancellation rules prior to derived creation rules increases the efficiency practically. If we apply creation rules first, there may be a case where a graph created by one creation rule of a triple production (p) is deleted by the corresponding forced cancellation of p triggered by the cancellations of the triple productions that p depends on. The back triggering operation in iteration k+1 of Proposition 8 show the case. This just creates and deletes the graphs for nothing; practically, it consumes unnecessary computational power. Our algorithm has better performance than the reverse order (i.e., derive creation rules first, and cancellation rules later).

Finally, the following proposition shows the number of execution steps of our algorithm so as to compare the efficiency with the worst case, $O(n^2)$.

Proposition 11. Given a triple graph transformation system with n triple productions in its production set, the number of execution steps of our algorithm is O(n).

Proof) An algorithm that generates a topological order for an acyclic graph in O(n) has been known (see [41]). In addition, step 2 and step 3 repeat the sub routines n times. Since all steps run in O(n), the entire algorithm also runs in O(n). □

4.6 Conclusion

In order to show multi layer model is theoretically possible, throughout this section, we prove the two key properties of TGG: recursive property, and determinism property. It is relatively easy to prove the first property. The monotonic production of triple production plays an important role in proving the properties. We reuse a number of theorems that have already been proven in other literatures.

The more significant contribution to is the second property: determinism property. To our knowledge, this is the first attempt to explore collective behaviors of TGG in terms of dependency and termination of graph transformation.

First of all, we introduce additional derived transformation rules so as to deal with more realistic situation of model transformation. Derived cancellation rules have been introduced to handle deletion operations. In addition, we indentify cross-over case where we need to use more than one triple production, which are related to one another, to derived transformation rules. We proposed how to obtain derived transformation rules in this case. We also find that the cross-over case causes a triggered creation issue, which makes analysis of dependency of triple productions much more difficult.

Second, we explore how derived transformation rules affect one another. In Definition 26, we define dependency between triple productions. Because of the monotonic production of TGG, we can assume one way dependency. The one way dependency allows us to develop an acyclic dependency graph, which shows the dependencies among all triple productions in a triple graph transformation system.

Third, we show a set of triple productions is terminating even if triggered creations occur. Lemma 4 shows that if a triple production p2 depends on another triple production p1, a triggered creation from p2 to p1 does not happen. This property ensures the termination of TGG by preventing a loop of triggered operations. With the termination property and connectivity assumption (Assumption 1), we proved the determinism property of TGG using local confluence theory.

Finally, we suggested an algorithm to get an efficient execution order of a set of triple productions using a topological order of the dependency graph.

In the following chapters, the key results of this chapter play important rules in implementing multi layer syntactical model transformation.

CHAPTER 5

SYNTACTICAL MODEL TRANSFORMATION

This chapter demonstrates the syntactical model transformation using the classical transportation problem, a typical problem in optimization. We use a concrete scenario where we store the data for the problem in an RDB and solve the optimization formulation using AMPL. As the technical framework, we use EMF (Eclipse Modeling Framework) because a great number of groups are actively developing various tools supporting formal modeling based on the common meta-modeling framework, Ecore [27].

We survey existing approaches for text-based representation of a model. Among a number of types of approaches, we use an EBNF-based hybrid approach because it supports a complete meta-model that we mentioned as the key part of the syntactical model transformation. The approach allows us to define the complete meta-model by incorporating the syntax definition capability of EBNF language into the semantic meta-modeling framework. We use Xtext, which is based on EMF, as the implementation tool [18].

We implement the syntactical model transformations for two different tools: MS Access™ and AMPL. Technically, they have different syntactical bases; MS Access™ is based on XML, whereas AMPL is based on plain text files. It turns out that our approach more efficiently supports MS Access™ than AMPL. To shed light on desired aspects that a good syntactic modeling framework should have, we explore what aspects of XML contribute to the efficiency.

Our contribution to improvement of general EBNF-based approach

We practically improve the EBNF-based approach. To our knowledge, no existing tool of this approach allows us to define a syntax that represents a reference between classes. In the meta-modeling framework, the reference plays a very important role in

describing relations between classes. Other modeling domains have analogous concepts for the same purpose, .e.g., foreign key in RDB domain, and indexing between sets in optimization domain. In spite of the conceptual similarity, tools use different syntaxes for the reference concept. Therefore, the EBNF-based approach has to be able to accommodate the syntactical diversity of the reference concept. To address this issue, we suggest an advanced way to define the syntax of the identifier of the referred class using the attributes of the class. As a demonstration, we apply this suggestion to Xtext.

5.1 Demonstration Example: Transportation Problem

In Chapters 5 and 6, we use a simple transportation model example, which is represented in both a relational database (RDB) and an optimization analysis tool. This simple example allows us to test whether our approach can support transformation of a key advanced modeling concept, a compound object that is defined by a combination of other objects. For example, 'Link' of the transportation problem is defined by a combination of an origin and a destination. Different modeling standards differently describe the compound object; e.g., a special type of table that links other tables using foreign keys in RDB, a compound set in optimization modeling, an association class in object oriented modeling. We demonstrate that our approach can cope with those differences.

Table 3. List of Technical Tools

Tool	Purpose
EMF (Eclipse Modeling Framework)	Meta-modeling framework
Ecore	Meta-model of EMF
ATL	Model transformation
AMPL	Optimization modeling language
MS Access™	Relational database
Xtext	Syntactical grammar definition tool
XML/XSD	Syntactic representation of MS Access™

In order to demonstrate our suggested approach, we use the scenario depicted in Figure 29 with technical tools listed in Table 3. In the scenario, models and instance data are developed and transformed in three steps:

(1) A domain expert develops the transportation problem model as a PIM (Platform Independent Model) using Ecore. The model is transformed to a schema model of RDB and a mathematical problem description in AMPL.

(2) Instance data is populated to the database tables defined by the schema models transformed in step 1. Since RDB is the most common data repository in practice, we assume that instance data is created and stored in RDB.

(3) The instance data of RDB is transformed to an AMPL data model for optimization analysis. The mapping rules that are necessary for the transformation are generated from the result of the model transformation of step 1 through the multi layer model transformation.

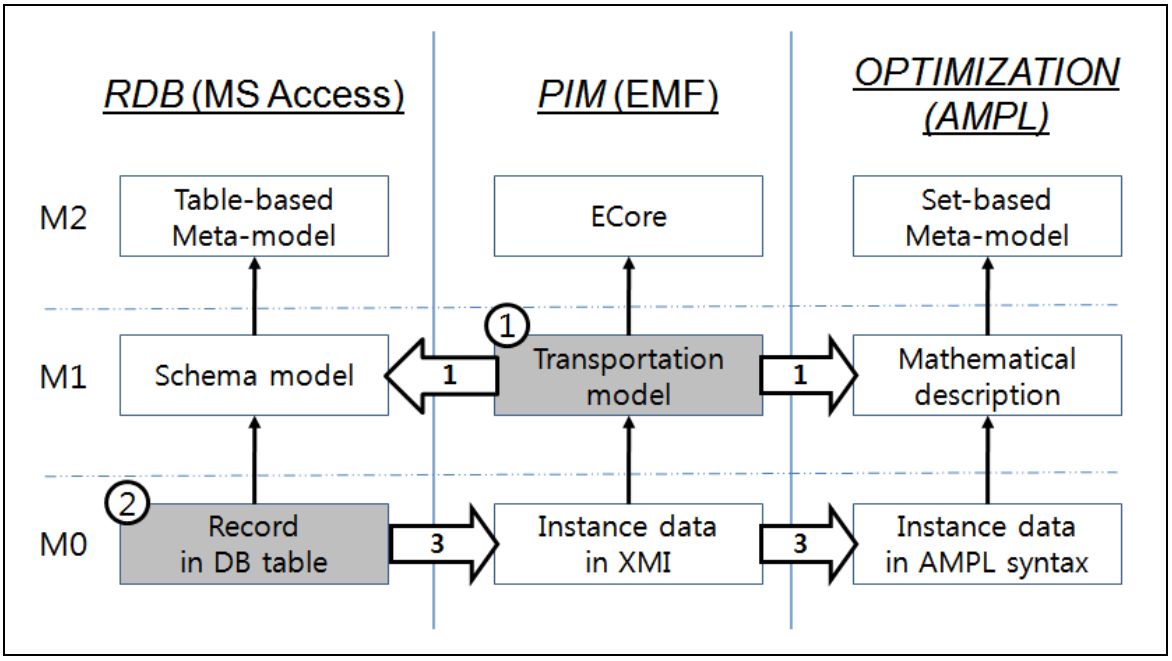

Figure 29. Implementation Scenario

Throughout the scenario, we apply our multi layer model transformation (the next chapter) and syntactical model transformation (this chapter) in the following ways:

(1) Multi layer model transformation is used to generate transformation rules for step 3 from the result of step 1. In step 1, we obtain correspondence associations as the result of the model transformation. The multi layer model transformation converts those associations into executable model transformation scripts that execute step 3, instance data integration. We will demonstrate this in Chapter 6.

(2) Syntactical model transformation is used to make user models and instance data compatible with the target tools in both M1 and M0. On the RDB side, the syntactical model transformation uses ATL to generate MS Access compatible schema model (M1) and record data (M0) in XML documents; in Optimization side, it uses Xtext to generate AMPL compatible mathematical description (M1) and instance data (M0) in the form of plain text. This chapter demonstrates the syntactical model transformation.

5.2 Linguistic Analysis of Target Modeling Domains

In order to properly apply our approach to the target modeling domains: RDB and optimization, we need to understand their modeling concepts and syntactic representations in terms of the layered language formalism. We analyze them in three language layers – meta-model, user model, and instance data - using a simple transportation example. In addition, we provide the syntactical representations of the models in terms of implementation tools: MS-Access for the RBD domain, and AMPL for the optimization domain.

Meta-model (M2)

A meta-model is a set of fundamental constructs which are used to describe a problem in a domain of interest; it is an abstract language used for developing domain

specific models. The optimization domain has set-oriented meta-model, whereas relational database (RDB) domain has table-oriented meta-model.

An optimization model is expressed using mathematical constructs. Parameters, variables, and equations are indexed over sets; hence, these can be regarded as the elements of the meta-model of the optimization domain. Table 4 lists the key components of the set-oriented meta-model from [13], which refers specifically to AMPL, but this set-oriented concept is commonly used in most major optimization modeling languages.

Table 4. Key Components of Set-oriented Meta-model

Component	Definition
Set	a collection of objects with common properties
Parameter	an attribute that characterize a set
Variable	a variable determined by solving the optimization problem
Constraint	a logical expression of the condition that must be satisfied
Index Relation	a relation that links a parameter, a variable, or a constraint to a set of objects that hold them

Figure 30 depicts a class diagram that describes the set-oriented meta-model. This diagram shows relations among the key components, and their attributes. The compound set is a combination of other sets and is used to define parameters or variables that do not belong to a single component. In the transportation example, a transportation link is a compound set; it is described as an ordered pair of a supply node and a demand node. The compound set refers to the base sets through ‘referenceSet’ relation. Hereafter this model will be used as the meta-model of the optimization domain.

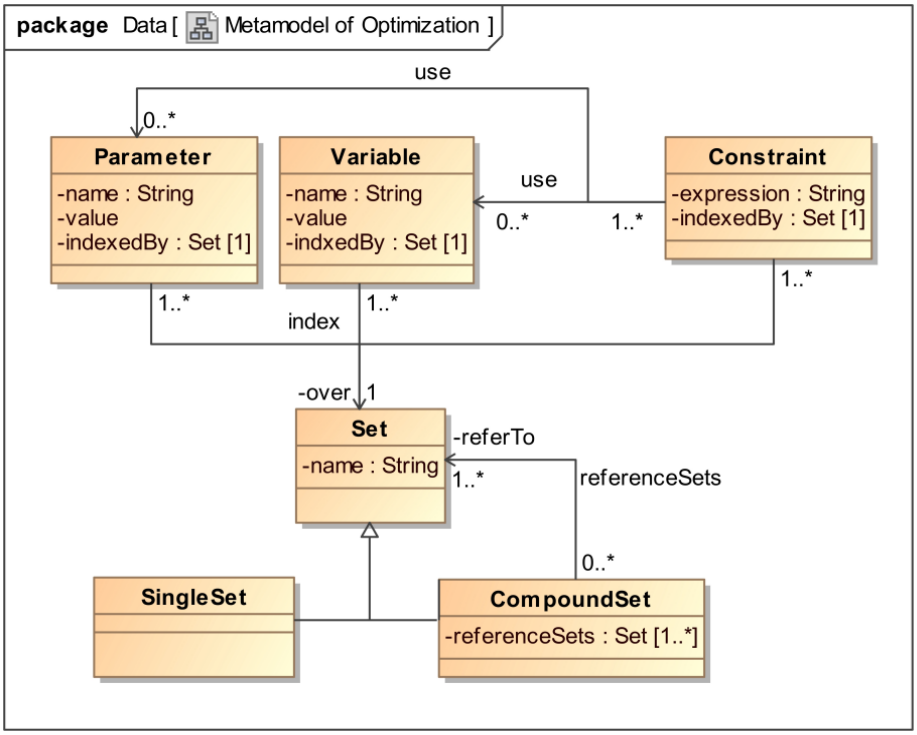

Figure 30. Class Diagram of Set-oriented Meta-model

Figure 31 displays the meta-model of RBD domain. This meta-model is simple, but it has been proven to have enough expressiveness to capture almost all schema models; it is commonly accepted by all relational database tools.

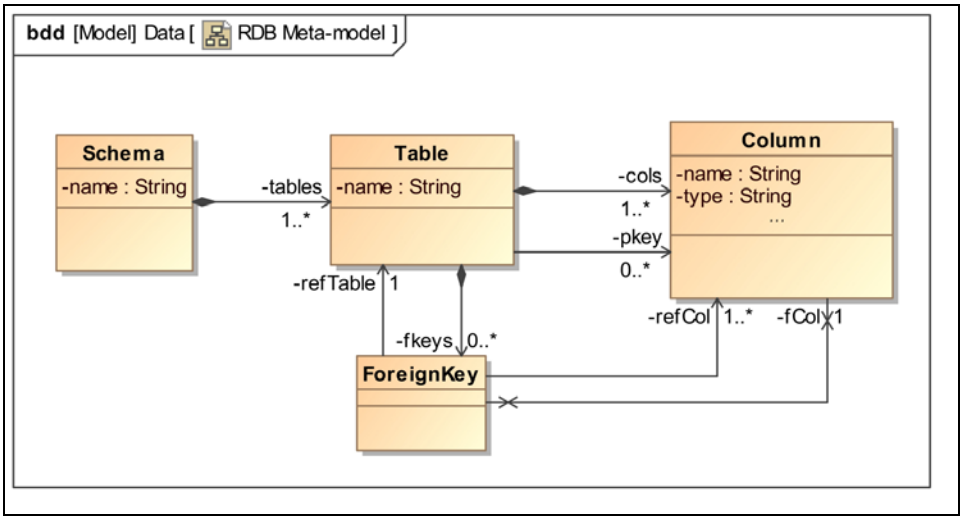

Figure 31. Class Diagram of Table-oriented Meta-model

We use these two meta-models throughout our implementation. As mentioned, these meta-models define modeling concepts with which the domain problem is described in the subsequent user model.

User model (M1)

In the user model layer, we concretely describe the transportation problem. The objective of the transportation problem is to determine minimum cost material flows that meet demands at destination nodes without exceeding capacity of supplying nodes, or the capacities of links between the nodes.

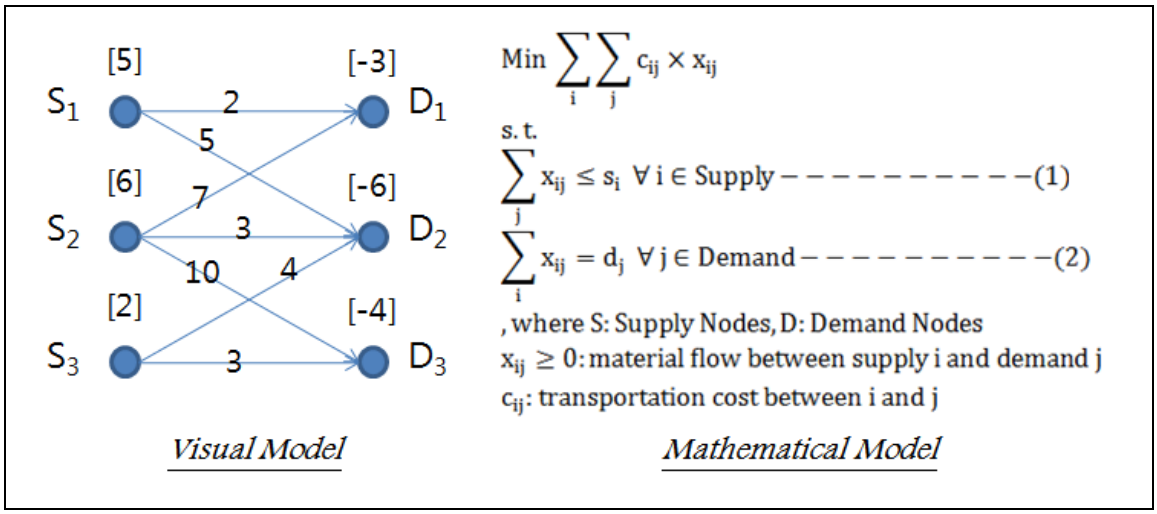

Figure 32. Transportation Model

Figure 32 gives both graphical and mathematical representations of the transportation problem. The source supplies and destination demands are remarked on nodes as positive or negative number. Arrows between the nodes represent possible links through which the materials can flow. Numbers noted on the arrows are transportation costs per unit material flow represented in the mathematical model as c_{ij}. Constraint (1) and (2) describe supply and demand balance conditions, i.e. a summation of outgoing material flows (or incoming material flows, respectively) of a node has to be equal to the supply (or demand, respectively) of the node. $\mathbf{x_{ij}}$ represents the amount of a flow from

node i to j. Thus it is straightforward that $\sum_j x_{ij}$ (or $\sum_i x_{ij}$, respectively) is the summation of all outgoing flows from node i (or the summation of all incoming flows to node j, respectively).

In the optimization domain, the mathematical representation in Figure 32 is captured using the set-oriented meta-model (Figure 30). Figure 33shows the model in AMPL syntax. The correspondence between the mathematical representation and the AMPL model is obvious, except for the link representation. The mathematical model describes the properties of the link by a combination of two subscript indexes (i.e., i and j), whereas AMPL explicitly defines Link as a compound set.

```
Set Supply; # Supply nodes
Set Demand; # Demand nodes
Set Link within{ Supply, Demand }; # Possible linkages
param s {Supply}; # amount of available materials
param d {Demand}; # amount of required materials
param c {Link}; # cost of links
var x {Link}; # amount of material flows
minimize Total_cost:
      sum {(i,j) in Link} c[i,j] * x[i,j];
subject to Supply{i in Supply}:
      sum {j in Demand} x[i,j] ≤ s[i]; --------------(1)
subject to Demand{j in Demand}:
      sum {i in Supply} x[i,j] = d[j]; --------------(2)
```

Figure 33. Transportation User Model in AMPL Syntax

The AMPL model of Figure 33 captures the underlying structure of the transportation problem without specific data. For example, the set Supply is just an abstract declaration of supply nodes without specifying concrete instance, and constraint (1) is imposed on the abstract declarations. This absence of specific instance data allows us to reuse the user model for other problems within the domain; we can obtain the concrete descript of our problem simply by filling the user model with problem specific instance data. In that sense, the use model captures domain specific knowledge in a reusable way. This is the reason people call the user model ‘domain specific model’ as well.

Figure 34 depicts the RDB schema model of the transportation problem. Rectangles represent tables, while texts in the rectangles are columns of the tables. Edges show foreign key relations between the columns in different tables.

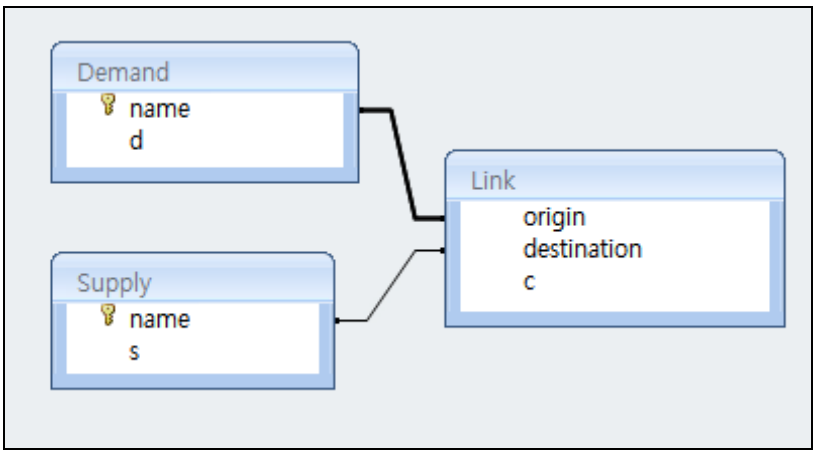

Figure 34. Transportation User Model in RDB Schema

Although the visual representation is easy to understand, we need the textual models of the schema for model transformation. MS Access supports various export and import formats. Among them, we use XML because it is a widely used standard textual data format. Recently, most COTS tools use XML as their primary data format, or input and output format. Furthermore, various APIs and tools have been developed for all important programming development environments such as JAVA, .NET, etc. Therefore, it is practically important to see how well model transformation handles XML format.

```
- <xsd:element name="Link">
    - <xsd:complexType>
        - <xsd:sequence>
            - <xsd:element name="origin" minOccurs="0" od:sqlSType="nvarchar" od:jetType="text">
                - <xsd:simpleType>
                    <xsd:restriction base="xsd:string"></xsd:restriction>
                  </xsd:simpleType>
              </xsd:element>
            - <xsd:element name="destination" minOccurs="0" od:sqlSType="nvarchar" od:jetType="text">
                - <xsd:simpleType>
                    <xsd:restriction base="xsd:string"></xsd:restriction>
                  </xsd:simpleType>
              </xsd:element>
              <xsd:element name="c" minOccurs="0" type="xsd:int" od:sqlSType="int" od:jetType="longinteger"/>
          </xsd:sequence>
      </xsd:complexType>
  </xsd:element>
```

Figure 35. Transportation User Model in XSD

Figure 35 represents 'Link' table of Figure 34 in the format of XSD. 'Link' table itself is defined using complexType, under which all the columns of the table are defined as xsd:element. The correspondence between the RDB schema and the XSD definition is quiet clear. The XSD also specifies how to describe XML in M0. We will discuss this in details in section 5.3.

The transportation problem is described differently in the two domains from two perspectives. First, different meta-models lead to different descriptions; the transportation problem is captured in different modeling concepts. Second, RDB model does not have constraints. As mentioned, mathematical constraints are specific to optimization model. We capture the constraints only where they are needed.

Instance Data (M0)

The instance data instantiates the abstract declarations defined in the user model (M1) by assigning concrete data. Figure 36 shows the instance data in AMPL data syntax.

```
Set Supply := S1, S2, S3 ;  ----------------------(1)
Set Demand := D1, D2, D3 ;  ----------------------(2)
Set Link := {S1,D1} {S1,D2} {S2,D1} {S2,D2} {S2,D3} {S3,D2} {S3,D3}; --(3)
param s := S1 5 S2 6 S3 2;
param d := D1 3 D2 6 D3 4;
param c := S1 D1 2 S1 D2 5 S2 D1 7 S2 D2 3 S2 D3 10 S3 D2 4 S3 D3 3;
```

Figure 36. Instance Data in AMPL Syntax

$x_{S1D1} + x_{S1D2} \leq 5$ The APML instance data specifies only sets, and parameters. This is because constraints can be determined at runtime of an optimization solver once parameters and sets that index the constraints are given. For example, the supply balance condition of supply node, S_1, can be expressed as '$x_{S1D1} + x_{S1D2} = 5$'. Link set, which indexes variable x, specify node S_1 is connected with node D_1 and D_2; the right term of the equation is the summation of the material flows in the two links. Supply capacity is given by parameter s. In this way, an optimization engine can generate the instances of the constraint equations at runtime. This allows the modeler to avoid effort to repeatedly

write down the constraint equations with the same pattern. It makes a significant contribution to efficiency in developing optimization models.

```
<Link>
      <origin>S1</origin>
      <destination>D1</destination>
      <c>2</c>
</Link>
<Link>
      <origin>S1</origin>
      <destination>D2</destination>
      <c>5</c>
</Link>
```

Figure 37. Instance Data in XML Syntax

In RDB, the instance data are stored as records of the tables defined in the database schema of M1. As the textual representation of the data, we also use XML for the same reason of M1. Figure 37 shows part of the instance data in XML. It has two links in top level. All the attributes of links are presented as nested elements, which are defined as xsd:element in Figure 35. Note that the rules with which the XML document must comply are defined in the XSD of Figure 35. The linguistic relation between the XSD and the XML will be discussed in the next section.

5.3 Syntax Generation in Modeling Framework

The syntactic issues identified above arise in many practical cases where COTS tools are involved. In order to address the issues, there have been efforts to incorporate syntactic manipulation capability into meta-modeling frameworks; e.g., OMG has issued an RFP for model to text transformation based on MOF [42].

There are two streams of these efforts. The first stream is model-to-text transformation, in which textual representations are generated by associating formal models with predefined string patterns. There are a number of branches within the stream depending on the way of specifying the associations [43].

Although these approaches are easy to use, there is a significant drawback; the approaches cannot be used for importing textual models. The main reason is that the association processes do not have formal ways to define language grammars; that is, the string patterns are just defined by mixtures of quoted text streams and dynamic parts referring to the information contained in the models without any grammatical definitions. The grammatical definitions are not critical when writing the textual models because the writing processes can be done simply by putting together the patterns and the information from the formal models. However, the reverse process, importing the models, is not simple; it requires a parser that understands input streams and can convert them into abstract models. This parser cannot be generated without formal grammatical definitions.

A second stream that addresses the lack of grammatical formalism has been suggested by several research groups [18, 44-46]. Basically, this stream incorporates language grammar definition forms, such as BNF (Backus-Naur Form) or EBNF (Extended BNF), into the meta-modeling frameworks. BNF and EBNF are meta-syntaxes that define context-free grammar of languages so that there is no ambiguity regarding what is allowed in a language and what is not [47]. With mathematical language theories, unambiguous grammars are used to automatically generate a parser for the languages. The key point of this second stream is to incorporate this capability of generating a parser into the meta-modeling framework. As an example of this stream, we describe how Xtext [18] works, because we use it for our implementation.

Hybrid Approach (Xtext)

Xtext implements the complete meta-model concept by combing an EBNF-based language development framework with EMF (Eclipse Modeling Framework), which is one of the major meta-modeling frameworks [48]. The key of the integration is a new grammar definition language that is extended from EBNF so that it can define features of EMF; that is, the extended EBNF is able to define not only a syntactic model for the

language grammar but also semantic definitions for the meta-model. That is what the complete meta-model has to do. Interestingly, EBNF and the meta-models (EMF, and MOF) have a close correspondence, which has been identified in [45]. Due to the correspondence, the extension has been accomplished by simply adding 'Assignment' that deals with definitions of attributes and cross references (e.g., name=ID, and indexedBy=[Set] in Figure 38). The Xtext framework converts the hybrid grammar definition into two components: an ANTLR parser [49] from the syntactical grammar, and a Ecore model from the semantic definitions. Technically, the two components are implemented in Java; the set of Java classes semantically has the equivalent structure to the Ecore, and the Java classes are able to parse and serialize models in accordance with the syntactic specification through the generated ANTLR parser.

The hybrid characteristic of Xtext enables us to take advantage of both EMF and EBNF. On the one hand, the generated Ecore model allows us to work with any EMF-based tools independently of the syntactic model. On the other hand, the generated parser enables us to read and write the permanent representation of the Ecore model in accordance with the grammar.

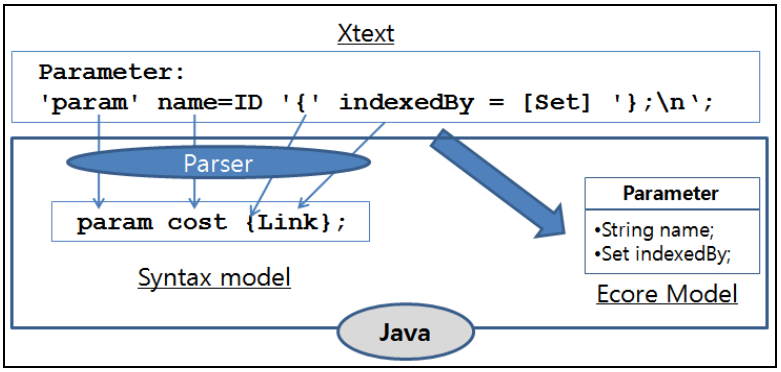

Figure 38. Hybrid Operation of Xtext

Figure 38 shows how a simple rule in an Xtext script generates the above two key components. For the Ecore model, 'Parameter', which is the name of the Xtext rule, goes to the name of the generated class. This rule also has the two assignments, i.e., 'name'

and ‘indexedBy’. They turn to a simple attribute and a cross-reference of the class, respectively. The cross-reference, ‘indexedBy’, refers to another class, ‘Set’. When it comes to the syntax model, the generated parser puts the quoted strings (i.e., ‘param’, ‘{‘, and ‘};\n’) into the syntax model as they are. For the assignments, it puts assigned values into the appropriate places; in the example, the second string, ‘cost’, is a value for the simple attribute, name, while ‘Link’ between brackets is the name of a ‘Set’ object that the cross-reference points to. For more detail, see [18].

Special Case: XSD-based Approach for XML

An XML schema document (XSD) is a document that defines schema for XML documentation [50]. The schema describes a set of rules that valid XML documents must conform to. The schema is used not only to populate XML documents in conformance with the schema but also to validate XML documents.

In this subsection, we describe the XSD-based approach - as a special case of the hybrid approach. As mentioned, the hybrid approach has to support the two aspects of the complete meta-model: the semantic meta-model and the meta-syntax (syntactic meta-model). XSD allows us to more simply specify the both meta-models than the general hybrid approach does. This is because the syntactic representation of an XSD construct is standardized; once we select a XSD construct to define a semantic component, the syntactic representation of the component is determined by the XML syntax standard. In this sense, XSD enables us to define the complete meta-model of a XML document without need of manually defining syntactic specifications.

The following two figures show how the syntax standard works between an XSD definition and its XML syntax. We use ‘Link’, one important component of the transportation problem, as an example. In Figure 39, we define the semantics of ‘Link’ using XSD components; ‘Link’ itself is described as ‘xsd:complexType’, while ‘cost’ (or capacity, respectively) is defined in ‘xsd:element’ (or ‘xsd:attribute’ respectively). The

syntaxes of the semantic components are determined in accordance with the XML syntax standard. ‘Link’ is represented as the top tag. ‘cost’, which is captured in ‘xsd:elment’, is turned into the separate tag nested in the top Link tag (see (2) in Figure 40). In contrast, ‘capacity’, which is defined as ‘xsd:attribute’, is converted into the inner tag attribute of the Link tag (see (1) in Figure 40).

```
<xsd:complexType name="Link">
      <xsd:sequence>
            <xsd:element name="cost" type="xsd:double"/>
            <xsd:attribute name="capacity" type="xsd:double"/>
      <xsd:sequence>
</xsd:complexType>
```

Figure 39. XSD Definition of Link

```
<Link capacity=10> ------------------------------(1)
      <cost>12.0</cost> -------------------------(2)
</Link>
```

Figure 40. XML Syntax of Link

The above example shows we can manipulate the syntax of XML by choosing different xsd components; semantically, both ‘cost’ and ‘capacity’ are simple attributes of ‘Link’, but they are described differently in Figure 40 because they are specified in different xsd components.

This standardization, of course, limits freedom of syntactic expression; the general syntaxes that the standard does not cover cannot be expressed. However, as long as we handle XML documents, the standardization makes it much easier to create the syntactical model specification; indeed, the implementation of the syntactical model transformation is much easier in XSD-based approach than in EBNF-based approach. We will discuss the details in section 5.5. The XSD-based approach is practically attractive in that more and more contemporary COTS tools support XML formatted data.

5.4 Syntactical model transformation with AMPL

We implement the syntactical model transformation for AMPL using a general EBNF-based tool, Xtext. As discussed in chapter 3, in order to implement the syntactical model transformation, we need to develop two complete meta-models for AMPL in M1 and M0. The following two figures shows the complete meta-model written in Xtext script.

```
OptModel:
       (declarations+=Declaration)*;  ------------------------------(1)
Declaration:
       Set | Parameter| Variable;
Set:
       SingleSet | CompoundSet;
SingleSet:       ----------------------------------------(2)
       'set' name=ID ';\n';
CompoundSet:         ----------------------------------------(3)
       'set' name=ID 'within {'
referenceSets+=[Set1] (','referenceSets+=[Set] )* '};\n';
Parameter:       ----------------------------------------(4)
       'param' name=ID '{' indexedBy = [Set] '};\n';
Variable:         ----------------------------------------(5)
       'var' name=ID '{' indexedBy = [Set] '};\n';
```

Figure 41. Xtext Script for the Complete Meta-model of AMPL User Model

Figure 41 is the Xtext script that defines the complete meta-model of the user model in optimization domain and AMPL syntax. In the notation introduced in Chapter 3, it can be denoted as CM(UM(Optimization, Syn(AMPL,1)). By the definition of complete meta-model, the Xtext script should define the two types of meta-models. On the one hand, the Xtext script semantically describes the set-oriented meta-model shown in Figure 30. On the other hand, the Xtext script syntactically specifies the set-oriented meta-model in a way that the user model is written in the syntax of AMPL user model in Figure 33.

In Figure 41, ‘OptModel’ is the top element. It includes ‘Declaration’, which is the super class of all other components (See line (1)); we can put any component under the top element. ‘CompoundSet’ is the most important element. As mentioned, it is formed by a combination of other base sets (see line 4). ComponentSet refers to the base

sets through ‘referenceSets+=[Set]’. Unlike ‘Declaration’ in line (1), Set is surrounded by []. This means that the base sets are referred through a cross-reference, which does not embed but does point to the base sets. In other words, [Set] is a list of pointers to base sets. These pointers are surrounded by ‘{}’ and separated by ‘,’.

```
OptModel:
       "Set Supply :=" supplys+=Supply* ";\n" ----------------------(1)
       "Set Demand :=" demands+=Demand* ";\n" ----------------------(2)
       "Set Link :=" links+=Link* ";\n"
       "param s :=" ss+=s* ";\n"
       "param d :=" ds+=d* ";\n"
       "param c :=" cs+=c* ";\n";
Supply:   ----------------------------------------------------(3)
       name=ID;
Demand:   ----------------------------------------------------(4)
       name=ID;
Link hidden(LinkIdRule): -------------------------------------(5)
       "{" supply=[Supply] "," demand=[Demand] "}";
s:
       indexedBy=[Supply] value=STRING;
d:
       indexedBy=[Demand] value=STRING;
c:
       indexedBy=[Link|"ByHiddenRule"] value=STRING;
LinkIdRule:   ------------------------------------------------(6)
       "<supply> <demand>";
```

Figure 42. Xtext Script for the Complete Meta-model of AMPL Instance

Figure 42 shows the Xtext script that defines the complete meta-model of AMPL instance data, i.e., CM(IM(Optimization, Syn(AMPL,0)). Unlike the previous Xtext script for AMPL user model, we put fixed string patterns (e.g., ‘Set Supply :=’) in the relation with the top component, ‘OptModel’, (line (1) and (2)), not within the definition of the components themselves (line (3) and (4)). Therefore, these string patterns appear once over multiple components embedded in the assignments of the top components, e.g., ‘supplys+=Supply*’. This complies with the grammar of AMPL instance data where more than one components of a type share the fixed string patterns. In Figure 36, supply nodes (S1, S2, S3) share ‘Set Supply :=’ as the collective fixed string patterns. The above Xtext script accommodates this structure well.

We extend Xtext script in the following practical sense. Existing Xtext script has no way to manipulate the identifier of a rule; i.e., ‘name’ is always used as an identifier.

However, in many cases, we need customized identifiers. In the AMPL instance data, the identifier of a compound set is the combined string of identifiers of the base sets with space separation; e.g., the identifier of Link (line (5)) is a combination of the identifiers of supply set and that of the demand set. In order to support this, we introduce a rule-based identifier which allows us to customize the identifier using the attributes of the rule. Line (6) shows the identifier rule for Link rule defined in line (5).

Model Transformation for Generating the Complete Meta-model of AMPL Instance

In Chapter 3, we suggest generating the complete meta-model of AMPL instance data, CM(IM(Optimization, Syn(AMPL,0)), through a special model transformation, instead of manually constructing it. As shown in Figure 42, the Xtext script contains information that is specific to the transportation problem (e.g., Link, Supply, Demand, etc). The fact that the problem specific information should be changed when switching to other optimization problems makes our generation approach attractive.

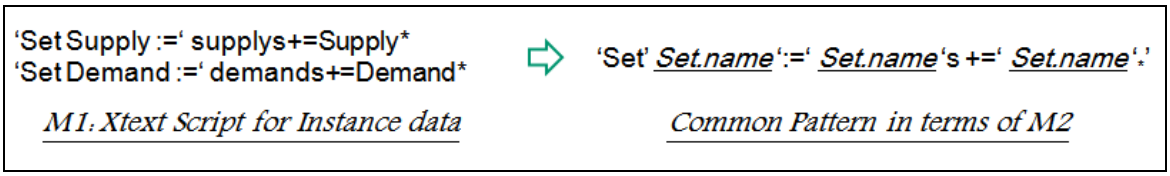

Figure 43. Syntactic Common Pattern of Xtext Scripts for Single Sets

This generation is possible because AMPL language grammar for instance data (in M0) has the same syntactic pattern as shown in Figure 43. The left side shows two Xtext scripts that specify the Supply set and Demand set, both of which are single sets. Although they have different concrete syntaxes, they have the common syntactic pattern shown in the right side; the fixed terms are exactly the same; and even the different terms can be considered as the name of the single sets (i.e., Set.name). We can write the common pattern from the right side of Figure 43 in terms of the meta-model of optimization model (in M2).

This common syntactic pattern plays an essential role in developing the special model transformation that generates the Xtext script. The model transformation is developed in a way that captures the common pattern in mapping rules; e.g., the common pattern in Figure 43 is used as the mapping rule for a single set. Figure 44 shows the details of the mapping rule in an ATL script.

```
rule Set2Set
{
from
  ss : OPT!SingleSet
to
  ts : XTEXT!ParserRule       ------------------------------------(1)
 (
   name <- ss.name,  ------------------------------------------(1-1)
   type <- ty,        ------------------------------------------(1-2)
   alternatives <-thisModule.createAssingment       -------------(1-3)
                    ('name','=',thisModule.getRuleCall
              (thisModule.getRuleByNameFromTerminal('ID')))
  ),
  ec : ECORE!EClass           ------------------------------------(3)
  (
    name <- ss.name
   ),
  ty : XTEXT!TypeRef
  (
       classifier <- ec,
       metamodel <- thisModule.topMeta
   )
   do
   {
     thisModule.topMeta.ePackage.eClassifiers <- OrderedSet{ec};
     thisModule.addTopRelationForSet(ts.name,ts); -----------------(4)
    }
}

rule addTopRelationForSet(setName : String, setRule : XTEXT!AbstractRule)
{
do    ------------------------------------------------------------(5)
 {
 thisModule.getTopRule().alternatives.elements
  <- OrderedSet{thisModule.createKeyword('Set ' + setName +' :='),
   thisModule.createAssingmentWithCard(setName.toLower()+'s','+=',
   thisModule.getRuleCall(setRule), '*'),
   thisModule.createKeyword(';\n')};

 }
}
```

Figure 44. ATL Mapping Rule of Syntactical Model Transformation for Single Set

Technically, the special model transformation is defined between the meta-model of DSM (domain semantic model) and the complete meta-model of EBNF language, i.e.,

the set-oriented meta-model and CMM(Xtext) in our example. For the implementation, we define the set-oriented meta-model, shown in Figure 30, using Ecore. In contrast, we do not need to develop CMM(Xtext) because it has been provided by Xtext developers as an Xtext script. Interestingly, the complete meta-model of the Xtext language is recursively defined in an Xtext script. This is similar to the self description property of meta-modeling frameworks; i.e., MOF (or Ecore, respectively) is defined by MOF (or Ecore, respectively).

In order to generate an Xtext script through a model transformation, we need to handle the detailed semantics of the meta-model of Xtext language. Manipulating the semantics is difficult because the semantics include a number of hidden parts that do not explicitly appear in the Xtext script. For example, semantically, a ParserRule contains its grammar expressions within an attribute, called ‘alternatives’; in Figure 44, we put an assignment, which denotes ‘name=ID’ in line (3) or (4) of Figure 42, into the ParserRule though the attribute, ‘alternatives’ in line (1-3). However, the Xtext script does not show existence of the attributes in Figure 42. You can get the detailed semantics of Xtext language from ‘Xtext.Xtext’ in Eclipse plug-in of the Xtext framework. ‘Xtext.Xtext’ is the Xtext script that recursively describes CMM(Xtext).

Figure 44 shows the mapping rule for a single set. It takes a SingleSet object as the input, and generates the grammar rules shown in line (3) and (4) of Figure 42. The output rule is declared as a ParserRule, (See line (1)). Although ParserRule itself does not explicitly appear in the syntax of (3) and (4), it semantically plays a role of container that includes all syntax parts of the rule in its properties.

In line (1-1), the name of ParserRule is set as the name of the input object. This name property appears as the declaration part of the grammar rule in Figure 42, i.e., ‘Supply:’ (or ‘Demand:’ respectively) for (3) (or (4) respectively). Since this is dynamically assigned by the property of the input object, what concretely appears in this portion depends on the input object. Indeed, line (3) and (4) have different declaration

words in our example. This dynamic assignment allows us to reuse this mapping rule for different objects that have a common syntactic pattern. In contrast, (1-3) in Figure 44 creates “name=ID”, which is a fixed part. Although it looks complicated, basically, it creates the fixed string using the semantic components of Xtext language.

Line (4) is the post action, which is executed after the output object is created. This allows us to carry out additional operations on the created output. In the script, the post action links the created grammar rule of single set into the top element, ‘OptModel:’ in Figure 42. The line calls ‘addTopRelationForSet’, which generate the syntax of the relation with ‘OptModel’. We already found the common pattern of the syntax in Figure 43. The function ‘addTopRelationForSet’ creates the concrete syntax based on the common pattern. For the dynamic term - Set.name, the function takes setName as its parameter.

Mathematical Equation

Mathematical equations are how optimization models are conceptualized. Usually, the analyst translates a system of mathematical equations into the language required by the optimization solver being used. We suggest a way to capture mathematical equations using a formal modeling framework. The formal description of mathematical equations is important because the equations are described in different forms across various tools; e.g., AMPL and GAMS have slightly different syntax for mathematical equations. If the equations are described in simply plain text, they can be used only for a particular tool. In contrast, the formally captured equations can be converted into different forms through a model transformation.

This capability of manipulating the equations becomes more important in the case that mathematical equations should be dynamically generated throughout a problem solving process. For instance, in Benders Decomposition [51], the constraints of the master problem are iteratively generated from the solutions of the sub problems.

In order to support formal modeling of mathematical equations, we develop the following complete meta-model for arithmetic expressions. The following figure shows the Xtext rules that define the complete meta-model for AMPL syntax.

```
Expression:  ---------------------------------------(1)
      Addition;
Addition returns Expression: ----------------------(2)
Multiplication (({Plus.left=current} '+' | {Minus.left=current} '-
') right=Multiplication)*;

Multiplication returns Expression: ------------------(3)
PrimaryExpression (({Multi.left=current} '*' | {Div.left=current}
'/') right=PrimaryExpression)*;

PrimaryExpression returns Expression:  --------------(4)
'(' Expression ')' | --------------------------------(4-1)
{NumberLiteral} value=NUMBER |
'sum' {SummationCall} '{' over=IndexofSet '}' '(' exp=Expression
')' |
{IndexedDeclar} decl=[Declaration]'[' index += [IndexofSet] (','
index += [IndexofSet])* ']';
```

Figure 45. Xtext Rules for Arithmetic Expression

First of all, we define four types of primary expressions: parenthesized expression, number, summation clause, and indexed variable (or parameter) (line (4)). Second, multiplication is defined between two primary expressions (line (3)). Finally, expression, in turn, is defined as an addition between multiplications (line (1) and (2)); the expression can describe formulations that combine the primary expressions with additions or multiplications. Interestingly, the first primary expression type (parenthesized expression) puts the expression back to the primary expression again (line (4-1)); the expression can be a primary type expression if it is enclosed in parentheses. This means any expression can be part of another expression. This recursive property allows us to describe any general arithmetic expressions using this meta-model.

```
Constraint:
'Subject to' name=ID '{' forall += IndexofSet (',' forall +=
IndexofSet)*  '}:' lhs = Expression opt=('=' | '=>' | '=<' ) rhs =
Expression ';\n' ;

IndexofSingleSet:
name = ID 'in' indexedSet = [Set];

IndexofMultiSet:
'(' name += ID (',' name += ID)* ')' 'in' indexedSet = [Set];
```

Figure 46. Xtext Rules for Constraints and Index

We introduce two additional concepts to incorporate arithmetic expressions into an optimization model as constraints. The first concept is, of course, a constraint; it consists of the left hand side, the right hand side, and the operation between them. We define a cross-reference ‘forall’ to specify the sets that the constraint is imposed on. The second concept is index letter for sets and variables. In an optimization model, the index letters are frequently used simply to refer to sets. We define two ‘indexofSet’ rules for a single set and a multi set respectively.

We demonstrate how efficiently the complete meta-model of arithmetic expression deals with constraints in the following two steps.

In the first step, we create the constraints of the transportation problem in Figure 33, and add them into the optimization model generated by the model transformation with PIM, i.e., MTM(2). MTM(2) cannot generate the constraints simply because the PIM, the source model, does not explicitly describe the constraints. The constraints are optimization specific components; they should be created in the optimization domain. In our implementation, we use the Java classes generated from the Xtext script that defines the complete meta-model of arithmetic expression in Figure 45 and Figure 46.

```
Constraint c2= outputfactory.createConstraint();
c2.setName("SupplyBalance");
IndexofSingleSet indexOfLink2 =
outputfactory.createIndexofSingleSet();
indexOfLink2.setName("i");
indexOfLink2.setIndexedSet((Set1)getObjByName((OptModel)topCopy,
"Supply"));
```

Figure 47. Creation of Supply Balance Constrain in Java

The above figure shows the Java code that creates supply side balance equations. The Java classes are generated from the semantic definitions of the Xtext script in Figure 46, i.e., the parts not surrounded by quotations; it is not difficult to recognize the Java classes have semantically identical structure with the Xtext script. These Java classes hide all the detailed syntactic specifications from us. We can use the Java classes without any knowledge of the syntactic specifications. This demonstrates how easily we can incorporate the constraints model into general programming languages. This makes it efficient for us to use our optimization model in general programming environment, which is usually demanded in many practical problem solving cases.

In the second step, we generate the syntactical representation of the semantic optimization model (DSM(Opt)) we obtained through the execution of MTM(2) and the additional creation of constraints in the previous step. We mentioned that a formal descript of a mathematical expression can be reused for various tools by converting the formal description into different syntactic representations. In order to demonstrate the reusability of our mathematical constraint model, we create two different syntactic representations of the optimization: AMPL syntax, and LaTex. Although LaTex is not an optimization tool, we select it because it is widely used as a mathematical expression authoring tool.

We develop Xtext scripts for both tools in a way that shares the common semantic model. The following simple example shows how it is done. All we need to do is to differentiate the syntactic parts, while keeping the semantic parts in common. The only

difference in the examples is '\\' in front of 'in'. In this way, we can accommodate all syntactical differences without changing the semantics.

```
For AMPL
IndexofSingleSet:
name = ID 'in' indexedSet = [Set1];

For LaTex
IndexofSingleSet:
name = ID '\\in' indexedSet = [Set1];
```

Figure 48. Comparison of Xtext Rules for AMPL and LaTex

Since the both Xtext scripts have the exactly same semantic model, they have the equivalent set of Java classes; DSM(Opt) can be use for the both tools. The syntactical differences between the tools are reflected in the parsers generated from the syntactic specifications of the Xtext script. These different parsers result in different syntactical representations from the same semantic model, DSM(Opt).

```
For AMPL
Subject to SupplyBalance { i in Supply }: sum { j in Demand } ( x
[ i , j ] ) =< s [ i ] ;

For LaTex
$\ \sum _{ j \in Demand } ( x _{ i , j } ) =< s _{ i } \forall{ i
\in Supply } $ : SupplyBalance \\

Visual Representation in LaTex
```

$$\sum_{j \in Demand}(x_{i,j}) =< s_i \forall i \in Supply : \text{SupplyBalance}$$

Figure 49. Syntactical Representations in AMPL and LaTex

The scripts in Figure 49 show the two different syntactic representations of the supply balance constraint for AMPL and LaTex. In addition, we put the visual representation of the constraint that LaTex complier generates in a PDF. This is a example where a formal mathematical model is used for computation in an optimization solver and documentation in an authoring tool.

5.5 Syntactical model transformation with MS Access™

We implement the syntactical model transformation for MS Access using an XSD-based approach. This approach has two advantages over the EBNF-based general approach: it is much easier to develop the special model transformation for generating the complete meta-model of instance data; more importantly, the generated the complete meta-model can be used as a user model. The second advantage makes it unnecessary for us to separately develop the complete meta-model of the user model.

The first advantage is attributed to the syntax standard of XSD/XML, which we discussed in section 5.3. In the EBNF-based approach, model transformation developers have to be concerned about even very trivial syntax elements, such as parenthesis, separator, etc. Moreover, as mentioned, the EBNF language has a lot of hidden semantic parts. These all make it very difficult to develop a model transformation for the EBNF-language. Figure 44 visually shows the complexity of the development; creating just two lines of syntax specifications for a single set object requires a quite lengthy script with complicated syntax specifications. The length is much longer if we include all sub-functions that the mapping rule uses. The syntax standard of XSD/XML frees developers from describing all these complicated details. This is because once modelers determine XSD constructs to describe the meta-model, the syntactical details can be automatically generated by the syntax standard.

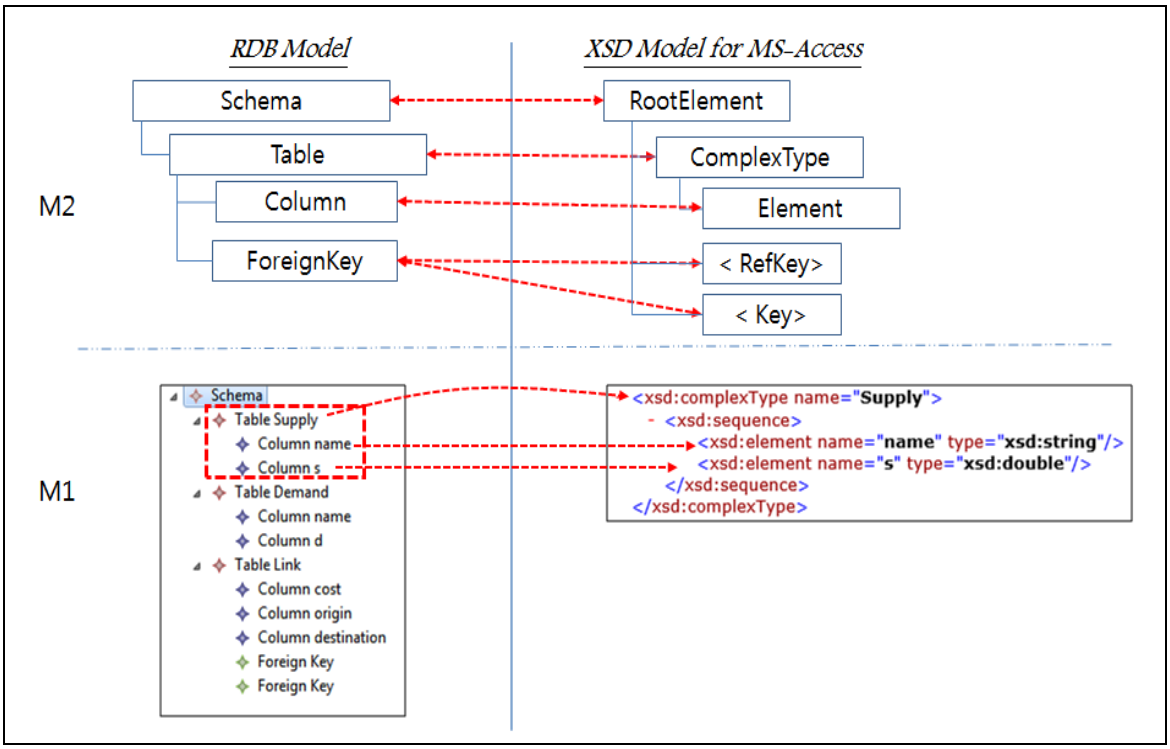

Figure 50. Syntactical Model Transformation to MS Access

This figure shows a model transformation that converts an RDB semantic model to an MS-Access specific syntactic model in XSD format. The mapping rule of the model transformation links semantic components with XSD constructs, e.g., 'Table' (or 'Column' respectively) is mapped to 'ComplexType' (or 'Element' respectively). Although this simple mapping rule is defined without specifying syntax, the syntax standard of XML enables the model transformation to generate a valid XSD representation in M1. This mapping is much simpler than that of general EBNF-language (Figure 44) in terms of complexity of defining syntactic specifications and mapping rules.

The meta-model of the target side is the complete meta-model XSD, i.e., CMM(XSD). Interestingly, CMM(XSD) is defined by another XSD model; it is analogous to Xtext, in which a special Xtext – Xtext.Xtext - recursively defines the Xtext language. Likewise, the XSD language itself is specified by XSD; this is technically possible because XSD is a type of XML documentation. W3C, which is responsible for

developing the XML standard, defined XSD.XSD [52]. We reuse the XSD.XSD in our implementation.

The second advantage of the XSD-based approach is that it reduces the effort to manually develop a complete meta-model of a user model (CMM(UM)), which is the first step in Figure 7. We capitalize on this advantage by reusing the result of step 2 of Figure 7, CMM(IM), as the user model in M1. Since the user model is generated as the byproduct of step 2, we do not need to separately construct CMM(UM) for the user model. However, this approach can be used in limited situations. In order to reuse CMM(IM) as the user model, CMM(IM) and the user model should be equivalent; that is, given a tool T in a tool domain TD, CMM(IM(TD,syn(T,0))) = UM(TD,syn(T,1)).

Practically, this condition means that a tool has to use XSD to specify the schema of XML instance of its user model. This is technically viable because XSD has equivalent expressiveness to class diagrams in UML [53]; XSD can express the same model as much as a class diagram can. Furthermore, XSD includes all semantic information that is necessary to define the user model because it is the complete meta-model of instance data. If a tool uses XSD to define its user model, it can provide a user model definition framework at no cost. Because of this, XSD plays roles of both the meta-model for the user model and a complete meta-model of XML documents in many COTS tools.

MS Access™ is a good example of this situation. It allows us to define a database schema, which is a user model in RBD domain, by importing an XSD file; this XSD file is used to define the XML format in which the database can import and export its records. The following figure depicts the two roles that the XSD plays as the complete meta-model of XML of M0 and the schema model of M1. Thanks to the dual roles of XSD, a model transformation (SM(2)) is able to generate both schema and records in the MS Access compatible syntax.

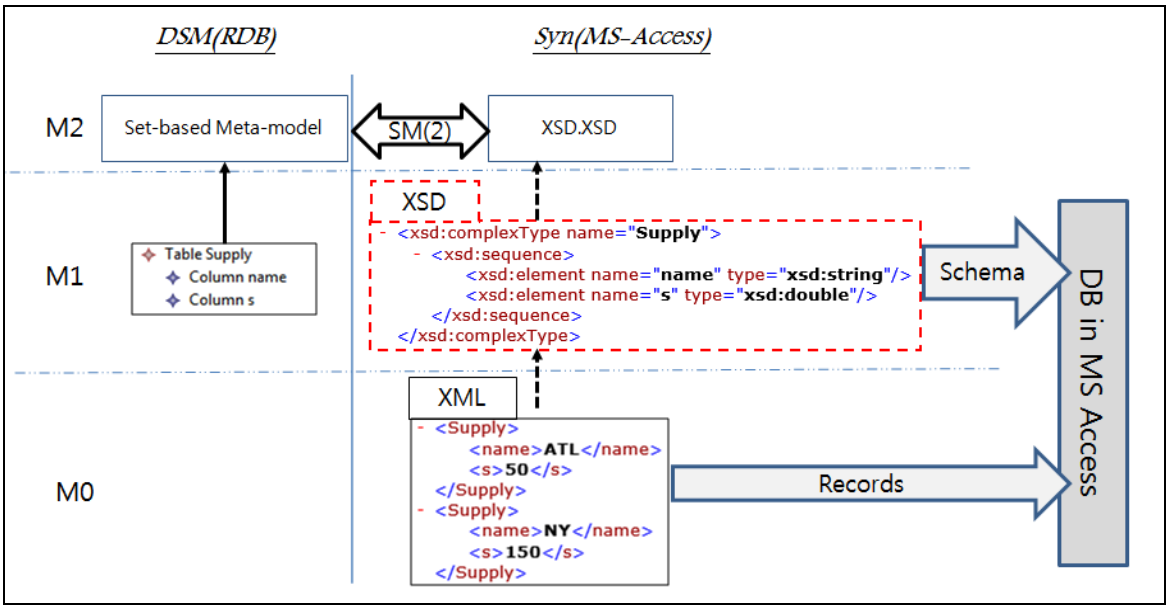

Figure 51. The Dual Roles of XSD in the Syntax of MS Access

However, MS Access has a technical shortcoming. In RDB domain, foreign keys play an important role in establishing reference relations between columns across tables; the reference relations are crucial to describe an object oriented model in a relation database. This means the foreign keys should be taken into account in the model transformation with object oriented modeling domains. XSD is able to accommodate well the foreign keys using a combination of 'Key' and 'RefKey' as shown in Figure 50. However, MS Access does not have capability of handling the foreign key concept in XSD import and export. Although the model transformation (SM2) generates the foreign keys using 'Key' and 'RefKey', MS Access ignores them.

Because of this technical issue, we cannot fully demonstrate our approach when the model has to deal with the foreign key relation. Unfortunately, our transportation example falls in this case; 'Link' refers to 'Supply' and 'Demand' through foreign keys in RDB domain. We can complete the demonstration for components without foreign keys such as 'Supply' and 'Demand'.

5.6 Conclusion

In this chapter, we proposed the demonstration scenario where RDB and optimization modeling domains are used to describe the transportation optimization problem. In the scenario, we demonstrated the syntactical model transformations, which generate the user models (M1) and the instance data (M0) in compatible forms to the target implementation tools: MS Access for RDB domain, and AMPL for optimization domain.

As the first step, we analyzed both modeling domains semantically and syntactically in terms of the layered language formalism. In M2, we developed the set-oriented meta-model (or the table-oriented meta-model respectively) for optimization domain (or RDB domain respectively). We also showed how the transportation problem can be described in M1 and M0; semantically, the meta-models in M2 are used to describe the user models (in M1) of the transportation problem, and then the user model is specified by instance data to describe an instance of the transportation problem in M0; syntactically, we reviewed how those models are described in the syntaxes of the implementation tools.

Understanding of the linguistic characteristics of the modeling domains, we developed the complete meta-models, which are the key part of the syntactical model transformation. Technically, we use two different types of frameworks to define the complete meta-models; Xtext framework is used for the optimization domain, while XSD/XML framework is used for the RDB domain. We view XSD/XML framework as a special case of Xtext framework; we found that XSD/XML framework allows us to very efficiently build up the complete meta-model in a limited condition where a modeling tool use one XSD file as its user model as well as the complete meta-model of XML documents written in M0.

For the optimization domain, we developed the complete meta-model of arithmetic expression in AMPL syntax. This complete meta-model is incorporated into

the set-oriented meta-model through indexing letters. Using the compete meta-model, we generated the mathematical constraints of the transportation problem as optimization domain specific components.

In addition, we demonstrated the reusability of the formally captured arithmetic models. We developed another complete meta-model in LaTex syntax in a way that has the same semantic meta-model but has different syntax specifications. We have shown the mathematical constraints described in the shared semantic meta-model can be converted into the two different syntactical representations through the different complete meta-models. We believe that the reusability can play an important role in data exchange among the various computation tools that collaboratively solve an optimization problem; they share the same problem but capture some part of the problem in different syntaxes.

CHAPTER 6

MULTI LAYER MODEL TRANSFORMATION

We complete the scenario introduced in Chapter 5 by implementing the multi layer model transformation incorporating the syntactical model transformation implemented in Chapter 5.

We start by showing that we can apply the correspondence model-based approach to our scenario. In Chapter 4, we conclude that we can use the approach if a model transformation model in M2, MTM(2), can be described in TGG. We show that our scenario falls into this case by completing step 1 of Figure 29 using TGG; i.e., we use TGG to define two model transformation models with the target domains RDB, and Optimization in M2.

In the rest of this chapter, we implement the correspondence model-based approach in the context of our scenario. The approach converts the correspondence associations that the above model transformations in step 1 generate into executable ATL model transformation scripts using Xtext.

6.1 General Implementation Process of Correspondence Model-based Approach

Although we implement the correspondence model-based approach in our technical environment, we suggest a general implementation process. The process consists of three steps:

i) develop the meta-model of correspondence model

ii) develop the complete meta-model of the model transformation tool that executes the model transformation model in M1 – MTM(1)

iii) build a model transformation model between the two meta-models defined in the previous steps.

We suggest step 1 in order to make the correspondence model independent of the model transformation tool for MTM(1). Since the meta-model of the correspondence model is one of the source meta models of MTM(2) (See Figure 10 in Chapter 3), if the meta-model depends on the model transformation tool, MTM(2) also becomes dependent on the tool.

This dependency is undesirable in that MTM(2) is a semantic transformation between user models in M1. It is intended that the semantic transformation should deal with domain specific knowledge of engineering problems; that is, it should be independent of technical specifications of the tools used in the implementation. Moreover, the independence allows us to reuse the meta-model of the correspondence model; that is, the meta-model we propose can be reused for other circumstances where different target domains and implementation tools are used. This makes the approach more efficient.

All model transformation tool specific aspects are incorporated through steps 2 and 3. In step 2, the complete meta-model of the model transformation tool is intended to define tool specific syntax for the key elements of the tool independent correspondence model developed in step1. Step 3 makes a model transformation model between the key elements and the corresponding syntax, and executes it to generate an executable model transformation script.

6.2 Model Transformation Models in M2: MTM(2)

We present two TGG-based model transformation models in M2 with the two target domains of our scenario - RDB, and Optimization. Practically, these model transformation models are used to implement step 1 of the scenario (See Figure 29 in Chapter 5). Theoretically, these ensure that we can apply the correspondence model-based approach to our scenario because of the conclusion we proved in Chapter 4.

We present the TGG mapping rules using the visual representation of triple production introduced in Figure 17. The visual representation is much easier to

understand than the mathematical definition of triple production is. This helps us easily figure out the mapping rules. Technically, we use ATL scripts to describe the TGG rules. We use some part of the ATL scripts to explain key points.

In the scenario, we use EMF as an object oriented (OO) modeling domain for describing PIM. For more general discussion, we use general terms of general OO modeling instead of EMF specific terms.

Model Transformation Model with RDB Domain

We present mapping rules between the Table-oriented meta-model defined in Section 5.2 and OO modeling domain. This following figure shows the two typical mapping rules for RDB domain: Class to Table, and Attribute to Column. The before condition of TP1-2 ensures that an attributed is converted into a column only when their parents (Class and Table, respectively) already have a mapping relation created by TP1-1. This prevents columns from being added to irrelevant tables.

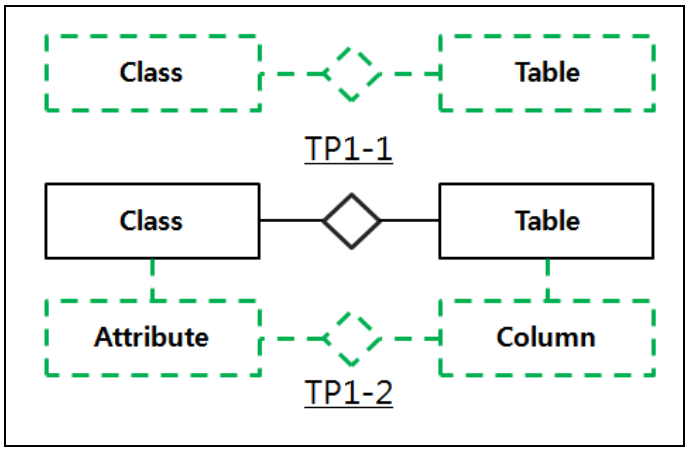

Figure 52. Triple Production for RDB Domain – Part 1

Figure 53. shows the mapping rule that handles a reference relation of a class. In the OO modeling domain, a class refers to other classes through the reference relations, which is mapped to foreign key in RDB domain. In the figure, we put notes on edges in order to make the rule clear. The class on the upper left corner contains the reference

through which it refers to the other class on the lower left. For the reference, the mapping rule creates the foreign key relation from the table on the upper right to the table on the lower right.

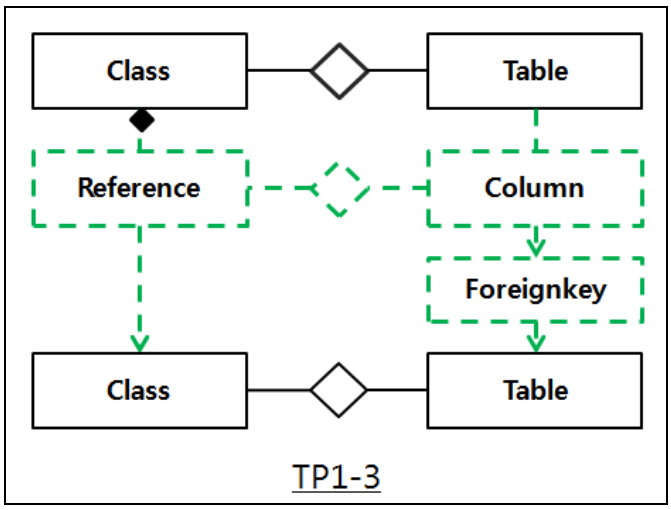

Figure 53. Triple Production for RDB Domain – Part 2

Unfortunately, this foreign key relation is not fully implemented in our demonstration because of the technical shortcoming of MS Access. As mentioned in Chapter 5, although XSD is able to accommodate the foreign key relation, MS Access does not provide any way to take the foreign key relation in XSD format. For this reason, we implement our demonstration only for the following optimization domain. Since we use more general EBNF-based approach in the optimization domain, it is enough to demonstrate our concept.

Model Transformation Model with Optimization Domain

We present mapping rules with the Set-oriented meta-model defined in section 5.2. Unlike the RBD domain, a class is mapped into different components depending on whether it has references or not. TP2-1 has a negative condition; if a class does not have any reference, TP2-1 generates a SingleSet. In contrast, if a class has references, TP2-2 creates a CompoundSet. One example of the second case is ‘Link’ of the transportation problem. In our transportation problem description (Figure 32), ‘Link’ refers to ‘Supply’

(or ‘Demand’ respectively) as its source (or destination respectively). The OO domain expresses these relations as reference relations from Link class to Supply class or Demand class, whereas optimization domain uses a compound set with ‘referenceSet’ relations to describe the relations.

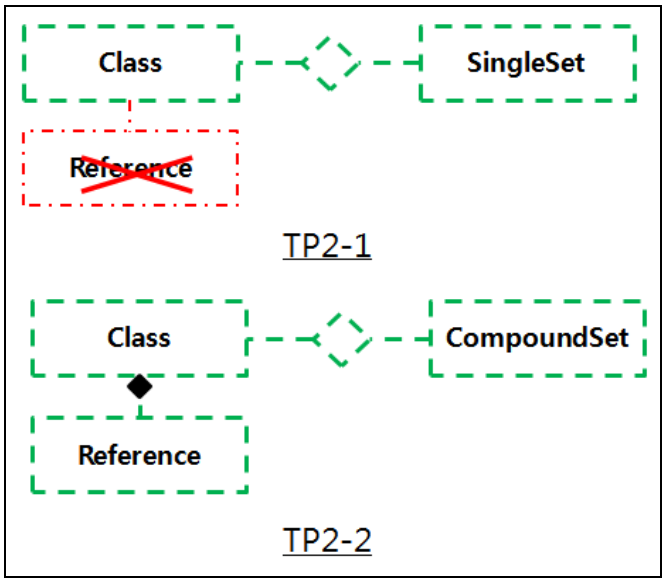

Figure 54. Triple Production for Optimization Domain – Part 1

Two triple productions in Figure 55 show two different transformation rules for ‘Attribute’. If it is marked as derived, it is converted into ‘Variable; otherwise, it is converted into ‘Parameter’. An attribute can be a parameter or a variable depending on the problem we try to solve. If the attribute is given as part of the problem, it should be considered as a parameter, whereas if the attribute is what we determine as the solution of the problem, it is a variable. In order to indicate this, we use ‘derived’ property.

Note that we use ‘Set’ on the right side instead of ‘SingleSet’ or ‘CompoundSet’. Since ‘Set’ is the super class of ‘SingleSet’ and ‘CompoundSet’ (see Figure 30), these rules are applicable to the both classes. Using a supper class allows us to avoid repeatedly developing mapping rules for all the sub classes if they have the same rules.

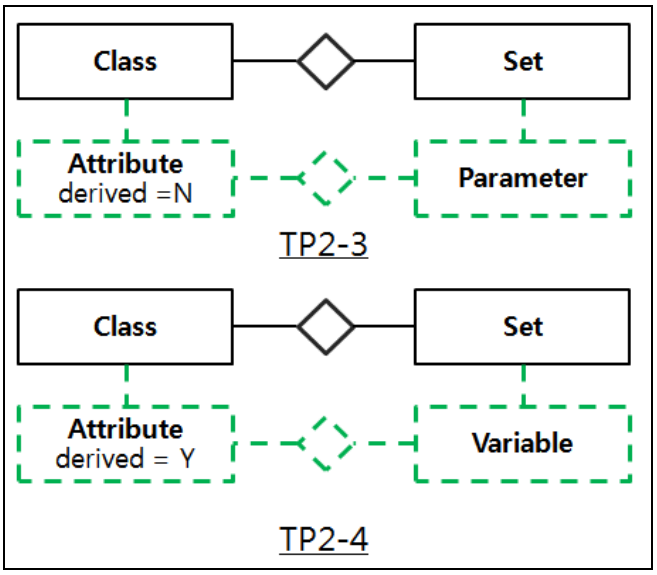

Figure 55. Triple Production for Optimization Domain – Part 2

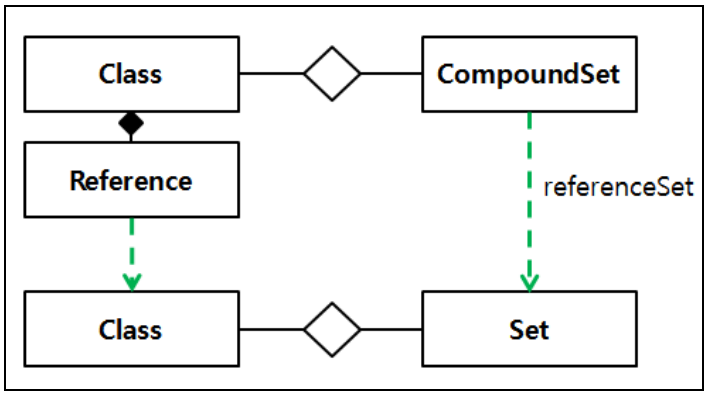

Figure 56. Triple Production for Optimization Domain – Part 3

The above mapping rule is for ‘CompoundSet’. If a class refers to other classes through references, the corresponding CompoumdSet, which has been transformed from the class, is associated with the sets that have been generated by the referred classes.

In this section, we have shown that we can develop MTM(2) for both of the domains in TGG. We can, therefore, apply the correspondence model-based multi layer model transformation to our scenario. We demonstrate this in the rest of this chapter.

6.3 Meta-model of correspondence model: Step 1

In this section, we develop a meta-model of the correspondence model. The meta-model consists of two parts: correspondence associations, and reference relations between the associations.

As discussed in Chapter 3, the first part, the correspondence association, captures the result of execution of a mapping rule as a linkage between the source component and the target component. This is included in the original definition of correspondence model (Definition 8) as the basic element. In this section, we explain how to incorporate the correspondence association into MTM(2) in order to actually capture the result of MTM(2).

The second part, the reference relation between the correspondence associations, is not the part of the definition of the correspondence model - Definition 8. We add it in order to deal with the execution order between the correspondence associations. This order is not obtained directly from the execution of MTM(2). Instead, we construct an efficient execution order using the algorithm we proposed based on a topological order of the dependency graph (see Figure 28). Since the correspondence associations are used as the mapping rules in the multi layer model transformation, the reference relation between them can capture the one way dependency between triple productions (Definition 26) in the dependency graph.

Proposed meta-model of correspondence model

By Definition 8, a correspondence association links the source component that a mapping rule takes as the input; and the target component that the mapping rule generates. Therefore, a correspondence association could be a simple linkage between the sources and the target components. However, we classify the linkage depending on the types of the end components of the associations; e.g., class-class, class-primitiveType, and so on. We suggest this classification because formal modeling framework provides different

ways of handling the class and the primitiveType; a reference relation is used to point at the class, whereas a simple attribute is used to store the value of the primitiveType. The classification contributes to dealing with the difference. The following figure shows the four classifications. Although they do not have additional information, they are transformed in different ways when we convert the correspondence model into an executable model transformation script.

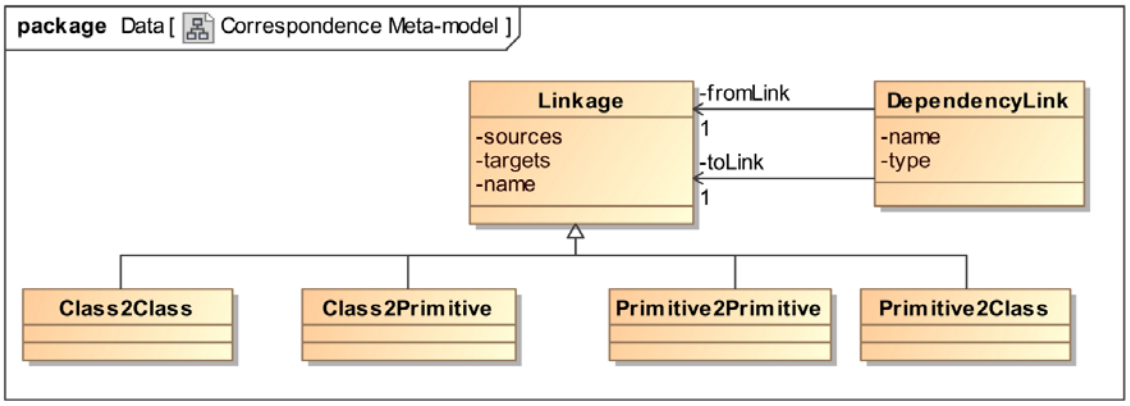

Figure 57. The Meta-model of Correspondence Model

'DependencyLink' handles the dependency relation between correspondence associations. Since the dependency relation is one way (Proposition 7), 'DependencyLink' has one source (fromLink) and one target ('toLink')..

In the introduction of this chapter, we say the meta-model of correspondence model should be independent of model transformation tools; this meta-model is designed not to include information related to any model transformation tool; it uses only general semantic terms that capture the key aspects of the correspondence model.

How to incorporate the correspondence meta-model into MTM(2)

Now we need to incorporate the correspondence meta-model into MTM(2) so that the correspondence model can capture necessary information that is generated as the result of execution of MTM(2). In section 6.2, we develop the mapping rules of MTM(2). The rules deal purely with the semantic model transformation between the modeling

domains. We add parts that handle the correspondence model into the pure mapping rule in ATL script.

```
rule Class2Set{
      from
      c : ECORE!EClass  ------------------------------(1)
      (  c. getReferenceClasses().size() = 0 and
      c.getIdAttributes().size() > 0
      )
      to
      s: OPT!SingleSet(  ------------------------------(2)
            name <- c.name),

      cor: LinkModel!Class2Class ----------------------(3)
      (
            sources <- OrderedSet{c},
            targets <- OrderedSet{s}
      )
      do{
            thisModule.topCorModel.links <- OrderedSet{cor};
      }
}
```

Figure 58. ATL Script for Class to SingleSet

Figure 58 describes the mapping rule that transformations a class without reference into a single set (TP2-1 in Figure 54). Line (1) and (2) accounts for this pure transformation part. Line (3) is added to create the correspondence association between the class and the single set that is generated from the class. Since both sides can exist independently, we use a Class2Class association.

In addition, we need to figure out dependency relations between the correspondence associations so as to determine the execution order. We suggested a formal definition of the dependency in Figure 24. In our implementation, we do not come up with an algorithm that determines dependency using the condition introduced in Definition 26. Instead we obtain the dependency relations from the underlying structure of the meta-models of OO domain. More specifically, we use ‘Reference’ and ‘Attribute’ as indicators of dependency.

```
rule ReferenceLink{
      from
      ref : ECORE!EReference
      to
      dLink : LinkModel!DependencyLink  ----------(1)
      (
            name <- ref.name,
            type <- 'C2C-C2C',
            fromLink <-
            thisModule.resolveTemp(ref.eContainingClass, 'cor'),
            toLink <- thisModule.resolveTemp(ref.eReferenceType,
            'cor')
      )
      do{  ---------------------------------------(2)
            thisModule.resolveTemp(ref.eContainingClass,'s')
            .referenceSets<-
            OrderedSet{thisModule.resolveTemp(ref.eReferenceType,'s
            ')
      };
}
}
```

Figure 59. ATL Script for Reference

This figure is the mapping rule for ‘Reference’ in Figure 56. This mapping rule does not create any independent component; it just creates a relation between the two existing sets. For this reason, there is no object creation in the output side. Instead, the connection operation between the set is carried out in the post condition (see Line 2). What this script creates in the output side (Line 1) is a dependency relation between the correspondence associations of the two classes associated through the reference. Since both ends of the reference, classes, can independently exist, the correspondence associations are independent; we can execute them in any order. We determine the dependency in the same direction of the reference (i.e., from the class containing the reference to the referred class) because it is technically much simpler to explore models in that direction; all we need to do to get the referred class (C2) from the containing class (C1) is just to call the following script: C1.C2; however, there is no direct way to go the other direction. Within the output dependency link in Line 1, we put the correspondence association of ‘ref.eContainingClass’ (or ‘ref.eReferenceType’ respectively) to ‘fromLink’ (or ‘toLink’ respectively).

```
rule Attribute2Parameter{
	from
	a : ECORE!EAttribute  --------------------(1)
	(
		not a.iD and
		not a.derived
	)

	to
	p : OPT!Parameter     --------------------(2)
	(
		name <- a.name,
		indexedBy <- a.eContainingClass
	),
	cor: LinkModel!Primitive2Class -----------(3)
	(
		sources <- OrderedSet{a},
		targets <- OrderedSet{p}
	),
	dLink : LinkModel!DependencyLink ---------(4)
	(
		name <- a.name,
		type <- 'C2C-A2C',
		fromLink <- ------------------(4-1)
		thisModule.resolveTemp(a.eContainingClass,'cor'),
		toLink <- cor ----------------(4-2)
	)

	do{
		thisModule.topObj.declarations <- OrderedSet{p};
		thisModule.topCorModel.links <- OrderedSet{cor};
	}
}
```

Figure 60. ATL Script for Attribute

Unlike ‘Reference’, ‘Attribute’ is converted into an independent component in the optimization domain, i.e., ‘Parameter’ or ‘Variable’. In the output side, we create the independent component (line 2) with the correspondence association (line 3), whose type is ‘Primitive2Class’. This script also generates the dependency link. Since an attribute cannot exist alone, mapping rules related to the attribute cannot precede the mapping rule of the class that has the attribute; the dependency link should be established from the containing class to the attribute. Line (4-1) and (4-2) create the link in that direction.

6.4 Syntactic transformation to an ATL script

In this section, we develop a HOT (Higher Order Model Transformation) that generates MTM(1) (see Definition 9 and Figure 11 for more details)..In our implementation, the HOT generates MTM(1) as an ATL script; the HOT is defined between the correspondence meta-model we defined in Section 6.2 and the complete meta-model of the ATL language we develop as step 2 in this section.

Complete meta-model of ATL language: Step 2

We define our own complete meta-model of the ATL language using Xtext rather than the meta-model defined by ATL developers. The Xtext script defines how the key components of the correspondence model (i.e., the four types of correspondence associations and the dependency link in Figure 57) are individually and collectively described in ATL syntax. We establish the syntactic specifications in the following steps. For clarity, we use 'Class2Class' correspondence association (denoted in C2C) as our example.

i) We manually develop the mapping rules of MTM(1) in ATL syntax for the correspondence associations in C2C; i.e., 'Link', 'Supply', and 'Demand' are the C2C type associations in our transportation example.

ii) We compare the mapping rules in order to understand their varying parts.

iii) We replace the varying parts by assignments of Xtext so that these parts can be dynamically changed by the information of the correspondence model.

iv) We put all other parts that are common in the mapping rules as static string patterns.

Figure 61 is a sample mapping rule for 'Demand' in the syntax of ATL. The parts that are bold and underlined should be changed in different contexts; they all depend on what component the C2C association represents in which modeling domain. All other parts are common for all C2C associations. Figure 62 shows the Xtext rule derived from

the manually constructed script. The varying parts are replaced by assignments with semantics meaning (see all underlined parts). They appear as the variables of the C2C association in our ATL complete meta-model, and are involved in the HOT. In contrast, the rest of the script is quoted as a static string. We do not need to care about the static parts in the HOT.

This approach has two advantages over using the original ATL complete meta-model. First, it is easy to develop. Using the original ATL meta-model requires us to understand all detailed semantics including even hidden components. In our approach, we can just write a ATL script in the usual way we use ATL, and then turn the script into the Xtext script in the above simple rules. This allows us to generate valid ATL scripts by HOT without understanding all detailed semantics of ATL. Second, our approach significantly simplifies the mapping rules of the HOT because our complete meta-model is developed in a way that has similar structure to the correspondence meta-model. As you will see in the next sub section, the HOT has very simple one-to-one mapping rule.

```
rule Demand {
from
so : OBJDOMAIN!Demand
to
ta : AMPL!Demand (
name<-so.name
)
```

Figure 61. ATL Mapping Rule between Demand Class and Set

```
Class2Class:
'rule' name =ID '{\n'
'from\n'
'so :' inputMetaModel = [Metamodel] '!' sourceComp = STRING '\n'
'to\n'
'ta :' outputMetaModel = [Metamodel] '!' targetComp = STRING '(\n'
(isNameID ?= 'name<-so.name\n')?
')'
'}';
```

Figure 62. Xtext Rule for Class2Class

We can easily develop Xtext rules for the other three types of correspondence associations in the same way. However, we need a different way to develop the ATL

syntactic representation of dependency link, another key component of the correspondence model. As discussed, we create the dependency link to specify execution order of mapping rules. Hence we need to figure out how we can control the execution order in ATL.

In ATL, there is no independent link component that specifies the execution order. Instead, ATL allows us to indirectly determine the execution order through an assignment that calls other rules. The following figure is the ATL mapping rule of Link, which has the calling assignments. Line (1) (or (2), respectively) assigns 'Origin' class (or 'Destination' class, respectively) of OO domain (the source domain) into 'Supply' set (or 'Demand' set). In order to accomplish the assignments, the classes should be transformed to the corresponding sets; this triggers the execution of mapping rules of 'Origin' and 'Destination'. In this way, the assignments affect the execution order.

```
rule Link {
from
so : OBJDOMAIN!Link
to
ta : AMPL!Link (
 supply <- so.origin ------------------------------(1)
,demand <- so.destination ------------------------(2)
)
do{
 thisModule.cost(so.cost).indexedBy <- ta ; ------(3)
}}
```

Figure 63. Mapping Rule of Link in ATL Syntax

In order to specify execution order, we associate a dependency link with a calling assignment via Xtext cross-reference in the following way. We describe the dependency link using a cross-reference; the correspondence association, from which the dependency link goes, refers to the correspondence association, to which the dependency link comes, through a cross-reference. Syntactically, Xtext uses the identifier of the referred associations as the pointer of the cross-reference. As discussed in Chapter 5, we extend

Xtext so that we can customize the identifier. We combine the customized identifier with string patterns in a way that generates the calling assignment shown in Figure 63.

```
Class2Class:
'rule' name =ID '{\n'
'from\n'
'so :' inputMetaModel = [Metamodel] '!' sourceComp = STRING '\n'
'to\n'
'ta :' outputMetaModel = [Metamodel] '!' targetComp = STRING '(\n'
(isNameID ?= 'name<-so.name\n')?
(',' tarRefname+=STRING '<-' 'so.' souRefName += STRING '--'
linkedR2R += [Class2Class] '\n')*          -------------(a)
')'
do{
'('\n thisModule.' derivedA2R += [Primitive2Class|"ByHiddenRule"]
'.indexedBy <- ta' ';' )*\n'                -------------(b)
'}'
'}';
```

Figure 64. Extended Xtext Rule for Class2Class

In Figure 64, we extend the Xtext Rule depicted in Figure 62 so as to deal with the calling assignments (i.e., line (1) through (3) in Figure 63). Line (1) and (2) are assignments that call mapping rules from class to table. Since this type of mapping is dealt with by Class2Class correspondence association, we represent the assignments by a cross-reference list for Class2Class associations (see (a) in Figure 64). The assignments do not have information from the referred correspondence associations. Therefore, we can generate the assignment script without customization of the identifier. Furthermore, since the identifier is unnecessary, we put the cross-reference list behind '- -', which is the comment indicator of ATL, so that ATL ignore it.

In contrast, line (3) connects 'Link' set, which is created from the mapping rule in Figure 63, with 'Cost' parameter through the 'indexedBy' relation. 'thisModule.cost(so.cost)' is the part that obtains the 'Cost' parameter by triggering the mapping rule from 'Cost' attribute in OO domain to 'Cost' parameter. As mentioned, this is a Primtive2Class association. A dependency link with the Primtive2Class association is used to generate the calling part; we put a cross-reference list with string patterns, and

customize the identifier of the Primtive2Class association as shown in the following figure.

```
Primitive2Class hidden (P2CRefRule, WS):   -----------------(1)
      'rule' name =ID '( val :String)\n{\n'
      'to ta :' outputMetaModel = [Metamodel] '!' targetComp =
      STRING  '(\n'
      'value <- val'
      '\n)'
      'do{'
      '\n thisModule.' getTopMethod = [TopModelHelper] '().'
      topPropName = ID '<- ta;\n'
            '\n ta; \n'
      '}
      }';
P2CRefRule:  ------------------------------------------------(2)
      "<name>( so.<name> )";
```

Figure 65. Xtext Rule for Primitive2Class and its Customized Indentifer

Line (1) defines the Xtext rule for a Primitive2Class association. Interesting, it does not have ‘from’ string pattern, which means that any ATL mapping rule created by this Xtext rule does not have any input component. This is because a primitive type cannot exist without a component containing the primitive type; the primitive type cannot be an input component of a mapping rule. So we define a special type of mapping rule that has only output components, and call the rule from the mapping rule of the component containing the primitive type. In our implementation, line (3) in Figure 63 is the calling point.

Line (2) defines the rule for the customized identifier of the Primitive2Class association. The strings surrounded by ‘< >’ are dynamically replaced by the value of the properties whose name is equal to the strings; that is, <name> is substituted for by the value of name property. The identifier rule in line (2) creates the calling statement (i.e., cost(so.cost)) using the name of the association.

Mapping rules of syntactical transformation: Step 3

We define the syntactical transformation model (HOT) between the meta-model of correspondence model and our complete meta-model of the ATL language. The HOT

generates model transformation models in M1 for instance data integration; they are denoted as MTM(1) in the notation of the multi layer model transformation we introduced in Chapter 3.

```
rule Class2Class
{
from
      corRel : COR!Class2Class
to
      mapRule : ATL!Class2Class (
      name <- corRel.sources->first().name,
      isNameID <- true,
      inputMetaModel <- thisModule.inMetaModel,
      outputMetaModel <- thisModule.outMetaModel,
      sourceComp <- corRel.sources->first().name,
      targetComp <- corRel.sources->first().name,
      getTopMethod <- thisModule.topHelper,
      topPropName <- corRel.sources->first().name.toLower() + 's'
            )
}
```

Figure 66. Class2Class Mapping in HOT

As discussed, the mapping rule of the HOT is simple because of the way of developing the complete meta-model of ATL language. Figure 66 shows the mapping for Class2Class association. The mapping rule just creates the corresponding Class2Class representation in ATL side and fills the necessary variables with the information from the correspondence model side. All other mapping rule can be developed in the same way.

6.5 Conclusion

In this chapter, we have shown that our example met the application condition of correspondence model based multi layer model transformation we have proven in chapter 4. The condition requires that the model transformation rules in M2 can be described in triple productions. In order to demonstrate it, we developed the two MTM(2) of our scenario using triple productions: MTM(2) between PIM and RDB, and MTM(2) between PIM and optimization. We presented the visual representations of the triple productions; further, we captured the triple productions in ATL script and used them to implement our scenario.

In order to implement the correspondence model-based approach, we developed a meta-model of the correspondence model. The meta-model consists of two key parts: correspondence association and dependency link. The first part deals with linkages between the source components and the target components, which are used as mapping rules for instance data. The second part specifies the dependency relations between the correspondence associations; they are used to determine an execution order of the associations when converting the correspondence model into an executable model transformation model. The meta-model of correspondence model has been incorporated into MTM(2) so that the result of MTM(2) can be captured in a way that can generate an executable model transformation.

As the last step of the scenario, we generated an executable ATL script that handles instance data integration from the correspondence model through a HOT (Higher Order Model Transformation). As the output meta-model, we developed our version complete meta-model of ATL language, which gives us efficiency in developing the HOT.

The HOT uses the two key parts of correspondence model in the following ways: i) It converts the correspondence associations into ATL mapping rules; it generates different styles of mapping rules depending on the types of associations we classified by the types of their end components. ii) It uses the dependency link to control the execution flow of the ALT script. Since there is no what explicitly specifies the order, we use an indirect way, i.e., a calling assignment through which a mapping rule trigger an execution of another mapping rule in ATL.

Interestingly, we can view the HOT as a good example of syntactical model transformation we demonstrated in Chapter 5. The correspondence model can be thought of as a domain semantic model (DSM(MT)) for model transformation model, and the complete meta-model of ATL specifies the syntactic representations of DSM(MT) in ATL syntax; using mapping rules between the meta-models, the HOT generates a

executable ATL script. The HOT can generate a valid model transformation model for another model transformation tool simply by replacing the complete meta-model of ATL by the new tool. This is exactly what the syntactical model transformation is intended to and supposed to do.

CHAPTER 7

CONCLUSION

7.1 Summary

Modeling is a common endeavor across all engineering domains. Engineers construct models to explain their engineering decision problems to other people and to solve the problems. However, every engineering domain uses and defines models in its own way. They have different modeling concepts and use them in different ways. These differences have been hindering communications among engineers in different disciplines.

In this dissertation, we proposed a model integration framework where engineers can work together and communicate with one another through model transformation. Since model transformation has been used in software engineering, its current state of the art has limitations as an integration tool for general engineering domains. We raised two key practical issues – instance data integration, and syntactic inconsistency- that are crucial to the successful adaptation of model transformation to the engineering domain.

In order to address the issues, we proposed multi layer syntactical model transformation. It consists of both a multi layer model transformation, and a syntactical model transformation. The multi layer model transformation addresses instance data integration by converting the result of a user model transformation into an executable model transformation model for instance data through a special type of model transformation, HOT. The syntactical model transformation deals with syntactical diversity of various engineering tools using model transformations based on complete meta-models, which allow us to define our models syntactically as well as semantically.

We have proven our approach theoretically and demonstrated it for a simple scenario.

In the theoretic part (Chapter 4), we found one necessary condition under which we can apply the correspondence model-based approach; i.e., MTM(2) should be described in a set of triple productions. In order to show this, we have proven the two properties of a set of triple productions under the disconnection assumption.

The first property, the recursive property, ensures that the results of triple productions are mathematically triple productions again; it says correspondence associations can be individually valid mapping rules.

The second property, the determinism property, says that a series of applications of triple productions results in an equivalent graph at termination regardless of the execution order. This determinism property allows us to construct the execution order, which cannot be obtained from the result of MTM(2). We can make a complete model transformation model by putting together the individual correspondence association in that order. In order to prove the property, we use a dependency graph, which is an acyclic graph that describes dependency relations among the triple productions. A topological order of the dependency graph plays an essential role of the order construction.

In the demonstrations (Chapter 5 and 6), we successfully demonstrated our proposed approach in the scenario where RDB and optimization are used as data storage domain and problem solving domain, respectively.

In Chapter 5, we semantically and syntactically analyzed the target modeling domains in terms of the layered language formalism. Based on the analysis, we developed the complete meta-models for the target tools (MS Access, and AMPL) using Xtext framework. For M1 (the user model), we manually constructed the complete meta-model of the user models in M2, while for M0 (instance data), we developed a special model transformation in M2 that generates the complete meta-model of the instance data in M1.

In practice, one important contribution is the extension of the Xtext framework in a way that allows us to customize the pointer of a cross-reference by setting up a rule.

This extension is crucial because engineering tools have a wide variety of ways of referring to other components; indeed, the rule based cross-reference pointer resolves not only the cross-referencing issue of multi sets in the AMPL syntax, which we originally intended it to deal with, but also the complicated cross-referencing between the mapping rules in ATL syntax, which is the key part of Chapter 6.

Finally, Chapter 6 completed the demonstration by generating an executable ATL script that integrates instance data from the correspondence model, the result of MTM(2); the generation has been done by a HOT. For the input of the HOT, we developed the meta-model of correspondence model independently of model transformation tools; this independence enables us to reuse the meta-model no matter what model transformation tool we work with. As the output meta-model of the HOT, we created a version of a complete meta-model of the ATL language. Furthermore, we suggested a general way to efficiently establish the complete meta-model of other model transformation tools, so that the entire process of our demonstration can be more easily applied to other implementation contexts.

One important advantage of our approach demonstrated in chapters 5 and 6 is that the model transformation models and the meta-models that we need to manually construct all exist in M2. For the syntactical model transformation, we constructed one complete meta-model in M2, and one model transformation in M2 that automatically generates another complete meta-model in M1. For the multi layer model transformation, we constructed MTM(2) and the complete meta-model of correspondence model and ATL syntax, which are all in M2. Our approach, therefore, is independent of the user model in M1; we can reuse all we constructed for other user models as long as M2 does not change, i.e., modeling domains remain same.

7.2 Contributions

Our research aims at an engineering modeling framework that supports effective decision making throughout contemporary interdisciplinary system design process. We focus on model transformation as an essential integration tool among engineering models in the framework. In this sense, this dissertation makes three significant contributions.

First, we extended model transformation methodology to deal with model integration in general engineering modeling domains. Specifically, we identified two key issues: instance data integration, and syntactical inconsistency. We successfully resolved the issues through the multi-layer syntactical model transformation where we use existing meta-modeling and model transformation frameworks in different ways with a simple but powerful change in perspective on model transformation (i.e., a model transformation model is one type of model). This perspective allows us to use the existing frameworks to address the issues of model transformation by manipulating model transformation models using other model transformations.

Second, we extended the theoretically foundation of model transformation. TGG succeeded in formally representing model transformation. However, there has been very little effort to explore theoretically the properties of model transformation using the mathematically formalism (TGG). In our dissertation, we have proven the two interesting properties of TGG: recursive property, and determinism property. The proofs of these properties not only show that the multi-layer model transformation is viable, but also provides a number of new concepts that can be generally used for further theoretical discussion on model transformation. The proof of the determinism property, especially, allows us to gain insight into the unique collective behaviors of a set of triple productions; we proposed a number of new concepts and theorems related to dependency among triple productions: the mathematical definition of the dependency, the effects of the dependency on collective behaviors of the triple productions, dependency graph, etc. These allow us to analyze how triple productions affect one another; it can be an

important theoretical foundation on which further theoretical discussion on model transformation can be made using TGG.

Finally, we reviewed a number of modeling and model transformation tools in practice. Furthermore, we suggested how to improve the tools in order to not only implement our demonstration, but also achieve the ultimate goal of our research. There are two sources of improvements. First, we technically added new functions to the tools. For example, we implemented the customized identifier in Xtext framework so as to manipulate the syntactical representation of cross-reference though which one object refers to another one. Second, we came up with new ways to use existing tools based on the perspective that a model transformation model is itself a type of model. Thus, we should be able to use existing tools to transform model transformation models without any technical extension. Indeed, in order to create an ATL script, we used a special ATL script that treats the ATL script as the output model; and we suggested a new way of using Xtext framework to define the complete meta-model of ATL language.

7.3 Future Research

In the future, we will apply our multi-layer syntactical model transformation to more realistic cases. In order to show that the new model transformation is practically viable, we need to see if our approach can handle large scale problems within reasonable computing time. In addition, we also have to show that our approach can support contemporary engineering decision making environment where highly heterogeneous models should be involved; we should demonstrate our approach with other modeling domains such as simulation.

The long term goal of our research is to establish an engineering design framework that effectively supports engineering decision making for developments of complex systems. We believe that effective modeling and model transformation

methodology plays essential roles in achieving our goal. We hope our research make significant contributions to the goal.

APPENDIX A

DERIVED CANCELLATION RULE FOR CROSS-OVER CASE

A given triple production p = ((SL,SR) ← sr – (CL,CR) – tr → (TL,TR)) and p_1 = ((SL_1,SR_1) ← sr_1 – (CL_1,CR_1) – tr_1 → (TL_1,TR_1)) with cross-over relation with p, the two intended cancellation rules are modified as follows:

((Ø,CR\CL,TR\TL), (Ø, Ø, Ø), NAC((SR\SL,CR\CL,TR\TL), (Ø, Ø, TR_1\$TL_1$))) and ((SR\SL,CR\CL,Ø), (Ø,Ø,Ø), NAC((SR\SL,CR\CL,TR\TL)), (SR_1\$SL_1$, Ø, Ø))).

We add an additional negative condition (Ø, Ø, TR_1\$TL_1$), or ($SR_1$\$SL_1$, Ø, Ø) in order to prevent the derived cancellation rule from completely rolling back p if the result of p1 exists.

REFERENCES

[1] G. Booch, *et al.*, *The Unified Modeling Language user guide*: Addison Wesley Longman Publishing Co., Inc., 1999.

[2] T. R. Gruber, "Toward Principles for the Design of Ontologies Used for Knowledge Sharing," *International Journal Human-Computer Studies*, vol. 43, pp. 907-928, 1993.

[3] P. Borst, *et al.*, "Engineering Ontologies," *International Journal of Human-Computer Studies*, vol. 46, pp. 365-406, 1997.

[4] T. Gruber, "Title," unpublished|.

[5] D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, *special issue on model-driven software development*, vol. 39, pp. 25-31, 2006.

[6] P. Integration. (2011, April 10th). *ModelCenter® 9.0*. Available: http://www.phoenix-int.com/software/phx_modelcenter.php

[7] H.-K. Lin, *et al.*, "Manufacturing system engineering ontology for semantic interoperability across extended project teams," *International Journal of Production Research*, vol. 42, 2004.

[8] J. A. Estefan, "Survey of Model-Based Systems Engineering (MBSE) Methodologies ".

[9] R. Cloutier, "Model Driven Architecture for Systems Engineering," in *Conference on Systems Engineering Research (CSER)*, University of Southern California, 2008.

[10] OMG, "Systems Modeling Language (OMG SysML™)," ed, 2008.

[11] A. A. Shah, *et al.*, "Enabling Multi-View Modeling With SysML Profiles and Model Transformations," presented at the International Conference on Product Lifecycle Management, University of Bath, UK, 2009.

[12] OMG, "XML Metadata Interchange (XMI®)," ed, 2011.

[13] R. Fourer, *et al.*, "A Modeling Language for Mathematical Programming," *MANAGEMENT SCIENCE*, vol. 36, pp. 519-554, 1990.

[14] J. Bisschop and A. Meeraus, "On the development of a general algebraic modeling system in a strategic planning environment," in *Applications*. vol. 20, J.-L. Goffin and J.-M. Rousseau, Eds., ed: Springer Berlin Heidelberg, 1982, pp. 1-29.

[15] L. Schrage, *Optimization Modeling With LINDO*: Duxbury Press, 1997.

[16] M. Tisi, *et al.*, "On the Use of Higher-Order Model Transformations," in *Model Driven Architecture - Foundations and Applications*. vol. 5562, R. Paige, *et al.*, Eds., ed: Springer Berlin / Heidelberg, 2009, pp. 18-33.

[17] A. Schürr, "Specification of graph translators with triple graph grammars," in *WG'94 Workshop on Graph-Theoretic Concepts in Computer Science*, 1994, pp. 151-163.

[18] (Jan). *Xtext Project Site*. Available: http://www.eclipse.org/Xtext/

[19] Eclipse. *ATL Project*. Available: http://www.eclipse.org/m2m/atl/

[20] OMG. (2003). *MDA Guide Version 1.0*. Available: http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf

[21] OMG, "Meta Object Facility (MOF) Core Specification," ed, 2006.

[22] OMG, "Common Warehouse Metamodel Specification," vol. 1,1, ed, 2003.

[23] C. Atkinson and T. Kühne. (2003) Model-Driven Development: A Metamodeling Foundation. *IEEE Software*. 36-41.

[24] OMG, "Query/View/Transformation Specification," ed, 2008.

[25] F. Jouault and I. Kurtev, "Transforming models with ATL," presented at the Satellite Events at the MoDELS 2005 Conference, Montego Bay, Jamaica, 2006.

[26] C. Amelunxen, *et al.*, "MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations," ed, 2006, pp. 361-375.

[27] *EMF Project*. Available: http://www.eclipse.org/modeling/emf/

[28] Y. Sun, *et al.*, "A Model Engineering Approach to Tool Interoperability," in *Software Language Engineering*. vol. 5452, D. Gaševic, *et al.*, Eds., ed: Springer Berlin / Heidelberg, 2009, pp. 178-187.

[29] I. Kurtev, *et al.*, "Model-based DSL frameworks," presented at the Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, Portland, Oregon, USA, 2006.

[30] J. Bézivin, *et al.*, "Model Transformations? Transformation Models!," in *Model Driven Engineering Languages and Systems*. vol. 4199, O. Nierstrasz, *et al.*, Eds., ed: Springer Berlin / Heidelberg, 2006, pp. 440-453.

[31] E. Jackson and J. Sztipanovits, "Formalizing the structural semantics of domain-specific modeling languages," *Software and Systems Modeling*, vol. 8, pp. 451-478, 2009.

[32] A. Konigs and A. Schurr, *Tool Integration with Triple Graph Grammars - A Survey* vol. 148: ELSEVIER, 2006.

[33] H. Ehrig, "Introduction to the algebraic theory of graph grammars (a survey)," in *Graph-Grammars and Their Application to Computer Science and Biology*. vol. 73, V. Claus, *et al.*, Eds., ed: Springer Berlin / Heidelberg, 1979, pp. 1-69.

[34] H. Ehrig, *et al.*, "Graph-grammars: An algebraic approach," presented at the Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973), 1973.

[35] T. W. Pratt, "Pair grammars, graph languages and string-to-graph translations," *Journal of Computer and System Sciences,* vol. 5, p. 36, 1971.

[36] H. Ehrig, *et al.*, *Fundamentals of Algebraic Graph Transformation*: Springer, 2005.

[37] R. Grzegorz, Ed., *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., 1997, p.^pp. Pages.

[38] H. Herrlich and G. Strecker, *Category Theory*: Allyn and Bacon, 1973.

[39] A. Habel, *et al.*, "Graph grammars with negative application conditions," *Fundamenta Informaticae,* vol. 26, pp. 287-313, 1996.

[40] H. Ehrig, *et al.*, "Termination Criteria for Model Transformation," in *Fundamental Approaches to Software Engineering*. vol. 3442, M. Cerioli, Ed., ed: Springer Berlin / Heidelberg, 2005, pp. 49-63.

[41] R. K. Ahuja, *et al.*, "Network flows: theory, algorithms, and applications," 1993.

[42] OMG, "MOF Model to Text Transformation Language RFP," ed, 2007.

[43] C. Krzysztof and H. Simon, "Classification of Model Transformation Approaches," in *Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[44] M. Alanen and I. Porres, "A Relation Between Context-Free Grammars and Meta Object Facility Metamodels," ed, 2004.

[45] A. Kleppe, "Towards the Generation of a Text-Based IDE from a Language Metamodel," in *Model Driven Architecture- Foundations and Applications*. vol. 4530, D. Akehurst, *et al.*, Eds., ed: Springer Berlin / Heidelberg, 2007, pp. 114-129.

[46] D. Hearnden, *et al.*, "Anti-Yacc: MOF-to-text," in *Enterprise Distributed Object Computing Conference, 2002. EDOC '02. Proceedings. Sixth International*, 2002, pp. 200-211.

[47] L. M. Garshol. (Dec.27). *BNF and EBNF: What are they and how do they work?* Available: http://www.garshol.priv.no/download/text/bnf.html

[48] D. Steinberg, *et al.*, *EMF: Eclipse Modeling Framework*, 2008.

[49] T. J. Parr and R. W. Quong, "ANTLR: A predicated-LL(k) parser generator," *Software: Practice and Experience,* vol. 25, pp. 789-810, 1995.

[50] E. T. Ray, *Learning XML*: O'Reilly & Associates, Inc., 2003.

[51] J. F. Benders, "Partitioning procedures for solving mixed-variables programming problems," *Numerische Mathematik*, vol. 4, pp. 238-252, 1962.

[52] S. S. Gao, *et al.* (2011). *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. Available: http://www.w3.org/TR/xmlschema11-1/

[53] T. Krumbein and T. Kudrass, "Rule-based generation of XML schemas from UML class diagrams," 2003, pp. 213-227.

VITA

Kysang Kwon

Kwon was born in ChungBuk, Korea. He received a B.S. in mechanical engineering from Seoul National University, Seoul, Korea in 2000. Afterward, he worked at Sunyang Tech Co. as a mechanical engineer, and worked at USG PLM Korea as a R&D process innovation consultant.

After over 6 year professional experience, he came back to school to pursue a doctorate, and received a M.S. in industrial engineering from Georgia Institute of Technology, Atlanta, Georgia in 2008.

His research interest includes model-based decision making framework, formal modeling framework for engineering models, and model transformation among engineering models. He is also interested in solving various decision problems annalistically and numerically.