Model Synchronization for Software Evolution

by

Igor Ivković

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Software evolution refers to continuous change that a software system endures from inception to retirement. Each change must be efficiently and tractably propagated across models representing the system at different levels of abstraction. Model synchronization activities needed to support the systematic specification and analysis of evolution activities are still not adequately identified and formally defined.

In our research, we first introduce a formal notation for the representation of domain models and model instances to form the theoretical basis for the proposed model synchronization framework. Besides conforming to a generic MOF metamodel, we consider that each software model also relates to an application domain context (*e.g.*, operating systems, web services). Therefore, we are addressing the problems of model synchronization by focusing on domain-specific contexts.

Secondly, we identify and formally define model dependencies that are needed to trace and propagate changes across system models at different levels of abstraction, such as from design to source code. The approach for extraction of these dependencies is based on Formal Concept Analysis (FCA) algorithms. We further model identified dependencies using Unified Modeling Language (UML) profiles and constraints, and utilize the extracted dependency relations in the context of coarse-grained model synchronization.

Thirdly, we introduce modeling semantics that allow for more complex profile-based dependencies using Triple Graph Grammar (TGG) rules with corresponding Object Constraint Language (OCL) constraints. The TGG semantics provide for fine-grained model synchro-

nization, and enable compliance with the Query/View/Transformation (QVT) standards.

The introduced framework is assessed on a large, industrial case study of the IBM Commerce system. The dependency extraction framework is applied to repositories of business process models and related source code. The extracted dependencies were evaluated by IBM developers, and the corresponding precision and recall values calculated with results that match the scope and goals of the research. The grammar-based model synchronization and dependency modelling using profiles has also been applied to the IBM Commerce system, and evaluated by the developers and architects involved in development of the system. The results of this experiment have been found to be valuable by stakeholders, and a patent codifying the results has been filed by the IBM organization and has been granted. Finally, the results of this experiment have been formalized as TGG rules, and used in the context of fine-grained model synchronization.

Acknowledgements

First and foremost, I would like to thank my PhD supervisor, Professor Kostas Kontogiannis. Without his support and understanding these past eight years, especially in the last five, I would not have made it through. He was relentless in helping me get to the finish line and motivated me to cross it. I am indebted to him for everything that he has done for me, and I hope we continue our research collaboration in the future. Thank you Kostas!

I would like to thank Professor Ladan Tahvildari, who generously agreed to co-supervise my thesis and help me handle the remaining thesis paperwork. Ladan's advice towards the end of the process was invaluable, and it helped me avoid some of the common mistakes that one goes through when scheduling their defence.

I am grateful to all the members of my PhD committee, Professor Paulo Alencar, Professor Krzysztof Czarnecki, Professor Derek Rayside, and Professor Rudolph Seviora, for their valued support in approving my extension forms and encouraging me to finish my dissertation. I am also indebted to Professor Scott Tilley for agreeing to serve as the external examiner.

I would like to thank my colleagues from the University of Waterloo, namely Suzanne Safayeni, Andrew Morton, Shaz Rahaman, Wendy Boles, Murray Zink, and Mirela Banica-Stanei, for their presence and support throughout the years. I would like to acknowledge my colleagues from IBM and SEI, namely Terry Lau, Tack Tong, Ross McKegney, Jennifer Hawkins, Gabby Silberman, Dennis Smith, Liam O'Brien, and Grace Lewis, for collaboration and support. I would also like to express my gratitude to Borisa Antonic, Aleksandar Gargenta, Marko Gargenta, and Chris Wilkins, for their encouragement and friendship.

I cannot forget my family, my father Dragan Ivkovic, my mother Zoja Ivkovic, my sister Sandra Ivkovic and Todd Metcalfe, and my grandmother Vukosava, for their love and awe-inspiring and neverending presence throughout my studies and my life. They never doubted that I would finish my degree, and I hope that this thesis means as much to them as it does to me. Furthermore, I cannot forget my other mommy, Narinder Sobti, who prayed and stood by me via phone and letters, and never stopped encouraging me to finish my work. I would also like to thank my extended family, namely Uttam Chawla, Rupinder Chawla, and Bobby Chawla, for their love and encouragement.

Finally, I dedicate this thesis to my wife, Shabnam, and our puppy, Shadow. Shabnam is a pillar of my existence, as I try to be for her, and without her I would not be writing this acknowledgement or submitting my thesis. And Shadow; he is not "just a dog". His intelligence, patience and human-like compassion for me have helped me when I struggled the most, and without him, I would not be where I am today.

For Shabnam and Shadow

The Two Fibers of My Life

Table of Contents

List of Figures

Chapter 1

Introduction

The beginning of knowledge is the discovery of something we do not understand.

— Frank Herbert

1.1 Motivation

From the Mariner 1 rocket disaster in 1962 and the Hartford Coliseum Collapse in 1978 to the failure of EDS IT system in 2004 and the Los Angeles Airport scheduling system collapse in 2007, all these disasters were caused by unpredictable behaviour of large and complex software-intensive systems [Dev10]. However, software is complex by its nature [Bro87], and it is not only the complexity of software that played a key part in these unfortunate incidents, but also the change that software goes through from inception to retirement. These factors increased the overall complexity and made the verification and validation of such systems even more difficult. This change that software incurs at each stage of its lifecycle is

referred to as software evolution [MBZR03]. The software entities in each evolution cycle are non-transient and they may inherit properties from their ancestors. In Darwinian evolution, change is manifested as natural selection that under the effect of time outcasts negative and upholds positive mutations of species in a given environment. In a sense, we can apply this principle to software and observe that natural selection of software systems is directed by different evolutionary forces. These forces in the context of software systems are manifested as different market conditions and changing business and technical requirements. Software evolution is also marked by different evolution cycles. Some of these cycles are considered major releases while others minor releases or even bug fixes. However, regardless of the nature of the evolution cycle, for large systems there is a need for a systematic and traceable management of the system's software artifacts, such as its source code, its requirements models, design models, and testing models, to name a few. Therefore, the main question that arises is,

> *How do we manage software models in a way that allows us to respond to the evolutionary forces in a systematic and traceable manner?*

To answer this question, we view evolution in the context of software models that not only are used to denote software artifacts, but also allow for Model Driven Development (MDD) to commence. Model Driven Development, according to the Rational Corporation [Ois02], is distilling business logic and design practice into a model or metadata that is used in application development, integration, and maintenance. This paradigm, also referred to as model-based software engineering (MBSE) [Jez03], has been hailed as a promising development paradigm both from the software engineering point of view and from object-oriented programming point of view. More specifically, modeling software takes the form of standardized processes (*e.g.*, Rational Unified Process (RUP) [IBM04]) and formalisms such

as Unified Modeling Language (UML) and Meta Object Facility (MOF). In MDD, modeling at each stage of development is the core activity, and software evolution is represented as continuous transformation of software models at different levels of abstraction. In such a development environment, it is evident that various software artifacts and models co-exist to form an integrated repository that has to be maintained in a consistent state. In this respect, a change applied to one model, such as a class diagram, may affect other models, such as sequence diagrams, state diagrams or test models. To achieve and maintain consistency among related software artifacts and models, each software change or transformation that is applied for development or evolution purposes must to be systematically applied, and its effects must be traced, analyzed, and propagated consistently to all other affected models. The main challenge, then, lies in systematically tracing and interpreting software transformations that are applied to specific models from one level of abstraction, such as design, to another level of abstraction, such as source code. Auxiliary challenges are related to overcoming different levels of expressiveness and semantics of models used at different stages of the software lifecycle.

1.2 Overview of Thesis Research Challenges

To answer the challenge of systematically tracing and interpreting software transformations from one level of abstraction to another, and maintaining consistency among all software artifacts and models, we introduce the mSYNTRA model synchronization framework [IK04a] [IK04b] [TML+04].

The framework is intended to complement and facilitate the activities of an iterative and incremental process model, such as the Rational Unified Process (RUP) [IBM04], in the

sense of facilitating the consistent management of models throughout the development and evolution phases.

As illustrated in Figure 1.1, the framework aims to assist software engineers on the following issues and challenges while developing or maintaining a software application.

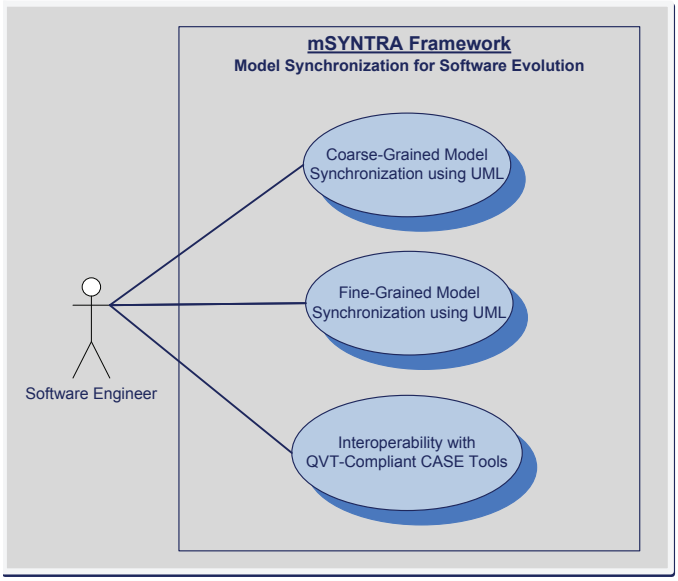


Figure 1.1: The mSYNTRA Framework Use Cases

Coarse-Grained Model Analysis for Model Dependency Extraction and Synchronization

In the context of this thesis, we consider coarse-grained model analysis as a collection of techniques that allow for the identification of dependencies between collections of different models. We refer to this type of analysis as coarse because it does not provide evidence of dependencies between individual model elements but rather between collections of model elements. Coarse-grained analysis is very useful as it can be very efficient when large models are involved and dependencies can be localized

only in smaller parts of the models instead of more accurate but computationally more expensive techniques. Coarse-grained analysis yields a collection of dependencies that can be used to build traceability links and thus facilitate tractable coarse-grained model synchronization. For example, in a large industrial system composed of millions of lines of code and hundreds or even thousands of models, coarse-grained analysis can answer questions of the form, “If I change this source code class, which design or business process models may be affected?”. Event though coarse-grained analysis allows for dependencies and traceability links to be established, it does not provide means to generate transformations that could be used to automatically synchronize models at a finer-grained element level.

Fine-Grained Model Synchronization In the context of this thesis, we consider fine-grained analysis and synchronization as a collection of techniques that allow for not only denoting model dependencies in a MOF-compliant formalism, but these also allow for synchronization of individual elements between different models. In this thesis, fine-grained synchronization takes the view of denoting model dependencies in a formalism that can be applied in an automated and verifiable way. More specifically, for fine-grained model synchronization, we take the view that of model dependencies denoted as grammar rules that can be used to map individual elements of one model to individual elements of the other model. The formation of the rules and the transformation process guarantee that the consistency requirements and constraints are valid after the synchronization process is applied.

Interoperability with QVT In the context of this thesis, we also consider that aforementioned issues have to be resolved by techniques that can be implemented in an automated tool and enacted as part of an algorithmic synchronization process. Over the

past few years, Query/View/Transformation (QVT) Standard [OMG11b] has been proposed as a standard collection of languages to facilitate model transformation. For this thesis we aim for integrating the proposed model synchronization techniques in a way that is compatible with QVT notation and process.

1.2.1 Introduction of Terms

We observe two repositories of MOF-compliant models, M and G, where concrete models are instantiated from domain-specific models DM_M and DM_G respectively (as shown in Figure 1.2).

The models $m_i \in M$ and $g_j \in G$ are models for the same software system, but at different levels of abstraction or different semantic detail. Each model m_i is composed of model elements $me^i_k \in m_i$, and each model g_j is composed of model elements $ge^j_l \in g_j$.

For a more specific synchronization scope, that is to synchronize two specific models and not model repositories, one can view M and G as models composed of individual model elements $m_i \in M$ and $g_j \in G$.

A model-change dependency between model repositories M and G is a set of tuples (m_i, g_j) of models $m_i \in M$ and $g_j \in G$, or between their corresponding model elements, such that m_i and g_j have associated attributes. We refer to this kind of model-change dependency within the context of this thesis as model dependency, in short.

1.2.2 Use Case: Coarse-Grained Model Synchronization using UML

In a situation where the repositories M and G are related, but the specific relations between individual models from M and G have either not been established, or have been lost over

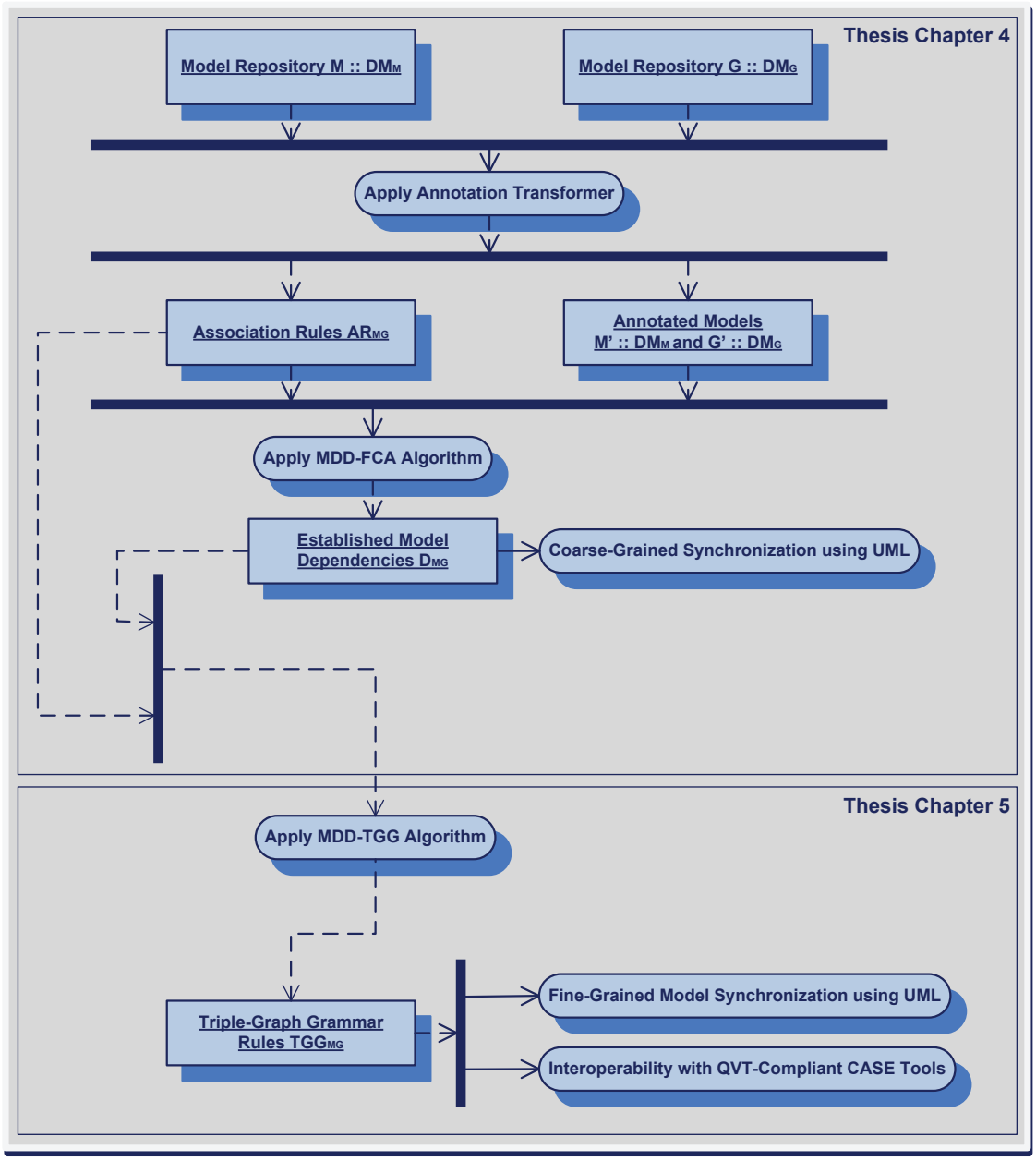


Figure 1.2: Thesis Overview

time due to lack of systematic synchronization, we first need to create a coarse-grained synchronization context. To provide such a coarse-grained model synchronization, we need to establish dependency relations between the models $m_i \in M$ and $g_j \in G$, or between their corresponding model elements. Consequently, we should allow for manual or tool-assisted tracing of such dependencies transformations that would assist on resolving any potential inconsistencies between related artifacts. As an example of the coarse-grained model synchronization, let us consider that there is a business process (*e.g.*, Business Process Execution Language (BPEL) [OAS07] document) that is enacted by a run-time system (*e.g.*, a web service), and the goal is to establish bidirectional coarse dependencies between business tasks and their implementation, either as a collection of run-time services or source-code packages. We refer to this analysis as coarse grained because it aims to establish dependencies and traceability links between models that have significant difference in the level of abstraction of the artifacts they specify (*i.e.*, business processes on one hand, and source code on the other). The basic elements of this use case for coarse-grained synchronization are presented below.

Apply Annotation Transformer

To establish the coarse-grained synchronization context for model alignment, we input the domain-specific models DM_M and DM_G to an annotation transformer to semantically align models. The transformer is discussed in detail in Chapter 4. It first annotates both domain models with relevant attributes to resolve semantic differences between them, and consequently establishes association rules, AR_{MG}, based on compatible domain types and relations. Finally, it applies the annotation attributes to individual models from the repositories M and G, thereby creating M' and G', that have their semantic differences resolved. An example of a semantic difference between model elements is when one model element has an

attribute *a1* and the other has an attribute *a2* that both refer to the same context or item. An association rule can be used to make these two attribute names isomorphic.

Apply MDD-FCA Algorithm

The output from the annotation transformer, the annotated models M′ and G′ and association rules AR_{MG}, are used as input to an algorithm that establishes specific dependencies between related models. The algorithm, named MDD-FCA, is also described in more detail in Chapter 4. The algorithm uses the rules from AR_{MG} and Formal Concept Analysis (FCA) to establish dependencies between M′ and G′, and consequently between M and G. The identified dependencies D_{MG} are used to support coarse-grained model synchronization. For example, let us assume that the consistent state between the repositories M and G is disrupted when model m_t is transformed into m_t' by changing some elements of m_t, such as $me_1^t \ldots me_k^t$. To identify elements of G that are affected by change to m_t, the tuples from D_{MG} are used to find all (m_i, g_j) that contain element m_t, or at the model element level, the tuples that contain $me_m^t \in \{me_1^t \ldots me_k^t\}$. For each identified tuple (m_t, g_j) or (me_m^t, ge_l^j) from D_{MG}, the model g_j or element ge_l^j respectively is identified as one of the elements that may need to altered to maintain the synchronization between M and G. Once all of the affected elements are found through iteration, they can be set for manual or tool-assisted updating. Since the analysis is focused on the identification of dependencies between sets, we consider the analysis as coarse as compared to fine grained.

Utilization of Standardized Modeling Infrastructure

Let us assume that M and G represent related repositories of models at different levels of abstraction, such as platform-independent and platform-specific models, specified in UML

and within the context of Model-Driven Architecture (MDA) [OMG01], and the specific relations between them are known. We would like to enable synchronization between models contained within M and G. For this purpose, we introduce a UML-based approach where the dependencies D_{MG} between models in M and G are encoded as UML stereotypes and relations defined within DM_M and DM_G are identified and encoded as corresponding association rules, AR_{MG}. More specifically, if model m $\in$ M and g $\in$ G are dependent, then m as the source model can stereotype g (the target model). In this way, dependencies are modeled and enacted at the UML / MOF level. The approach is described in more detail in Chapter 4. The identified association rules, AR_{MG}, are previously known (*e.g.*, provided by software developers who are tasked with maintaining them) or are created by applying the annotation transformer. Similarly, specific dependencies between model elements D_{MG} are previously known or are created by applying the MDD-FCA algorithm. With AR_{MG} and D_{MG} available, the algorithm establishes specific profile-based dependencies between UML model elements, DP_{MG}, where the left-hand side of an association rule $ar_i \in AR_{MG}$ is encoded as a UML stereotype or stereotype package that is applied to the right-hand side of the same rule. The correspondence between individual models is then achieved by, first, applying the stereotype mappings for types and relations, and then second, by applying dependency relations from D_{MG} for mapping model-specific attributes (*e.g.*, class names).

1.2.3 Use Case: Fine-Grained Model Synchronization using UML

Let us assume that the repositories M and G represent related models at different levels of abstraction, for which a corresponding set of profile-based dependencies, DP_{MG} has been established. In this context, we would like to establish a frameworks to support fine-grained model synchronization. To fulfill this objective, we introduce a technique based on Triple

Graph Grammars (TGG). The technique, named MDD-TGG, is described in more detail in Chapter 5. The basic phases of this use case are discussed below.

Apply MDD-TGG Algorithm

The set of profile-based dependencies, DP_{MG}, is either previously known or is created by applying the approach described above. For each $t_i \in DP_{MG}$, a new triple graph grammar rule, tgg_i is created with the source stereotype or stereotype package of t_i representing the left-hand side of the new rule and the the target package representing the right-hand side of the new rule, with any other constraints of t_i encoded as the correspondence node (*i.e.*, the node linking two related nodes) of tgg_i. The new set of grammar rules, TGG_{MG}, is created as the output. To perform fine-grained model synchronization, we assume that the consistent state between the repositories M and G is disrupted when m_t is transformed into m_t' by changing some elements of m_t, such as $me_1^t \ldots me_k^t$. However, instead of just identifying elements of G that are affected by change, we identify rules in TGG_{MG} that pertain to $me_m^t \in \{me_1^t \ldots me_k^t\}$, and then, apply those rules to identify specific inconsistencies in models and model elements of G. Before any changes are committed, the effect of each rule can be investigated, and then, the rule can be applied; or it can be ignored if it is found to be redundant (*e.g.*, overlapping rules).

1.2.4 Use Case: Interoperability with QVT-Compliant CASE Tools

With the TGG_{MG} set of rules available, one can utilize a TGG-to-QVT mapping, as described in Chapter 5, to export the created transformations into the QVT format. The exported transformations can then be utilized within many CASE tools that are compliant with the QVT

standard, to enable interoperability with other modelling frameworks and integration into different software lifecycle models.

1.3 Research Contributions

Model synchronization activities that are needed to achieve systematic and traceable change propagation and interpretation across different levels of abstraction are still not adequately identified and defined. In this context, the mSYNTRA framework aims to address the following issues and challenges:

1. To provide a method for bridging the semantic gap between domain models at different levels of abstraction;
2. To devise a method for identifying dependencies between model elements;
3. To enable coarse-grained model synchronization by utilizing identified dependency relations between models or model elements;
4. To introduce a method for representing domain models as domain-specific context-free grammars and their dependencies in terms of UML classifiers; and
5. To enable fine-grained model synchronization by utilizing identified triple graph grammar rules.

1.4 Limitations

Limitations of the mSYNTRA framework include:

- Handling circular dependencies, where we do not explicitly address possible circular dependencies present within models (*e.g.*, inheritance chain, where A is a superclass of B, B is a superclass of C, and C is a superclass of A). Instead, if presence of circular dependencies is detected, it can be addressed through additional user-defined constraints, specifically aimed at preventing circular dependencies from occurring.

- Conflict resolution for conflicting synchronization actions, where we do not explicitly address a scenario of two or more synchronization actions that occur at the same time and have conflicting side effects. If such a scenario is detected, it can be resolved through serialization of synchronization actions, or through additional user-defined constraints, aimed at preventing conflicting synchronization actions from occurring.

- Pre-selection of applicable rules, where we do not explicitly specify which synchronization rules need to be applied given a specific change (*i.e.*, localized model synchronization). Instead, our approach reapplies all of the available rules, and relies and source pattern matching for rule selection (*i.e.*, global model synchronization).

1.5 Thesis Scope

The systems targeted with this research are software-intensive applications that utilize models, and not just source code, as their primary artifacts. The models to be considered covered include MOF-compliant models, in the form of UML, and include models that pertain to the development of the software, from requirements to deployment to maintenance phases.

1.6 Thesis Organization

This thesis is organized as follows:

- Chapter 2 describes previously conducted research that relates to the mSYNTRA framework.
- Chapter 3 introduces formal notation for the representation of domain models and model instances.
- Chapter 4 discusses the approach for establishing model dependencies using formal concept analysis, and introduces the method for representing model dependencies using UML profiles and constraints.
- Chapter 5 presents the theory for mapping profile-based dependencies as triple graph grammar rules.
- Chapter 6 covers the case study results of applying the mSYNTRA framework to a large and complex industrial case study of the IBM Commerce systems.
- Chapter 7 provides the final conclusions and directions for future research.

Chapter 2

Related Research

How inappropriate to call this planet Earth when it is quite clearly Ocean.

— Arthur C. Clarke

2.1 Chapter Overview

In this chapter we present related research used as the basis for this thesis. We present a detailed discussion on software models, model transformations, grammar-based model transformations, model dependency extraction, and model synchronization.

2.2 Software Models

Due to the inherent complexity of software, even for a modest size system, we are inclined to use various models as simplified versions of the system for purposes of understanding,

planning, and evolution. From our viewpoint, a model is a model element that contains a group of model elements representing an abstraction of a system or its parts from a specific perspective and at a particular level of detail. For example, one system might be represented by more than one model on the same level of abstraction, and different models might share corresponding elements [Cor03].

Software engineering models can be categorized according to the stage of development to which they relate: from requirements to source code to maintenance models. Relationships between models and original entities that they represent can be classified as *descriptive*, those that mimic an underlying original; *prescriptive*, those that specify something to be created; and *transient*, those that first describe and then prescribe changes to an underlying original. Making changes to a system represented through models is not done directly; instead, it is done on an abstraction level of the model and then mapped to the underlying system at hand [Lud03].

2.2.1 Architecture Models

When it comes to software architecture, several definitions exist including:

- “Software architecture [is a level of design that involves] the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on those patterns.” [SG96]
- “The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.” [BCK98]

To model software architecture, various Architectural Description Languages (ADL) are available, including UML [OMG10]. When modelling software architecture, different architectural views are used to represent specific aspects of interest (*e.g.*, 4+1 View Model [Kru95]), such as concurrency or data flow. Architectural views in practice are geared towards key stakeholders with the understanding of their concerns, and the understanding of how they model and deal with those concerns. Different types of architectural views that exist include [Bre98, CKK02]:

Conceptual/Logical View Major functional components are identified and responsibilities of those components allocated. Concrete View identified in [BHB99] also shows the logical decomposition but from the implementation perspective (*i.e.*, it shows the logical structure that resulted after the conceptual/logical view was implemented).

Concurrency/Execution/Process View Runtime component instances are assigned to processes, threads, and address spaces. It demonstrates how the runtime components communicate and coordinate, and how they share physical resources among them.

Code View Presents classes, objects, procedures, and functions along with their abstractions and compositions into subsystems, layers, and modules. Typical relationships include function calls, method invocations, and containment such as is-a-sub-module-of.

Development/Implementation View Shows a hierarchical structure of files and directories in the implementation of a software system. It shows directories along with the source and the header files contained in them.

Use Cases/Scenarios View Externally-visible interfaces are mapped onto system subcomponents. It shows an interaction among elements of other views in the context of a

particular functionality (*i.e.*, a scenario for a particular use case).

Architectural concerns can also be modeled using architectural design patterns, as described by Alencar *et al*. in [ADL96].

2.2.2 MOF and UML

With creation of the MOF specifications [OMG06], a four-level, layered architecture for model engineering was introduced (see Figure 2.1). The architecture consists of the following layers [PZB00]:

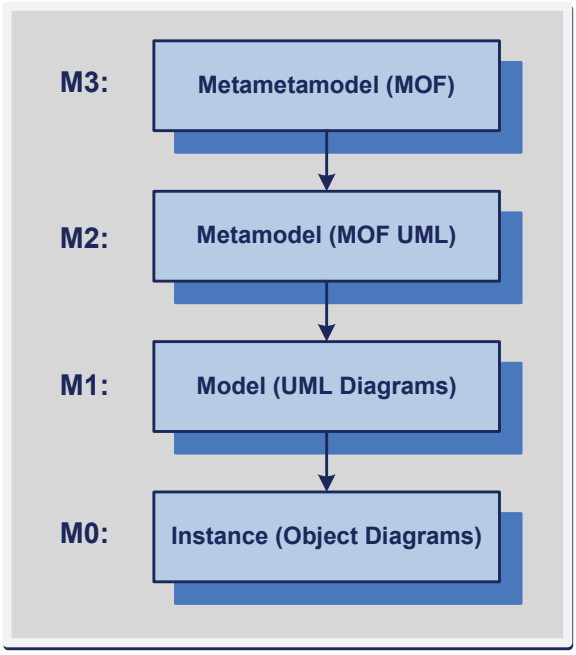


Figure 2.1: Four Level Architecture for MOF-Compliant Models

- M3 — The metameta level, that is, a self-defined language used for defining other languages at level M2. This level contains only one recommendation and that is MOF.
- M2 — The meta level, that is, a set of domain specific metamodels. From OMG's perspective, any metamodel is defined in terms of MOF. A domain specific metamodel defines a language to write models at level M1. For example, a recognized metamodel is the UML metamodel, but others exist including a Java metamodel.
- M1 — The model level. Any model is compliant with a specific metamodel. For example, a specific UML diagram of a web browser describing its components would be one of the M1-level models. Another example would be a Java implementation of one of the web browser components.
- M0 — The instance level. This levels represents instances (objects) derived from the specific M1 model. For example, an execution of the Java implementation of one of the web browser components could be described using UML object diagrams.

In this architecture, all levels are considered to have instance relation with their parent level. As an example, the M3 level enables a MOF model to define various metamodels that can appear at M2 level. However, if by using the MOF model, one can express all metamodels, then it is possible to transform a model based on one metamodel to another.

The Unified Modeling Language (UML) [OMG10] is a specification, visualization, construction, and documentation language for software system artifacts and business modelling. UML is meant to represent a standardized collection of proven engineering practices for modelling of large and complex systems. One of the primary design goals of UML is to provide users with an expressive visual modelling language for creation and exchange of software models.

UML 2.0 defines three main categories of diagrams [OMG11a].

Structure Diagrams These include the Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram. The emphasis is on modelling static aspects of the system.

Behavior Diagrams These include the Use Case Diagram, Activity Diagram, and State Machine Diagram. The emphasis is on modelling dynamic aspects of the system, at a higher level of abstraction.

Interaction Diagrams These include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram. The emphasis is on modelling dynamic aspects of the system, at a lower level of abstraction.

All of the above diagrams are part of the UML 2.0 formal specifications as described in [OMG10].

2.2.3 Source Code Models

Source code models provide structured techniques for representing source code information at a higher level of abstraction than source code text. Various source code representation formalisms exists [Mam00], and here we are list the ones that pertain to our research.

Abstract Syntax Tree (AST) The AST is a tree structure that represents the syntactic information contained in the source code [ASU86]. A node in a tree is an element of the language, where non-leaf nodes represent operators and leaf nodes represent operands. An AST does not include syntactic information of punctuation tokens that

are implicit from the AST structure. The AST notation is commonly used by compilers for internal representation of source code for analysis, optimization, and binary code generation.

Abstract Semantic Graph (ASG) The ASG is defined as an AST with embedded semantic information. To specifically distinguish between the two, a reference to an entity in an AST is represented by an edge pointing to a leaf node that holds the name of the entity reference. In an ASG, a reference is represented by an edge pointing to the root of the subgraph in the ASG that represents the declaration of the entity.

Program Dependence Graph (PDG) The PDG combines control and data dependence information, where nodes represent statements, expressions, and regions of code, and the edges represent control or data passed from one expression to another along with control conditions that influence order of execution. Unlike the AST that directly represents source code information, the PDG represents information derived from the source code. The information stored in a PDG is commonly used for software testing purposes as well as developing code optimization algorithms.

The Control Flow Graph (CFG) is used for encoding control flow information [All70]. Within a CFG, the nodes represent statements and the edges represent transfer of control between statements.

Construction of a PDG precludes extraction of control flow information, which can be derived from a CFG.

2.3 Model Transformations

Model transformations research spans research on propagating change across models on the same level and on different levels of abstraction. Mapping elements from one model to another is a crucial activity in this process, where entities affected by change in one model reflect the change on entities of other models. The mappings are also a basic tool for MDA model-based development [KWB03]. Techniques for the mapping description identify the elements of the source model that are mapped and the destination model that correspond to the source elements, along with conditions that must be fulfilled to apply this type of transformation. In current research, the mapping process is put in practice in several different ways including [MES02]:

1. Script Languages — Certain UML tools include imperative script languages with meta-model navigation facilities similar to Object Constraint Language (OCL) [WK98] navigation expressions. Such languages, which serve as support to implement mapping scripts, are flexible but suffer from the deficiency of tool dependence.

2. XML and XMI-Based Mapping — Certain tools provide support for deriving XML and XMI files. These files include the metadata of UML models, and are used to facilitate mapping. The mapping is independent of the UML tool but is dependent on the specific transformation technique supported by the tool that is used for file creation.

3. MOF Transformation Facilities — There are tools that include MOF transformation facilities based on rules. These rules enable identification of the elements in the source model to which the rule applies, and the destination elements that can be generated with the rule.

When discussing model transformations, two main concepts are relevant: the modelling languages used for source and destination modelling, and the mapping between modelling languages [IK06]. MDA addresses these aspects particularly using UML extensions where the matching UML transformations can be used for the following purposes [MES02]:

1. UML Model Transformation and Refinement — MDA proposes the refinement and transformation of models as a basic technique to extend or specialize a model. To avoid platform dependencies in models, Platform Independent Models (PIMs) can be transformed into Platform Specific Models (PSMs) to introduce platform specific concepts, where some concepts are automatically introduced in the generated model and others are updated manually. Intra-level model refinements — PSMs to PSMs and PIMs to PIMs — enable improvements of models in the same modelling language space.

2. UML Model Evaluation — UML standards that address UML extensions and facilities for the transformation of UML extended models into other types of modelling techniques.

3. Implementation Generation — Code generators that develop platform specific implementations can be used to implement the PSM. These generators translate UML model into a selected programming language and middleware constructors (*e.g.*, Java and CORBA interfaces along with Enterprise Java Beans (EJB) component descriptors).

2.4 Grammar-Based Model Transformations

The research related to grammar-based model transformations is divided into the following categories: model transformation taxonomy, coordination theory, metamodels as grammars,

and model transformation management.

2.4.1 Model Transformation Taxonomy

A classification of model transformation approaches is provided by Czarnecki and Helsen in [CH03]. Following this classification, each transformation rule consists of two distinct parts: a left-hand side (LHS), which refers to the source model, and a right-hand side (RHS), which refers to the target model. To create both the LHS and RHS, one can use a combination of

1. patterns, such as string, term, and graph patterns;
2. logic, such as computations and constraints on model elements; and
3. variables, which hold model elements of source, target, or some intermediary model.

We also follow a taxonomy of model transformations provided in [MCG05] that classifies transformations as

1. endogenous, if they transform models in the same language, or exogenous, if they transform models between different languages; and
2. horizontal, if the transformed models are at the same, or vertical, if the transformed models are at different levels of abstraction.

In this thesis, we view transformations as components of the model synchronization framework. Our focus is on effects of exogenous, vertical transformations that we represent in pattern-like format using attributed context-free grammars.

2.4.2 Coordination Theory

Our interpretation of the problem of synchronizing heterogenous entities is related to Coordination Theory originally published by Malone [Mal90]. In this work, coordination is defined as the act of managing interdependencies (generally viewed as constraints) between activities performed to achieve a goal.

We adopt this theory to define model synchronization as the process of managing constraints among software artifacts, which are based on established model interdependencies.

2.4.3 Metamodels as Grammars

The idea of representing software models as context-free grammars was previously described by Metayer [Met96], where software architecture styles are formalized as n-ary relations and represented through context-free grammars by identifying a role as unary relation and a link between entities as a binary relation. The representation of types as relations, even though plausible for architectural styles, when extended to more specific metamodels or domain models does not address the problem of ambiguity in the resulting formalisms. Alanen and Porres [AP03] have derived a method for interpreting MOF metamodels directly as Extended Backus-Naur Form (EBNF) grammars, and have also identified an inadequacy of EBNF to handle attributed edges, a crucial feature of many object-oriented models (*e.g.*, attributed and directed associations between classes).

We adopt Metayer's view of models as n-ary relations, but we interpret a MOF-compliant domain model uniquely as tuples of types, relations, connectors, and attributes. From the tuples, we generate a context-free grammar that is capable of handling attributed edges, and hence, is capable of representing specialized associations such as aggregation, composi-

tion, and generalization.

2.4.4 Model Transformation Management

Akehurst [Ake00] attends to the problem of model translation by utilizing a combination of UML and OCL to specify transformation relations between two object-oriented models. A similar technique is proposed by Milicev [Mil02], which utilizes extended UML object diagrams to specify translation between source and target metamodels. The Fujaba approach [Fuj05] is based on an extended-through-action stereotypes combination of UML activity and collaboration diagrams as story diagrams [FNTZ98] for the specification of model transformations.

In [Tah04], Tahvildari focuses on legacy system re-engineering. In the proposed approach, the evolution is driven by both functional and non-functional software requirements. To represent software qualities and software transformations that may affect them, the NFR framework is used [CNYM00]. A soft-goal interdependency graph (SIG) is used to model software architecture design, where the leaves of the SIG are design decisions that positively (++) or negatively (–) affect the soft-goals above them.

We also base our approach on adoptable UML/OCL representation for models. However, we use context-free grammars as abstractions of models to create a generic approach that can systematically be adopted for translation of any two related MOF-compliant software models.

2.5 Model Dependency Extraction

The research related to dependency extraction is divided into the following categories: software reuse, hierarchical data management, and program flow analysis.

2.5.1 Software Reuse

In the approaches that focus on software reuse, Spanoudakis and Constantopoulos [SC94] measure similarity through a distance metric in order to evaluate the reuse potential of software artifacts. Engels *et al*. [EHSW99] discuss the transformations between Unified Modeling Language (UML) Class Diagrams and UML Collaboration Diagrams [OMG10] and Java source code. The approach considers the structural and behavioral mappings using transformation patterns. The patterns used are not trivial to extract and the pattern repository needs to be updated as new transformations are introduced. The approach in this paper uses formal concept analysis to establish the mappings at the level of model elements. For objects that belong to more than one concept, conflict resolution is performed using a similarity metric, represented for instance as a sum of weighted scores.

2.5.2 Program Flow Analysis

In [Sev87], Seviora describes the use of program flow analysis and understanding in knowledge-based debugging systems. Two different program understanding approaches are recognized. In the code-driven (bottom-up) approach, symbolic evaluation and recognition of standard programming constructs are used to form an abstract representation of the program and its individual parts. In the problem-driven (top-down) approach, using the existing program specification the structure of the program is derived and refined until it can be verified

against the code. Tilley *et al*. in [TSP96] follow this categorization of program understanding, and also identify iterative hypothesis refinement and hybrid approaches (*i.e.*, combinations of top down and bottom up) as two additional categories. They also view reverse engineering and program analysis in a canonical fashion as a three-step approach: (1) modelling, where domain-specific models of the application are constructed; (2) extraction, where data is gathered from the subject system using appropriate extraction techniques; and (3) abstraction, where abstract representations are created that facilitate understanding and exploration of the considered information structures.

In the research published by Rich and Wills, subgraphs are used to recognize program design [RW90]. In their approach, several categories of problems related to establishing model dependencies are recognized. These include non-contiguousness — adjacent elements from one flow may be separated in another related flow, implementation variation — the same design under differing contexts may be represented by different implementations, overlapping implementations — two or more implementations may overlap in implemented functionality, and unrecognizable implementations — no relevant semantic information can be extracted from an information flow. This view is extended through our research by considering challenges related to: n-ary relations — dependencies are not only one-to-one or one-to-many, but also many-to-many mappings of model elements, partial dependencies — only parts of model elements are related, and non-applicable dependencies — an element from one model is not directly mapped to any element in a related model. These challenges are addressed through the following:

- Non-contiguousness, overlapping implementations and partial dependencies are addressed through mappings of individual model elements instead of flow patterns.
- Implementation variation is addressed through mappings of interfaces and recognition

of differing contexts for each mapping.

- N-ary relations are addressed through definition of model dependencies as tuples of model elements.
- Unrecognizable implementations and non-applicable dependencies are addressed through inclusion of user feedback.

In [RMJ06], Rayside *et al*. introduce an approach to program flow analysis where object ownership is used to cluster related objects. The key aspect of this approach is control, where one object (x) is said to own another object (y) if x is an immediate dominator of y (*i.e.*, the dominator relationship implies that every path from the root node to y passes through x) in the corresponding program control graph.

2.5.3 Model Management

In the research area of hierarchical data management, an approach by Faid *et al*. [FMG99] uses formal concept analysis to discover concepts and rules based on structured complex objects. Gianolli and Mylopoulos [RGM01] perform semantic mapping of XML data stores using a common DTD schema. In this paper, the semantics of hierarchical data structures are mapped using intermediate models. However, the relations among individual model elements are also inferred based on the mappings of related attributes.

In [SME08, SME09], Salay, Mylopoulos and Easterbrook propose a framework in which formally defined model relationships are used to express relationships between models at a high-level of abstraction by constructing a macromodel while maintaining comprehension and consistency during evolution. The approach is limited in that it works with models within a collection only.

In view consistency management, Sabetzadeh and Easterbrook [SNEC06] address the problem of incompleteness and inconsistency in graph-based view integration towards gaining a unified perspective in conceptual modeling. The authors present a flexible and mathematically rigorous framework based on structure-preserving maps and a general algorithm for merging views systematically and traceably. However, identifying potential interconnections between views is still a manual process, the merges do not function on a semantic axis, and hierarchical structures are not supported.

2.6 Model Synchronization

Within the context of model synchronization related research, we identify three categories: triple graph grammar (TGG) approaches, approaches based on other model transformation languages, and software co-evolution approaches.

2.6.1 TGG-based Model Synchronization

In TGG-based model synchronization, triple graph grammar rules [SK08] are used to establish relations between source and target models. Each rule consists of the source pattern, the correspondence nodes, and the target pattern. The synchronization is accomplished by

- Matching the source pattern in the source instance model,
- Using the correspondence nodes to check applicable constraints and to identify the target pattern, and
- Applying the target pattern to the target instance model, to either generate missing model elements or identify the ones that need to be altered or removed.

The emphasis of related research in this area is on improving the run-time efficiency of the application of the TGG rules. Many of the related approaches assume that the rules have already been established, and focus on exploring various synchronization algorithms [KLKS10].

In [GW06, GW09], Giese and Wagner have introduced an approach for incremental model synchronization using triple graph grammars. The use of TGGs allows for the formal and bidirectional synchronization of larger models as compared to the traditional batch approaches. They use additional links between correspondence nodes to interpret the correspondence model as a directed acyclic graph. Their approach does not declaratively address node deletions. Instead, in their implementation, for unsatisfied transformation rules the related correspondence node and all created elements are deleted. The drawback to the approach seems to be that the expressiveness of the grammar variant does not include non-local properties, and as such, operational or hybrid transformation languages may be more expressive.

Giese and Hildebrandt in [GH08] present an improvement of batch processing of changes for multiple updates. The original incremental algorithm's performance is severely compromised if many modifications are made because their algorithm synchronizes changes in the order that they occur. Their improvement to the incremental algorithm accelerates synchronization for a large number of changes in the models, and is comparable to the speed of the batch algorithm. This is possible because their algorithm starts the synchronization at the modified correspondence nodes by sorting the modifications by their depth and synchronizing at the correspondence node that is closest to the root.

In [GK07, GK10], Greenyer and Kindler compare and contrast the philosophy and the concepts of the declarative languages of QVT and TGG, to enable tool-supported TGG-based synchronization. They conclude that relational QVT-Core mappings can be converted

to TGG rules and executed by a TGG transformation engine. They propose an initial approach for mapping QVT-Relations to TGGs. They also use the semantic foundation of TGG to better understand the semantic gaps of QVT. We base our mapping of TGG rules to QVT on this approach.

Dang and Gogolla in [DG08] propose a language for the integration of TGG rules and OCL based on QVT using their USE tool. The approach is based on declarative OCL pre- and post-conditions for operation contracts, and imperative command sequences for operational realization. This integration is demonstrated by discussing examples supporting model co-evolution and consistency. The model does not handle large-scale syntax and semantics translations, and cannot definitively comment on the correctness of correspondence models.

2.6.2 Synchronization based on Model Transformation Languages

Other approaches to model synchronization that do not utilize TGG instead employ other model transformation languages, such as Atlas Transformation Language (ATL) [JK05], to identify and synchronize corresponding model properties. The approaches in this area focus on refining the transformation languages to more efficiently suit the needs of the model synchronization (*e.g.*, creation of synchronization rules from model transformation rules), and where possible enable bidirectional synchronization.

Xiong, Liu, *et al*. in [XLH+07] discuss an automatic approach, that satisfies stability, preservation and composability properties, to synchronizing models based on a unidirectional transformation between the related metamodels in the Atlas Transformation Language (ATL). While the algorithm produces clear synchronization semantics, the approach seems rather limited in that it is based on the prerequisite that model transformations from one model to another be given. The proposed algorithm is a promising start, but is also limited in

that reflectable insertions on the target side cannot be handled.

Diskin in [Dis09] presents an algebraic framework for specification and design of model synchronization tools based on the algebraic operations being diagrammatic and resulting in compositions of tiles. Two approaches — abstract (or black box) and concrete (or white box) — provide a formal notation for synchronization. One of the obvious limitations is that the approach considers very simple constraints of single multiplicities, and does not handle multi-layer mappings.

Eramo, Malavolta et al, in [Mal10], discuss that description of any software architecture requires multiple Architecture Description Languages (ADL), which results in the need to maintain consistency in the multiple notations. The authors propose a generic convergent and scalable change propagation approach between multiple architectural languages. The approach is implemented within Eclipse, and supports model mappings by using bidirectional transformations. The authors are working towards a learning approach that deduces information to be used for further mappings from user choices, and that is based in collaborative modeling.

Mens in [Men05] postulates the representation of models as graphs and model transformations as graph transformations for model refactoring. This is implemented in the Attributed Graph Grammar System (AGG) [Tae03] using critical pair analysis and Fujaba using round-trip engineering for a simplified version of UML class diagrams and statecharts. Improvements to this work include the expressiveness of the approach enhanced by a mix of graph and tree representations, the ability to compose more complex and composite refactorings, and the ability to maintain consistency between models as they evolve.

In another publication [Men], Mens addresses the issue of working with models that have design defects in the context of graph-transformation based software evolution. A tool

developed as a front-end of the AGG Engine is proposed, and is based on properties such as critical pair analysis and reasoning about sequential dependencies. The approach does not address the termination of the iterative and interactive defect introduction in the models, or the order of resolving the defects, though it does comment on initial thoughts about both issues.

2.6.3 Synchronization through Software Co-Evolution

Finally, there are related approaches that do not focus on model transformations as a change apparatus, but instead focus on other aspects of co-evolution, such as interpreting intention of change from one abstraction layer to another [Kön10], consideration of aspects in the context of co-evolution [DT10], domain-specific synchronization using a custom-built synchronization environment [RÖV10], and design patterns as facilitation of change [LLM09].

More specifically, Konemann in [Kön10] presents a discussion on capturing user's intention of the atomic changes made to a model, specifically, MOF-compliant ones. The author presents a framework for producing the semantic aspect of the atomic changes (*i.e.*, capture the user's intention in making these changes). The goal is to abstract changes which may be applicable to other models as well, and is achieved by identifying individual and flexible matching strategies defined for each semantic change through two stages: (1) an automated initial matching; and (2) an interactive refinement step that is optional. One obvious drawback is that this approach is not automated. Also, of the refactorings applied, less than 50% were only applicable if the models are similar to the sample models, and about 65% of the design patterns applied had to be adjusted by the user after application.

Dahanayake and Thalheim [DT10] discuss multi-model information systems modeling from the point of view of 'facets' and 'aspects' where model synchronization is necessary for

creating harmonized and coherent models needed for specifying these facets and aspects. The authors introduce the theory of model suites as a set of models with explicit associations among the models, which include tracers and explicit controllers for coherence and application schemata for harmonization during evolution. The experimentation is based in MetaCASE to generate toolkits. The limitation of the approach seems to be that it is applicable only to highly structured models, such as UML diagrams, extended ER models, and XML models.

In [RÖV10], Ráth, Ökrös and Varró discuss the issue of the limitation of restrictions imposed on traceability links between abstract concrete syntax by most Domain-Specific Modelling (DSM) frameworks and propose a syntax-driven domain-specific model which generates complex mappings under a novel DSM environment. Also, they demonstrate bidirectional synchronization. While the approach is highly scalable, the approach is driven by syntax only.

In [LLM09], Levendovszky, Lengyel and Mészáros propose the possibility of design patterns as efficient solutions for recurring issues with the rapid development of domain-specific modeling languages. On the basis of constructs that weaken instantiation rules, the authors discuss the appropriateness and sufficiency of these rules to express patterns. The theory does not handle non-relaxable metamodels for OCL-like constraints.

2.7 Chapter Summary

In this chapter we have presented topics in software models, model transformations, grammar-based model transformations, model dependency extraction, and model synchronization, as related to our research. In the next chapter, we introduce a formal notation for the repre-

sentation of domain models and model instances. This notation is based on the previously conducted research discussed in this chapter.

Chapter 3

Model Synchronization Notation

The problems of language here are really serious. We wish to speak in some way about the structure of the atoms. But we cannot speak about atoms in ordinary language.

— Werner Heisenberg

3.1 Chapter Overview

In this chapter, we introduce formal notation for the representation of domain models, model instances, and model stereotypes that will be used throughout the thesis to form the theoretical basis for the proposed model synchronization framework (mSYNTRA).

Besides conforming to a generic MOF metamodel, we consider that each software model also relates to an application domain context (*e.g.*, operating systems, web services). In this

respect, we are addressing the problems of model synchronization by focusing on domain-specific contexts. Each such domain context consists of domain-specific types and relations that are associated with corresponding attributes and values. A formally defined abstraction of such a domain context is referred to as, a domain model. A domain model may be available as part of the design documentation or may need to be extracted using domain analysis techniques such as the Feature-Oriented Domain Analysis (FODA) technique [Kea97]. In addition to domain types and relations, a domain model may also contain additional meta information such as ontologies and feature maps [CK06]. Based on formally defined domain models, concrete software model instances can be represented using these concepts that are specific to a domain context. With model elements represented in terms of domain types and relations and further contextualized with the meta information, the models become more lucid and the ensuing evolution and maintenance activities are simplified. Examples of domain models that conform to specific contexts are domain models that represent SOA e-commerce systems, CORBA banking applications, and e-procurement web services, to name a few.

In the proposed approach, we define a formal context for domain-specific model synchronization by representing a domain model as a semantically-annotated context-free grammar that we refer to as, domain-model grammar. In this respect, context-free grammars provide a flexible and extensible way of generating model instances in a formal manner and consequently verifying the syntactic and semantic validity of these instances with respect to their corresponding domain model or metadata constraints. More specifically, domain types and relations are represented as grammar productions while additional semantic content is denoted by “semantic heads” (constraints specifying semantic properties) attached to each of the production rules [SNPM90]. Dependencies between models that belong to different domains are viewed as associations between corresponding grammar rules and are

formally represented through an association grammar. The semantic heads and the association grammar are derived from a synchronization relation, which is established in reference to specific domain models. The synchronization relation specifies properties and constraints that must hold for two models from the corresponding domains to be synchronized. Using the association grammar, the source model can be used to automatically generate a version of the target model that is synchronized with the source (*i.e.*, to translate a model from one domain to another). In this context, an analogy the to the synchronization of two models is the synchronization of two sentences in two different languages such as English and French. When one sentence changes, the other needs to be changed as well in order to preserve structural and semantic properties. These properties can be defined as associations between corresponding grammars.

The goal is to represent models as “sentences” that comply to corresponding grammars, and then view model synchronization as grammar-based language translation, analogous to translation of sentences between natural languages (*e.g.*, translation between French and English).

3.2 Motivation: Using Model Dependencies in Grammar-Based Model Synchronization

Our research in the area of model synchronization uncovered the following research issues and challenges:

1. Resolving model inconsistency through traceability consists of several types of manual or, at best, semi-automated activities that include:

(a) identifying and maintaining dependency relations among models and their elements,

(b) tracing model transformations as they are performed,

(c) mapping transformations from one domain context to another, and

(d) ensuring that the source and target models are finally synchronized.

2. Automating the identification of model dependency relations requires the creation of concept mapping rules.

3. Mapping of individual transformations across domain boundaries requires complex syntax and semantic mappings and validation.

4. Validating the synchronization between the considered models requires the definition of hierarchical synchronization relations.

5. Enabling pragmatic model synchronization requires significant manual overhead and creates feasibility and scalability detriments.

To resolve some of these challenges and make our approach to model synchronization more practical, we have opted to make use of context-free grammars for automated generation of synchronized target models from source models [ASU86]. These grammars can either be used directly or can be used to identify inconsistencies in existing target abstractions. In this respect, by making use of context-free grammars, we aim to address the following issues:

1. Generality and Completeness — unlike graph transformation-based approaches such as [Fuj05], we intend on providing a generic mechanism for expressing model syntax

formally as grammars. More specifically, we define an algorithm for mapping domain models to domain model grammars, and provide automated creation of grammar productions in contrast to graph transformation rules that are in general manually created.

2. Domain-Specific Model Generation and Model Synchronization — by using domain contexts as the basis for the encompassing grammars, we aim to interpret each model in terms of domain-specific concepts therefore allowing creation of more precise dependency relations and in turn enabling automated generation of desired models and domain-specific model synchronization.

3. Semantic Consistency and Relations – given that each of the grammar productions is associated with a corresponding semantic head, which provides information with respect to constraints and conditions that dictate whether a rule is applicable or not, we are providing a method for encoding such semantic head information in the form of metadata semantic properties that can be used to define the model synchronization context. In this respect, establishing whether a synchronization rule is applicable in a given context is performed not only by matching the left-hand side of the rule, but also, by evaluating whether the semantic head metadata and constraints, hold in that context.

4. Extensible and Adaptable Formalism — by choosing attributed context-free grammars as the conceptual basis, the proposed framework can be extended with additional properties encoded as attributes for the chosen production rules. Moreover, by selecting recognized MOF and UML metamodels as the base, we also address adaptability in existing tools such as the Eclipse environment [Fou10].

3.3 Domain Models as Grammars

The conceptual view of MOF-compliant models that is taken in this thesis is that of sentences generated through the corresponding grammar. In this respect, we interpret domain models as context-free grammars. Using the graph metamodel for synchronization (GMS), which we have presented in [IK04b] and illustrated in Figure 3.1, we view domain models as collections of attributed nodes (domain types) and attributed directed edges (ordered domain relations), which represent types for instantiated concrete models. The domain model elements are then viewed as nonterminals, and the concrete model elements are viewed as terminals in the domain model grammar (DMG).

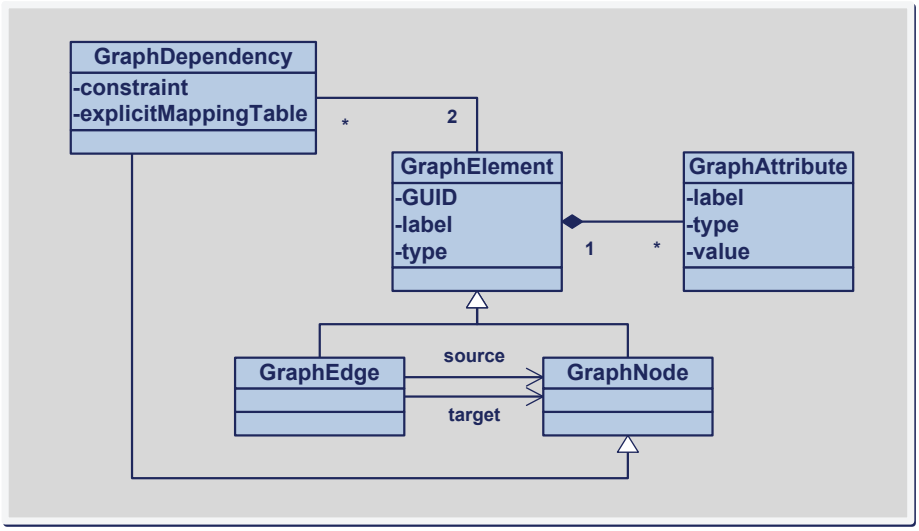


Figure 3.1: Graph Metamodel for Synchronization (GMS)

As we stated above, domain models can be represented as context-free grammars that allow for the generation of concrete model instances conforming to the corresponding domain model, that is, can be viewed as “sentences” belonging to the language generated by

the corresponding grammar. However, in order to formally define domain models as context-free grammars, we first have to formally define domain model elements. In this respect, we consider that domain models are composed of domain model elements, which are defined as follows.

As stated in Chapter 2, we adopt Metayer's view [Met96] of models as n-ary relations, but we propose to denote a MOF-compliant domain model as a collection of tuples of types, relations (associations between types), connectors (edges of association relations), and attributes.

Domain Model Elements: A domain model is defined as a tuple (DT, DR, DC, DA, T_{Names}, R_{Names}, C_{Names}, A_{Names}, Values).

- Domain Types DT := { (<enum>, t_i, {a_j}, {r_k}) |
  - <enum> is the enumeration of tuples as an ordered sequence of types,
  - type name $t_i \in T_{Names}$,
  - type attributes $a_j \in$ DA, and
  - type relations $r_k \in$ DR}.
- Domain Relations DR := {(<enum>, r_i, {a_j}, {$a_k{}^l$}, {c_m}) |
  - <enum> is the enumeration of tuples as an ordered sequence of relations,
  - relation name $r_i \in R_{Names}$,
  - relation attributes $a_j \in$ DA,
  - relation attributes $a_k{}^l$ for the relation type $t_l \in$ DT, and
  - relation connectors $c_m \in$ DC}.

- Domain Connectors DC := { (<enum>, c_i, {a_j}, {t_k}) |
  - <enum> is the enumeration of tuples as an ordered sequence of connectors,
  - connector name $c_i \in C_{Names}$,
  - connector attributes $a_j \in$ DA, and
  - domain type names $t_k \in T_{Names}$}.
- Domain Attributes DA := {(<enum>, a_i, {v_j}) |
  - <enum> is the enumeration of tuples as an ordered sequence of attributes,
  - attribute name $a_i \in A_{Names}$, and
  - attribute values $v_j \in$ Values with v_1 as the initial value if defined }.
- Unique type names T_{Names},
- Relation names R_{Names},
- Connector names C_{Names},
- Attribute names A_{Names}, and
- Values as an alphabet of domain values.

Object Model Elements: An object model that complies with a specified domain model is defined as a tuple (DO, DR, DC, DA, O_{Names}, T_{Names}, R_{Names}, C_{Names}, A_{Names}, Values), where

- Domain Objects DO := { (<enum>, o_i:t_i, {a_j}, {r_k}) |
  - <enum> is the enumeration of tuples as an ordered sequence of objects,

- object name $o_i \in O_{Names}$,
- type name $t_i \in T_{Names}$,
- type attributes $a_j \in$ DA, and
- type relations $r_k \in$ DR}.

- Unique object names O_{Names}.

The other elements, DR, DC, DA, T_{Names}, R_{Names}, C_{Names}, A_{Names}, and Values, are defined in the same manner as discussed above for the domain model elements definition.

To represent the elements of a given domain model using this definition, a typical approach would be to:

1. Derive the elements of T_{Names}, R_{Names}, C_{Names}, and A_{Names} based on the UML metamodel and their usage context;
2. Create in the following order tuples of attributes DA with consideration for unconstrained attributes, tuples of distinct connectors DC, tuples of distinct relations DR, and tuples of distinct types DT; and
3. Eliminate redundant tuples.

To illustrate the DM formalization, we make use of the domain model example from Figure 3.2 and formally define it according to the domain model elements definition.

DM = (DT, DR, DC, DA, T_{Names}, R_{Names}, C_{Names}, A_{Names}, Values),

- T_{Names} = {T1, T2, T3, T4, T5},

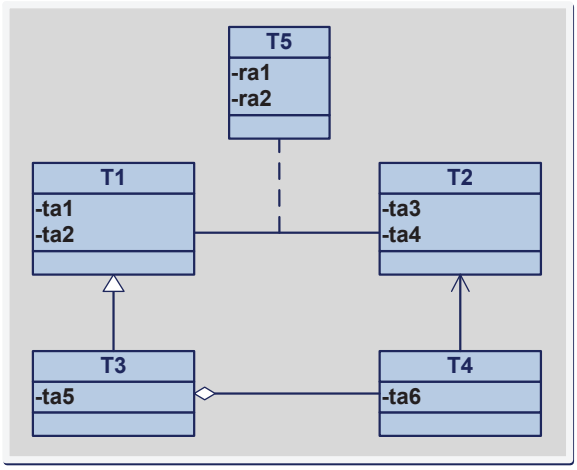


Figure 3.2: Domain Model Example

- R_{Names}= {Association, Generalization},
- C_{Names} = {AssociationEnd, GeneralizationEnd},
- A_{Names} = {Name, ta1, ta2, ta3, ta4, ta5, ta6, ra1, ra2, Aggregation, IsNavigable, Visibility, Multiplicity, Role, Constraint}, and
- Values = an alphabet of domain values.

DT = {(<t_1>, T1, {a_1, a_2, a_3}, {r_1, r_4}), (<t_2>, T2, {a_1, a_4, a_5}, {r_1, r_2}), (<t_3>, T3, {a_1, a_6}, {r_2, r_3}), (<t_4>, T4, {a_1, a_7}, {r_3, r_4}), (<t_5>, T5, {a_1, a_8, a_9}, $\emptyset$)}

DR = {(<r_1>, Association, {a_1}, {a_8, a_9}, {c_1, c_2}), (<r_2>, Association, {a_1}, $\emptyset$, {c_3, c_4}), (<r_3>, Association, {a_1}, $\emptyset$, {c_5, c_6}), (<r_4>, Generalization, {a_1}, $\emptyset$, {c_7, c_8})}

DC = {(<c_1>, AssociationEnd, {a_1, a_{10}, a_{11}, a_{12}, a_{13}}, {T1}), (<c_2>, AssociationEnd, {a_1, a_{10}, a_{11}, a_{14}, a_{13}}, {T2}), (<c_3>, AssociationEnd, {a_1, a_{10}, a_{15}, a_{16}, a_{13}}, {T2}), (<c_4>, AssociationEnd, { a_1, a_{10}, a_{11}, a_{16}, a_{13}}, {T3}), (<c_5>, AssociationEnd, {a_1, a_{10}, a_{11},

a_{16}, a_{13}}, {T1}), (<c_6>, AssociationEnd, {a_1, a_{17}, a_{11}, a_{16}, a_{13}}, {T3}), (<c_7>, GeneralizationEnd, {a_1, a_{18}, a_{19}}, {T3}), (<c_8>, GeneralizationEnd, {a_1, a_{20}, a_{19}}, {T1})}

DA = {(<a_1>, Name, Values), (<a_2>, ta1, Values), (<a_3>, ta2, Values), (<a_4>, ta3, Values), (<a_5>, ta4, Values), (<a_6>, ta5, Values), (<a_7>, ta6, Values), (<a_8>, ra1, Values), (<a_9>, ra2, Values), (<a_{10}>, Aggregation, none), (<a_{11}>, IsNavigable, false), (<a_{12}>, Multiplicity, 1), (<a_{13}>, Visibility, public), (<a_{14}>, Multiplicity, *) (<a_{15}>, IsNavigable, true), (<a_{16}>, Multiplicity, none), (<a_{17}>, Aggregation, shared), (<a_{18}>, Role, child), (<a_{19}>, Constraint, Values), (<a_{20}>, Role, parent)}

As demonstrated through the previous example, the defined formalization is based on cross-indexed tuples, where individual elements are uniquely identified through combinations of tuple elements. For example, relations are uniquely identified through the combination of the relation name, relation attributes tuples, relation attribute tuples for the relation type, and relation connector tuples. Constrained attributes are identified through the combination of the attribute name and defined value while unconstrained attributes, which are defined at the concrete model level, are identified through the combination of the attribute name and *Values* set. Therefore, it follows that the defined formalism is a nonambiguous representation of MOF-compliant domain models.

We have chosen this representation of domain models to allow for easier storage in a relational or object-relational database, and querying using database-manipulation language such as SQL. This representation is also comparable to eCore, but it is a more abstract representation than eCore, and it can be processed outside of EMF.

The proposed formalism can also be used to represent model instances. As shown in Figure 3.3, object instance o3 of type T3 was created and associated with object instance

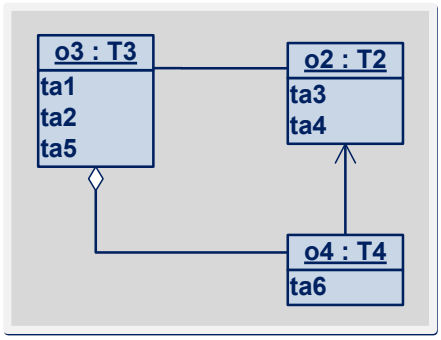


Figure 3.3: Object Model Example

o2 of type T2. Furthermore, object o4 of type T4 was created and associated with the object o2 through directed association, and object o3 through aggregation association.

OM = (DO, DR, DC, DA, O_{Names}, T_{Names}, R_{Names}, C_{Names}, A_{Names}, Values),

- O_{Names} = {o2, o3, o4},
- R_{Names}= {Association, Generalization},
- C_{Names} = {AssociationEnd, GeneralizationEnd},
- A_{Names} = {Name, ta1, ta2, ta3, ta4, ta5, ta6, ra1, ra2, Aggregation, IsNavigable, Visibility, Multiplicity, Role, Constraint}, and
- Values = an alphabet of domain values.

DO = {(<o_1>, o3:T3, {a_1, a_2, a_3}, {r_1, r_4}), (<o_2>, o2:T2, {a_1, a_4, a_5}, {r_1, r_2}), (<o_3>, o4:T4, {a_1, a_6}, {r_2, r_3})}

DR = {(<r_1>, Association, {a_1}, {a_8, a_9}, {c_1, c_2}), (<r_2>, Association, {a_1}, $\emptyset$, {c_3, c_4}), (<r_3>, Association, {a_1}, $\emptyset$, {c_5, c_6}), (<r_4>, Generalization, {a_1}, $\emptyset$, {c_7, c_8})}

DC = {(<c_1>, AssociationEnd, {a_1, a_{10}, a_{11}, a_{12}, a_{13}}, {T1}), (<c_2>, AssociationEnd, {a_1, a_{10}, a_{11}, a_{14}, a_{13}}, {T2}), (<c_3>, AssociationEnd, {a_1, a_{10}, a_{15}, a_{16}, a_{13}}, {T2}), (<c_4>, AssociationEnd, { a_1, a_{10}, a_{11}, a_{16}, a_{13}}, {T3}), (<c_5>, AssociationEnd, {a_1, a_{10}, a_{11}, a_{16}, a_{13}}, {T1}), (<c_6>, AssociationEnd, {a_1, a_{17}, a_{11}, a_{16}, a_{13}}, {T3}), (<c_7>, GeneralizationEnd, {a_1, a_{18}, a_{19}}, {T3}), (<c_8>, GeneralizationEnd, {a_1, a_{20}, a_{19}}, {T1})}

DA = {(<a_1>, Name, Values), (<a_2>, ta1, Values), (<a_3>, ta2, Values), (<a_4>, ta3, Values), (<a_5>, ta4, Values), (<a_6>, ta5, Values), (<a_7>, ta6, Values), (<a_8>, ra1, Values), (<a_9>, ra2, Values), (<a_{10}>, Aggregation, none), (<a_{11}>, IsNavigable, false), (<a_{12}>, Multiplicity, 1), (<a_{13}>, Visibility, public), (<a_{14}>, Multiplicity, *) (<a_{15}>, IsNavigable, true), (<a_{16}>, Multiplicity, none), (<a_{17}>, Aggregation, shared), (<a_{18}>, Role, child), (<a_{19}>, Constraint, Values), (<a_{20}>, Role, parent)}

Based on this hierarchical definition of a DM, it is now possible to more directly represent DM types and relations through a context-free grammar that uses grammar productions to represent each of the domain model elements. In the following section, we discuss how we represent a domain model and its elements as a context-free grammar.

3.4 Creating Domain-Model Grammars

In this section, we address the process of creating a domain-model grammar by proposing the following DM2DMG algorithm. We also prove selected properties of domain-model grammars that are necessary for their usage in the context of domain-specific model synchronization.

Domain Model Grammar: A domain model grammar (DMG) for a domain model DM := (DT, DR, DC, DA, T_{Names}, R_{Names}, C_{Names}, A_{Names}, Values) is a tuple (NT,

T, P, AX), where a set of nonterminals NT := $NT_T \cup NT_R \cup NT_C \cup NT_A$. NT_T = {F(T_{Names}) | F [1] is a mapping of strings to nonterminals to generate enumeration of nonterminals for type names}, NT_R = {F(R_{Names})}, NT_C = {F(C_{Names})}, and NT_A = {F(A_{Names})}. A set of terminals T := {t | t ∈ Values}, a finite set of production rules P := {(LHS, RHS) | where LHS ∈ NT, RHS ∈ (NT ∪ T)*} inferred from DT, DR, DC, and DA, and AX is the axiom that represents the origin for the derivation.

3.4.1 An Algorithm for Representing Domain Models as Domain Model Grammars

Algorithm DM2DMG Representing DM as DMG

Input:

1. Domain Model DM

Output:

1. Domain Model Grammar DMG

Steps:

[1] F is a lexicographical "identity" mapping F:T→NT, that keeps the same lexicographical notation for all the elements of the original field to the mapped elements of the target field. For instance, F(a:T) = *a*:NT, where a in the origin is considered to be a type while in the target is considered to be a nonterminal.

Step 1. Let NT_T = {F(T_{Names}) | F is a mapping of strings to nonterminals} be enumerated set of nonterminals for type names, NT_R = {F(R_{Names})} be enumerated set of nonterminals for relation names, NT_C = {F(C_{Names})} be enumerated set of nonterminals for connector names, NT_A = {F(A_{Names})} be enumerated set of nonterminals for attribute names.

Step 2. Let NT = $NT_T \cup NT_R \cup NT_C \cup NT_A$, let T be a set of terminals defined as elements from Values, and let AX be the starting symbol for derivation.

Step 3. Create the start rule by placing the start symbol AX on the LHS and iterate through the elements of $nt_i \in NT_T$ and $nr_j \in NT_R$ to derive the RHS by adding nt_i and AX nt_i and nr_j and AX nr_j as choices of the start rule (*e.g.*, AX $\rightarrow$ nt_1 | nt_1 AX | nt_2 | nt_2 AX ... | nr_1 | nr_1 AX | nr_2 | nr_2 AX ...).

Step 4. Create type name production rules P_T by iterating through the elements of NT_T:

Step 4.1 For each $nt_i \in NT_T$, with the corresponding domain-model tuple (<enum>, t_i, {a_j}, {r_k}) $\in$ DT where nt_i = F(t_i), create a production rule $p_i \in P_T$ with nt_i on the LHS and elements from {a_j} as a string on the RHS (*e.g.*, $nt_i \rightarrow a_1 \ldots a_n$).

Step 5. Create relation name production rules P_R by iterating through the elements of NT_R:

Step 5.1 For each $nr_i \in NT_R$, with the corresponding domain-model tuple (<enum>, r_i, {a_j}, {$a_k{}^l$}, {c_m}) $\in$ DR where nr_i = F(r_i), create a production rule $p_i \in P_R$ with nr_i on the LHS and attributes from {a_j} as the starting string on the RHS followed by the attributes from {$a_k{}^l$}, attributes of the type relation t_l, and connector names c_m (*e.g.*, $nr_i \rightarrow a_1 \ldots a_p\ t_l.a_1{}^l \ldots t_l.a_q{}^l\ c_1 \ldots c_r$).

Step 6. Create connector name rules P_C by iterating through the elements of NT_C:

Step 6.1 For each $nc_i \in NT_C$, with the corresponding domain-model tuple (<enum>, c_i, {a_j}, {t_k}) $\in$ DC where nc_i = F(c_i), create a production rule $p_i \in P_C$ with nc_i on the LHS and choices on the RHS for each t $\in$ {t_k} where each choice is a string of attributes from {a_j} followed by the type name nonterminal, nt_T = F(t) (*e.g.*, $nc_i \rightarrow a_1 \ldots a_p\ nt_{T1} \mid \ldots \mid a_1 \ldots a_p\ nt_{Tn}$).

Step 7. Create attribute name rules P_A by iterating through the elements of NT_A:

Step 7.1 For each $na_i \in NT_A$, with the corresponding domain-model tuple (<enum>, a_i, {v_j}) $\in$ DA where na_i = F(a_i), create a production rule $p_i \in P_A$ with na_i on the LHS and lexicographically-sorted attribute values from {v_j} as choices on the RHS (*e.g.*, $na_i \rightarrow v_1 \mid v_2 \ldots$).

Step 8. Let P = {ordered set of production rules from P_T, P_R, P_C, P_A}, that is, the rules appear always in the same order by first listing the rules P_T then P_R, P_C, P_A. Let the resulting grammar DMG = (NT, T, P, AX).

Step 9. Output DMG and terminate.

3.4.2 Representing Standardized Modeling Infrastructure

To enable representation of entire domain models, we first define the representation of atomic domain model relations as context-free grammar rules. The atomic or single relations considered are the association (association), shareable aggregation (aggregation) composite aggregation (composition) and generalization (inheritance). These relations are illustrated in Figure 3.4 as R_1, R_2, R_3, and R_4, respectively.

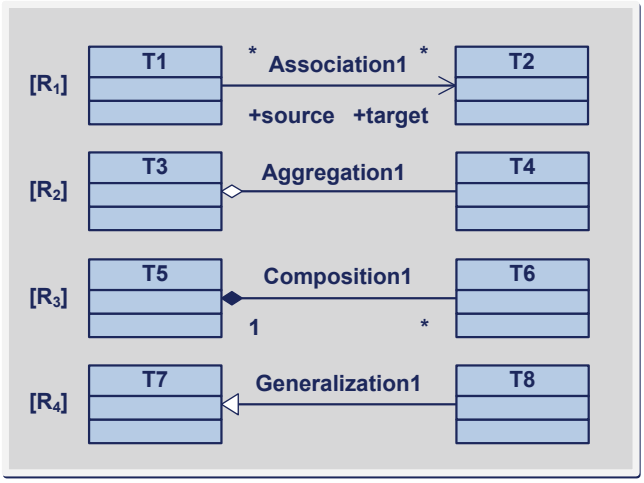


Figure 3.4: Domain Model Atomic Relations

Using a particular section of the UML metamodel as input [OMG10], a grammar required to represent the elements and features of R_i is as follows:

Step 1-2 :

NT_T := {Class}, NT_R := {Association, Generalization}, NT_C := {AssociationEnd, GeneralizationEnd}, NT_A := {Name, Role, Type, Constraint, isNavigable, isComposite, isDerived, MultiplicityElement, isOrdered, isUnique, lower, upper, LiteralInteger, range, star, Visibility}, NT := $NT_T \cup NT_R \cup NT_C \cup NT_A$, T := {alphabet of valid UML element names}, AX := M.

Step 3. :

p_1 : M $\rightarrow$ Class | Class M | Association | Association M | Generalization | Generalization M

Step 4. :

p_2 : Class $\rightarrow$ Name Constraint | Name:Type Constraint

Step 5. :

p_3 : Association $\rightarrow$ Name Constraint AssociationEnd AssociationEnd

p_4 : Generalization $\rightarrow$ Name Constraint GeneralizationEnd GeneralizationEnd

Step 6. :

p_5 : AssociationEnd $\rightarrow$ Role Constraint isNavigable isComposite isDerived MultiplicityElement Visibility Member:Class

p_6 : GeneralizationEnd $\rightarrow$ Constraint General:Class | Constraint Specific:Class

Step 7. :

p_7 : Name $\rightarrow$ T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | Association1 | Aggregation1 | Composition1 | Generalization1 | an element from the alphabet of other element names for the corresponding Namespace

p_8 : Role $\rightarrow$ an element from the alphabet of element roles for the corresponding Namespace

p_9 : Type $\rightarrow$ an element from the alphabet of element types for the corresponding Namespace

p_{10} : Constraint $\rightarrow$ an element from the alphabet of constraints for the corresponding Namespace

p_{11} : isNavigable $\rightarrow$ true | false

p_{12} : isComposite $\rightarrow$ true | false

p_{13} : isDerived $\rightarrow$ true | false

p_{14} : MultiplicityElement $\rightarrow$ isOrdered isUnique lower upper

p_{15} : isOrdered $\rightarrow$ true | false

p_{16} : isUnique $\rightarrow$ true | false

p_{17} : lower $\rightarrow$ none | LiteralInteger | range | star

p_{18} : upper {upper >= lower} $\rightarrow$ none | LiteralInteger | range | star

p_{19} : LiteralInteger $\rightarrow$ a constant Integer value

p_{20} : range $\rightarrow$ LiteralInteger..LiteralInteger

p_{21} : star $\rightarrow$ *

p_{22} : Visibility $\rightarrow$ public | protected | private | package

Step 8 :

P := {p_1, p_2, . . . p_{22}} and DMG := {NT, T, P, AX}.

Step 9 :

Output DMG and terminate.

We illustrate the validity of the grammar through derivation of individual relations as follows:

Association An association defines a semantic relationship between classes, and an instance of an association is a set of tuples relating instances of the classes. R_1 demonstrates an association between nodes T_1 and T_2. Based on the derived grammar, a derivation tree R_1 is illustrated in Figure 3.5.

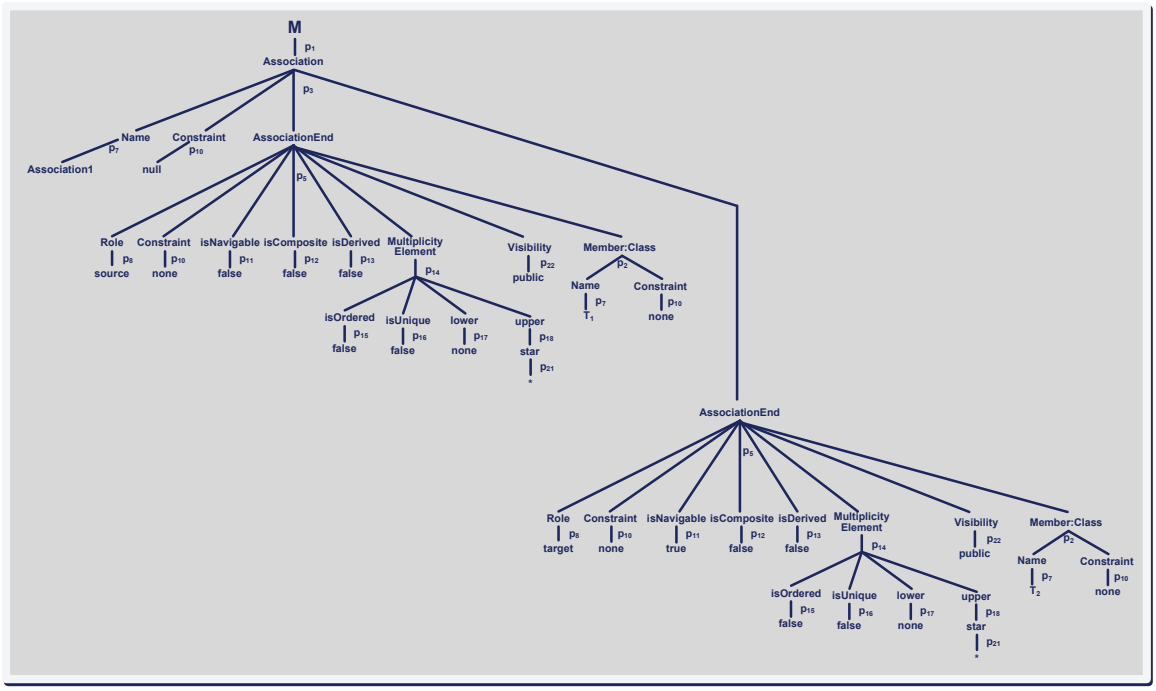


Figure 3.5: A Derivation Tree for R_1

Aggregation An aggregation is a specialized form of association that represents a whole/part relationship. The end symbol as shown in R_2 of Figure 3.4 is attached to the whole element (T_1) and the other association ends (T_2) are attached to its parts. A derivation tree for R_2 is analog to the one shown for R_1.

Composition A composition is a specialized form of aggregation, where the relation between the whole, indicated by the end symbol as shown in R_3 of Figure 3.4, and its parts is exclusive. Each part can only belong to one whole and the parts cannot exist without the whole. A derivation tree for R_3 is analog to the one shown for R_1.

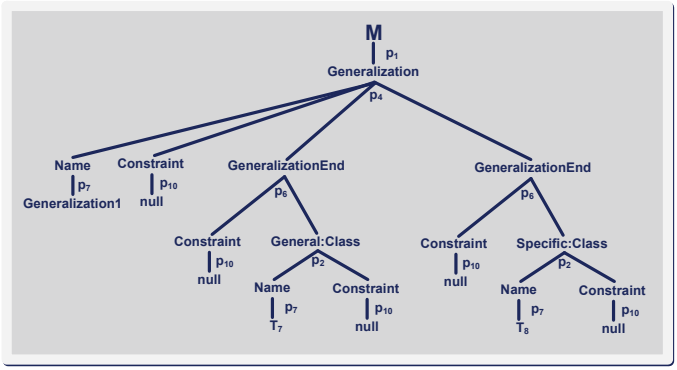


Figure 3.6: A Derivation Tree for R_4

Generalization A generalization is a taxonomic relationship between a more general, indicated by the end symbol as shown in R_4 of Figure 3.4, and a more specific element. The more specific element is consistent with the more general element by having all of its properties, members, and relationships, but it may contain additional attributes and features. A derivation tree for R_4 is illustrated in Figure 3.6.

3.4.3 Illustrative Example

To illustrate the DMG creation algorithm, we use the following application scenario, derived from work by Miller [ABMM07]. In this scenario, there are two related domain models as shown in Figure 3.7. In the source domain model, each instance of “Person” type is related through the “writes” relation to zero or more “Book” instances, and each “Book” instance is related to one or more “Person” instances. Each “Book” instance is also related to zero or more “Library” instances through the “heldAt” relation, and each “Library” instance is related to one or more “Book” instances. In the target domain model, each instance of “Author” type is related through the “hasBookAt” relation to zero or more “Library” instances, and each “Library” instance is related to one or more “Author” instances.

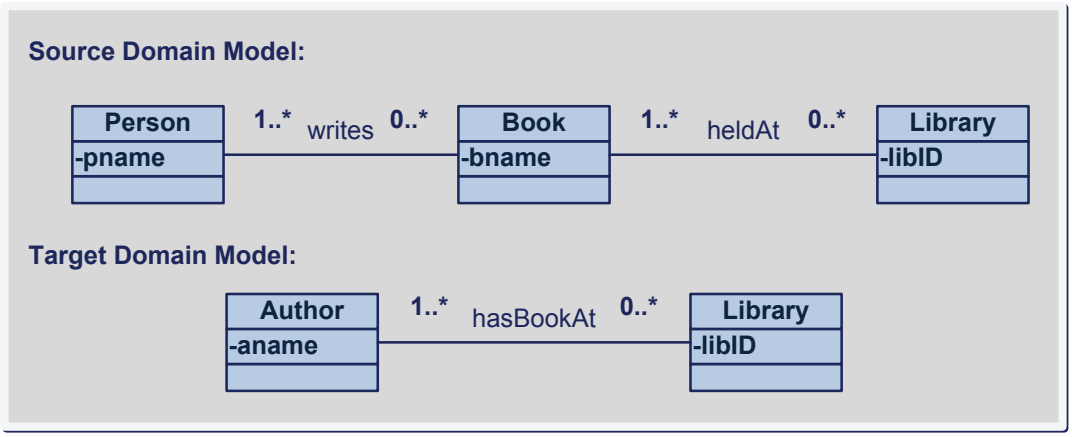


Figure 3.7: Illustrative Example Overview

Using the DM to DMG algorithm described above, the following domain model grammars for the source and target domain models are created.

Source Domain Model Grammar

Step 1-2 :

NT$_T$:= {Person, Book, Library}, NT$_R$:= {writes, heldAt}, NT$_C$:= {Connector$_1$, Connector$_2$}, NT$_A$:= {pname, bname, libID, star}, NT := NT$_T$ $\cup$ NT$_R$ $\cup$ NT$_C$ $\cup$ NT$_A$, T := {alphabet of valid attribute values}, AX := M.

Step 3. :

p$_1$: M $\rightarrow$ Person | Person M | Book | Book M | Library | Library M | writes | writes M | heldAt | heldAt M

Step 4. :

p$_2$: Person $\rightarrow$ pname

p$_3$: Book $\rightarrow$ bname

p$_4$: Library $\rightarrow$ libID

Step 5. :

p$_5$: writes $\rightarrow$ Connector$_1$ Connector$_2$

p$_6$: heldAt $\rightarrow$ Connector$_1$ Connector$_2$

Step 6. :

p$_7$: Connector$_1$ $\rightarrow$ 1..star

p$_8$: Connector$_2$ $\rightarrow$ 0..star

Step 7. :

p$_9$: pname $\rightarrow$ an element from the alphabet of person names for the corresponding Namespace

p_{10} : bname $\rightarrow$ an element from the alphabet of book names for the corresponding Namespace

p_{11} : libID $\rightarrow$ an element from the alphabet of library IDs for the corresponding Namespace

p_{12} : star $\rightarrow$ *

Step 8 :

P := {p_1, p_2, . . . p_{12}} and DMG := {NT, T, P, AX}.

Step 9 :

Output DMG and terminate.

Target Domain Model Grammar

Step 1-2 :

NT_T := {Author, Library}, NT_R := {hasBookAt}, NT_C := {$Connector_1$, $Connector_2$}, NT_A := {aname, libID, star}, NT := $NT_T \cup NT_R \cup NT_C \cup NT_A$, T := {alphabet of valid attribute values}, AX := M.

Step 3. :

p_1 : M $\rightarrow$ Author | Author M | Library | Library M | hasBookAt | hasBookAt M

Step 4. :

p_2 : Author $\rightarrow$ aname

p_3 : Library $\rightarrow$ libID

Step 5. :

p_4 : hasBookAt $\rightarrow$ $Connector_1$ $Connector_2$

Step 6. :

p_5 : Connector$_1$ $\rightarrow$ 1..star

p_6 : Connector$_2$ $\rightarrow$ 0..star

Step 7. :

p_7 : aname $\rightarrow$ an element from the alphabet of author names for the corresponding Namespace

p_8 : libID $\rightarrow$ an element from the alphabet of library IDs for the corresponding Namespace

p_9 : star $\rightarrow$ *

Step 8 :

P := {p_1, p_2, . . . p_9} and DMG := {NT, T, P, AX}.

Step 9 :

Output DMG and terminate.

3.4.4 Domain Model Instances

The described DMG formalism is intended to provide a set of grammar production rules that can parse a specific domain model. To ensure that our notation is capable of parsing specific model instances, that is models which are compliant with the corresponding domain models, we introduce a specialized domain model grammar, $\overline{DMG}$, that is capable of parsing specific model instances.

Domain Model Grammar for Domain Model Instances $\overline{DMG}$: For a domain model DM := (DT, DR, DC, DA, T_{Names}, R_{Names}, C_{Names}, A_{Names}, Values) and a set of domain objects DO, $\overline{DMG}$ is a tuple (NT, T, P, AX), where a set of nonterminals NT := O_{Names}:$T_{Names} \cup R_{Names} \cup C_{Names} \cup A_{Names}$, where O_{Names}

$\in$ DO, a set of terminals T := {t | t $\in$ Values}, a finite set of production rules P := {(LHS, RHS) | where LHS $\in$ NT, RHS $\in$ (NT $\cup$ T)*} inferred from DO, DT, DR, DC, and DA, and AX is the axiom that represents the origin for the derivation.

To derive a $\overline{DMG}$, one can make use of the algorithm from Section 3.4.1, with the exception that for Step 4, we create object production rules P_O by iterating through the elements of NT_O. For each $nt_i \in NT_O$, with the corresponding tuple (<enum>, $o_i : t_j$, {a_k}, {r_l}) $\in$ DO where $nt_i = F(o_i : t_j)$, create a production rule $p_i \in P_O$ with nt_i on the LHS and elements from {a_j} as a string on the RHS (*e.g.*, $nt_i \rightarrow a_1 \ldots a_n$).

To illustrate the change, we use the following generic example.

Step 1-2 :

NT_O := {o:Class}, NT_R := {Association, Generalization}, NT_C := {AssociationEnd, GeneralizationEnd}, NT_A := {Name, Role, Type, Constraint, isNavigable, isComposite, isDerived, MultiplicityElement, isOrdered, isUnique, lower, upper, LiteralInteger, range, star, Visibility}, NT := $NT_O \cup NT_R \cup NT_C \cup NT_A$, T := {alphabet of valid UML element names}, AX := M.

Step 3. :

p_1 : M $\rightarrow$ o:Class | o:Class M | Association | Association M | Generalization | Generalization M

Step 4. :

p_2 : o:Class $\rightarrow$ Name Constraint | Name:Type Constraint

Step 5. :

p_3 : Association $\rightarrow$ Name Constraint AssociationEnd AssociationEnd

p_4 : Generalization $\rightarrow$ Name Constraint GeneralizationEnd GeneralizationEnd

Step 6. :

p_5 : AssociationEnd $\rightarrow$ Role Constraint isNavigable isComposite isDerived MultiplicityElement Visibility Member:Class

p_6 : GeneralizationEnd $\rightarrow$ Constraint General:Class | Constraint Specific:Class

Step 7. :

p_7 : Name $\rightarrow$ T1 | T2 | Association1 | Aggregation1 | Composition1 | Generalization1 | o1:T1 | o2:T2 | o3:T2 | o4:T2 | an element from the alphabet of other element names for the corresponding Namespace

p_8 : Role $\rightarrow$ an element from the alphabet of element roles for the corresponding Namespace

p_9 : Type $\rightarrow$ an element from the alphabet of element types for the corresponding Namespace

p_{10} : Constraint $\rightarrow$ an element from the alphabet of constraints for the corresponding Namespace

p_{11} : isNavigable $\rightarrow$ true | false

p_{12} : isComposite $\rightarrow$ true | false

p_{13} : isDerived $\rightarrow$ true | false

p_{14} : MultiplicityElement $\rightarrow$ isOrdered isUnique lower upper

p_{15} : isOrdered $\rightarrow$ true | false

p_{16} : isUnique $\rightarrow$ true | false

p_{17} : lower $\rightarrow$ none | LiteralInteger | range | star

p_{18} : upper {upper >= lower} $\rightarrow$ none | LiteralInteger | range | star

p_{19} : LiteralInteger $\rightarrow$ a constant Integer value

p_{20} : range $\rightarrow$ LiteralInteger..LiteralInteger

p_{21} : star $\rightarrow$ *

p_{22} : Visibility $\rightarrow$ public | protected | private | package

Step 8 :

P := {p_1, p_2, . . . p_{22}} and DMG := {NT, T, P, AX}.

Step 9 :

Output DMG and terminate.

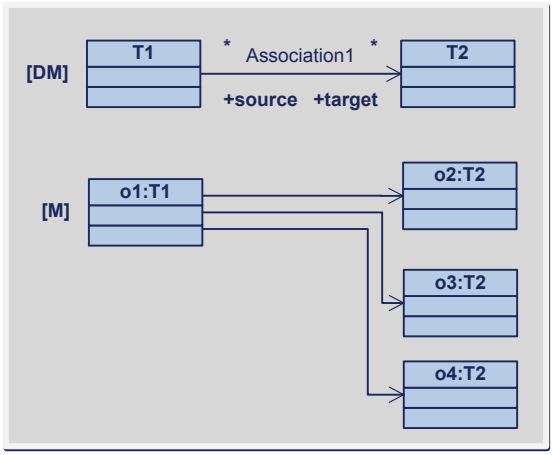


Figure 3.8: A Domain Model Instance Example

The Figure 3.8 illustrates a domain model instance, where one instance of type T1 is matched to three instances of type T2, following the many-to-many relation in the applicable domain model. The corresponding derivation tree for the same model is shown in the Figure 3.9.

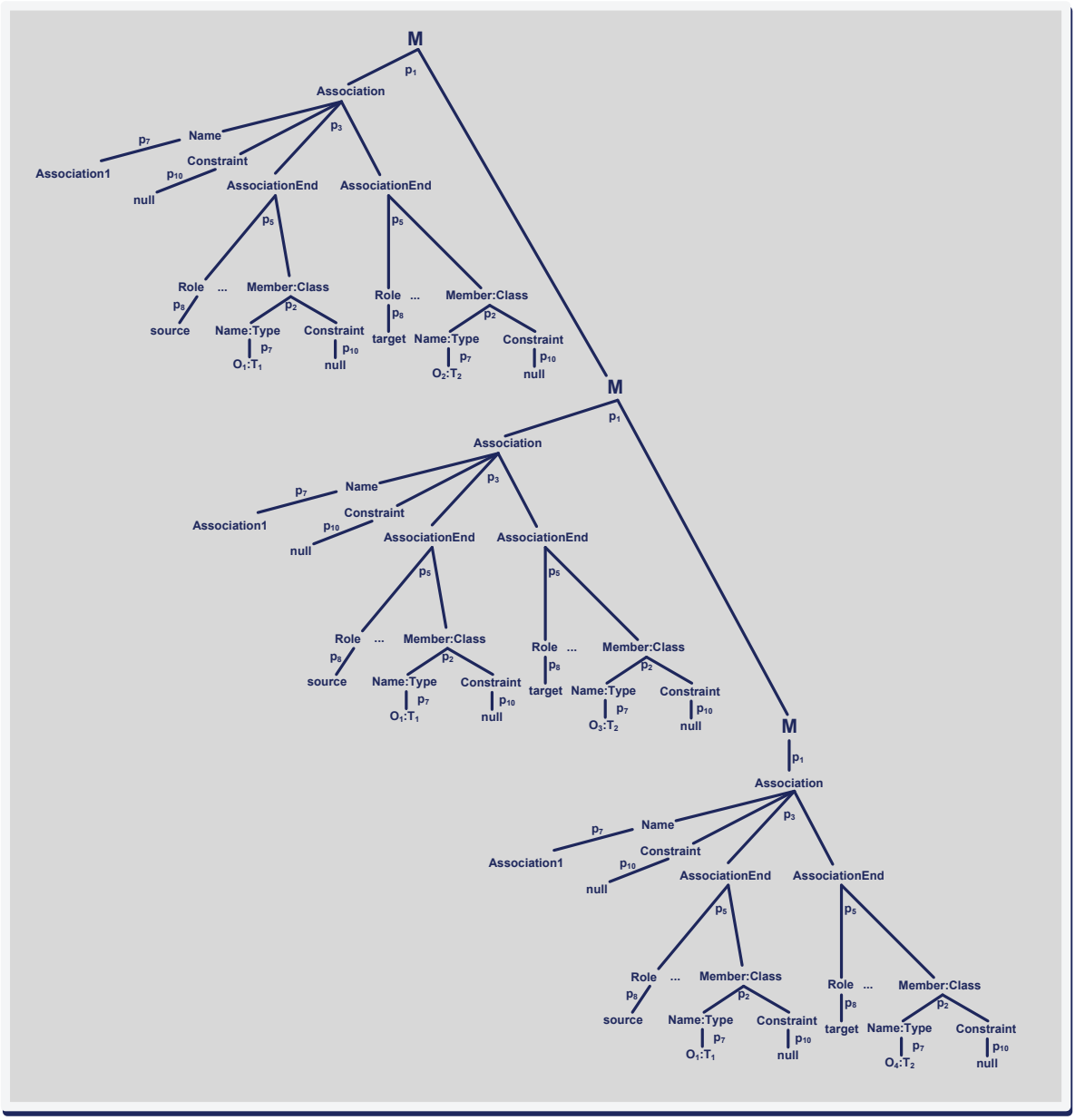


Figure 3.9: A Domain Model Instance Derivation Tree

3.4.5 DMG in the Context of Domain-Specific Model Synchronization

We evaluate the representation of domain models as domain-model grammars in two steps: first, by evaluating the DMG construction, and second, by evaluating the DMG properties with respect to "parsing" or "generation" of specific models. The criteria for evaluation are based on the requirements for enabling grammar-based model synchronization. For example, having a grammar representation that cannot be constructed in bounded time (*i.e.*, intractable grammar) or one that is not an adequate representation of a domain model (*i.e.*, grammar undergeneration or overgeneration) would be detrimental to the model synchronization process.

DMG Construction Evaluation

To evaluate the grammar construction algorithm, we evaluate the following qualities: (1) uniqueness, (2) determinism, (3) tractability, (4) incrementality, and (5) reverse incrementality. We also evaluate the following qualities with respect to the grammar parsing capabilities: (1) undergeneration, (2) overgeneration, (3) tractability, (4) completeness, and (5) soundness.

Uniqueness For each set of DM elements and an enumeration algorithm, the production rules are unique.

Proof: The proof is by construction, based on four cases: for DT domain type tuples, for DR domain relation tuples, for DC domain connector tuples, and for DA domain attribute tuples.

For domain type tuples, each domain type is enumerated and related attributes are lexicographically sorted. Hence, each domain type t_i from ($<$enum$>$, t_i, {a_j}, {r_k}) is uniquely mapped to a production rule $nt_i \rightarrow a_1 \ldots a_n$ in a ordered sequence based on the tuple enu-

meration, where nt_i = F(t_i) with F being a mapping of strings to nonterminals, and ($a_1 \ldots a_n$) $\in$ {a_j} being a lexicographical and unique ordering of attributes.

For domain relation tuples, each domain relation is enumerated, relation attributes and relation type attributes are lexicographically sorted, relation type attributes are indexed based on the relation type, and relation connectors follow their respective enumeration. Hence, each domain relation r_i from (<enum>, r_i, {a_j}, {$a_k{}^l$}, {c_m}) is uniquely mapped to a production rule $nt_i \rightarrow a_1 \ldots a_p\ t_l.a_1{}^l \ldots t_l.a_q{}^l\ c_1 \ldots c_r$, where nt_i = F(r_i), ($a_1 \ldots a_p$) $\in$ {a_j} being a lexicographical and unique ordering of attributes, ($t_l.a_1{}^l \ldots t_l.a_q{}^l$) $\in$ {$a_k{}^l$} also being a lexicographical and unique ordering of attributes, and ($c_1 \ldots c_r$) $\in$ {c_m} being an enumeration of related connectors.

For domain connector tuples, each domain connector is enumerated, connector attributes are lexicographically sorted, and connector types follow their respective enumeration. Therefore, each domain connector c_i from (<enum>, c_i, {a_j}, {t_k}) is uniquely mapped to a production rule $nt_i \rightarrow a_1 \ldots a_p\ nt_{T1} \mid \ldots \mid a_1 \ldots a_p\ nt_{Tn}$, where nt_i = F(c_i), ($a_1 \ldots a_p$) $\in$ {a_j} being a lexicographical and unique ordering of attributes, $nt_{T1} \ldots nt_{Tn}$ being an enumeration of related connector types.

Finally, for domain attribute tuples, each domain attribute is enumerated, and attribute values are lexicographically sorted. Therefore, each domain attribute a_i from (<enum>, a_i, {v_j}) is uniquely mapped to a production rule $nt_A \rightarrow v_1 \mid v_2 \ldots$, where nt_i = F(a_i), (v_1, $v_2 \ldots$) $\in$ {v_j} being a lexicographical and unique ordering of values.

Based on the four cases above, which comprise the base construction blocks of a DMG, it follows that for each set of DM elements, the resulting DMG production rules are unique (please see the enumeration process in the DM2DMG algorithm and the unique ordering of the rules).

Determinism For distinct domain models DM_1 and DM_2 and domain-model grammars DMG_1 and DMG_2 that are generated from DM_1 and DM_2 respectively, it holds that $DMG_1 \neq DMG_2 \Leftrightarrow DM_1 \neq DM_2$, iff DM_1 and DM_2 are allowed to differ in one of the core elements, such as different types, relations, or attributes.

Proof: *Case 1 ($DM_1 \neq DM_2 \Rightarrow DMG_1 \neq DMG_2$).* $DM_1 \neq DM_2$ means that DM_1 and DM_2 differ either on their types or on their attributes or on the relations among the different types. In this respect, any such difference will result in to generating different tuples for DM_1 and DM_2 for all the elements that DM_1 and DM_2 differ. If there are tuples from DM_1 and DM_2 that differ and because the rules that are generated are shown to be unique by the uniqueness property, this means that the resulting grammar will be different. Any other possibility of the grammars being equal would violate the uniqueness property.

Case 2 ($DMG_1 \neq DMG_2 \Rightarrow DM_1 \neq DM_2$). We use the proof by construction and consider four sets of tuples that comprise the domain model elements.

For domain type production rules P_T, if there is a difference in any of the elements of $nt_i \rightarrow a_1 \ldots a_n$, such as different types or different attributes, then the resulting DM tuples are also different since the domain types are enumerated and attributes are lexicographically sorted.

Similarly, for domain relation production rules P_R, if there is a difference in any of the elements of $nt_i \rightarrow a_1 \ldots a_p\ t_l.{a_1}^l \ldots t_l.{a_q}^l\ c_1 \ldots c_r$, such as different relations, different attributes, different type attributes, or different connectors, then the resulting DM tuples rules are different since the domain relations and domain connectors are enumerated, and all domain attributes are lexicographically sorted.

Also, for domain connectors production rules P_C, if there is a difference in any of the

elements of $nt_i \rightarrow a_1 \ldots a_p\ nt_{T1} \mid \ldots \mid a_1 \ldots a_p\ nt_{Tn}$, such as different connectors, different attributes, or different connector types, then the resulting DM tuples are different since domain connectors and domain types are enumerated and attributes are lexicographically sorted.

Finally, for domain attribute production rules P_A, we distinguish two cases. The first case is when attributes in the two domain models are allowed to have different values as choices, and the second case is when attributes in the two domain models are allowed to have only one value. In the first case, it is possible that the resulting DM tuples are the same even though the related DMG rules are different if the specific values from the alphabet of domain values do not apply to DMs in question. In this case, it may hold that even though $DMG_1 \neq DMG_2$, the corresponding domain models DM_1 and DM_2 be still equal ($DM_1 = DM_2$). For example, domain values of 0 or * for Multiplicity attribute could be left out as choices in P_A production rule. Domain models that do not have Multiplicity specified or that use other values such as 1 could still be parsed by the changed DMG rules without changing any of the DM element tuples.

In the second case, where the attributes in the two domain models are allowed to have only one value then we can say that when $DMG_1 \neq DMG_2 \Leftrightarrow DM_1 \neq DM_2$

Therefore, we identify two cases of domain models with respect to the determinism property: for domain models that differ in one of the core elements, such as different types, relations, connectors, or attributes, it holds that $DM_1 \neq DM_2 \Leftrightarrow DMG_1 \neq DMG_2$; and for domain models that differ only in domain values but have the same core elements, it holds that $DM_1 \neq DM_2 \Rightarrow DMG_1 \neq DMG_2$ only.

Tractability The creation of a domain-model grammar DMG from the given domain model DM is tractable.

Proof: Using the DM2DMG algorithm, the creation of a DMG from the given DM involves iterating through the tuples of domain types DT, domain relation DR, domain connectors DC, and domain attributes DA. The algorithmic complexity of DMG creation is then O(|DT| + |DR| + |DC| + |DA|). Consequently, as the DMG can be created in bounded time, the presented approach is tractable.

Incrementality For a domain model DM, a corresponding domain-model grammar DMG, and a subset DM′ of DM, there exists a domain-model grammar DMG′ that corresponds to DM′ such that DMG′ is a subset of DMG.

$$\forall DM' \subseteq DM \Rightarrow \exists\, DMG' \mid DMG' \subseteq DMG \tag{3.1}$$

Proof The proof is by construction. Based on the uniqueness property, each of the tuples t_i in DM′ would represent a unique production rule in DMG′. Since DM′ is a subset of DM, DM would contain all the tuples of DM′. That means, assuming the same enumeration process, the rules in DMG will have all the rules stemming from the tuples of DM′ plus all the rules that stem from the extra tuples of DM. Hence, DMG′ ⊆ DMG.

Reverse Incrementality For a domain-model grammar DMG and a corresponding domain model DM, there exists a subset DMG′ of DMG generated from a domain model DM′ such that DM′ is a subset of DM.

$$\exists DMG' \subseteq DMG \mid DM' \subseteq DM \tag{3.2}$$

Proof It is possible to select a collection of rules from DMG yielding DMG′ such that DMG′ is a subset of DMG, in a way that the rules from DMG′ will correspond to types, relations, connectors, and attributes from DM that form a subset DM′ of DM. Please note that not any

subset DMG′ of DMG can correspond to a valid DM′ as a subset of DM. The valid subsets DMG′ of DMG where the reverse incrementality property holds is when the rules are selected in a way that the generated model DM′ is a valid MOF model.

Undergeneration All concrete models M that can be instantiated from a domain model DM can also be parsed by the corresponding grammar $\overline{DMG}$.

$$\forall M :: DM \underset{gen}{\rightarrow} M \subseteq L(\overline{DMG}) \tag{3.3}$$

Proof: The proof is by contradiction. Let us assume that there exists a model M′ that can be instantiated from a domain model DM but cannot be parsed by the corresponding domain-model grammar DMG. This implies that M′ complies with all of the tuples from DM, but it violates one of the rules from $\overline{DMG}$. That means, M′ has a tuple in its representation that has not been used to create a rule in $\overline{DMG}$. However, since M′ is an instance of DM, all tuples of M′ have to have a rule in $\overline{DMG}$, which contradicts with the hypothesis.

Overgeneration All concrete models M that can be parsed by a domain-model grammar $\overline{DMG}$ can also be instantiated from a domain model DM.

$$\forall M \subseteq L(\overline{DMG}) \Rightarrow M :: DM \tag{3.4}$$

Proof: The proof is by contradiction. Let us assume that M can be parsed by $\overline{DMG}$, but M is not an instance of DM. That means there exists a type, relation, connector, or attribute, that exists in M and does not exist in DM. In this case, since $\overline{DMG}$ is generated from DM, it will contain rules from DM tuples that do not correspond to tuples representing M. In this case, it will not possible to use the $\overline{DMG}$ rules to fully parse M, which is a contradiction.

Tractability The parsing by a domain-model grammar $\overline{DMG}$ of all concrete models M that can be instantiated from a domain model DM is tractable.

Proof: Since DMG and $\overline{DMG}$ are defined as context-free grammars, the asymptotic time complexity for parsing sentences from both L(DMG) and L($\overline{DMG}$) is O(n^3) (*e.g.*, using the CYK algorithm [Mar97]), where n is the size of the parsed string that represents a concrete model instantiated from the DM. Each parsed string includes types followed by type attribute tuples, relations followed by relation attribute tuples, and connectors followed by connector attribute tuples. It follows that the asymptotic size complexity for sentences from L($\overline{DMG}$) is O(|DO| + |DR| + |DC| + |DA|). Hence, the $\overline{DMG}$ parsing of concrete models M instantiated from DM is tractable.

Completeness Any MOF-compliant DM can be represented as a DMG.

Proof: We consider that MOF-compliant domain models are composed of types, relations, connectors, attributes, and values. Based on the construction algorithm 3.4.1, each one of these elements yields a tuple which then is transformed into a rule.

Soundness A generated DMG is a sound representation of a MOF-compliant DM.

Proof: This property follows from the DMG construction algorithm and the tuple representation of DM. More specifically, for every element (type, relation, connector, and attribute) of DM, a corresponding rule is generated for DMG. That means, every element of DM or an instance of every element of DM can be parsed by a corresponding DMG rule. Parsing wise and representational-equivalence wise, DMG is a sufficient representation of DM.

3.5 Chapter Summary

This chapter presents the formal notation for the representation of domain models and model instances in the context of domain-specific model synchronization. As part of the notation, we have represented a domain model as a set of domain model elements, that is, enumerated and sorted tuples of domain types, relations, connectors, and attributes. We have then presented a method whereby domain model elements are represented as grammar productions of the corresponding domain-model grammar, and where models are considered as “sentences” generated by that grammar. We have then discussed how the generated context-free grammar can be used to parse model instances that conform to a specific domain model. Finally, we have evaluated our approach by reflecting on specific properties regarding domain grammar construction and domain grammar parsing capabilities.

Chapter 4

Coarse-Grained Model Synchronization: Establishing and Representing Model Dependencies

Every word or concept, clear as it may seem to be, has only a limited range of applicability.

— Werner Heisenberg

4.1 Chapter Overview

In model-driven development, software evolves iteratively and incrementally through modifications of artifact models, specified at different abstraction levels. These modifications are performed independently, but the objects to which they are applied to, are in most cases

mutually dependent. To avoid creation of evolution-induced inconsistencies and drift, among related software models, the effects of each model transformation need to be systematically and proactively identified, recorded, and propagated to other affected models. For large software-intensive systems there may be a great number of model dependencies. Manual extraction of a large number of dependencies would not be feasible, and some degree of tool-supported automation would be required.

In this chapter, we address the problem of automating the extraction of model dependencies among related software models and their elements. We utilize the extracted dependency relations in the context of coarse-grained model synchronization.

4.2 Establishing Model Dependencies using Formal Concept Analysis

The approach for establishing model dependencies using formal concept analysis was introduced in our work presented in [IK05] and [IK06].

In our approach, each software model besides being denoted as a domain model grammar (DMG), it is also viewed as a *context* in terms of objects and attributes. Formal Concept Analysis (FCA) can be used as a formal method for identifying groups of objects that share common attributes and are hence considered dependent. Given that the models are possibly from different domains, we also introduce for that case, the notion of *attribute association rules* for creating mappings among heterogeneous attributes. These rules can be defined both at the domain model and, at the concrete model level. At the domain model level, the rules represent mappings of types, attributes and associations between two models, while at the model level, they represent mappings of object names, attribute values and annotations.

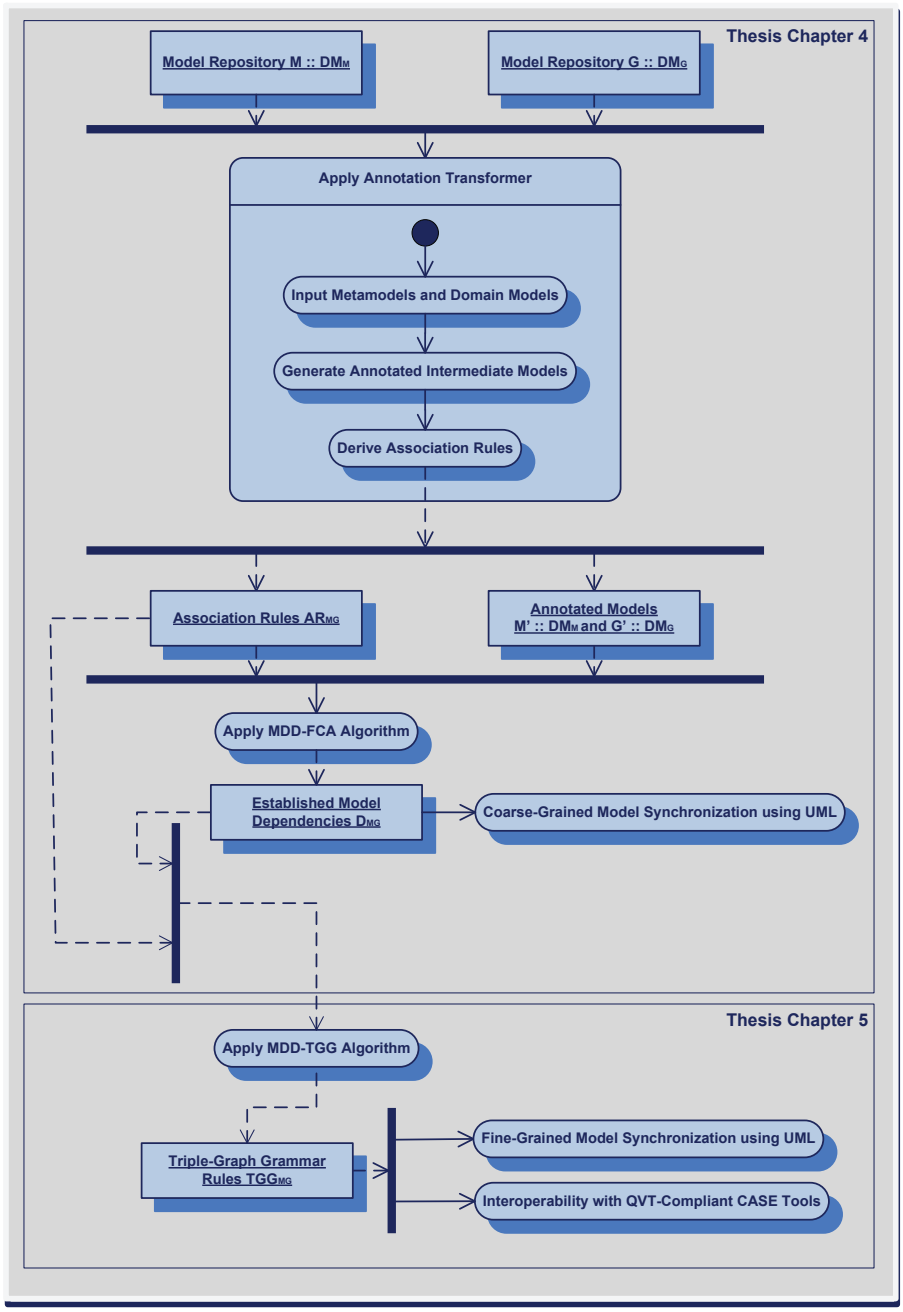


Figure 4.1: Establishing Model Dependencies using FCA

In the proposed approach, we consider that one or more models conform to a specific domain model that denotes relations and properties between model elements. Since models may pertain to software artifacts at different levels of abstraction, they may have a significant semantic gap when compared to each other. We propose that domain model information can be used to generate a sequence of transformations that can alter these models to a level where they can be compared and their dependencies can be extracted. These domain model transformations can take the form of association rules or type mappings from one domain model to another.

Schema mapping and the identification of association rules is out of the scope of the thesis. This topic has been investigated extensively in the area of databases, such as the work reported in Miller [ABMM07]. For this work, we consider that models are either from the same domain or the association rules have already been defined. Nevertheless, for the purpose of completeness, we provide some heuristic methods for defining association rules as discussed in the Section 4.3.1.

Figure 4.1 illustrates this approach, where two repositories of models, M (source) and G (target), contain models instantiated from domain models DM_M and DM_G respectively. The corresponding association rules, AR_{MG}, between the domains are established based on compatible domain types and relations from models M and G. Using FCA, dependencies D_{MG} between models and model elements from M and G are identified.

To validate the approach and demonstrate its applicability in a practical scenario, we make use of a case study where the goal is to synchronize business process models (BPM) with the enacting Java and Enterprise Java Beans (EJB) [Ora08] source code. The business models are represented as business workflows that include processes, tasks, decisions, data, *etc*. Examples of such diverse domain models that represent EJB source code and business workflows are depicted in Figure 4.2 and 4.3, respectively. Due to signifi-

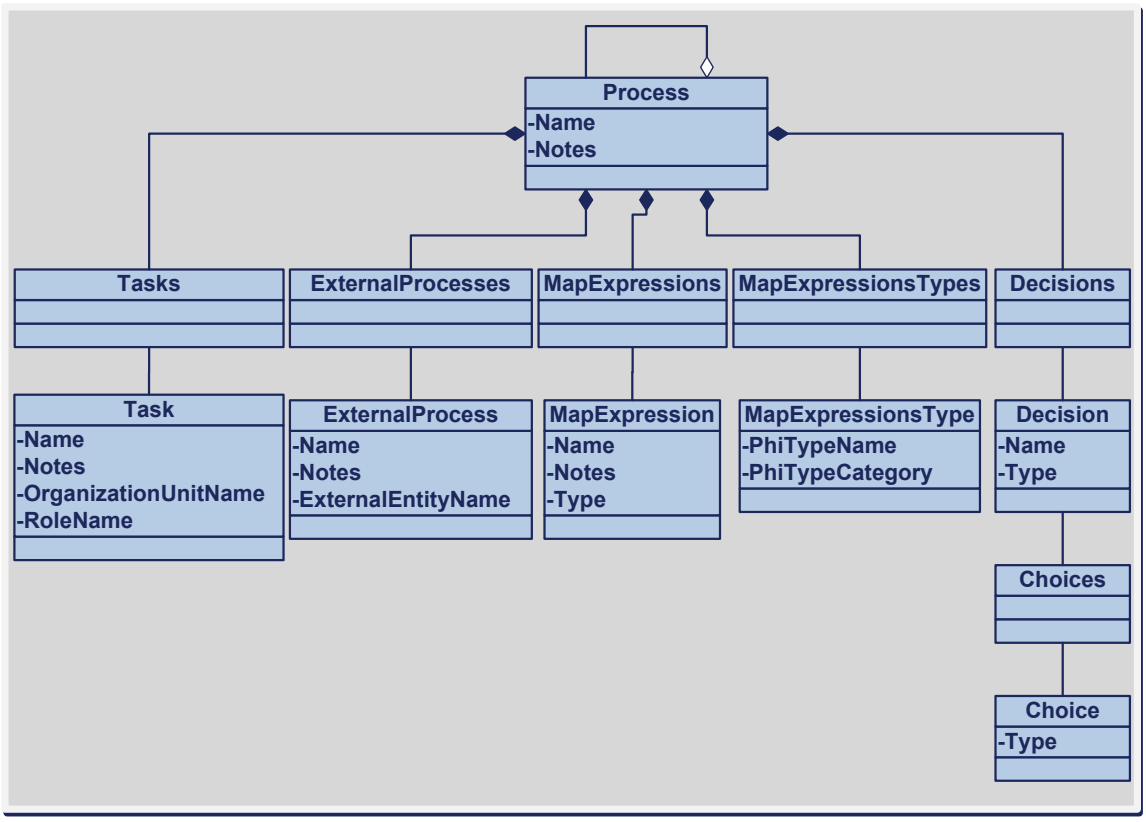


Figure 4.2: Business Process Domain Model

cant semantic gap between the two domains, before dependencies can be established, it is necessary on one hand to augment the representation of the business workflows with implementation information, and, on the other hand, to abstract the code representation in terms of related business functionality. Once models, domain models and, metamodels are represented in a MOF-compliant form [OMG06], they can then be represented as XML documents and consequently their transformations can be encoded as XSLT transforms [Hol00]. The details of this case study are presented in Chapter 6.

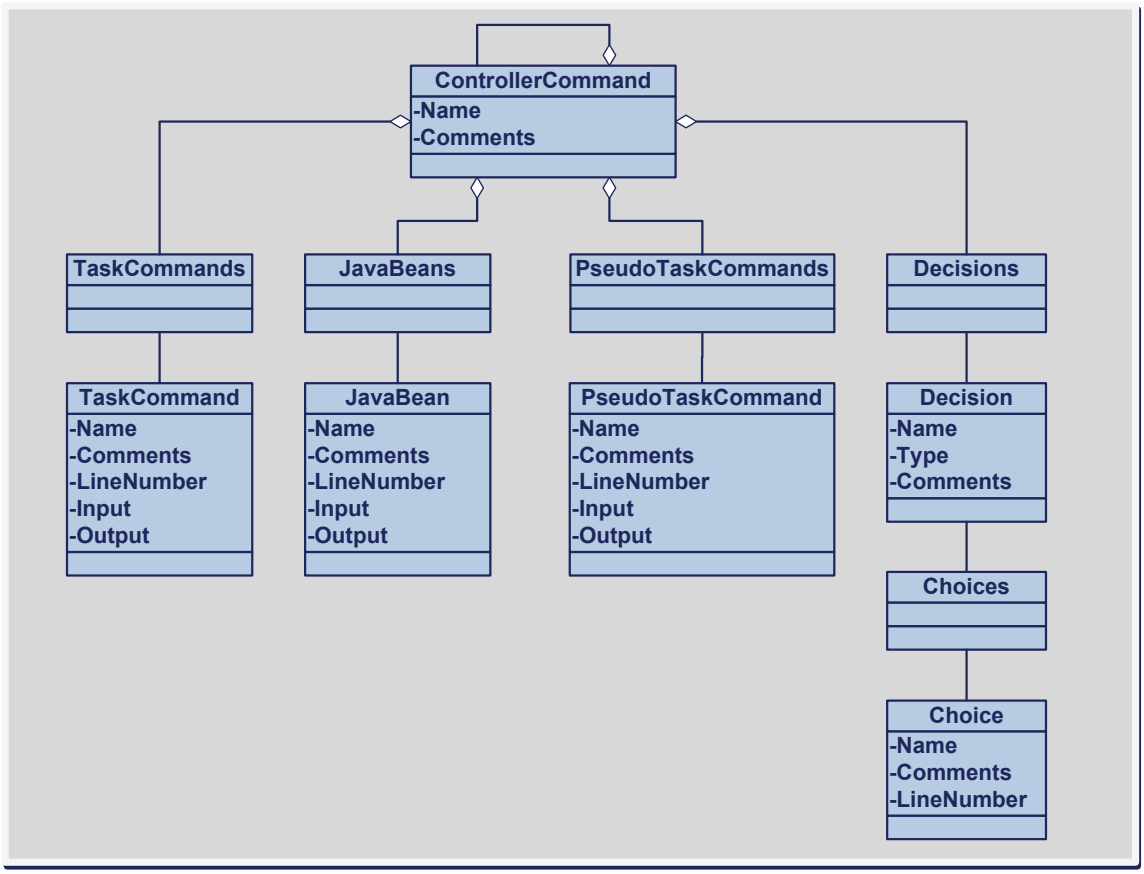


Figure 4.3: Source Code Domain Model

Figure 4.3 illustrates an example of an abstracted source code domain model. The source code elements such as classes and methods are represented through abstraction as ControllerCommand that are containers for other ControllerCommand and second-level abstractions such as JavaBeans for invocation of Java Beans, TaskCommands for external invocation, PseudoTaskCommands for clustered fragments of code, and Decisions. The source code files are represented in XML, where this domain model is encoded as a DTD schema [ZLK+04]. The mappings between source code and other models can thus be performed as mappings of XML Document Object Model (DOM) trees.

4.3 Applying Annotation Transformer

Before FCA can be used, attribute associations need to be established for purposes of model alignment. To establish relations between heterogeneous attributes, we make use of attribute association rules to map attributes and values between models that relate and conform to different domain models. The rules are established both at the domain model level and at the concrete model level. The differences in levels of expressiveness and semantics are augmented through generation of association models. Once the rules are established, FCA is used for “clustering” objects based on shared attributes. A “clustered” group of objects constitutes a concept and the objects are then said to be dependent.

The elements of domain models such as types, relations, and attributes, are extracted through domain analysis and represented using MOF syntax. To bridge possible syntactic and semantic gaps between domain models, the more abstract ones are iteratively annotated and enhanced into more specific ones, while the more concrete ones are iteratively

abstracted and refined into more abstract ones in an attempt to bridge the syntactic and semantic gap between the models that need to be associated.

4.3.1 Defining and generating association metamodels

The models that need to be synchronized are based on domain model schemas. As presented in the previous chapter, the elements of such domain models are represented using MOF syntax, as types, relations, and attributes that are specific to a particular domain. The domain models need to accurately represent their domain constituents in a format that provides for easy access and manipulation. Domain models based on MOF could be for instance represented in XML [W3C00], where the domain model elements are used to define the corresponding DTD schema.

As an example, a domain model for business workflows, shown in Figure 4.2, is a schema represented in UML that denotes Processes, Tasks, Decisions, Data, *etc*. as UML classes, where process is a container class for itself and other classes.

A domain model can be instantiated to yield a concrete model that describes a specific process or a source code segment. Relationships between one or more domain models can be represented by the use of association metamodels. In this context, models, domain models, and metamodels can be considered as typed, attributed, labeled, directed graphs. If the domain models are not available, as part of the requirements or design documentation, they may need to be extracted using a technique such as Feature-Oriented Domain Analysis (FODA) [Kea97].

For two models that need to be synchronized, the relations between them can be established first at the domain-model level, and then at the model-instance level. To establish dependencies at the domain-model level, types, relations, and attributes that are dependent

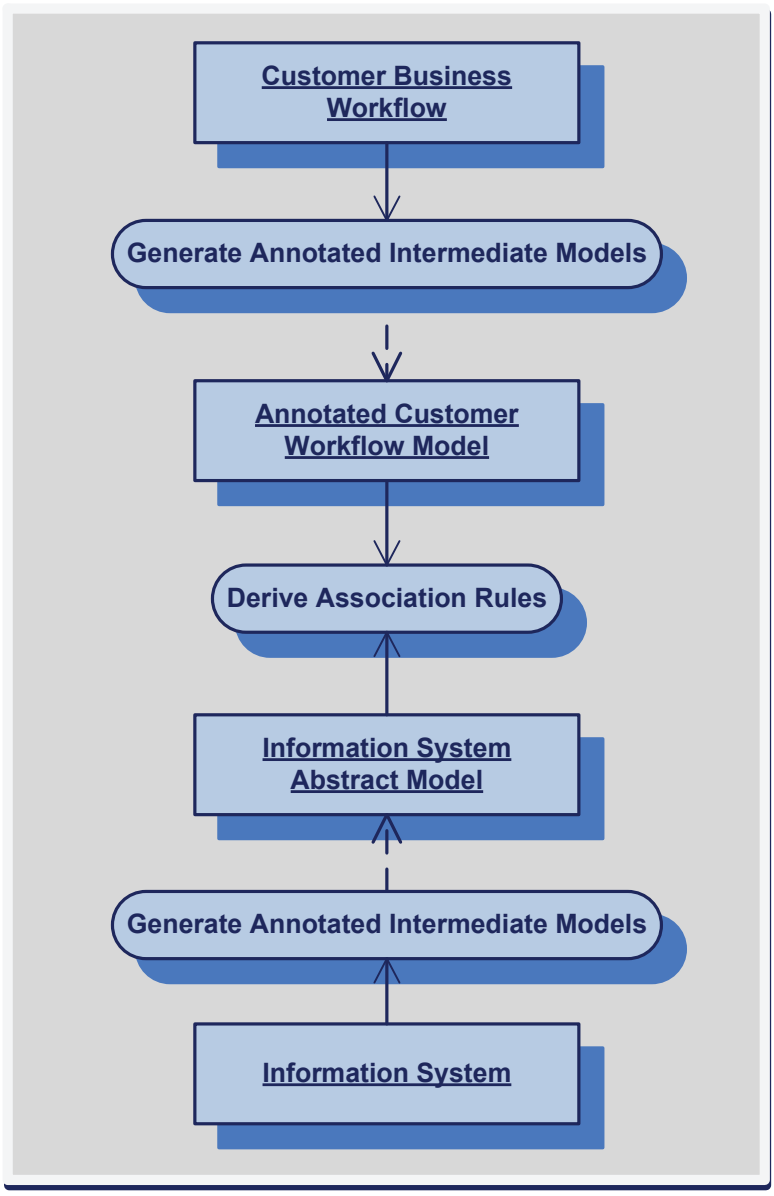


Figure 4.4: Synchronizing Business Processes with Source Code

need to be derived and recorded as association rules. To establish relations at the concrete model level, the rules established at one of the higher levels are used to identify tuples of model elements that are related.

Since the approach is focused on heterogeneous models at different levels of abstraction, establishing relations directly without refinement may be quite difficult. To overcome the differences in model expressiveness and semantics, we propose to generate more closely related association models. These models would be less syntactically and semantically diverse and therefore, establishing relations among them would be easier.

For example, in the problem of synchronizing business processes with source code, we annotate and enhance business process models with concrete data flow and control flow information, and at the same time, we abstract the source code to yield activity-like models. In Figure 4.4, this convergence is illustrated, where Customer Business Workflow models are annotated to yield Customer Workflow Detailed models, and Information System models are abstracted to yield Information System Abstract models. This is a problem that falls in the context of schema mapping, where the objective is to associate schema elements from two different schemas, and to identify mappings between such elements. Work conducted by Miller [ABMM07] falls in this category. Since schema mapping fall beyond the topic of this thesis, we propose a simple heuristic process that can be semi automated in order to establish associations between elements of domain models of different schemas. Of course, this heuristic approach can be replaced by a more sophisticated schema approaches, such as the one presented in [ABMM07]. We describe this heuristic process that is based on attribute association rules, such as hierarchical association rules, type-based association rules, spatial resolution rules, text-based association rules, more formally as follows:

1. Let M and G be two concrete models, instantiated from domain models DM_M and

DM_G, respectively. Analyze DM_M and DM_G to establish compatible domain elements such as domain types, relations, events, *etc*.

2. Create convergent DM_M' and DM_G' from DM_M and DM_G respectively, so that DM_M' and DM_G' are more closely related and relations among them can now be established.

3. From M and G, generate M′ and G′ based on DM_M' and DM_G', respectively.

The sections below will discuss this process in more detail.

Attribute association rules

The domain models for a particular domain at a particular level of abstraction based on our assumption would have common properties such as consistent features maps, lexicons, ontologies, *etc*. The attribute association rules can thus be viewed as mappings between attributes of heterogeneous models based on the mappings of the domain-specific properties. The classification of the association rules, and the examples for each of the rules based on the domain models for business workflows and source code (Figures 4.2 and 4.3 respectively), are discussed below.

- **Hierarchical Association Rules** — Models are parts of model hierarchies and feature maps for the hierarchies are extracted. Based on the containment relations between models and features, two association types are recognized:
  - direct, where models M and G are associated directly since they implement related features F_M and F_G, or

- indirect, where models M and G are associated indirectly since M contains a model M′ that implements a feature F_M that is related to a feature F_G that G implements. In this case, M′ and G are directly related.

To illustrate hierarchical association rules, let M be a process "Process order" and let M′ be process "Prepare inventory" that is contained within M and that implements a feature F_M "Inventory management". Also, let G be a source code class "InventoryAllocation" that implements a feature F_G "Inventory management", and let features F_M and F_G be related. From the relation between F_M and F_G, it follows directly that M′ and G are related, and indirectly that M and G are related since M contains M′.

- **Type-Based Association Rules** — Domain elements such as domain types, relations, and attributes defined in different domain models are associated based on compatible structural properties (*e.g.*, equivalent domain model, metamodel, or even metametamodel classification).

  To illustrate type-based association rules, let us consider the types Process and ControllerCommand of the respective domain models (Figure 4.2). The compatible properties are that they are both root types of each model, that they contain themselves and other subtypes, and that they have related attributes (Name, Name) and (Notes, Comments). As a result, Process and ControllerCommand are associated.

- **Spatial Association Rules** — A flow of data in a model is represented as order attributes, such that object o_1 precedes object o_2. The associations are established based on the order attributes.

  To illustrate spatial association rules, let us consider a flow (m_1, m_2) for a model M and a flow (g_1, g_2) for a model G. The attributes of the first flow would be for m_1 "Preceded by null" and "Followed by m_2", and for m_2 "Preceded by m_1" and "Followed by

null". The attributes for the second flow would be analogous, so m_1 and g_1 could be matched since they are both preceded by null, and m_2 and g_2 could be matched since they are both followed by null.

- **Text-Based Association Rules** — Informal information attributes (*i.e.*, comments, descriptions, variable names, types, *etc*.) are viewed as strings of text. The difference in syntax and semantics are resolved through:
  - thesaurus replacements — related synonyms are mapped,
  - stemming — each word is reduced to its root (*e.g.*, resource, resources, resourceful to resource),
  - abbreviation expansion — abbreviations that are recognized for a particular domain are expanded,
  - stop-word elimination — words with no semantic meaning, locally or globally, are eliminated,
  - word-matrix matching — groups of attributes that share words with semantic meaning are recognized, and
  - n-gram matching — strings are divided into substrings of size n and matched accordingly.

  To illustrate text-based association rules, let m_a := "Verify the order line item is still old" and g_b := "Verify that this order item still meets the criteria for being stale" be related attributes for elements m $\in$ M and g $\in$ G. We perform text-based matching as follows:

  1. Eliminate stop words and replace synonyms to obtain $m_a{}'$:= "verify order line item stale" and $g_b{}'$:= "verify order item meets criteria stale"

2. Use 3-gram matching and the match level of 0.6 on m_a'' := “ver eri rif ify ord rde der lin ine ite tem sta tal ale” and g_b'' := “ver eri rif ify ord rde der ite tem mee eet ets cri rit ite ter eri ria sta tal ale” to obtain average(3-gram-match(m_a'', g_b”$''$), 3-gram-match(g_b”$''$, m_a”$''$)) = 0.744 for the successful match.

The attribute association rules are formally represented as OCL descriptions [WK98]. The following illustrates in OCL the preceding 3-gram matching rule at the match level of 0.6:

M$->$iterate(m : ModelElement |
  G$->$iterate(g : ModelElement;
    result : Boolean = NGramMatching.ApplyRule(m$->$a, g$->$b, 3, 0.6)))

Other than the proposed method for establishing relations, there are other techniques such LSI [DDF$^+$90].

Here we introduce a sample association rule, A_R, which is defined as follows.

A_R := {TypeMatching(SourceType, TargetType),
  NGramMatching.ApplyRule(SourceType->ElementName,
    TargetType->ElementName, 3, 0.6),
  NGramMatching.ApplyRule(SourceType->ElementDescription,
    TargetType->ElementDescription, 3, 0.4)}

This rule denotes that there is a type matching between source type and target type, and there is text-based association rule, where the 3-gram matching is applied for matching source and target model element names with the threshold level of 0.6. The value of 0.6 here indicates that we would like the 3-gram matching to be performed at a relatively high degree of text-matching precision. Please note the threshold values range from 0 to 1, 0 meaning anything matches and 1 meaning exact matching. Similarly, the third rule indicates

the 3-gram matching is applied for matching source and target model element descriptions with the threshold level of 0.4.

Unmatched objects

The unmatched objects may be recognized through association of their immediate neighbors based on additional or refined attributes and rules. For instance, in the following two flows (m_1, m_2, m_3, m_4) and (g_1, g_2, g_3, g_4), tuples (m_1, g_1), (m_1, g_2), (m_3, g_4) are found as model dependencies. It may be possible to ascertain dependencies for unmatched objects m_2, g_3, and m_4 by performing additional clustering on these objects and their matched neighbors with new or changed attributes and rules. If for instance m_2 is now found to be related to m_1 and g_1, a tuple (m_2, g_1) would be created.

Conflict resolution rules

For the objects that are a part of two or more clusters, it may be necessary for practical reasons to decide to which cluster they more strongly belong. This conflict resolution may be achieved by weighted scoring of individual rules, where some rules are assigned a higher value through initial experiments for a particular domain. The best match is selected by maximizing the weighted score. In case that more than one cluster is found with the maximum score, all of the tuples from such clusters would be encoded as model dependencies.

More formally, let A_R be a set of applicable attribute association rules and let MG := M x G be a set of tuples of model elements. For a tuple (m_p, g_q) $\in$ MG, the weighted score WS_{pq} := $\sum$ w_i * ar_i(m_p, g_q) where ar_i : Boolean $\in$ A_R and w_i are domain-specific weight parameters. The maximum score for an element m_p is WS_{pmax} := $\max${WS_{p0}, WS_{p1} ...} and the top matches for an element m_p are tuples (m_p, g_i) for which the WS_{pi} := WS_{pmax}.

4.4 Establishing Model Dependencies using FCA

After semantic gaps (*i.e.*, schema mappings) are bridged manually or semi automatically utilizing the heuristics as discussed above or by schema mapping techniques as the ones proposed in [ABMM07], the association rules are defined based on compatible properties. Using FCA and the association rules, clusters of objects that share common attributes are identified, and represented as tuples of model dependencies.

This section describes the framework for automatically establishing model dependencies using formal concept analysis (FCA), which was introduced in [IK05]. As a part of this view, models are viewed as collections of objects and attributes.

The complexity and the accuracy of individual mappings between models depends on the type and the amount of information that is available from each model. If we compare concurrent mapping of structural and temporal properties to mapping of only structural or only temporal properties, it holds that the number of eligible elements available would increase in the former case as would the complexity of the mapping. As a result, the accuracy of the mapping in the latter case may be higher.

Our focus is on a set of specific domain model properties based on the corresponding structural, temporal, spatial, and behavioral attributes. As part of the approach, all domain models and all instantiated concrete models are viewed as objects and attributes. All model properties are then viewed as labels and attribute-value pairs. With regards to semantic homogeneity of considered domains, we assume that models belonging to a particular domain, such as database management, represented at a particular level of abstraction would be based on consistent features and types, and would also contain specific lexicons and ontologies. Based on this assumption, we create matches of models and model elements by associating related lexical and ontological concepts as part of the attribute association rules.

4.4.1 Introduction to FCA

Before using FCA to establish model dependencies, we first need to introduce related FCA definitions [GW99].

> **Formal Context**: A *formal context* K := (O, A, I) contains two sets O and A, and a relation I between O and A. The elements of the first set O are called the objects, and the elements of the second set A are called the attributes of the context.

If we need to express that an object o from O is in a relation I with an attribute a from A, we would write this as oIa and read it as "the object o has the attribute a". The relation I in this case is called the incidence relation of the context M.

We apply this definition to the previously introduced definition of domain model elements (see Section 3.3), to interpret domain models elements as domain-model contexts DMC := (O_{DM}, A_{DM}, I_{DM}), where O_{DM} := { o_i | domain types $o_i \in$ DT }, A_{DM} := { a_i | domain attributes $a_i \in$ DA $\cup$ DC }, and I_{DM} := { i_i | domain relations $i_i \in$ DR }.

The models instantiated from respective domain models are also contexts M := (O_M, A_M, I_M), where O_M := { o_j | domain type names $o_j \in T_{Names}$ }, A_M := { a_j | domain attribute names $a_j \in A_{Names} \cup C_{Names}$ }, and I_M := { i_j | domain relation names $i_j \in R_{Names}$ }

It follows then that a domain-model context DMC_M is a metacontext for a model M that is instantiated from DMC_M. Since FCA dictates binary contexts, we provide a mapping of n-ary to binary relations through combinatorial scaling. For example, an object o and n possible values v_1, v_2... are represented as binary relations (o, $\emptyset$), (o, v_1), (o, (v_1, v_2)), ... (o, (v_1, v_2, ... v_n)).

For a set $O_1 \subseteq$ O of objects, let

$$A_1 := \{a \in A \mid oIa, \forall o \in O_1\} \tag{4.1}$$

be the set of attributes common to the objects in O_1. Also, for a set $A_2 \subseteq A$ of attributes, let

$$O_2 := \{o \in O \mid oIa, \forall a \in A_2\} \tag{4.2}$$

be the set of objects which have all the attributes in A_2.

> **Formal Concept**: A *formal concept* of the context (O, A, I) is a tuple (O_1, A_2) with $O_1 \subseteq O$, $A_2 \subseteq A$, $O_1 = O_2$ and $A_2 = A_1$, where O_1 is the extent and A_2 is the intent of the concept (O_1, A_2).

The relations between attributes in A are represented through a relation of format "$A_i \rightarrow A_j$", where A_i and A_j are subsets of A utilizing the schema mapping heuristics presented in the previous section. This statement implies that an object that has the attributes in A_i also has the attributes in A_j. To visualize these relations, a concept lattice can be built.

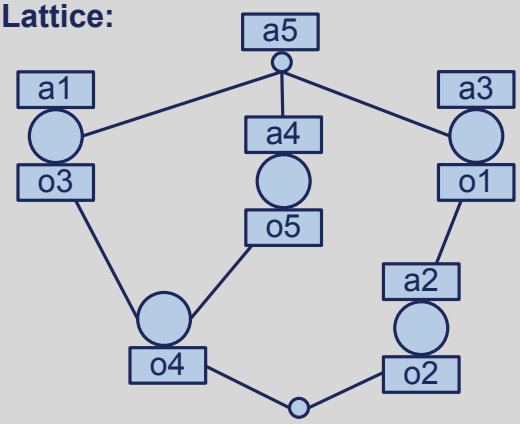
In Figure 4.5, a context of five objects O := {o_1, o_2, o_3, o_4, o_5} and five attributes A := {a_1, a_2, a_3, a_4, a_5} is represented as a matrix. Each cell in the matrix represents a relation between an object and an attribute, where a filled cell indicates that a particular object has a particular attribute (*e.g.*, $o_3 I a_1$ since the cell (3, 1) is filled). From the context, the attribute associations can be inferred as shown in the figure (*e.g.*, $\{\} \rightarrow a_5$ implies that all objects in the context exhibit attribute a_5). The concept lattice is built from the attribute associations, and from the concept lattice concepts can be read by reading the objects from the bottom and the attributes from the top (*e.g.*, object o_3 and attributes a_1 and a_5 are a concept, objects o_3, o_4, o_5 and attributes a_1, a_4, a_5 are a concept).

We refer to the rules for defining associations between attributes of different contexts as *attribute association rules*.

Context:

	a1	a2	a3	a4	a5
o1			x		x
o2		x	x		x
o3	x				x
o4	x			x	x
o5				x	x

Concept Lattice:



Association Rules:

```
1 < 5 > { } ==> a5;
2 < 0 > a3 a4 a5 ==> a1 a2;
3 < 1 > a2 a5 ==> a3;
4 < 0 > a1 a3 a5 ==> a2 a4;
```

Figure 4.5: FCA Example

4.4.2 Defining Model Dependency

Before discussing the method for establishing dependencies, we first define model dependency as follows.

> **Model Dependency**: A *model dependency* between models M and G is a set of tuples (m_i, g_j) of elements obtained from models M and G such that m_i and g_j have associated attributes.

More formally, let M and G be two models at different levels of abstraction with corresponding domain-model contexts DMC_M and DMC_G. Let O_M and O_G be the objects and let A_M and A_G be the attributes for M and G. Also, let

$$A_{MG} := \{a_M, a_G \mid a_M \subseteq A_M, a_G \subseteq A_G, a_M \Rightarrow a_G\} \tag{4.3}$$

be related attributes from A_M and A_G. The attribute logic for attribute associations $a_M \Rightarrow a_G$ of attributes from different contexts is defined as a set A_R of attribute association rules.

For nonempty sets ${O_M}' \subseteq O_M$ and ${O_G}' \subseteq O_G$ of objects, let

$$A'_{MG} := \{a \in A_{MG} \mid o_1 I a, o_2 I a, \forall o_1 \in O'_M, \forall o_2 \in O'_G\} \tag{4.4}$$

be the set of attributes common to the objects in ${O_M}'$ and ${O_G}'$. Also, for a nonempty set ${A_{MG}}' \subseteq A_{MG}$, let

$$B'_{MG} := \{o_1 \in O_M, o_2 \in O_G \mid o_1 I a, o_2 I a, \forall a \in A'_{MG}\} \tag{4.5}$$

be the set of objects which have all the attributes in ${A_{MG}}'$.

> **Model Dependency via FCA**: An concept between two contexts M and G is a set (${O_M}'$, ${O_G}'$, ${A_{MG}}'$) with ${O_M}' \subseteq O_M$, ${O_G}' \subseteq O_G$, ${A_{MG}}' \subseteq A_{MG}$, and B_{MG} = (${O_M}'$ $\cup$ ${O_G}'$).

4.4.3 Algorithm MDD-FCA:

Establishing Model Dependencies Through FCA

The algorithm for establishing model dependencies then includes the following steps.

Algorithm MDD-FCA Establishing Model Dependencies Through FCA

Input:

1. Model M := (O_M, A_M, I_M), 2. Model G := (O_G, A_G, I_G)
3. Domain-Model Context DMC_M, 4. Domain-Model Context DMC_G

Output:

1. Established Model Dependencies D

Steps:

Step 1. Identify a set A_{MG} of related attributes from A_M and A_G based on compatible properties such as matched types, associations, data flow, spatial properties, informal information, *etc*. of DMC_M and DMC_G.

Step 2. Given A_{MG}, derive a set A_R of corresponding association rules based on the matched properties of DMC_M and DMC_G. For example, for attributes matched based on spatial properties, use the spatial-matching rules.

Step 3. Iterate through elements of M and G to identify an initial set of model dependencies D that share the attributes from A_{MG} using the rules from A_R for association of concrete attributes.

Step 4. Exclude from D all of its irrelevant (*e.g.*, include only the objects from one context) and redundant (*e.g.*, equivalent association results based on different sets of attributes) elements.

Step 5. For those elements of M and G that cannot be matched using attribute associations, attempt matching using their already-matched neighbors.

Step 6. For those elements of M and G that are matched to more than one cluster, select the matchings using a corresponding conflict resolution rule. For example, each association rule could be assigned a weight and those elements with a maximum weighted score are selected as the top results.

Step 7. Return D as the set of established model dependencies.

4.4.4 Illustrative Examples

To illustrate the algorithm, we apply it to the application scenario (see Figure 3.7) introduced in the previous chapter. As shown in Figure 4.6, each instance of Person is matched to an instance of Author, and each instance of Book is matched to an instance of Author using type matching rules. An instance of Library from the source domain model is matched to an instance of Library from the target domain model, also using type matching. Finally, attribute pairs (Person->pname, Author->aname), (Book->person->pname, Author->aname), and (Library->libID, Library->libID) are matched using lexicographical nGram matching rules.

To further clarify the algorithm, we also demonstrate its usage on a simple scenario of two models, a workflow M and a source code class G, that were matched based on suitable properties. We focus on matching of their elements, so we apply the algorithm. For Step 1, we identify a set of compatible attributes A_{MG} := ((Process Name, Class Name), (Notes, Comments), (Subsystem, Package), (Previous Element in the Process Flow, Previous Element in the Information Flow), (Next Element in the Process Flow, Next Element in the Information Flow)) from the workflow and abstracted source code domain models. For Step 2, based on A_{MG} we identify a set of association rules A_R that includes: text-based matching using

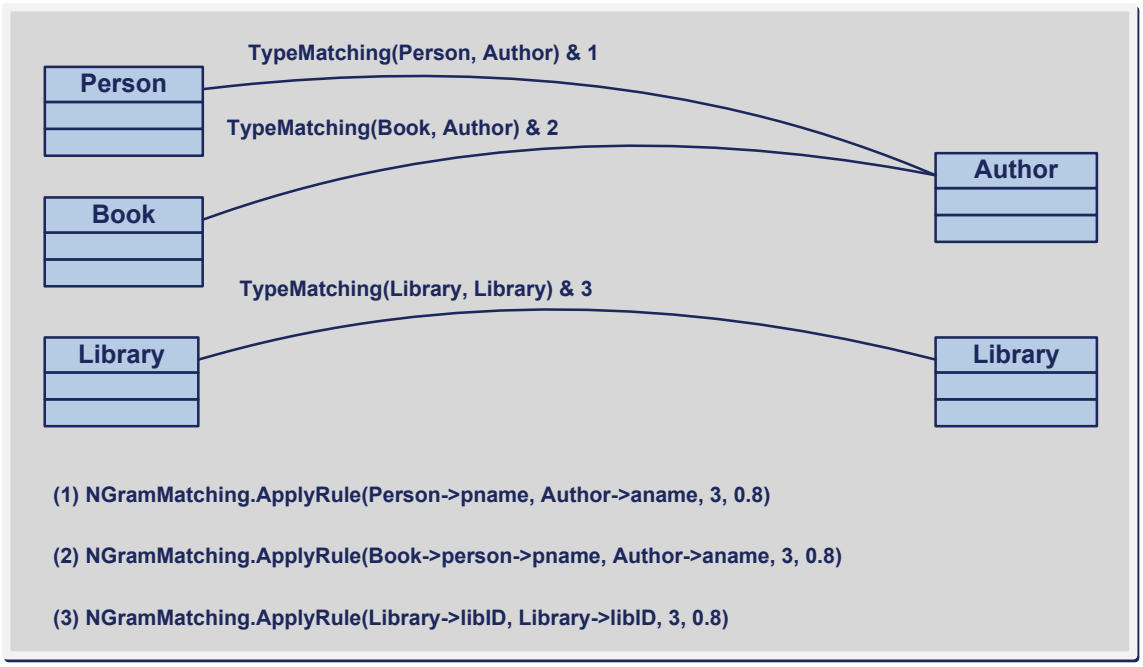


Figure 4.6: FCA Illustrative Example 1

a combination of techniques including stemming, abbreviation expansion, stop-word elimination, n-gram matching, *etc*. on attribute pairs (Process Name, Class Name) and (Notes, Comments); type-based and hierarchical matching by extracting workflow and source code hierarchies on attribute pair (Subsystem, Package), and type-based and spatial matching using position and data flow on attribute pairs (Previous Element in the Process Flow, Previous Element in the Information Flow) and (Next Element in the Process Flow, Next Element in the Information Flow). In Step 3, we iterate through elements of M and G to identify those that cluster together. For example, we matched the following two model elements:

- o_M := ((Process Name, “Find stale order line items”), (Notes, “Find order line items that are stale”), (Subsystem, “Order”), (Previous Element in the Process Flow, “Time to execute” (Task)), (Next Element in the Process Flow, “Are there more order items”(Decision))) and

- o_G := ((Class Name, “abOrderJDBCHelper.findStaleOrderItems”), (Comments, “Call the Order query to get the list of expired order items”), (Package, “Order”), (Previous Element in the Information Flow, “startUse” (Task)), (Next Element in the Information Flow, “hasMoreElements” (Decision))).

The matching of elements, if we consider text-based matching rules, is based on:

- n-gram distance between semantically relevant words using thesaurus replacements of (Process Name, “Find stale order line items”) and (Class Name, “abOrderJDBCHelper. findStaleOrderItems”),

- n-gram distance and hierarchical (*i.e.*, Subsystem to Package) mapping of (Subsystem, “Order”) and (Package, “Order”), and

- n-gram distance, type-matching (*i.e.*, Decision to Decision), and spatial matching of (Previous Element in the Process Flow, “Time to execute” (Task)) and (Previous Element in the Information Flow, “startUse” (Task)), and (Next Element in the Process Flow, “Are there more order items” (Decision)) and (Next Element in the Information Flow, “hasMoreElements” (Decision)).

Figure 4.7 illustrates attribute-value pairs used to match elements of M and G, and establish a model dependency between them.

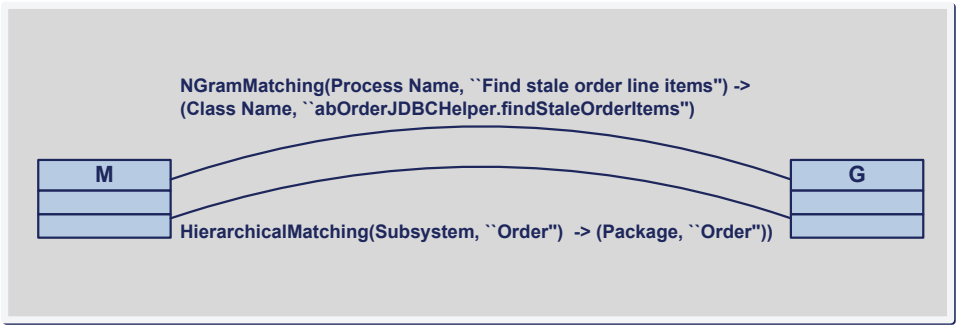


Figure 4.7: FCA Illustrative Example 2

4.5 Representing Model Dependencies using Profiles and Code Templates

Once model dependencies have been identified using Formal Concept Analysis (FCA), the problem becomes how to represent such dependencies in a MOF-compliant way so that these dependencies can be used by a programmatically-automated model synchronization

framework. More specifically, in this section, we aim to contribute to the development of a generalized MDA-based framework that can be used for fine-grained model synchronization. Namely, the framework first proposes a method for modeling structural and programmatic elements of Platform-Independent Models (PIMs) (*i.e.*, source models) and Platform-Specific Models (PSMs) (*i.e.*, target models) in an open and extensible manner. Second, the framework provides a method for encoding source-to-target model associations using UML profiles and stereotypes. Finally, the platform-specific models are automatically mapped to source code for the specific domain using customizable code templates.

In this context, the following challenges need to be addressed:

- How to define a customizable and extensible model-driven process that is based on iterative and incremental transformation of software artifacts and models.
- How to represent domain-specific models that can be customized for a given application. The approach bridges the dichotomy between the MDA approaches that utilize UML extensions (through profiling) and the approaches that utilize Domain Specific Languages (DSL).
- How to represent associations between source models and arbitrary target model entities in a way specific mapping languages or model weaving techniques are not required. The approach allows for constraints to be defined in terms of an open language such as OCL so that different mappings can be possible by considering the context these mappings are applied in.
- How to generate source code for the target application in a way that no restrictions are placed with respect to the target language or programmatic paradigm to be used.

4.5.1 Model Dependencies in the Context of MDA

Model Driven Architecture (MDA) is a methodology developed by the Object Management Group (OMG) for streamlining the design and implementation of large software-intensive systems. In MDA, software artifacts of large software systems are represented as a collection of Platform Independent Models (PIM) that can be transformed to a collection of Platform Specific Models (PSM). The collection of PSMs can finally be transformed into source code that complies with specific programmatic patterns. A PIM represents the elements of a software system in a way that is not bound or dependent to a specific implementation technology. Moreover, a PSM represents the elements and components of a software system in a way that directly relates to the implementation technology that will be used for implementing such a system.

MDA as a methodology has made progress from a "guideline" towards "standardized process" that is meant to improve productivity of software engineers and improve overall system robustness. However, some important questions regarding MDA still remain. For example, what specific types of models pertain to PIMs and PSMs, how to encode and enact PIM-to-PSM transformations, how to denote constraints, and how to generate source code, are some of the open questions today.

The existing CASE tools that support MDA process can be divided into two categories: full MDA-capability tools that provide modeling, transformation, and code generation infrastructure; and limited MDA-capability tools that incorporate only code generation infrastructure using specific models such as UML as input.

The main drawbacks of existing MDA tools include the following:

1. Existing MDA tools [Tea10a, Tea10b] focus on generating systems that are dependent on specific platforms such as J2EE [Ora10], Corba [OMG02], or .NET [Mic10].

2. The tools have limited capability on defining higher level transformation models since they utilize either specific model transformation languages such as QVT, or proprietary transformation formats.

3. The tools do not address interoperability and integrate only with specific modeling frameworks. And, they do not provide infrastructure to integrate with other tools and environments.

To address some of the open questions related to the MDA methodology, we describe the following approach for representing model dependencies using profiles and code templates, as part of our overall mSYNTRA framework. The approach was first introduced in [HIK+08]. This approach is intended as a full MDA-capability framework, including modeling, transformation, and code generation as it pertains to the MDA process.

More specifically, by modeling PIMs (source models) and PSMs (target models) in an open way as MOF models that can be extended incrementally through inheritance, associations, and aggregation to denote specific applications (e.g. Gift Registry) in a given domain (IBM WebSphere Commerce), we allow for the decoupling of a core source model from any model extensions that pertain to specific properties of the process or application (e.g. Gift Registry) being specified. This provides greater flexibility on generating richer target models and consequently, facilitates the generation of code and systems to several target platforms. Furthermore, by encoding source and target models using UML profiles and stereotypes, we provide a simple, yet expressive way of modeling source-to-target model associations at a higher level than QVT or any other rule-based model transformation formalism. This novel approach also allows for the transformation rules themselves to be modeled in a MOF compliant way, and therefore the automatic generation of QVT or any other formalism for model transformation rules is possible. This addresses the second MDA tool drawback pertaining

to the limited capability MDA tools have on denoting model transformation rules in an open and extensible way.

For example, a source model from a Commerce application can be represented by entities such as Service, Service Description, Binder, Invocation Protocol, while the corresponding target model entities that relate to SOA as the selected underlying implementation technology will contain corresponding entities such as Web-Service, WSDL, UDDI and, SOAP respectively.

Finally, by utilizing a template based approach for code generation combined with profile-based source-to-target model associations, we allow for the framework to be portable and usable by any modeling environment or IDE that accepts MOF compliant models. This aims to address the third drawback of existing MDA tools pertaining to the integration of MDA tools with other environments.

4.5.2 Representing FCA-Based Dependencies using Profiles

To model specific dependencies extracted using formal concept analysis, we make use of the Model Driven Architecture (MDA) methodology. Namely, we represent specific association rules as mappings between UML profiles, where individual types are represented as matching stereotypes.

We consider four core mapping cases: one-to-one, one-to-many, many-to-one, and many-to-many. For each mapping, we consider a set of attribute association rules, as described in Section 4.3.1.

- Type-based association rule, where the corresponding source and target classification types are matched.

- Text-based association rule, where the 3-gram matching is applied to model element names at the match level of 0.6 (higher precision).
- Another text-based association rule, where the 3-gram matching is applied to model element descriptions at the match level of 0.4 (higher recall).

In Figure 4.8, we demonstrate how a one-to-one type-matching rule and several text-matching rules are represented as profiles. Namely, each type-matching rule is represented as a profile mapping, where the source type represents the source stereotype and the target type represent the target package. Other rules that do not pertain to types are denoted as OCL constraints attached to the profile mapping.

In Figure 4.9, we demonstrate how a one-to-many type-matching rule and several text-matching rules are represented as profiles. The main difference here is that each dependency tuple is represented as an additional target element. Other rules that do not pertain to types are still denoted as OCL constraints attached to the profile mapping.

In Figure 4.10, we demonstrate how a many-to-one type-matching rule and several text-matching rules are represented as profiles. The main difference here is that each dependency tuple is represented as an additional source stereotype. Other rules that do not pertain to types are still denoted as OCL constraints attached to the profile mapping.

In Figure 4.11, we demonstrate how a many-to-many type-matching rule and several text-matching rules are represented as profiles. The main difference here is that each dependency tuple is represented as a corresponding additional source stereotype or additional target element. Other rules that do not pertain to types are still denoted as OCL constraints attached to the profile mapping.

One-to-One Model Dependencies

Let us assume, as illustrated in Figure 4.8, that there are the following elements. Source type represents a type from the source domain model DM_M, target type represents a type from the target domain model DM_G, A_R is a collection of association rules, and C is a collection of constraints stemming from the association rules A_R. Let us also assume that the dependency between the source type element and the target type element is a one-to-one dependency mapping. According to the theory presented in Section 4.4.2, the dependency is then denoted as $d_{MG} = \{o_i \in O_M, o_j \in O_G \mid o_i Ia, o_j Ia, \forall a \in A_{MG}\}$, which represents all model elements o_i from M and o_j from G that have attributes a from A_{MG}, where A_{MG} = {(SourceType, TargetType), (SourceType->ElementName, TargetType->ElementName), (SourceType->ElementDescription, TargetType->ElementDescription)}.

More specifically, the matching attributes from A_{MG} are represented as association rules A_R := {TypeMatching(SourceType, TargetType), NGramMatching.ApplyRule(SourceType-> ElementName, TargetType->ElementName, 3, 0.6), NGramMatching.ApplyRule(SourceType-> ElementDescription, TargetType->ElementDescription, 3, 0.4)} denotes that there is a type matching between source type and target type, and there is text-based association rule, where the 3-gram matching is applied for matching source and target model element names with the threshold level of 0.6. The value of 0.6 here indicates that we would like the 3-gram matching to be performed at a relatively high degree of text-matching precision. Please note the threshold values range from 0 to 1, 0 meaning anything matches and 1 meaning exact matching. Similarly, the third rule indicates the 3-gram matching is applied for matching source and target model element descriptions with the threshold level of 0.4.

This dependency is then denoted by a denotational function $\delta_P(d_{MG})$ using profiles as follows. SourceType becomes the stereotype profile for the TargetType element. The constraints C are attached to the profile, and the association rules A_R provide the details of the

possible mappings between elements of the source model to the target model (please see Section 4.3.1). For example, in the case of one-to-one model dependencies the $\delta_P(d_{MG})$ is equal to the model of Figure 4.8.

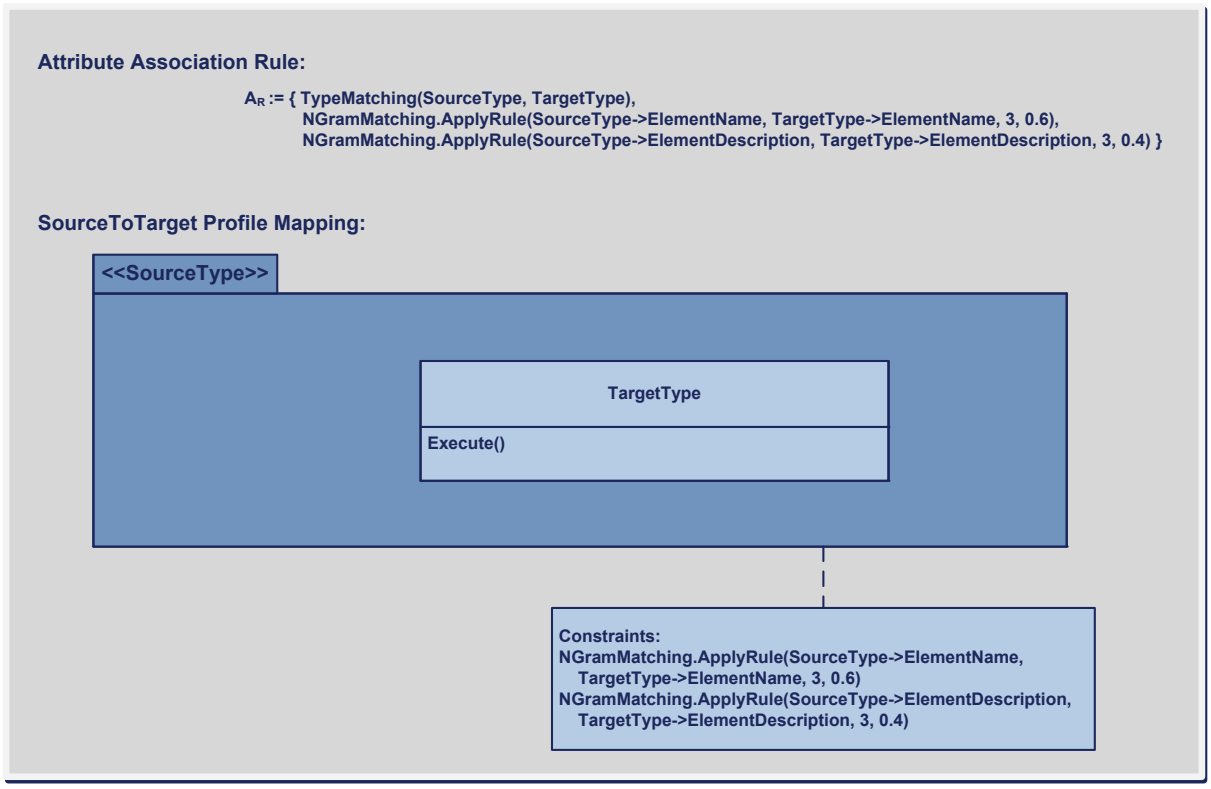


Figure 4.8: The One-to-One Model Dependencies as Profiles

One-to-Many Model Dependencies

Let us assume, as illustrated in Figure 4.9, that there are the following elements. Source type represents a type from the source domain model DM_M, three target types represent types from the target domain model DM_G, A_R is a collection of association rules, and C is a collection of constraints stemming from the association rules A_R. Let us also assume

that the dependency between the source type element and the target type element is a one-to-many dependency mapping. According to the theory presented in Section 4.4.2, the dependency is then denoted as $d_{MG} = \{o_i \in O_M, o_j \in O_G \mid o_i Ia, o_j Ia, \forall a \in A_{MG}\}$, which represents all model elements o_i from M and o_j from G that have attributes a from A_{MG}, where A_{MG} = {(SourceType, TargetType1), (SourceType, TargetType2),(SourceType, TargetType3), (SourceType->ElementName, TargetType->ElementName), (SourceType-> ElementDescription, TargetType->ElementDescription)}.

This dependency is then denoted by a denotational function $\delta_P(d_{MG})$ using profiles as follows. SourceType becomes the stereotype profile for the TargetType1, TargetType2, and TargetType3 elements. The constraints C are attached to the profile, and the association rules A_R provide the details of the possible mappings between elements of the source model to the target model (please see Section 4.3.1).

Many-to-One Model Dependencies

Let us assume, as illustrated in Figure 4.10, that there are the following elements. Three source types represent types from the source domain model DM_M, the target type represent a type from the target domain model DM_G, A_R is a collection of association rules, and C is a collection of constraints stemming from the association rules A_R. Let us also assume that the dependency between the source type element and the target type element is a many-to-one dependency mapping. According to the theory presented in Section 4.4.2, the dependency is then denoted as $d_{MG} = \{o_i \in O_M, o_j \in O_G \mid o_i Ia, o_j Ia, \forall a \in A_{MG}\}$, which represents all model elements o_i from M and o_j from G that have attributes a from A_{MG}, where A_{MG} = {(SourceType1, TargetType), (SourceType2, TargetType),(SourceType3, TargetType), (SourceType->ElementName, TargetType->ElementName), (SourceType-> ElementDescription, TargetType->ElementDescription)}.

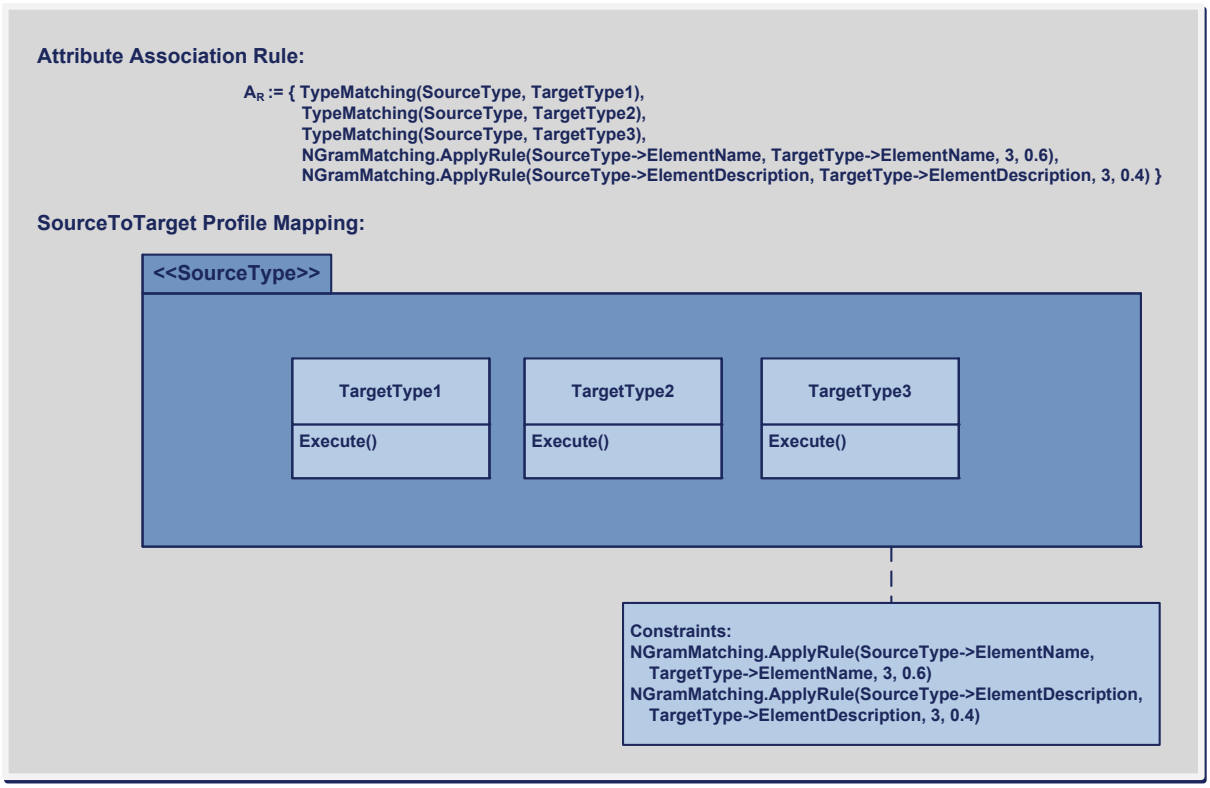


Figure 4.9: The One-to-Many Model Dependencies as Profiles

This dependency is then denoted by a denotational function $\delta_P(d_{MG})$ using profiles as follows. SourceType1, SourceType2, and SourceType3 jointly become the stereotype profile for the TargetType element. The constraints C are attached to the profile, and the association rules A_R provide the details of the possible mappings between elements of the source model to the target model (please see Section 4.3.1).

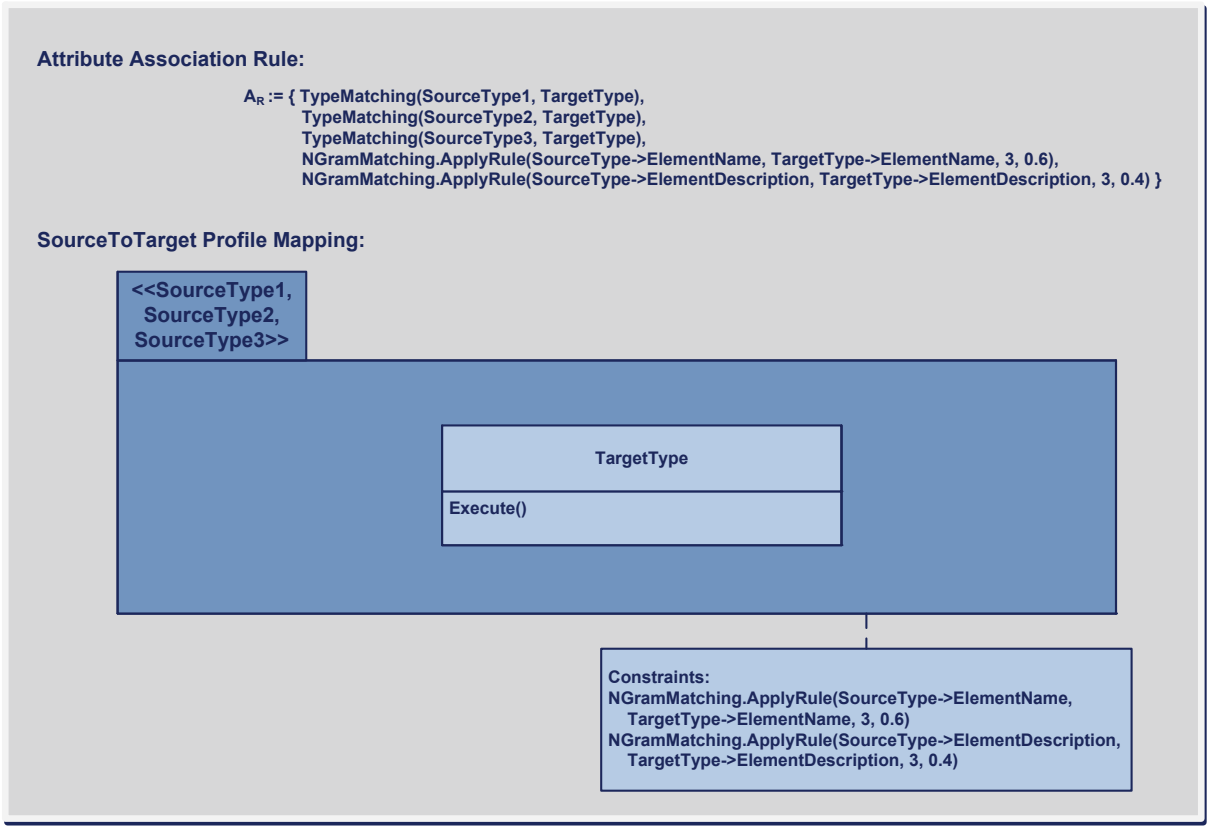


Figure 4.10: The Many-to-One Model Dependencies as Profiles

Many-to-Many Model Dependencies

Let us assume, as illustrated in Figure 4.11, that there are the following elements. Three

source types represent types from the source domain model DM_M, three target types represent types from the target domain model DM_G, A_R is a collection of association rules, and C is a collection of constraints stemming from the association rules A_R. Let us also assume that the dependency between the source type element and the target type element is a many-to-many dependency mapping. According to the theory presented in Section 4.4.2, the dependency is then denoted as $d_{MG} = \{o_i \in O_M, o_j \in O_G \mid o_i Ia, o_j Ia, \forall a \in A_{MG}\}$, which represents all model elements o_i from M and o_j from G that have attributes a from A_{MG}, where A_{MG} = {(SourceType1, TargetType1), (SourceType1, TargetType2),(SourceType1, TargetType3), (SourceType2, TargetType1), (SourceType2, TargetType2), (SourceType2, TargetType3), (SourceType3, TargetType1), (SourceType3, TargetType2), (SourceType3, TargetType3), (SourceType->ElementName, TargetType->ElementName), (SourceType-> ElementDescription, TargetType->ElementDescription)}.

This dependency is then denoted by a denotational function $\delta_P(d_{MG})$ using profiles as follows. SourceType1, SourceType2, and SourceType3 jointly become the stereotype profile for the TargetType1, TargetType2, and TargetType3 elements. The constraints C are attached to the profile, and the association rules A_R provide the details of the possible mappings between elements of the source model to the target model (please see Section 4.3.1).

4.5.3 Illustrative Example

To clarify how the FCA-based dependencies are mapped to profiles, we apply the profile-based representation to the illustrative example shown in Figure 4.6. The resulting stereotype mappings, which are based on the many-to-many model dependency mapping, are shown in Figure 4.12.

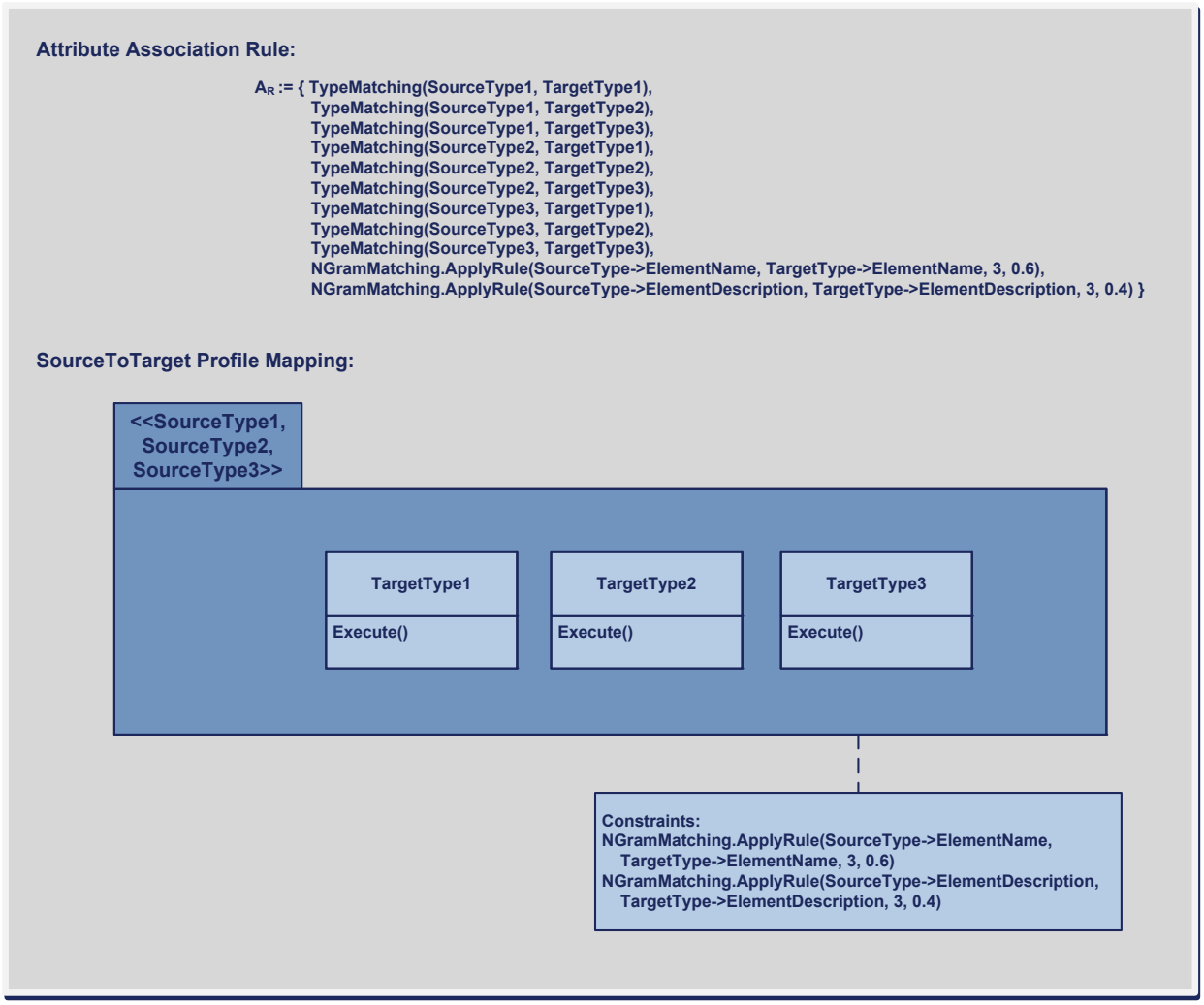


Figure 4.11: The Many-to-Many Model Dependencies as Profiles

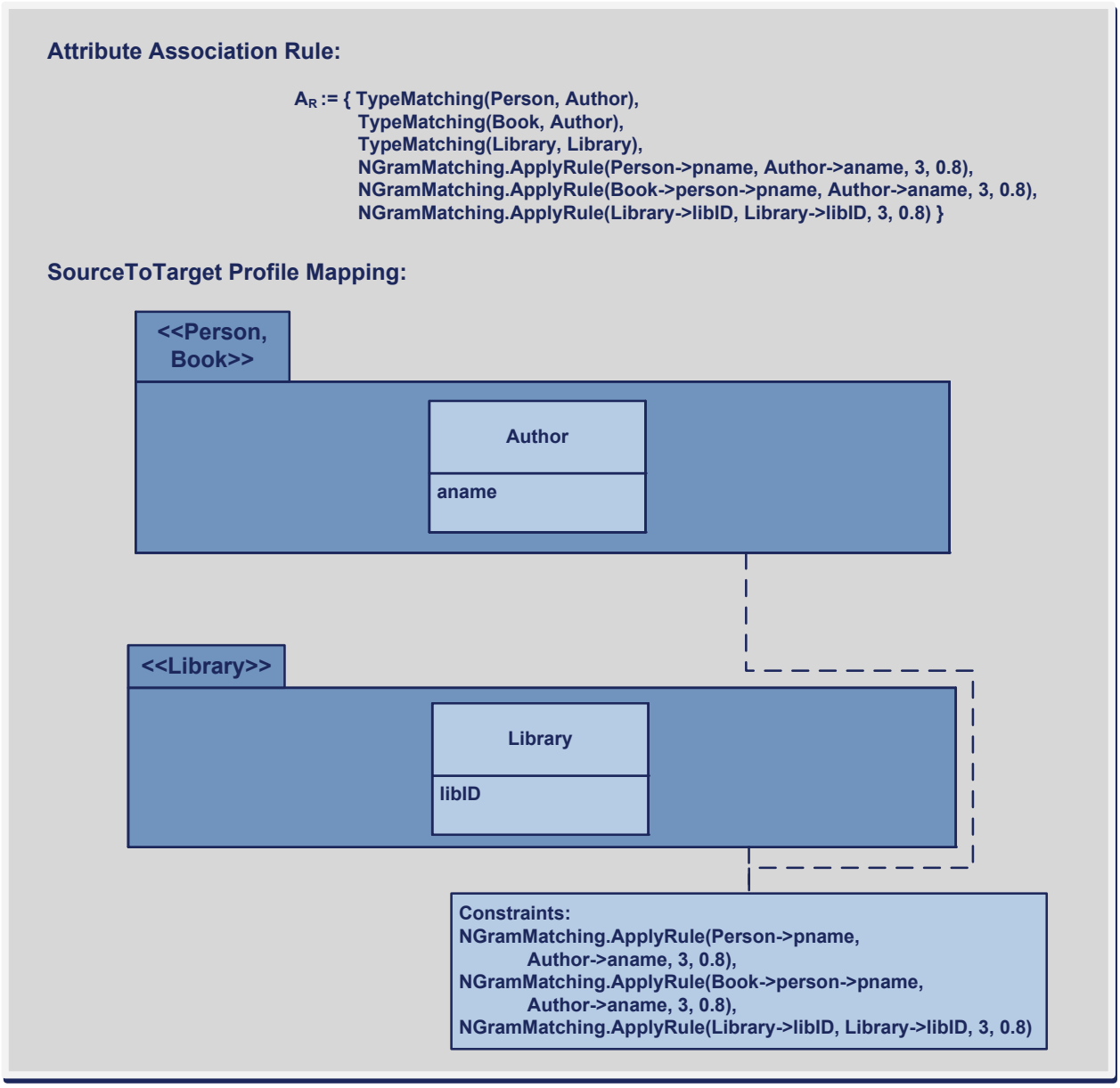


Figure 4.12: FCA Model Dependencies as Profiles Example

4.6 Validity of Profile-Based Model Dependency Representations

For the representation of FCA-extracted dependencies using profiles, we have abstracted model dependencies to provide more semantically precise representation (*e.g.*, the use of annotations and constraints). The notation provided in Chapter 3 serves as a formal foundation for the framework, but the use of profiles enables structural adoption in practice with more detailed semantic detail. The profile-based representation is then mapped to triple graph grammars in Chapter 5, to provide model synchronization capabilities and compliance with other related methodologies, such as QVT. Below, we outline the assessment of profile-based model dependency representations with respect to the properties of completeness and soundness.

Completeness All model dependencies D that are extracted using the proposed FCA-based approach can also be represented using profiles.

Proof: The proof is by construction on structural properties and element associations.

Structural properties: A tuple d $\in$ D has to be one of the following cardinality cases: 1-to-1, 1-to-many, many-to-1, and many-to-many. If d $\in$ D, it has to follow one of these cardinality rules. For any such rule, a profile model can be created as indicated in the construction process above.

Element associations: For element associations, we can differentiate two cases: known and unknown mappings. For known mappings, these are denoted by attribute association rules, as illustrated in Figures 4.8 to 4.11. For unknown mappings, we create explicit matching constraints as discussed in the heuristic process related to unmatched objects in the Section 4.3.1.

Soundness All model dependencies represented using profiles are sound representations of the corresponding model dependency tuples D.

Proof: The proof is analogous to the previous one, by construction on structural properties and element associations.

For structural properties, a profile mapping would denote any of the four cardinality rules. Let us assume that a profile mapping denotes a cardinality rule. There must be by construction a corresponding dependency tuple of the same cardinality.

For element associations, we distinguish two cases: known and unknown mappings. For known mappings, in the profile we must have explicit association rules that should appear by construction in the schema associations between the source and target models. And for unknown mappings, by construction constraint representations (Figures 4.8 to 4.11). Let us assume that a dependency is modeled through profiles. By construction, there should be an associated dependency tuple.

4.7 Chapter Summary

In this chapter, we have discussed a framework that first, makes use of Formal Concept Analysis (FCA) to systematically establish dependency relations among models and their elements, and second, denotes model dependency relations as UML Profiles. The steps for establishing model dependency relations include (1) the definition and generation of association models to bridge possible representational and abstraction gaps between domains, (2) the establishment of model dependencies based on attribute association rules using FCA, and (3) the validation of established dependencies. The process of denoting model dependencies as profiles is based on the constructing appropriate profiles for one-to-one,

one-to-many, many-to-one, and many-to-many dependency relations.

In the following chapter, we make use of the extracted profile-based dependencies, and represent them as triple graph grammar rules, for the purposes of fine-grained model synchronization.

Chapter 5

Fine-Grained Model Synchronization: Mapping Profile-Based Dependencies as Triple Graph Grammar Rules

Even if there is only one possible unified theory, it is just a set of rules and equations. What is it that breathes fire into the equations and makes a universe for them to describe?

— Stephen Hawking

5.1 Chapter Overview

In the previous chapter, we have introduced a process for representing dependencies between models by using UML profiles, as well as, mappings between profiles through UML

stereotypes.

In this chapter, we introduce modeling semantics that allow for more complex profile-based dependencies using Triple Graph Grammar (TGG) rules with corresponding OCL constraints.

5.2 Mapping Profile-Based Dependencies as Triple Graph Grammar Rules

In this section, we present how profile-based model dependencies can be represented as triple graph grammars. The motivation behind this concept is to allow for fine-grained model synchronization using a language-based approach, similar to the one used in natural language translation. The basics of the approach are illustrated in Figure 5.1. The set of profile-based model dependencies, D_{MG}, is either previously available or is created by applying the approach described in the previous chapter. For each $d_i \in D_{MG}$, a new triple graph grammar rule, tgg_i is created with the source stereotype or stereotype package of d_i representing the left-hand side of the new rule and the the target package representing the right-hand side of the new rule, with any other constraints of d_i encoded as the correspondence node of tgg_i. The new set of grammar rules, TGG_{MG}, is created as the output. Consequently, these rules can be encoded as QVT rules and used for fine-grained model synchronization.

To perform fine-grained model synchronization, let us assume that the consistent state between model repositories M and G is disrupted when a model $m_t \in M$ is transformed into m_t' by changing some elements of m_t, such as $me_1^t \ldots me_l^t$. For all $me_1^t \ldots me_l^t$ model elements that are affected by change, we apply rules from TGG_{MG} that pertain to $me_m^t \in \{me_1^t \ldots me_l^t\}$ to transform them to generate ge_j^k element of g_j so that m_t and g_j can be

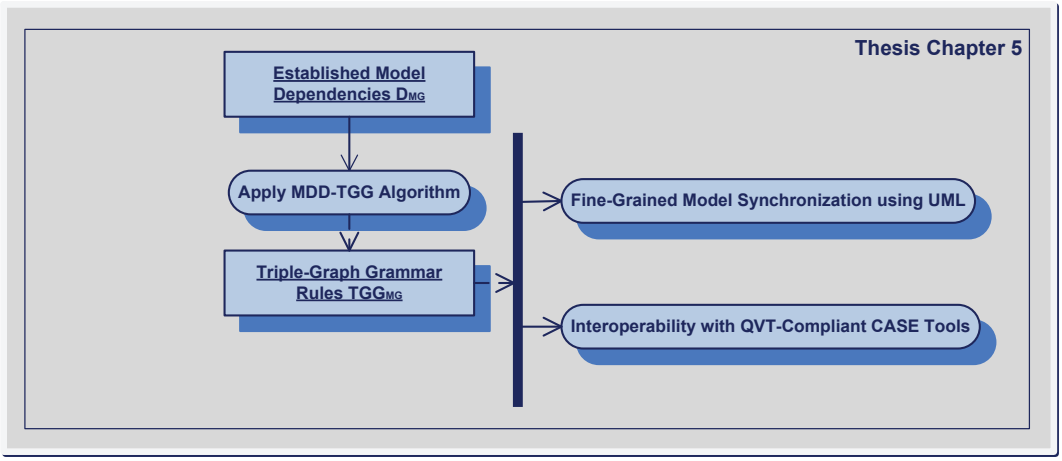


Figure 5.1: MDD-TGG Overview

again synchronized.

Before representing model dependencies as TGG rules, we first need to specify how the individual source and target metamodels are related using TGG [SK08].

In Figure 5.2, we specify the relation between the source metamodel, and the target metamodel, related through a correspondence metamodel specified using TGG rules.

Each TGG rule consist of the left-hand side graph, derived from the source metamodel, and the right-hand side graph, derived from the target metamodel, related through a middle graph node (as shown in Figure 5.3). Optional precondition constraint, specified in OCL for example, can be attached to the middle node, providing additional specification on when the TGG rule can be applied. For example, multiplicity constraints between source and target classes, role-based constraints, or domain-specific constraints such as the ones discussed later in this chapter.

The nodes and arcs marked with '++' symbols represent graph elements that can be

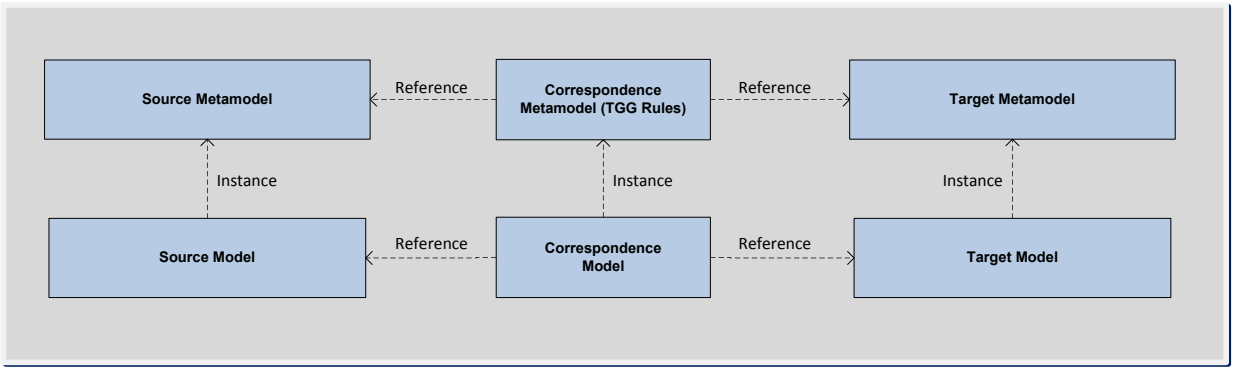


Figure 5.2: Mapping Profiles to TGG Rules Overview

created as part of the translation or synchronization. For forward translation, the right-hand side of the rule is created when the presence of the left-hand side (source) is detected. For backward translation, the left-hand side of the rule is created when the presence of the right-hand side (target) is detected. This approach effectively is equivalent to yielding two grammars, TGG_f for forward and TGG_b for backward translation. For synchronization, both left and right-hand sides are compared against the source and target models.

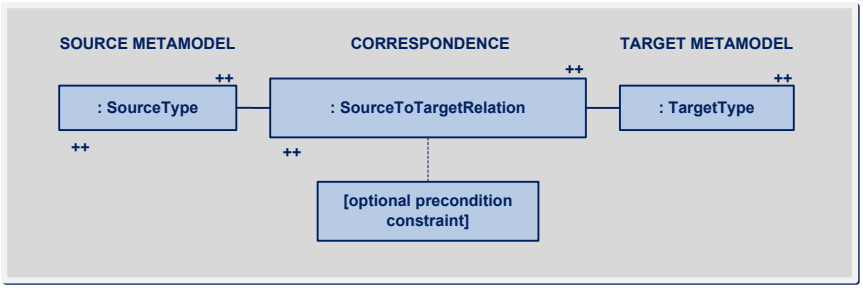


Figure 5.3: TGG Rule Example 1

The rule in Figure 5.3 also has no context nodes, and represents one of the base mapping

rules. If a TGG rule contains context nodes, it becomes a context-dependent rule, applied only when the presence of context nodes is detected in both the source and target. The rule in Figure 5.3 is used as a context rule for sub-elements of the SourceClass and TargetClass, as shown in Figure 5.4.

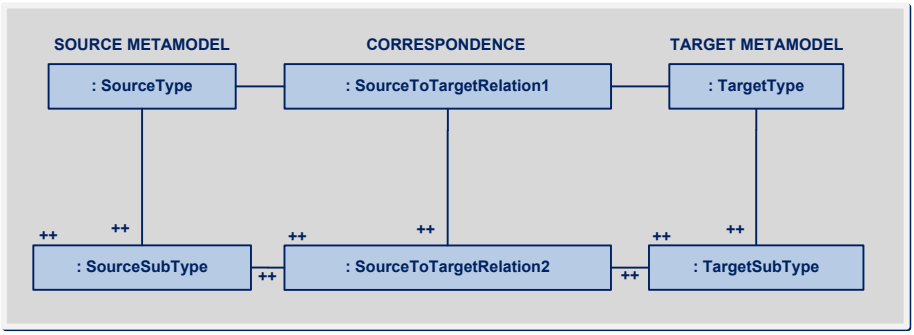


Figure 5.4: TGG Rule Example 2

Now, having a collection of model dependencies represented as profile mappings, we proceed with the process of generating the TGG rules denoting these dependencies. To help illustrate the corresponding algorithm, we shall use a sample mapping as shown in Figure 5.5. In this mapping, three stereotype mappings are included, (Source1, Target1), (Source2, Target2), and (Source3, Target3). The (Source1, Target1) tuple is a container mapping while the other two tuples depend on the (Source1, Target1) mapping as their base/context.

5.2.1 Structured Representation of Profile-Based Dependencies using TGG

To provide a structured representation of profile-based dependencies as TGG rules, we consider four core mapping cases: one-to-one, one-to-many, many-to-one, and many-to-many.

Where applicable, we have chosen to use one correspondence node for each mapping to allow encoding of type-specific mapping constraints (*i.e.*, one constraint for one type-to-type mapping), and to provide more consistent translation to other notations such as QVT. For each core mapping, we consider a denotational function δ_p that encodes the meaning of mapping dependencies D_{MG} between M and G as source-to-target profiles (*e.g.*, as illustrated in the top part of the Figure 5.6). We also consider a denotational function δ_{tgg} that encodes the meaning of mapping profile-based mappings to TGG (*e.g.*, as illustrated in the bottom part of the Figure 5.6).

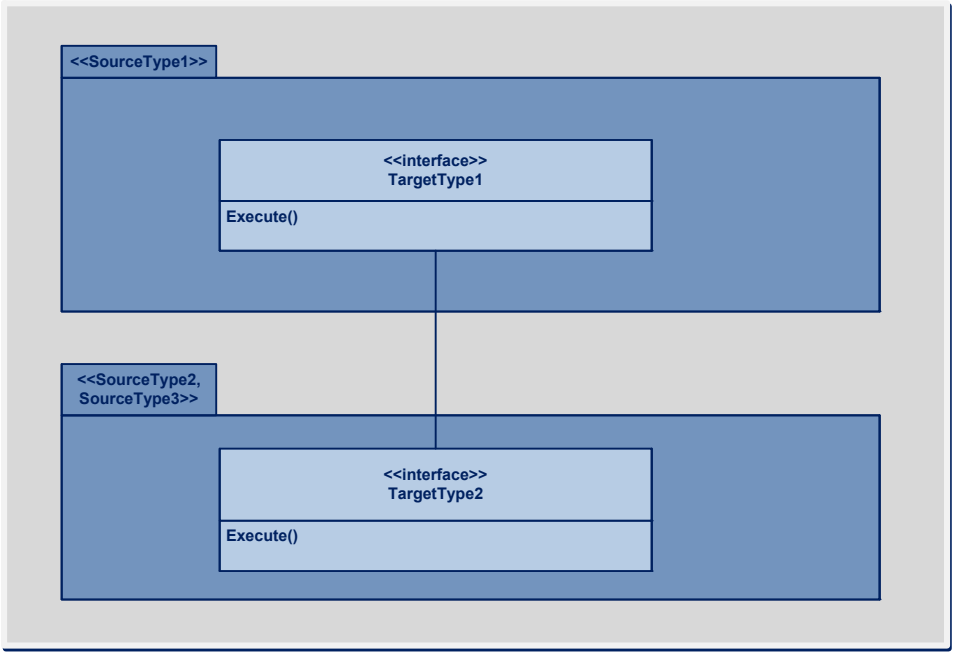


Figure 5.5: Simple Source-to-Target TGG Mapping

In Figure 5.6, we demonstrate how a one-to-one profile-based dependencies are represented as TGG rules. Namely, each profile-based dependency is represented as a TGG

rule, where the source stereotype represents the left-hand side and the target stereotype represent the right-hand side of the corresponding TGG rule. Other constraints, such as applicable OCL constraints, are copied over and they apply to the entire TGG rule. The function δ_p represents a denotational function that can be used to interpret the profile-based dependency in terms of model element dependencies as these were defined in Section 4.4.2. For example, with respect to Figure 5.6, the δ_p represents the semantics of mapping a model element s, which is an instance of SourceType, to a model element t, which is an instance of TargetType, using profiles. Similarly, the function δ_{tgg} represents a denotational function δ_p(D$_{MG}$) that can be used to interpret how we represent profile-based dependencies as TGG rules. For example, regarding the Figure 5.6, the δ_{tgg} represents the semantics of the mapping of the structure of the top of the figure to the structure to the bottom of the figure. The SourceToTarget correspondence node in Figure 5.6 is empty, but can be annotated with specific context-sensitive information that can be used to differentiate on the conditions SourceType is mapped to TargetType. For example, context-sensitive condition may indicate that an element o1 of SourceType will be mapped to an element o2 of TargetType while another context-sensitive condition may dictate that a model element o1 of SourceType into model element o3 of TargetType.

In Figure 5.7, we demonstrate how a one-to-many profile-based dependencies are represented as TGG rules. The main difference here is that each target stereotype is represented as an additional correspondence mapping in the TGG rule. Other constraints, such as applicable OCL constraints, are copied over and they still apply to the entire TGG rule.

In Figure 5.8, we demonstrate how a many-to-one profile-based dependencies are represented as TGG rules. The main difference here is that each source stereotype is represented as an additional correspondence mapping in the TGG rule. Other constraints, such as applicable OCL constraints, are copied over and they still apply to the entire TGG rule.

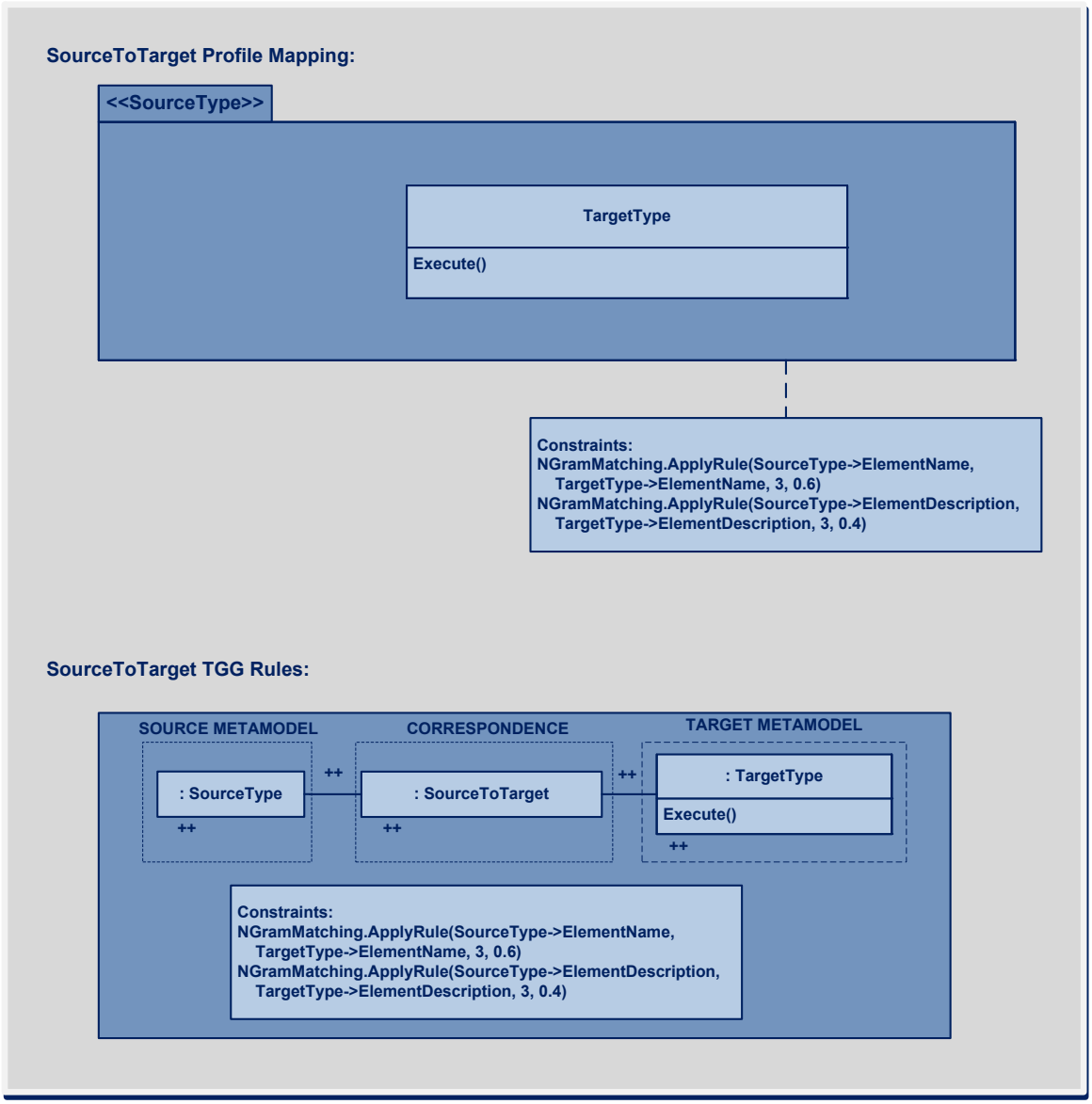


Figure 5.6: The One-to-One Profile-Based Dependencies as TGG Rules

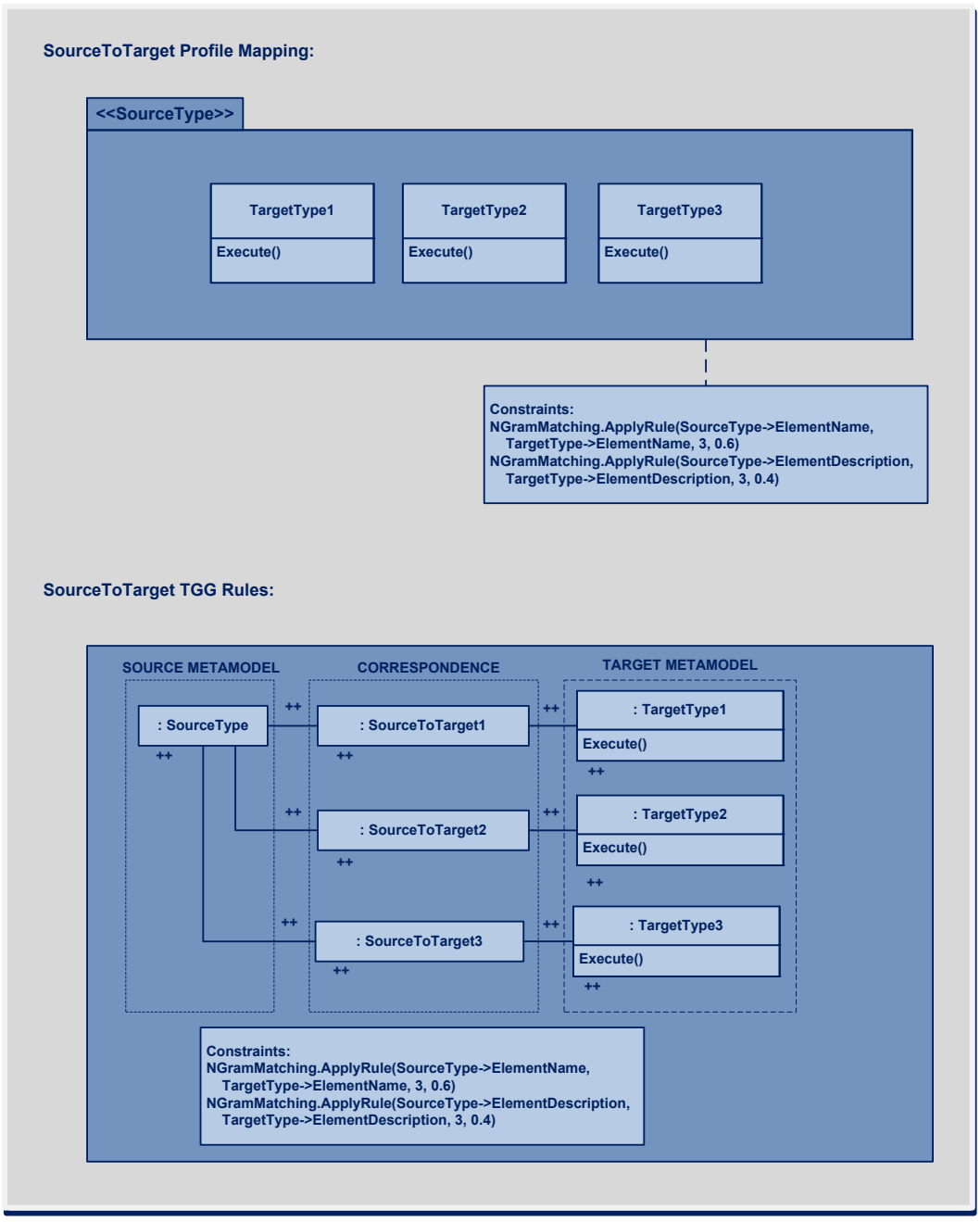


Figure 5.7: The One-to-Many Profile-Based Dependencies as TGG Rules

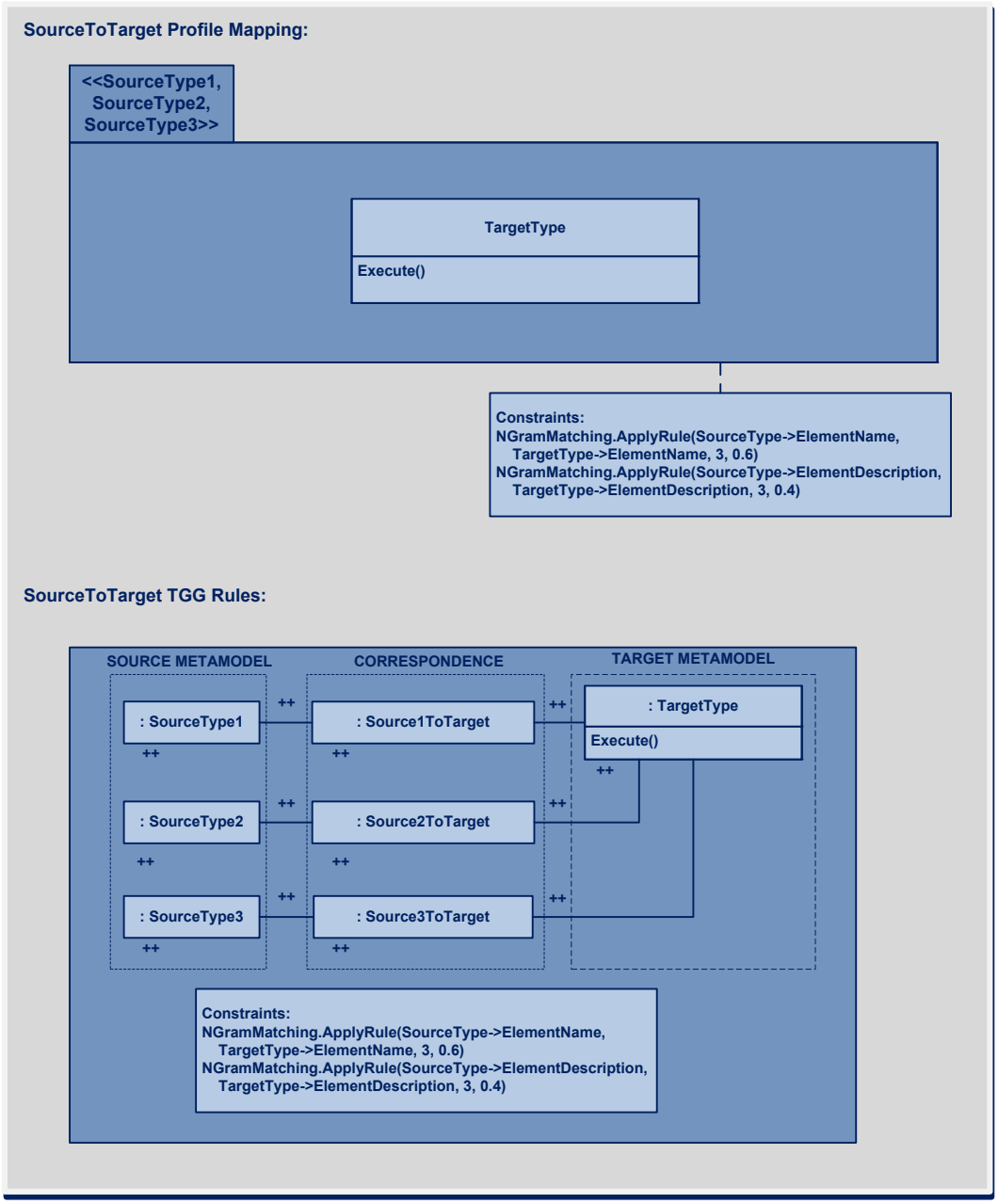


Figure 5.8: The Many-to-One Profile-Based Dependencies as TGG Rules

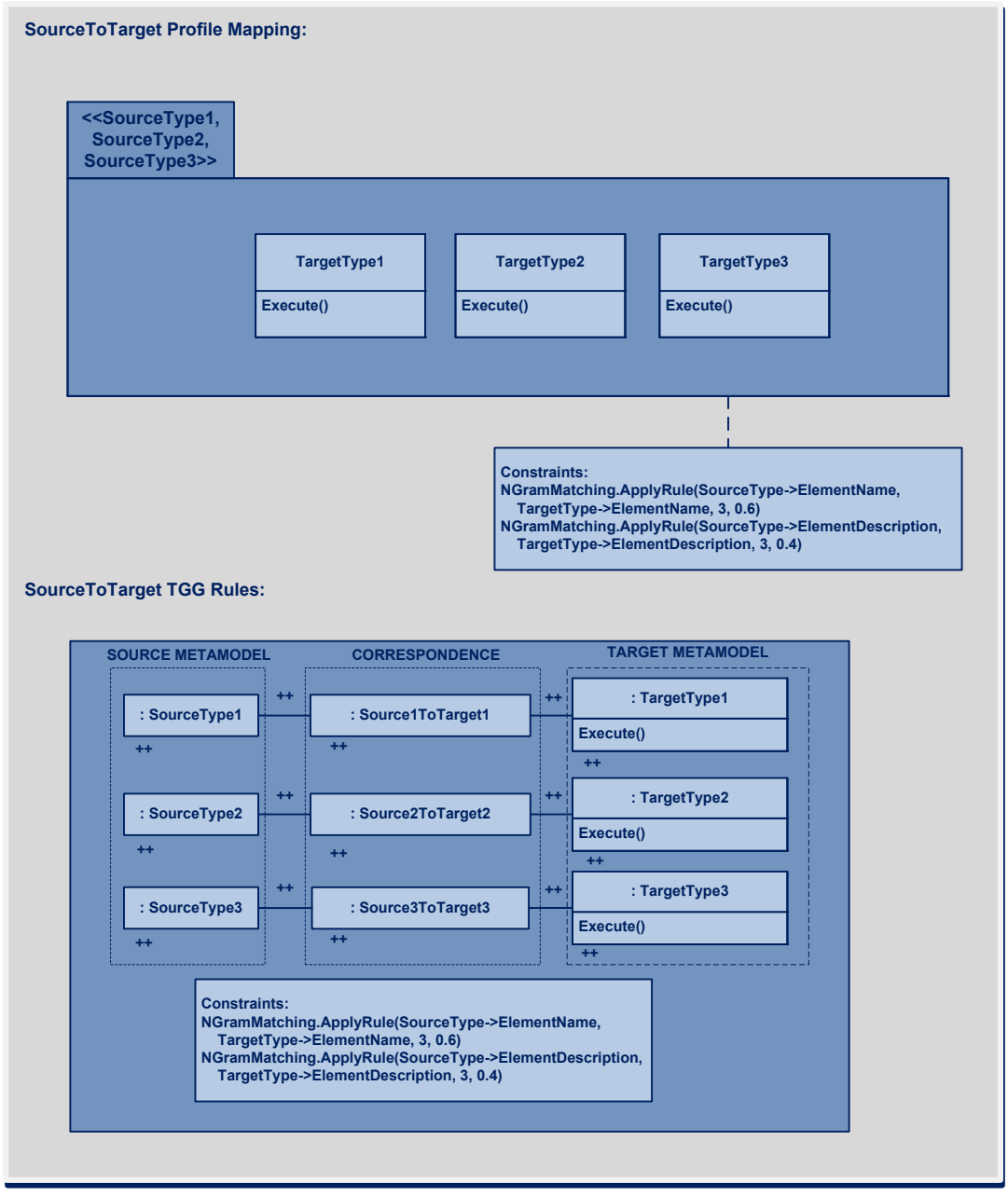


Figure 5.9: The Many-to-Many Profile-Based Dependencies as TGG Rules

In Figure 5.9, we demonstrate how a many-to-many profile-based dependencies are represented as TGG rules. The main difference here is that each source-to-target stereotype mapping is represented as an additional base rule in the overall TGG rule. Other constraints, such as applicable OCL constraints, are copied over and they still apply to the entire TGG rule.

To clarify how the profile-based dependencies are mapped to TGG rules, we apply the TGG-based representation to the illustrative example shown in Figure 4.12. The resulting TGG rules, which are based on the many-to-many profile-based dependencies as TGG rules mapping, are shown in Figure 5.10.

5.2.2 Mapping Profile-Based Dependencies as Triple Graph Grammars

In this section, we outline the algorithm that is used for mapping profile-based dependencies as triple graph grammar rules. The algorithm is composed of five main steps and takes as input the source domain model S, the target domain model T, profile-based model dependencies D_{ST}, and produces as output a triple graph grammar TGG_{ST}. The algorithm is presented in more detail below.

Algorithm MDD-TGG Mapping Profile-Based Dependencies as TGG

Input:

1. Source Domain Model, S
2. Target Domain Model, T
3. Model Dependencies, D_{ST}

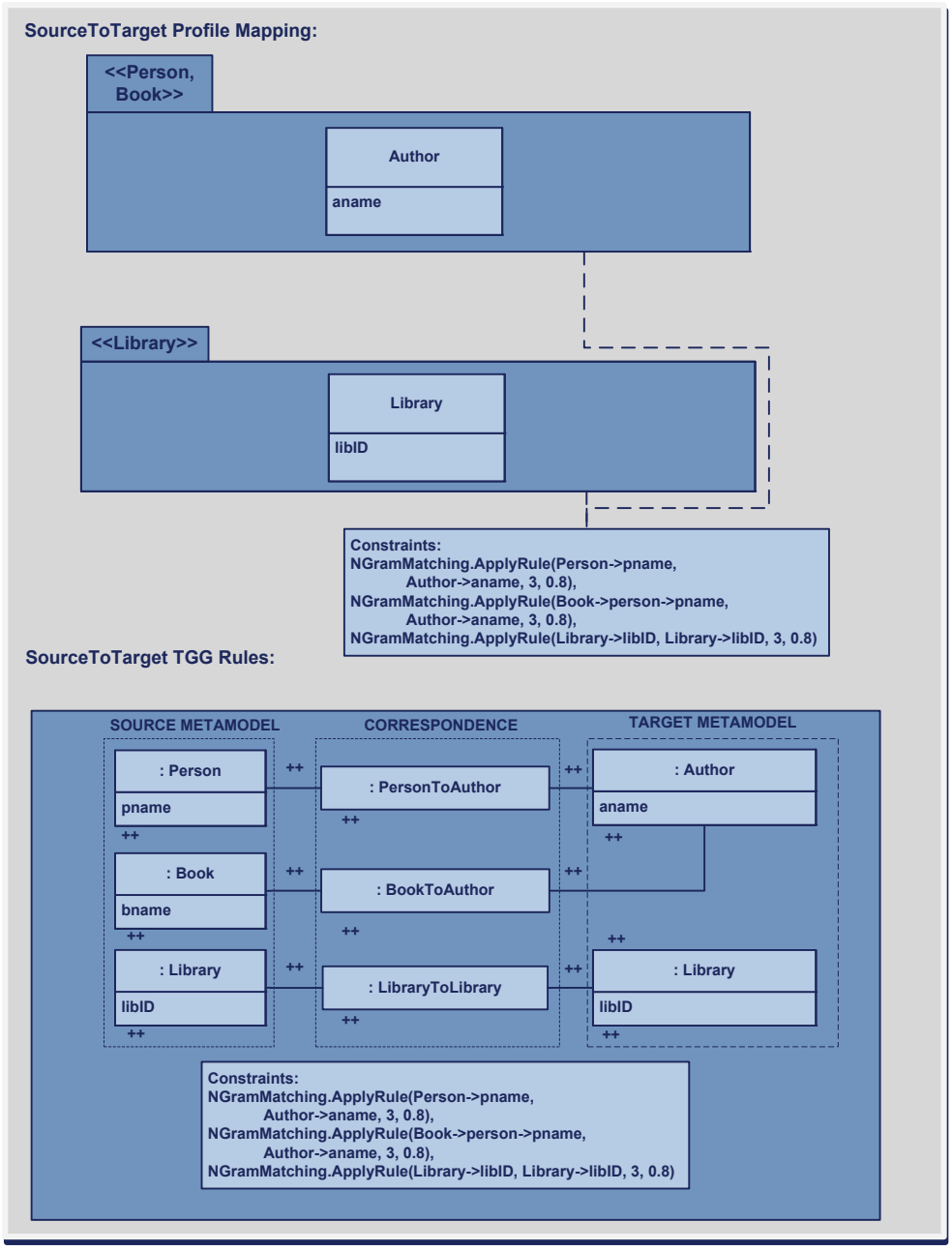


Figure 5.10: Profile-Based Dependencies as TGG Rules Example

Output:

1. Triple Graph Grammar, TGG_{ST}

Steps:

Step 1. Using the DM2DMG algorithm, as defined in Section 3.4.1, represent domain models S and T as domain-model grammars DMG_S and DMG_T, respectively.

Step 2. Using DMG_S and DMG_T as source, let NT be a set of nonterminals defined as a union $\text{NT}_S \cup \text{NT}_T$, let T be a set of terminals defined as a union $\text{T}_S \cup \text{T}_T$, let AX be a starting symbol for derivation equal to AX_S, and let P_S and P_T be enumerated sets of source and target production rules respectively. Also, let D_{ST} be a set of profile-based model dependencies.

Step 3. Iterate through the elements of P_S and P_T and create attributed productions:

Step 3.1 For each $\text{r}_S \in \text{P}_S$, augment the rule by inserting (as its identifier) a semantic head sh_S for the nonterminal on the LHS from a set of applicable semantic heads SH_S (*e.g.*, sh_S r_S).

Step 3.2 For each $\text{r}_T \in \text{P}_T$, augment the rule by inserting (as its identifier) a semantic head sh_T for the nonterminal on the LHS from a set of applicable semantic heads SH_T (*e.g.*, sh_T r_T).

Step 4. Iterate through the elements of D_{ST}:

Step 4.1 For each $d_i \in T_{ST}$, create a new triple graph grammar rule, tgg_i:

Step 4.1.1 Represent the source stereotype or stereotype package as a corresponding production $r_S \in P_S$, and set r_S as the LHS production of the tgg_i rule.

Step 4.1.2 Represent the target package as a corresponding production $r_T \in P_T$, and set r_T as the RHS production of the tgg_i rule.

Step 4.1.3 Create a mapping of semantic heads (sh_S, sh_T) for the two rules, r_S and r_T, and set the mapping as a correspondence node of the tgg_i rule.

Step 4.1.4 Augment the correspondence node with constraint predicates if the association only holds under certain conditions that cannot be specified via attribute matches (*e.g.*, (sh_S, sh_T, {predicates})).

Step 4.1.5 Add tgg_i to the set of TGG productions, TGGP.

Step 5. Output TGG_{ST} = (NT, T, TGGP, AX).

5.2.3 Illustrative Examples

To illustrate application of the algorithm, we utilize the mapping from Figure 5.5. For Step 1, we apply the DM2DMG algorithm as defined in Section 3.4.1.

Input : DM_S based on Figure 5.5.

Step 1-2 :

NT_T := {SourceType1, SourceType2, SourceType3}, NT_R := {Connector}, NT_C := {Connector-Source, ConnectorTarget}, NT_A := {Name, Notes, Constraint}, NT := $NT_T \cup NT_R \cup NT_C \cup NT_A$, T := {alphabet of valid BPM element names for the corresponding Namespace}, AX := M.

Step 3. :

p_1 : M $\rightarrow$ SourceType1 | SourceType1 M | SourceType2 | SourceType2 M | SourceType3 | SourceType3 M | Connector | Connector M

Step 4. :

p_2 : SourceType1 $\rightarrow$ Name Notes

p_3 : SourceType2 $\rightarrow$ Name Notes

p_4 : SourceType3 $\rightarrow$ Name Notes

Step 5. :

p_5 : Connector $\rightarrow$ Name null ConnectorSource ConnectorTarget

Step 6. :

p_6 : ConnectorSource $\rightarrow$ Name Constraint SourceType1 | Name Constraint SourceType2 | Name Constraint SourceType3

p_7 : ConnectorTarget $\rightarrow$ Name Constraint SourceType1 | Name Constraint SourceType2 | Name Constraint SourceType3

Step 7. :

p_8 : Name $\rightarrow$ alphabet of valid element names (values)

p_9 : Notes $\rightarrow$ alphabet of valid notes

p_{10} : Constraint $\rightarrow$ alphabet of valid constraints

Step 8 :

P := {p_1, p_2, ... p_{10}} and DMG_S := (NT, T, P, AX).

Step 9 :

Output DMG and terminate.

Input : DM_T based on Figure 5.5.

Step 1-2 :

NT_T := {TargetType1, TargetType2}, NT_R := {Connector}, NT_C := {ConnectorSource, ConnectorTarget}, NT_A := {Name, Comments}, NT := $NT_T \cup NT_R \cup NT_C \cup NT_A$, T := {alphabet of valid source code element names}, AX := M.

Step 3. :

p_1 : M $\rightarrow$ TargetType1 | TargetType1 M | TargetType2 | TargetType2 M | Connector | Connector M

Step 4. :

p_2 : TargetType1 $\rightarrow$ Name Comments

p_3 : TargetType2 $\rightarrow$ Name Comments

Step 5. :

p_4 : Connector $\rightarrow$ Name null ConnectorSource ConnectorTarget

Step 6. :

p_5 : ConnectorSource $\rightarrow$ Name null TargetType1 | Name null TargetType2

p_6 : ConnectorTarget $\rightarrow$ Name null TargetType1 | Name null TargetType2

Step 7. :

p_7 : Name $\rightarrow$ alphabet of valid element names (values)

p_8 : Comments $\rightarrow$ alphabet of valid comments

Step 8 :

P := {p_1, p_2, ... p_8} and DMG_T := (NT, T, P, AX).

Step 9 :

Output DMG and terminate.

With DMG_S and DMG_T derived, now we can apply the MDD-TGG algorithm.

Input : DM_S, DM_T, T_{ST} based on Figure 5.5.

Step 1. : Created domain-model grammars, DMG_S and DMG_T, as shown above.

Step 2. : NT := $NT_S \cup NT_T$, T := $T_S \cup T_T$, AX:= AX_S.

Step 3.1. : (UML-Type=class)(SourceType1 $\rightarrow$ Name Notes), (UML-Type=class)(SourceType2 $\rightarrow$ Name Notes), (UML-Type=class)(SourceType3 $\rightarrow$ Name Notes).

Step 3.2. : (UML-Type=class)(TargetType1 $\rightarrow$ Name Comments), (UML-Type=class)(TargetType2 $\rightarrow$ Name Comments).

Step 4. :

tgg_1 : LHS := SourceType1 $\rightarrow$ Name Notes, RHS := TargetType1 $\rightarrow$ Name Comments, Correspondence := UML-Type=class (Rule 1 in Figure 5.11).

tgg_2 : LHS := SourceType2 $\rightarrow$ Name Notes, RHS := TargetType2 $\rightarrow$ Name Comments, Correspondence := UML-Type=class (Rule 2 in Figure 5.11).

tgg_3 : LHS := SourceType3 $\rightarrow$ Name Notes, RHS := TargetType2 $\rightarrow$ Name Comments, Correspondence := UML-Type=class (Rule 3 in Figure 5.11).

Step 5. : Output TGG_{ST} = (NT, T, TGGP, AX) (shown in Figure 5.11).

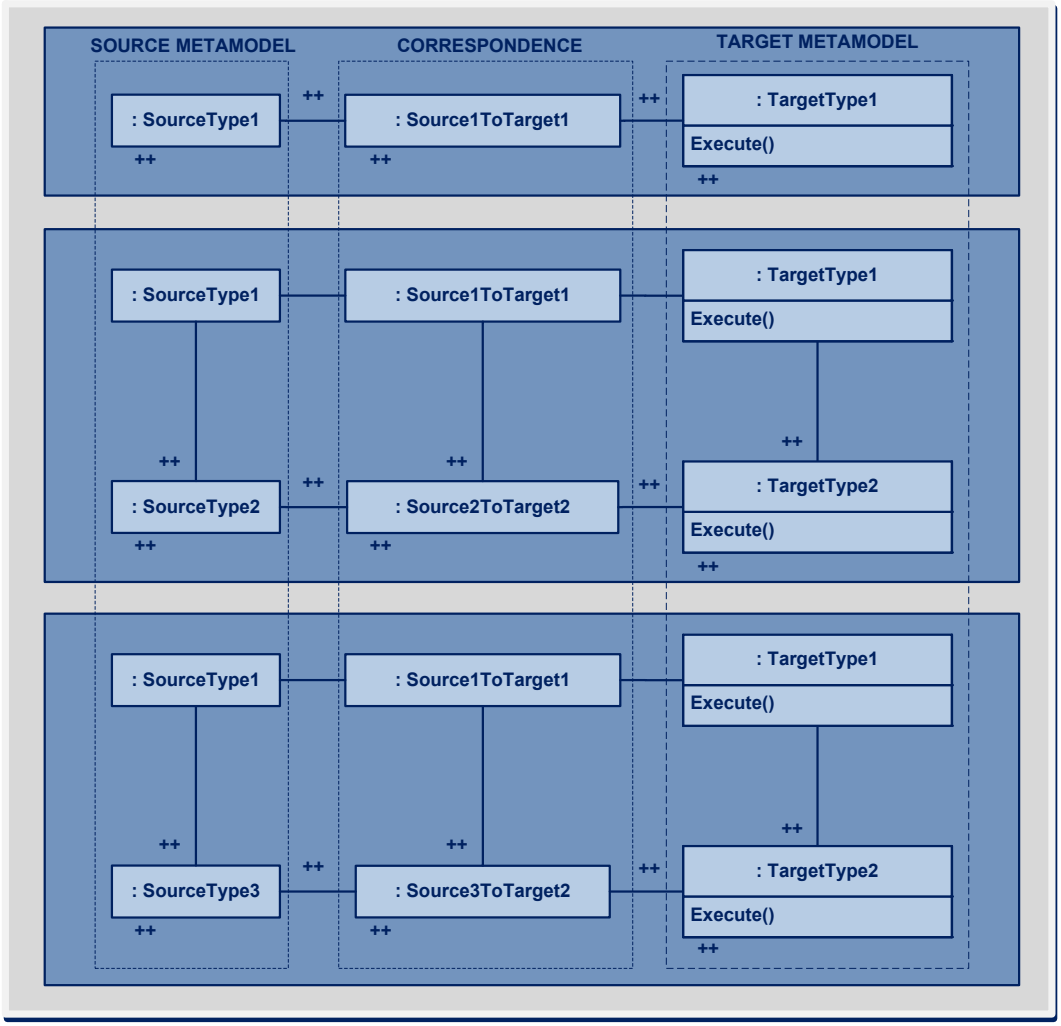


Figure 5.11: TGG Rule Example 3

5.3 Representation of Model Dependencies using QVT

To address interoperability with QVT-compliant CASE Tools, we provide a structured mapping of the derived TGG rules to QVT-based representation. To this end, we address four base mapping cases: one-to-one, one-to-many, many-to-one, and many-to-many.

Figure 5.12 demonstrates how a one-to-one dependency formulated as a TGG rule is mapped to a QVT-compliant transformation. The correspondence node SourceToTarget is represented as a QVT class SourceToTarget, with the left-hand side of the TGG rule representing the left-hand side of the QVT “map” method, and the right-hand side of the TGG rule representing the right-hand side of the QVT “map” method. The applicable OCL constraints are retained as additional constraints for the derived QVT transformations.

Figure 5.13 demonstrates how a one-to-many dependency formulated as a TGG rule is mapped to a QVT-compliant transformation. Each correspondence node, such as SourceToTarget1, is represented as a QVT class with the same name, such as SourceToTarget1. Within each class, the left-hand side of the applicable TGG rule represents the left-hand side of the QVT “map” method, and the right-hand side of the TGG rule represents the right-hand side of the QVT “map” method. The applicable OCL constraints are retained as additional constraints for the derived QVT transformations.

Figure 5.14 demonstrates how a many-to-one dependency formulated as a TGG rule is mapped to a QVT-compliant transformation. Each correspondence node, such as Source1ToTarget, is represented as a QVT class with the same name, such as Source1ToTarget. Within each class, the left-hand side of the applicable TGG rule represents the left-hand side of the QVT “map” method, and the right-hand side of the TGG rule represents the right-hand side of the QVT “map” method. The applicable OCL constraints are retained as additional constraints for the derived QVT transformations.

Figure 5.15 demonstrates how a many-to-many dependency formulated as a TGG rule is mapped to a QVT-compliant transformation. Each correspondence node, such as Source1To Target1, is represented as a QVT class with the same name, such as Source1ToTarget. Within each class, the left-hand side of the applicable TGG rule represents the left-hand side of the QVT “map” method, and the right-hand side of the TGG rule represents the right-hand side of the QVT “map” method. The applicable OCL constraints are retained as additional constraints for the derived QVT transformations.

5.3.1 Illustrative Example

To demonstrate how TGG rules are mapped to QVT-compliant transformations, we make use of the TGG rules shown in Figure 5.10. The resulting QVT transformations, which are based on the many-to-many TGG rules as QVT transformations mapping, are shown in Figure 5.16.

5.4 Evaluation

In this chapter, we have mapped profile-based dependencies to triple graph grammars, to provide model synchronization capabilities and to operationalize these capabilities with a run-time enjoinment such as QVT. By extending the profile-based semantics of the profile-based model dependencies, we have provided a more precise way of encoding model-to-model dependencies. Furthermore, by making use of the TGG rules, which were shown to be in correspondence with the QVT modeling syntax, we have extended the applicability of the profile-based representation by making it QVT compliant. Now, it is possible for the profile-based dependency mappings to be represented in QVT syntax, and used in a growing number of CASE tools that support QVT modeling notation.

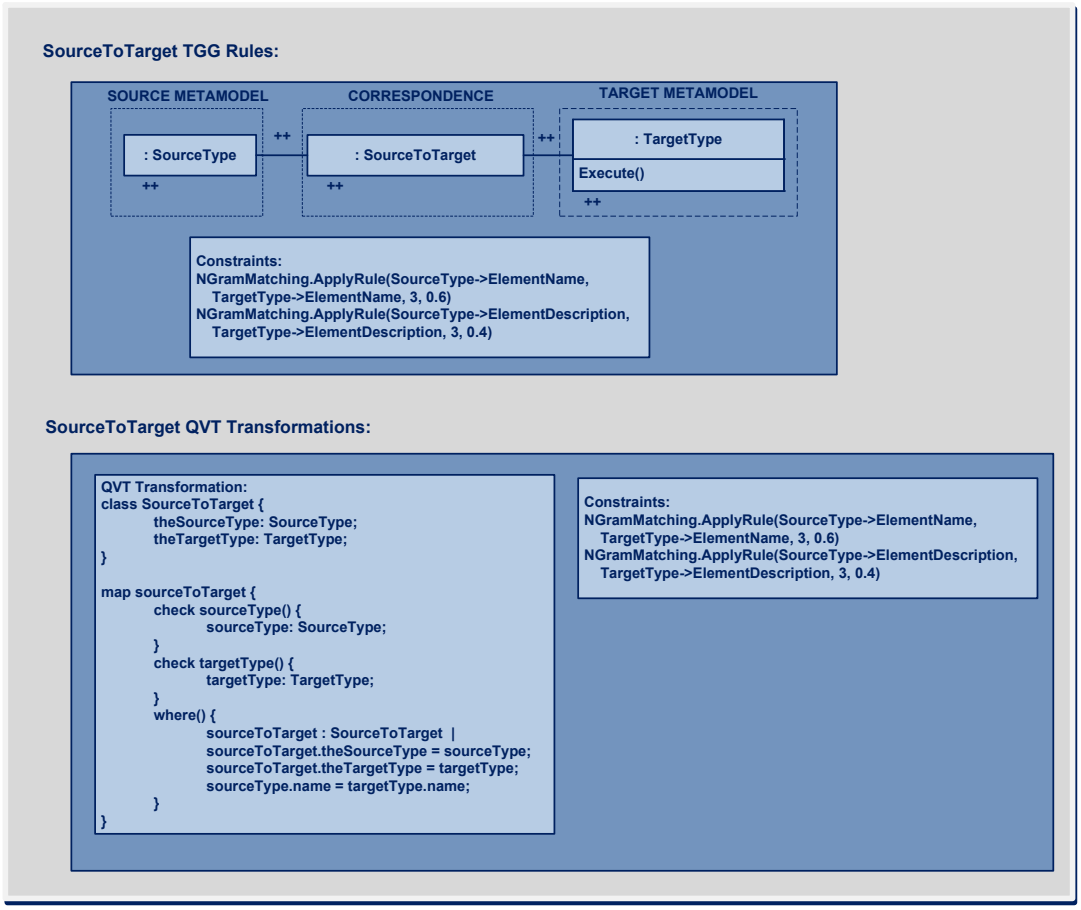


Figure 5.12: The One-to-One TGG Rules as QVT Transformations

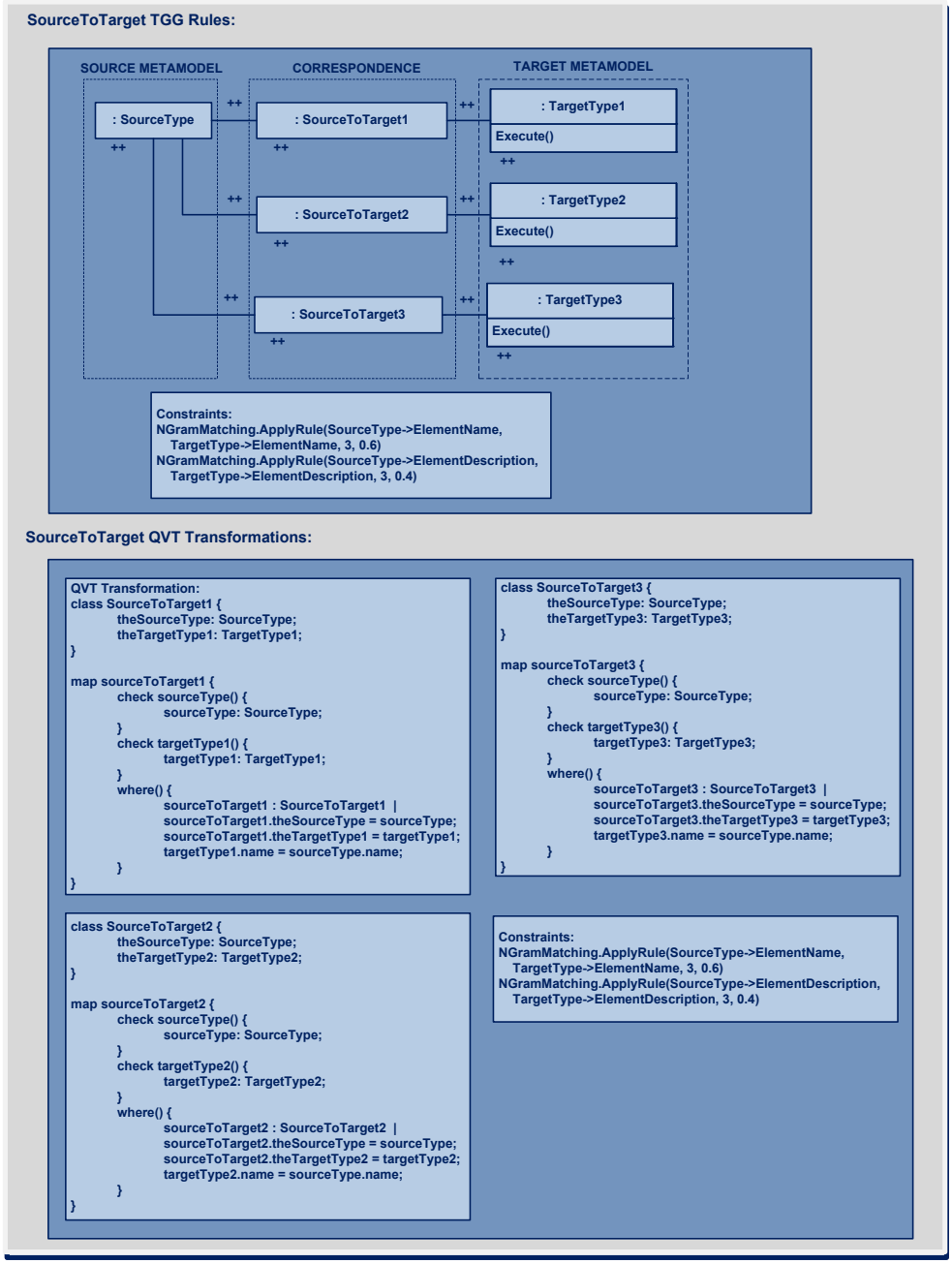


Figure 5.13: The One-to-Many TGG Rules as QVT Transformations

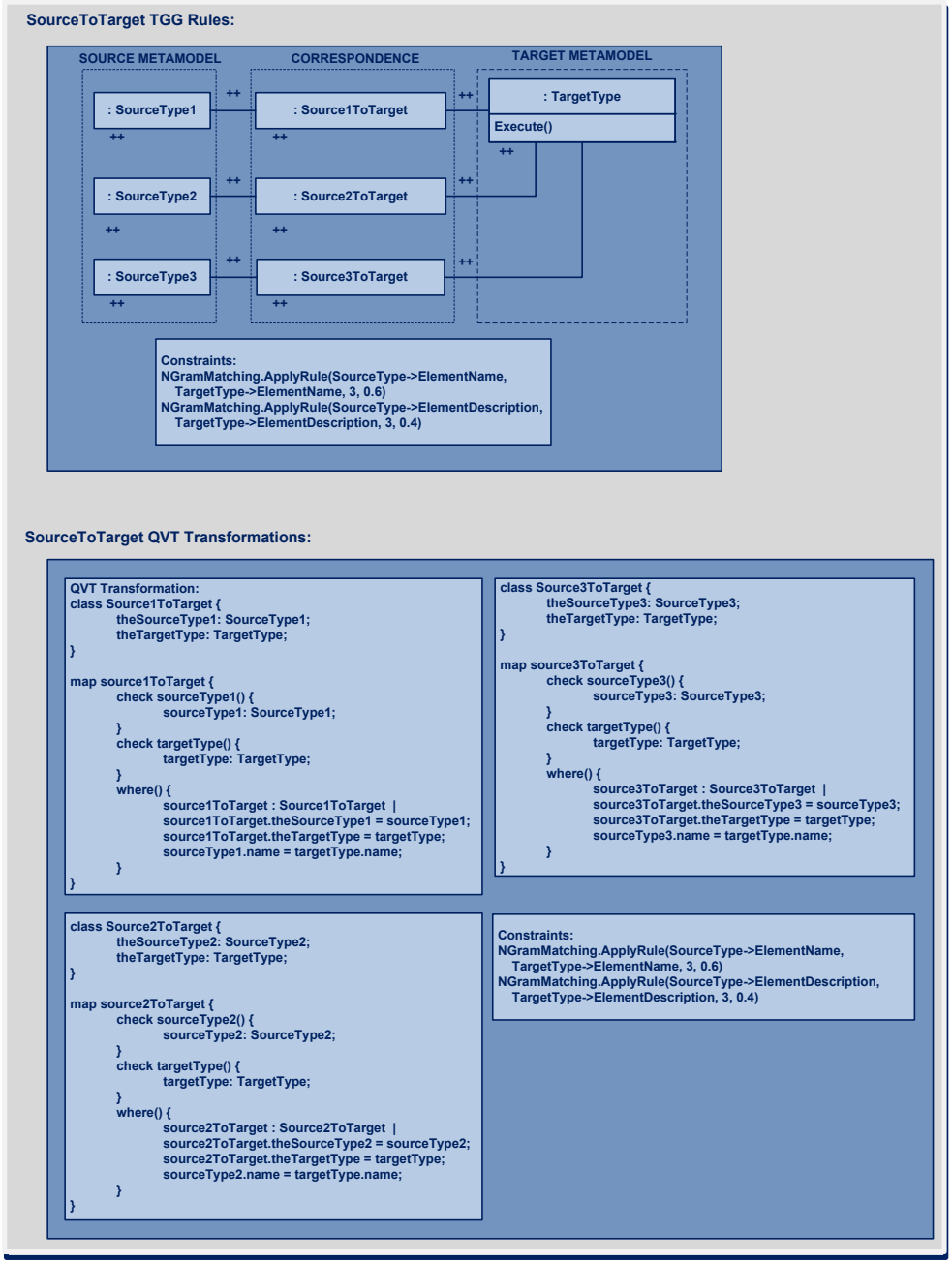


Figure 5.14: The Many-to-One TGG Rules as QVT Transformations

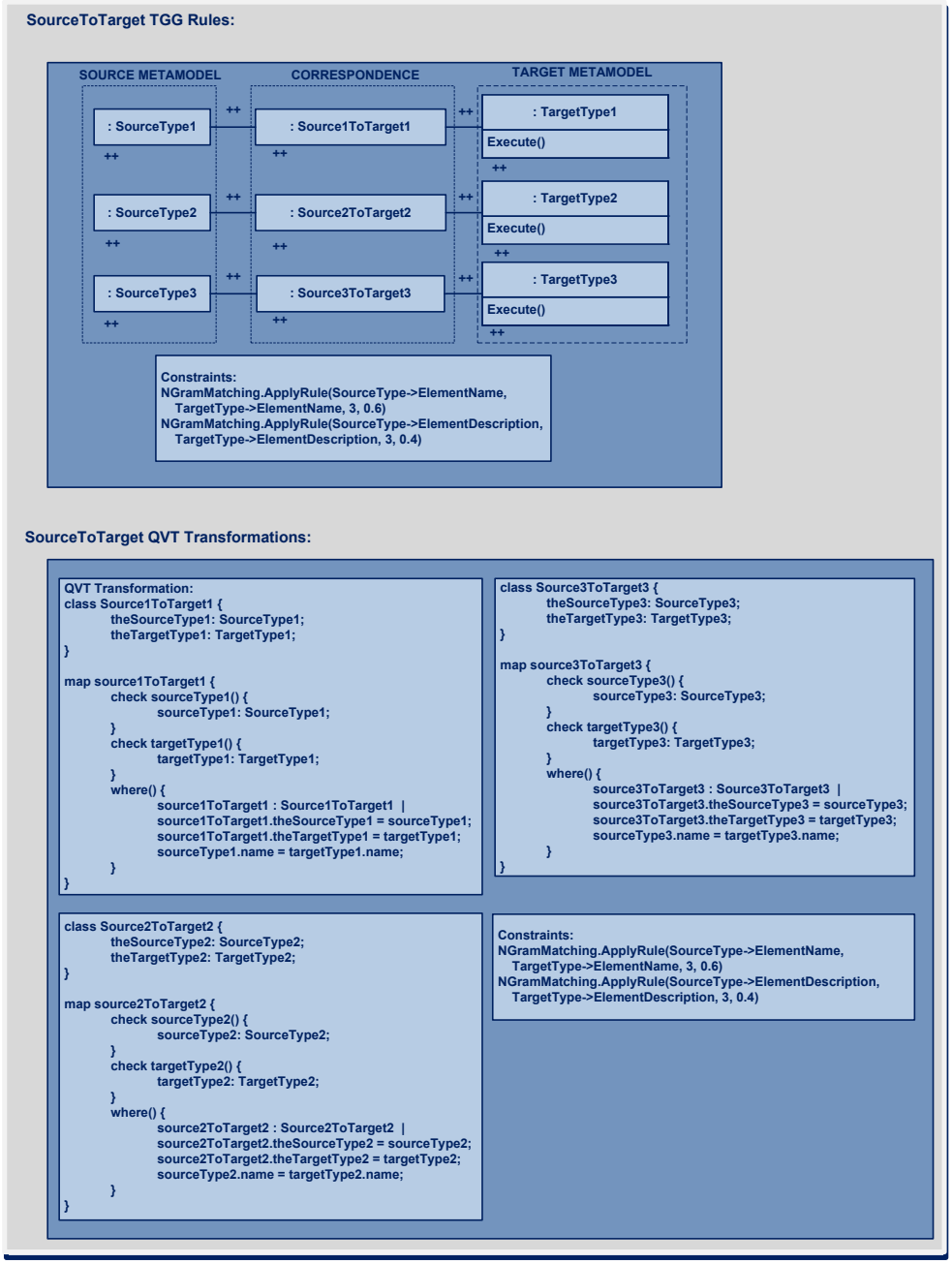


Figure 5.15: The Many-to-Many TGG Rules as QVT Transformations

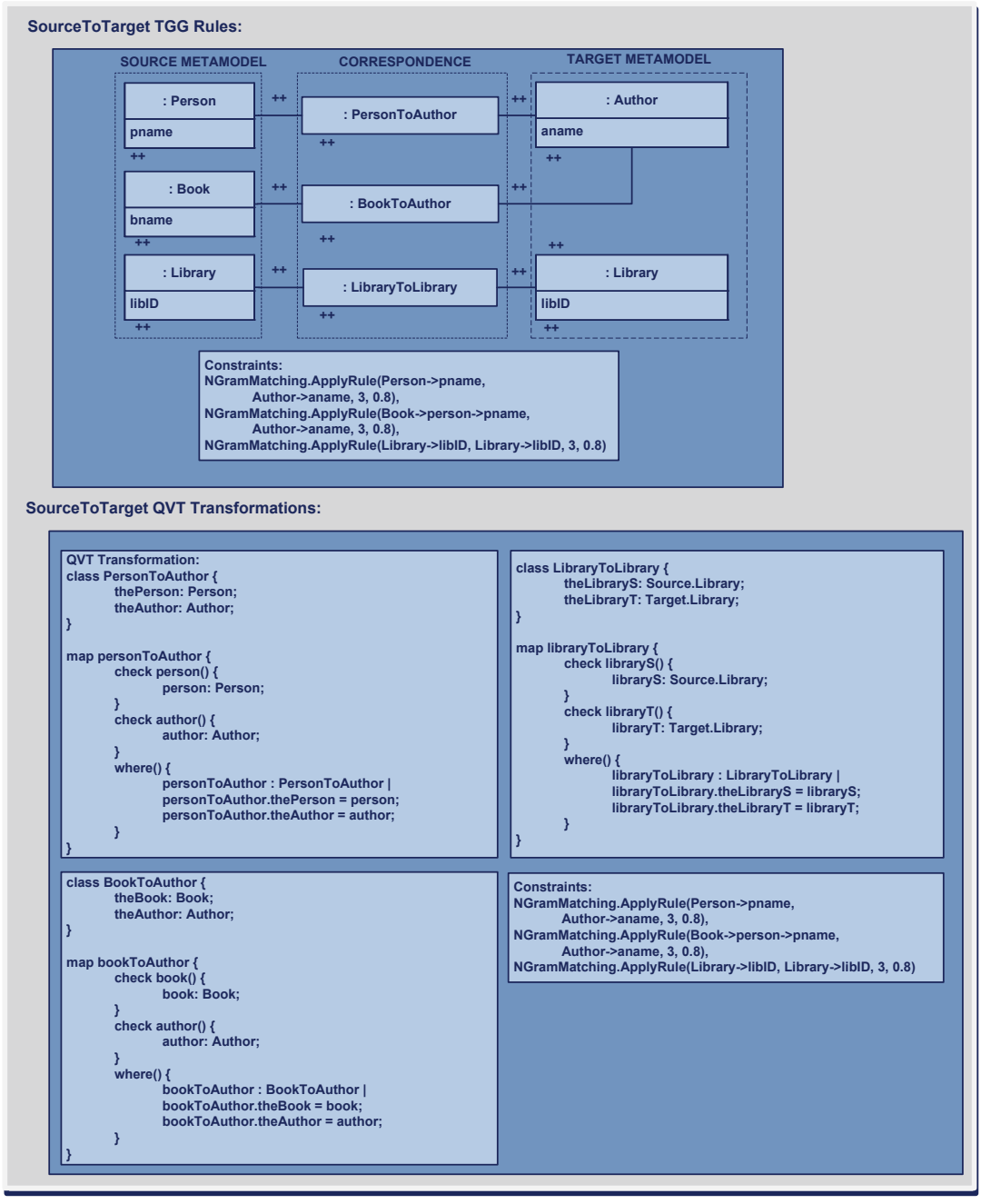


Figure 5.16: TGG Rules as QVT Transformations Example

We now examine the proposed approach with respect to the following properties: completeness and soundness, as part of the evaluation.

5.4.1 Completeness

Property *All for types of model dependencies D that are represented using profiles can also be represented using TGG rules*

Proof: The proof is by construction. We consider that there exist four types of dependencies, one-to-one, one-to-many, many-to-one, and many-to-many. By the construction algorithm, any such type of dependency can be represented as a TGG rule. If we assume that there exists a dependency d, where d $\in$ D that cannot be represented using TGG rules, the dependency d is then not compliant with any of the one-to-one, one-to-many, many-to-one, and many-to-many base cardinality rules. According to the definition of a model dependency tuple (see Section 4.4.2), one-element tuples such as zero-to-one or zero-to-many are not allowed (*i.e.*, they have no semantic meaning), so the dependency tuple has to comply with one of the four base cardinality rules. This is a contradiction, so it follows that any dependency tuple d that follows the definition from the Section 4.4.2 can be represented using profiles.

5.4.2 Soundness

Property *All model dependencies represented using TGG rules are sound representations of the corresponding profile-based dependency mappings*

Proof: Let us consider a denotational function δ_p that maps the source profile-based dependency to target profile-based dependency (*e.g.*, as shown in the top part of the Figure

5.6). Let us also consider a denotational function δ_{tgg} that maps the source TGG rule to target TGG rule (*e.g.*, as shown in the bottom part of the Figure 5.6). The function δ_p(d$_{ST}$,C), where S is the collection of source types, T is the collection of target types, d$_{ST}$ are the corresponding model dependency tuples, and C is the collection of constraints, represents the semantics of the profile-based representation of mappings between S and T under constraints C. More specifically, δ_p(d$_{ST}$,C) = $\delta(S \underset{C}{\rightarrow} T)$, where δ denotes the meaning of transforming model S to model T under constraints C. That is, $\delta(S \underset{C}{\rightarrow} T)$ = $S \underset{C}{\rightarrow} T$. On the other hand, δ_{tgg}(T$_S$,T$_T$,Corr,C) where T$_S$ is the rule representing the collection of source model elements S, T$_T$ is the rule representing the collection of target model elements T, Corr is the correspondence node in the TGG rule, and C is a collection of constraints, represents the semantics of the triple graph grammar representation of mappings under constraints C. More specifically, δ_{tgg}(T$_S$,T$_T$,Corr,C) = $\delta(T_S) \underset{C,Corr}{\rightarrow} \delta(T_T)$ where δ is the denotional function that provides the meaning of the representation of model S as a context-free grammar as discussed in Chapter 3. That is, $\delta(T_S) \underset{C,Corr}{\rightarrow} \delta(T_T)$ = $\delta(S) \underset{C,Corr}{\rightarrow} \delta(T)$ = $S \underset{C}{\rightarrow} T$. The last equality relation is based on the determinism property of representing domain models as domain model grammars, and vice versa, as described in Section 3.4.5.

Therefore, it follows that the profile-based dependencies represented using a denotational function δ_p can be soundly represented using TGG rules represented via a denotational function δ_{tgg}.

5.5 Chapter Summary

In this chapter, we have introduced an approach for mapping profile-based dependencies using Triple Graph Grammar (TGG) rules with corresponding OCL constraints. We have shown an algorithm that given a source domain model S, target domain model T, and model

dependencies D, outputs corresponding triple graph grammar (TGG) rules. Finally, we have shown how to represent the TGG rules in a QVT-compliant notation. In the next chapter, we will demonstrate application of this approach to experimentation scenarios.

Chapter 6

Application Case Study

> *To attempt seeing Truth without knowing Falsehood. It is the attempt to see the Light without knowing the Darkness. It cannot be.*
>
> *— Frank Herbert*

In this chapter, we present case study results used to assess the proposed theory. Namely, we first discuss the case studies conducted to confirm the applicability of our formal concept analysis approach, which was introduced in Chapter 4. Secondly, we discuss several application scenarios for our TGG based approach, as discussed in Chapter 5, and demonstrate the applicability of the approach. Finally, we present measurements such as the precision and recall information for the FCA approach, and the corresponding memory and run-time performance discussion.

6.1 Dependency Extraction Case Study

We have implemented a prototype of our framework in Java, and we have integrated it with the previously developed mSynTra plug-in for the Eclipse development environment [Fou10].

Figure 6.1 depicts the prototype as a Workflow Synchronization Plug-In (WSP) Perspective in Eclipse. The dependency view shows two model repositories, the BPMs on the left and the source code models on the right. Clicking on one of the model elements on the left will show the corresponding dependencies selected on the right, with the details of the match including positively matched rules, the weighted score, and other possible matches in the middle. Additional interface is provided for reparsing of model elements based on the changed criteria such selecting or deselecting particular model attributes, and increasing or decreasing the threshold level for the weighted score. Through these settings users can change the size of the domain of matching, and hence, iteratively improve the accuracy of the results.

The prototype is used in a case study of synchronizing business process models (BPM) represented as business workflows with the enacting Java 2 Enterprise Edition (J2EE) platform compliant source code [Ora10]. Based on the domain analysis and discussion with different stakeholders, it was concluded that the business workflows are typically created independently of source code. It was also noted that the mappings between the related models are not consistently recorded, and as a result, they are incomplete and out of date. Thus, developers and architects of the system would be required to validate the extracted model dependencies.

The main target for the case study is to enable bidirectional change propagation, where changes from a workflow are traced to underlying source code and changes from a source code file are traced to related workflows. The first step in this process is to establish relations

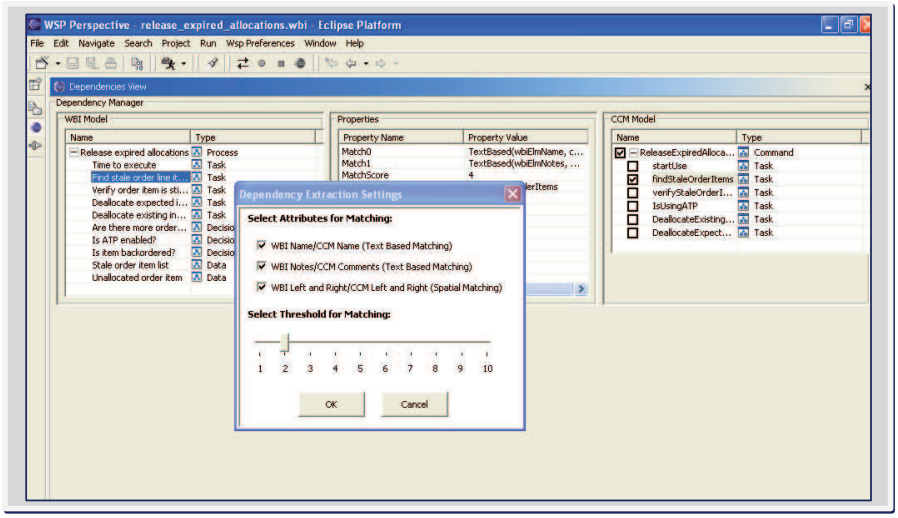


Figure 6.1: Prototype Implementation in Eclipse

among models and model elements, and the approach presented in this paper is used to fulfill the requirements of this goal.

In the ensuing text, we describe how each of the steps of the approach is applied. Namely, we discuss the process of recovery of intermediate models to bridge syntactic and semantic gaps between the workflows and the source code. The derived intermediate models are used as a basis for extraction of inter-context concepts, derived as clusters of objects that share related attributes. The extracted dependencies are validated based on the corresponding feedback, from which the precision and recall levels are also measured.

6.1.1 Generating intermediate models

In this step, we transform the business workflows by enriching and annotating their representations. We also transform the source code models through abstraction and annotation.

The workflow transformation includes:

1. Analyzing the workflow models and deriving their simplified representation in relation to source code artifacts. The workflow functionality that is not performed by a machine along with elements that have no source code representation are excluded from the analysis. However, the data flow and control flow information is preserved. The call information for processes and run-time modules, which can be found in the source code, is also preserved.

2. Adding annotations as attributes to workflow elements for which additional information may be derived. The annotations may include attributes for context, role, hierarchical relation, implemented features, *etc*.

3. Extracting content from workflows automatically into XML, with the domain model schema as a DTD for the XML files.

The source code transformation includes:

1. Analyzing the source code models and deriving their simplified representations in relation to business process artifacts. The source code functionality that has no relation to business processes along with elements that have no workflow representation are excluded. The data flow and control flow information is preserved. The call information for task commands, pseudo task commands, and Java Beans is also preserved.

2. Adding annotations as attributes to source code elements for which additional information may be derived. The annotations may include attributes for parameter passing, informal text, hierarchical relation, implemented features, *etc*.

3. Extracting content from source code files automatically into XML, with the domain model schema as an XML DTD.

6.1.2 Establishing model dependencies

From the refined domain models, the attribute association rules may be created. For the BPM domain model, type-based and hierarchical associations are created. The types are mapped to the compatible source code types, while processes are recognized as parts of the overall hierarchy and relations are established based on directly or indirectly implemented features. For the source code domain model, hierarchical associations could not be established in addition to the type-based ones since no meaningful hierarchy of source-code models could be identified. For the concrete models, the compatible attributes are identified and spatial and text-based association rules are defined. The settings for each rule are adjusted through initial experimentation on a representative set of related workflow and source code models. Using FCA and the association rules, dependency tuples are identified and the results are stored in XML.

Figure 6.2 shows the mapping of the BPM and the source code attributes and properties. Different levels of mapping, from Level 0 to Level 3, indicate different steps in the matching process: at Level 0, hierarchical and type-based clustering of top-level elements (*i.e.*, processes and classes); at Level 1, clustering of top-level elements using specified attributes; at Level 2, clustering of model elements using specified attributes; and at Level 3, matching of unmatched elements using spatial information.

Figure 6.3 demonstrates a successful mapping between a workflow (at the top) and a corresponding source code model (at the bottom). As part of this figure, two spatial association rules are presented in OCL.

	comparing	Business Flow	Source Code
Level 0:	**Objects**	A set of workflow processes	A set of Java source code files
	Properties	Hierarchical and type-based associations	Type-based associations
Level 1:	**Objects**	Workflow process	A set of corresponding Java classes
	Attributes	Process name, process notes	Class name, top-level comments
Level 2:	**Objects**	Elements of the workflow process	Elements of the source code flow
	Attributes	Element name, type, description, data input and output, spatial info	Element name, type, comments, method params, spatial info
Level 3:	**Objects**	Unmatched elements of the workflow process	Unmatched elements of the source code flow
	Attributes	Spatial information	Spatial information

Figure 6.2: Attribute and Property Mappings

6.1.3 Validating established dependencies

Since the previously established model relations are not available, or are, at best, incomplete, the validation process is based on feedback from developers and architects. The feedback is used to identify all relevant model dependencies, and to confirm the relevance of the identified model dependencies. Several iterations are performed, each with the goal of improving precision and recall levels through adjustment of corresponding settings (*e.g.*, change of attribute associations, addition of new rules).

The estimated precision is measured as:

> Estimated Precision = the number of relevant model dependencies that were identified / the number of all model dependencies that were identified.

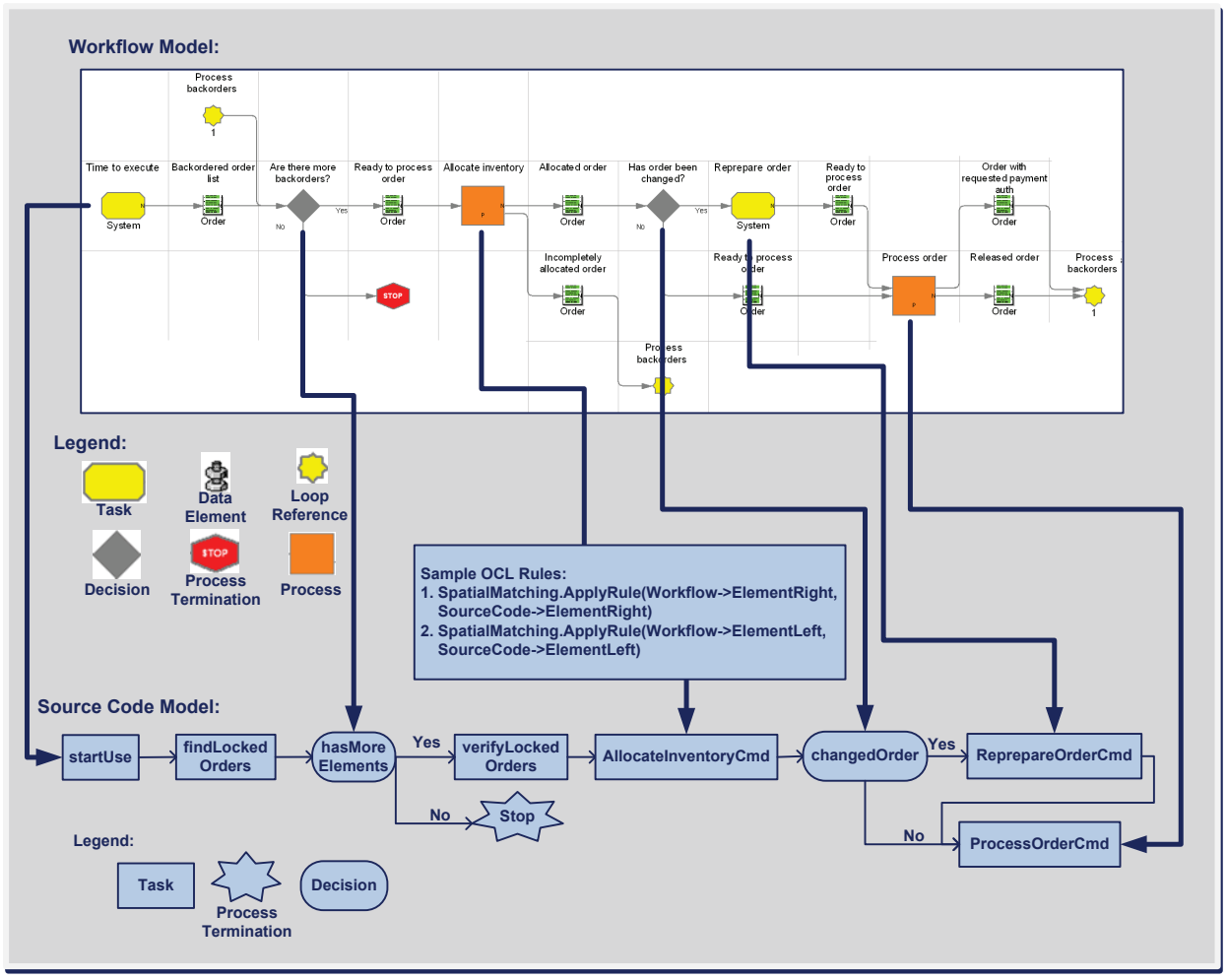


Figure 6.3: Dependency Mapping Example

The estimated recall is measured as:

> Estimated Recall = the number of relevant model dependencies that were identified / the number of all known relevant model dependencies.

Based on the set of four queries, for the subsystems that were available to us, we have measured and analyzed the precision and recall levels for the top-level matches derived using our framework.

For the first and smallest data set, 9 out of 34 identified model dependencies were relevant (26.47% estimated precision level) while 9 out of 15 of all known relevant model dependencies were identified (60% estimated recall level). This data set also exhibited the highest level of nonconformity between related workflows and source code flows.

For the second data set, 37 out of 140 identified model dependencies were relevant (26.43% estimated precision level) while 37 out of 51 of all known relevant model dependencies were identified (72.55% estimated recall level).

For the third and largest data set, on the first iteration 78 out of 300 identified model dependencies were relevant (26% estimated precision level) while 78 out of 100 of all known relevant model dependencies were identified (78% estimated recall level). On the second iteration, after corresponding association rules adjustments, 82 out of 317 identified model dependencies were relevant (25.9% estimated precision level) while 82 out of 100 of all known relevant model dependencies were identified (82% estimated recall level).

Figure 6.4 illustrates the estimated precision (X axis) versus estimated recall levels (Y axis). Based on the plotted data, there is a noticeable decline in recall levels from approximately 80% to approximately 60% with the precision slightly increasing but remaining close to 26%.

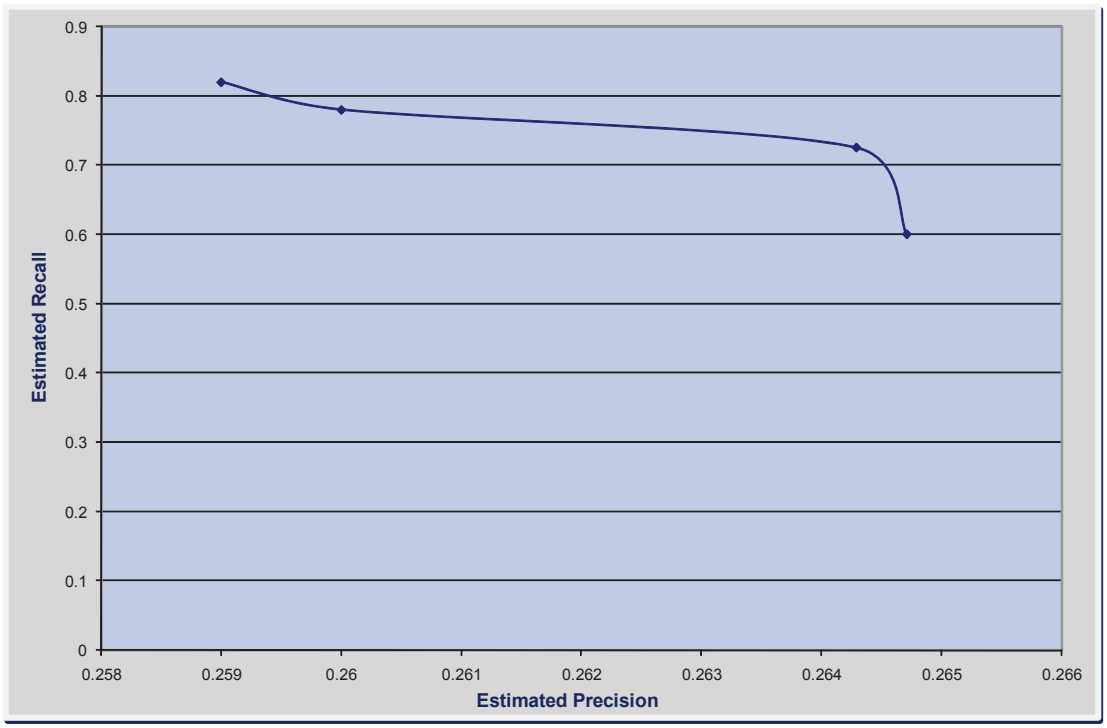


Figure 6.4: Estimated Precision and Recall

The decline in recall levels may depend on several factors including:

- attrition of information when considering source code elements that have indirect or partial relations to their business workflow counterparts,
- nonconformity of certain related workflows and source code flows, and
- inconsistency and drift that may have occurred over time between related workflow and source code models.

6.2 TGG Generation Case Study

To evaluate the applicability of the TGG generation method presented in Chapter 5, we provide a more complex profile mapping as shown in Figure 6.5. Here we identify the following tuples (Service, (ComponentService, RegistryService, RegistryComponent ServiceImpl, RegistryServiceBeanBase)), (Operation, (BusinessObjectDocumentCmd, BusinessObject DocumentCmdImpl, UpdateRegistryCmd, UpdateRegistryCmdImpl)), (Task, (BODCmdRootInterface, ReportInvalidRegistryDataCmd, ReportInvalidRegistryDataCmdImpl, ValidateRegistry ValueCmd, ValidateRegistryValueCmdImpl)).

The first tuple, (Service, (. . .)) makes use of EJB-based target model to represent a source Service element. We would like to encode this selection as a constraint, and make that part of the matching TGG rule. In Figure 6.6, we demonstrate this more precise rule. This constraint is also domain-specific, as it makes use of a domain-specific attribute, Service.useEJB.

To further elaborate this constraint semantics, we also map the second tuple (Operation, (. . .)) as a TGG rule, and add a multiplicity constraint. Namely, we dictate that an Operation element can be mapped to a corresponding pattern shown in Figure 6.5 only once; that is, only one instance of BusinessObjectDocumentCmd can be created. We illustrate this rule in Figure 6.6. The first rule from that Figure serves as a base rule, representing the context for the second rule.

Finally, we map the third tuple (Task, (. . .)) as a TGG rule. We illustrate this rule in Figure 6.6. The second rule from that Figure serves as a base rule, representing the context for the third rule.

Now, we again consider a denotational function δ_p that denotes the mapping of the source profile-based dependency to target profile-based dependency, and a denotational function

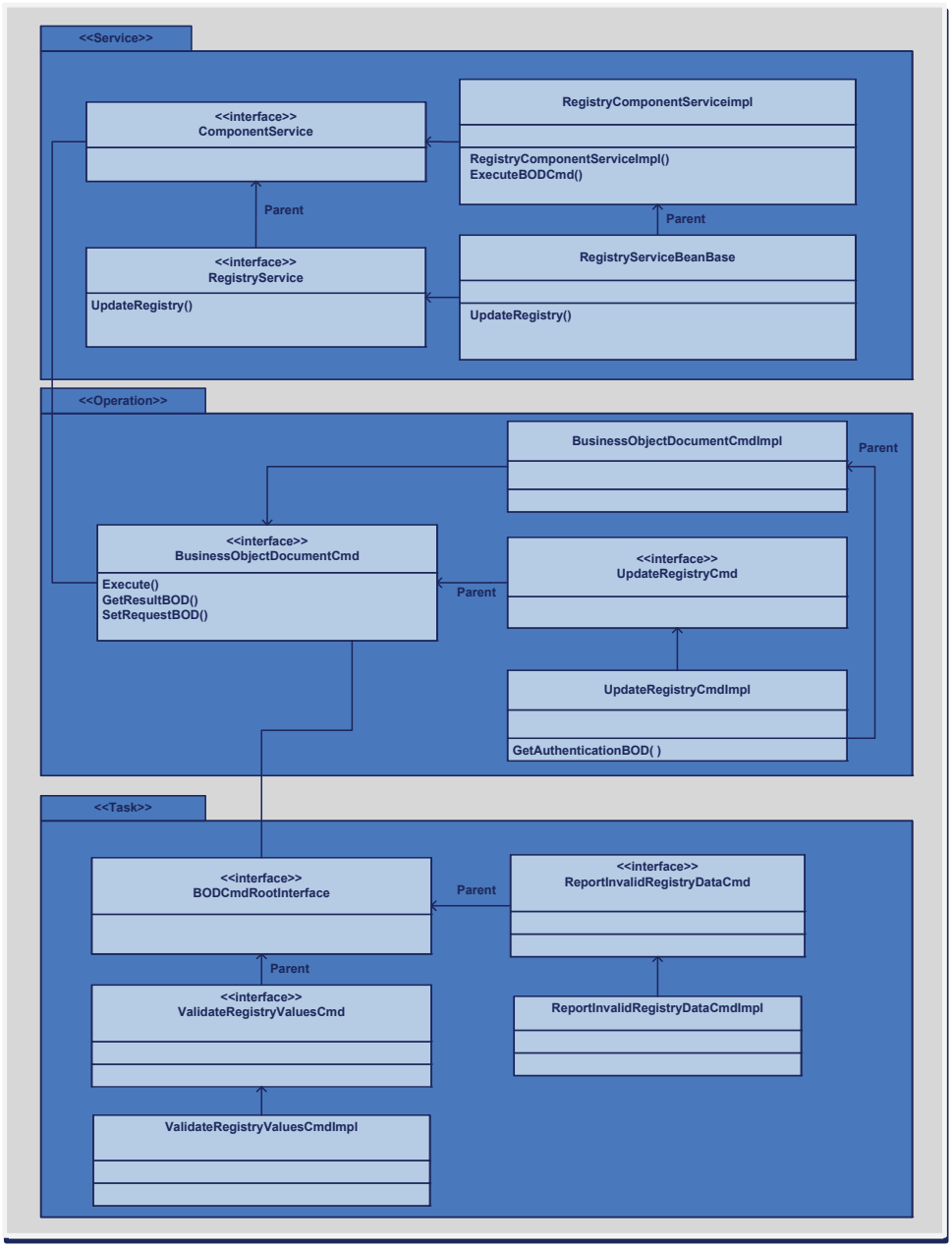


Figure 6.5: TGG Generation Application Case Study: Source

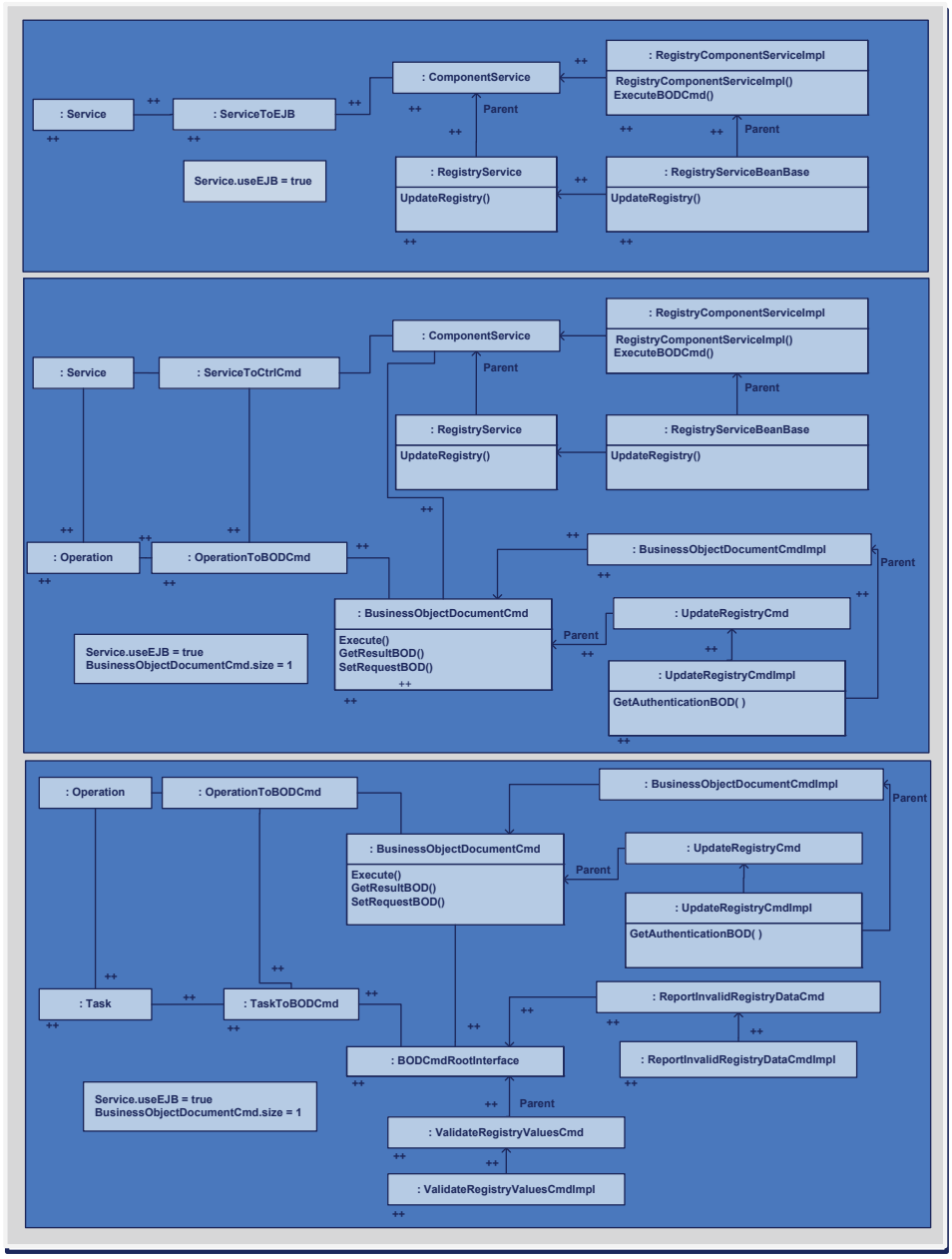


Figure 6.6: TGG Generation Application Case Study: Target

δ_{tgg} that denotes the mapping of the source TGG rule to target TGG rule. From the first rule of Figure 6.5, the function δ_p represents a mapping where for each s1, which is an instance of Service, there exists tmap1, which is an instance of the (ComponentService, RegistryService, RegistryComponent ServiceImpl, RegistryServiceBeanBase) pattern. Similarly, from the first rule of Figure 6.6, the function δ_{tgg} represents a mapping where for each s2, which is an instance of Service, there exists tmap2, which is an instance of the (ComponentService, RegistryService, RegistryComponent ServiceImpl, RegistryServiceBeanBase) pattern.

From the second rule of Figure 6.5, the function δ_p represents a mapping where for each o1, which is an instance of Operation, there exists tmap3, which is an instance of the (BusinessObjectDocumentCmd, BusinessObject DocumentCmdImpl, UpdateRegistryCmd, UpdateRegistryCmdImpl) pattern. Similarly, from the second rule of Figure 6.6, the function δ_{tgg} represents a mapping where for each o2, which is an instance of Operation, there exists tmap4, which is an instance of the (BusinessObjectDocumentCmd, BusinessObject DocumentCmdImpl, UpdateRegistryCmd, UpdateRegistryCmdImpl) pattern.

From the third rule of Figure 6.5, the function δ_p represents a mapping where for each t1, which is an instance of Task, there exists tmap5, which is an instance of the (BODCmdRootInterface, ReportInvalidRegistryDataCmd, ReportInvalidRegistryDataCmdImpl, ValidateRegistry ValueCmd, ValidateRegistryValueCmdImpl) pattern. Similarly, from the third rule of Figure 6.6, the function δ_{tgg} represents a mapping where for each t2, which is an instance of Task, there exists tmap6, which is an instance of the (BODCmdRootInterface, ReportInvalidRegistryDataCmd, ReportInvalidRegistryDataCmdImpl, ValidateRegistry ValueCmd, ValidateRegistryValueCmdImpl) pattern.

Therefore, based on these three equivalences, it follows that these two functions represent the equivalent semantics for the models represented in Figures 6.5 and 6.6.

Through this case study, we have demonstrated the application of our TGG generation approach to a complex practical case study. We have demonstrated the robustness of the approach to handle a greater variety of mapping scenarios, and illustrated additional semantic expressiveness through domain-specific and multiplicity-based constraints.

6.3 Fine-Grained Model Synchronization Case Study

To demonstrate and evaluate our approach to fine-grained model synchronization using TGG rules, we study the problem of synchronizing business process models with the underlying source code models. Specifically, we are interested to keep business process models and source code synchronized when one of these models changes due to evolution and maintenance activities.

Our case study is an industrial size system that consists of a hierarchy of business processes, for which the functionality is enacted through Java and EJB source code components. Element types of the business workflows that are related to source code include process, subprocess, task, decision, choice, and data (see Figure 6.7).

These element types are mapped to the elements of the source code domain model for controller commands, which include task command, task, and decision abstractions at the source code level (see Figure 6.8). Controller commands represent the first point of invocation and they, in turn, invoke task commands, tasks and decisions as needed.

To demonstrate how fine-grained model synchronization is applied to this case study, we have created two TGG rules, as shown in Figure 6.9.

The first rule (tgg1) indicates that for each instance of type Process in the source model, there should exist one instance of type ControllerCommand in the target model.

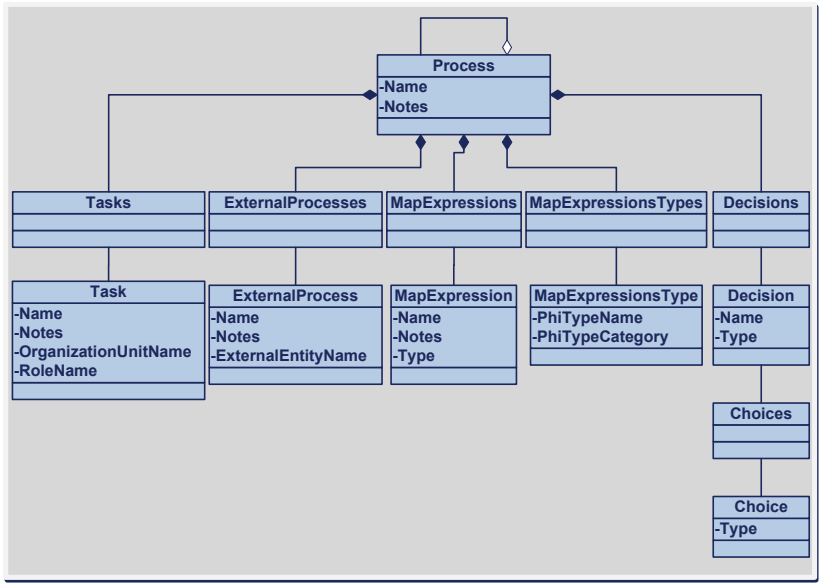


Figure 6.7: Business Process Domain Model

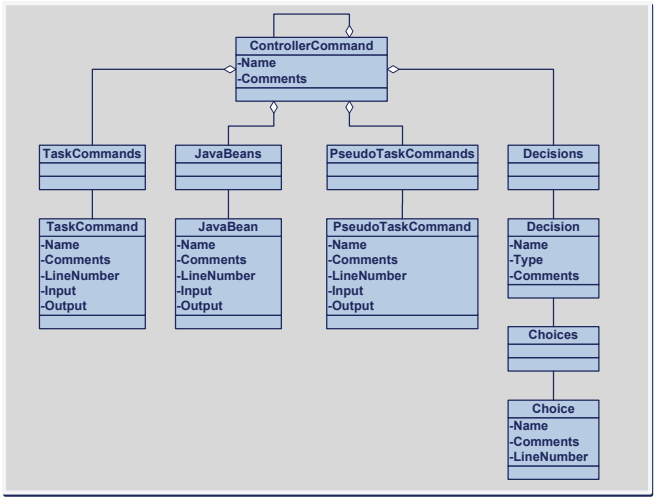


Figure 6.8: Source Code Domain Model

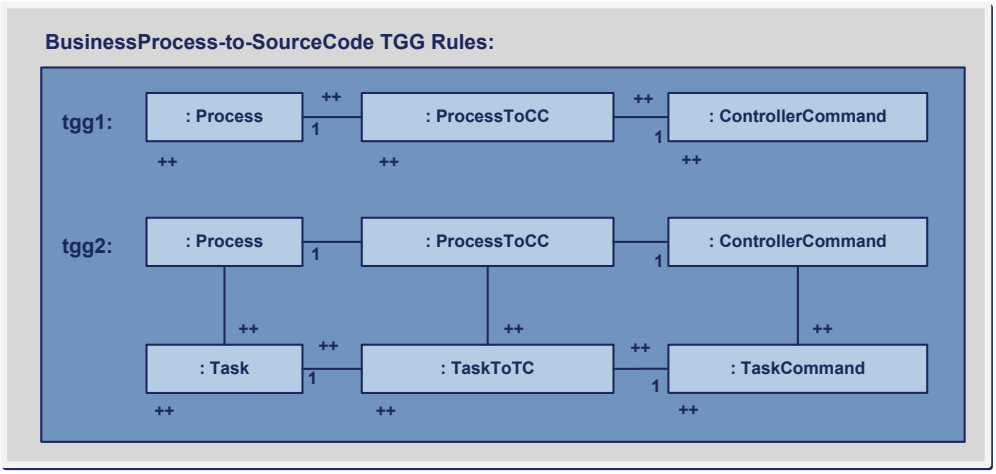


Figure 6.9: BusinessProcess-to-SourceCode TGG Rules

The second rule (tgg2) indicates that for each instance of type Task that is connected to an instance of type Process, there should exist one instance of type TaskCommand in the target model that is connected to an instance of type ControllerCommand.

To illustratable the process of synchronization, we consider the following scenario (see Figure 6.10):

Step1. An instance p1 of type Process is inserted into a source model source1.0. The target model target1.0 is empty at the start. Through pattern matching of the source, the TGG rule tgg1 is identified as applicable, and applied to the source and target. The target model is modified to reflect the rule, which enforces that for each instance of type Process in the source model there should exist one instance of type ControllerCommand in the target model. The result of the synchronization is the target model target1.1, which contains an instance cc1 of type ControllerCommand.

Step2. An instance t1 of type Task is inserted into the source model source1.0, and con-

nected to the previously inserted instance p1 of type Process, thereby resulting in the modified source model source1.1. Through pattern matching of the source, the TGG rule tgg2 is identified as applicable, and applied to the source and target. The target model is modified to reflect the rule, which enforces that for each instance of type Task that is connected to an instance of type Process, there should exist one instance of type TaskCommand in the target model that is connected to an instance of type ControllerCommand. The result of the synchronization is the target model target1.2, which contains an instance tc1 of type TaskCommand connected to instance cc1 of type ControllerCommand.

Step3. An instance t2 of type Task is inserted into the source model source1.1, and connected to the previously inserted instance p1 of type Process, thereby resulting in the modified source model source1.2. Through pattern matching of the source, the TGG rule tgg2 is identified as applicable, and applied to the source and target. The target model is modified to reflect the rule. The result of the synchronization is the target model target1.3, which contains an instance tc2 of type TaskCommand connected to instance cc1 of type ControllerCommand.

Step4. An instance tc3 of type TaskCommand is inserted into the target model target1.3, and connected to the previously inserted instance cc1 of type ControllerCommand, thereby resulting in the modified target model target1.4. Through pattern matching of the source, the TGG rule tgg2 is identified as applicable, and applied to the source and target. The source model is modified to reflect the rule. The result of the synchronization is the source model source1.3, which contains an instance t3 of type Task connected to instance p1 of type Process.

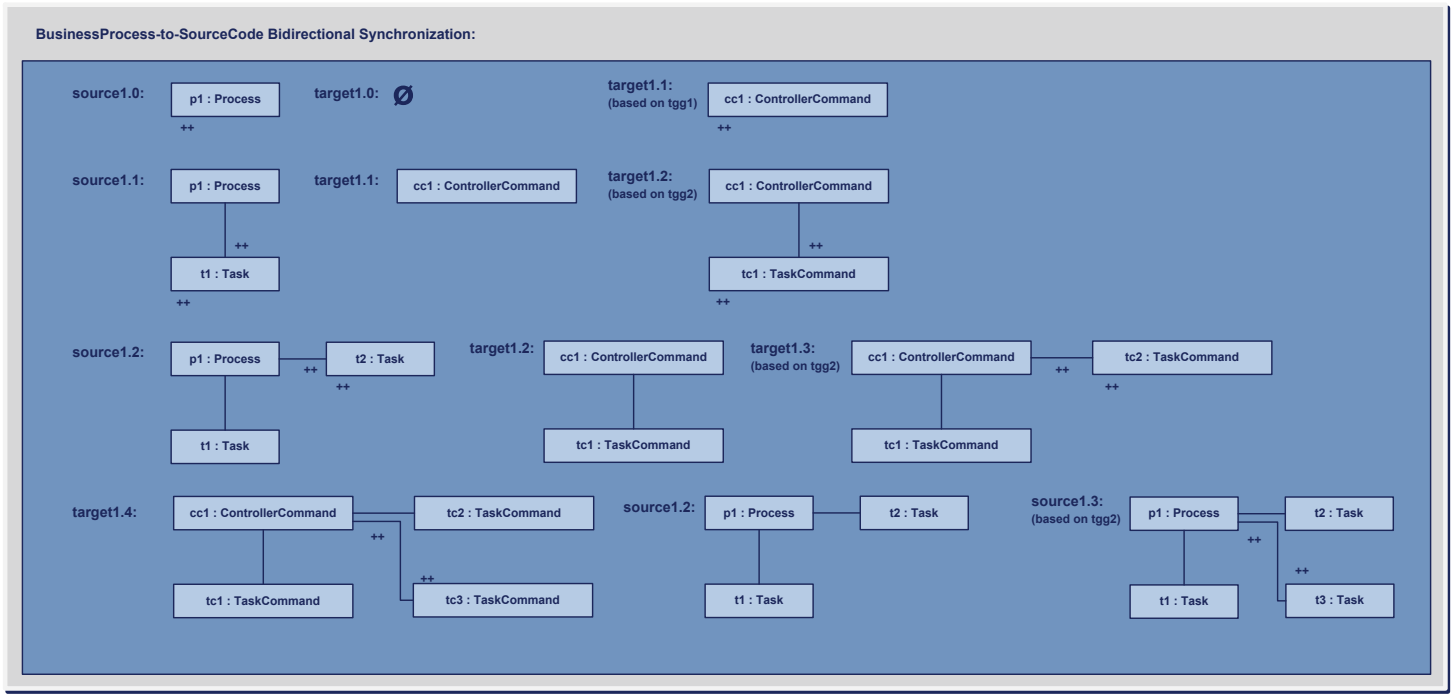


Figure 6.10: BusinessProcess-to-SourceCode Model Synchronization

Through this scenario, we have demonstrated the applicability of the proposed fine-grained model synchronization approach to an example case study of synchronizing business process models with the underlying source code. Even though the case study is small, it is illustrative as it encompasses all cases of one-to-one, one-to-many, and many-to-many synchronization using the corresponding TGG rules.

6.4 Computational Efficiency Discussion

In this section, we present the memory and run-time performance discussion, for each of the experimentation sections.

6.4.1 Dependency Extraction: Computational Efficiency

To analyze run-time and memory performance of the framework, we have performed a set of four subsystem queries, and we have included the results as Figure 6.11.

The run-time performance was measured in seconds (X axis) versus number of element combinations per model (Y axis) that were parsed in the matching process.

- The first query required 8.01 seconds to process 3,740.43 element combinations per model.
- The second query required 168.28 seconds to process 22,587.19 element combinations per model.
- The third query required 212.18 seconds to process 45,220.03 element combinations per model.

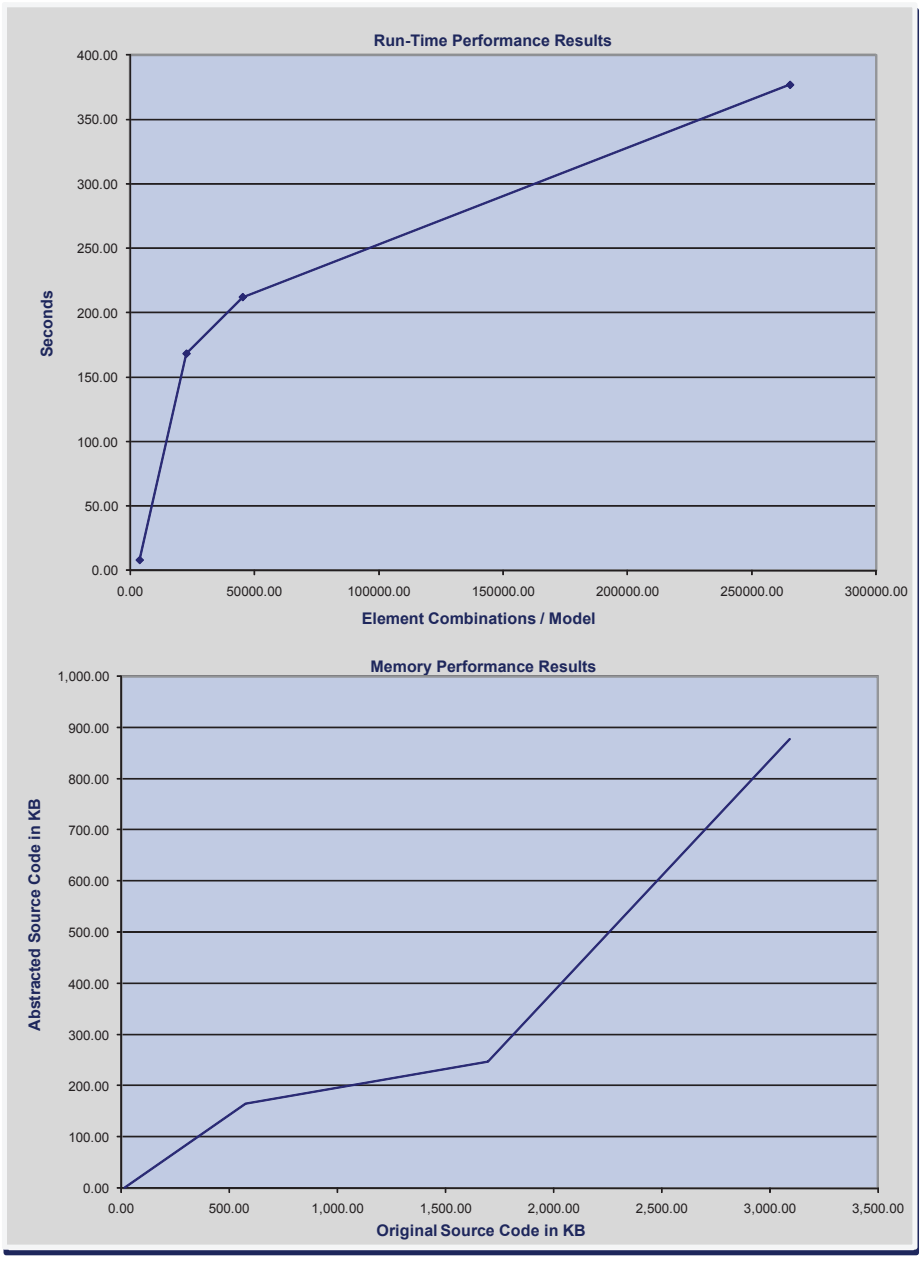


Figure 6.11: Run-Time and Memory Performance Results

- The fourth query, which included combined data set of models from all the available subsystems, required 376.93 seconds to process 265,414.80 element combinations per model.

The plotted results, as shown in Figure 6.11, indicate an approximately logarithmic O(log(n)) runtime performance.

The memory performance was measured in kilobytes of source code (X axis) versus kilobytes of abstracted source code (Y axis).

- The first data set contained 15.95 kilobytes of source code versus 1.94 kilobytes of abstracted source code.
- The second data set contained 574.60 kilobytes of source code versus 165.28 kilobytes of abstracted source code.
- The third data set contained 1,695.85 kilobytes of source code versus 247.95 kilobytes of abstracted source code.
- The fourth data set contained 3,091.95 kilobytes of source code versus 877.21 kilobytes of abstracted source code.

The plotted results, as shown in Figure 6.11, indicate an approximately linear O(n) memory performance.

6.4.2 Fine-Grained Model Synchronization using TGG: Computational Efficiency

To generate TGG rules, we iterate through profile-based dependencies DP_{MG}, and create a new triple graph grammar rule, tgg_i, for each dependency $t_i \in DP_{MG}$. As such, we estimate the average runtime and memory performance to generate the required TGG rules as O(n), where n is the number of profile-based dependencies.

The proposed framework relies on TGG rules for synchronization. Within the defined scope, we do not use Negative Application Conditions (NAC) [SK08], that is conditions that forbid specific graph patterns to be present before or after applying a specific TGG rule. However, such patterns can be specified within the framework as additional constraints.

Hence, we can utilize the runtime and memory performance of efficient TGG translator, as discussed in [KLKS10]. That is, the proposed model synchronization framework will exhibit polynomial O(m x n^k) worst-case runtime and memory performance, where m is the number of TGG rules, n is the size of input graph, and k is the maximum number of elements in each rule. This estimate is based on two assumptions: first, n^k is the worst-case estimate of the application pattern matching of a rule with k elements, and second, there is an ordering of rule applications that prevents any one element from being processed more than once.

6.5 Chapter Summary

In this chapter we have presented case studies used to support the proposed theory. We have conducted the case studies in order to asses the applicability of our formal concept

analysis approach. We have presented corresponding measurements such as the memory and run-time performance, and precision and recall information. We have also discussed a commerce-based reference architecture to which our model-drive synchronization approach is applied. Finally, we have discussed several application scenarios for our TGG based approach, to demonstrate the applicability of the approach.

Chapter 7

Conclusions

I don't pretend we have all the answers. But the questions are certainly worth thinking about.

— Arthur C. Clarke

In this chapter, we summarize the major findings and contributions of the thesis, and we discuss directions for future research.

7.1 Summary and Conclusions

In this thesis, we have focused on software model synchronization problem. The software model synchronization problem is defined as the problem of maintaining consistency between software models that co-evolve due to iterative development or maintenance activities. Model synchronization is a major issue in the context of model-driven software engineering and automated code generation.

The question that we are attempting to answer is how to manage software artifacts in a way that allows software engineers to respond to the evolutionary forces in a systematic and traceable manner. We have viewed this question within the context of model-driven frameworks, and in particular, in the context of the Meta Object Facility (MOF) framework, and identified the main challenge as the systematic tracing and interpretation of software transformations that are applied to specific models at different levels of abstraction (*e.g.*, design level, implementation level, and source code level).

To address these challenges, we have introduced the mSYNTRA model synchronization framework. The major issues that the framework is aiming to address along with the solutions that have been proposed are summarized below.

1. To provide a method for reducing the semantic gap between domain models at different levels of abstraction.

   For this challenge, we have introduced the concept of incremental transformations that aim to annotate software models that pertain to different abstraction levels, in order to bridge the semantic gap between them. In this respect, annotations provide implicit semantic links between models, so that these can be easier compared and dependencies between them can be established. For this thesis, we have experimented with business process models, obtained from IBM's WebSphere Commerce suite, and the Java source code implementation models that enact these business models. Since there is a significant abstraction difference between such models, we provided annotations that allow for business tasks to be annotated with input and output data type information that can be traced to code. We have also provided annotations that allow for source code to annotated with higher level operations that can be traced to business processes. Association rules have been used to facilitate schema mappings, so

that model comparison can commence.

2. To devise a method for identifying dependencies between model elements and facilitating coarse-grained model synchronization.

   For this challenge, we have introduced the use of Formal Concept Analysis (FCA) as a possible solution for identifying coarse-level software model dependencies between co-evolving models. The basic premise is that model elements that share common attributes and features may relate to the same concept or cluster. Models that fall under the same concept or cluster are considered dependent. In this thesis, we have experimented with a number of WebSphere Commerce processes and the corresponding Java implementation models, and the results indicated high level of recall for the identified dependencies. This result provides evidence that Formal Concept Analysis may be a valid technique for establishing dependencies between different software models. Once model dependencies have been established, coarse-grained model synchronization can be achieved by tracing dependencies from one model to another.

3. To introduce a method for representing domain models as domain-specific grammars and their dependencies in terms of UML classifiers.

   For this challenge, we have first proposed a technique for representing domain models and instance models as collections of cross-referenced tuples. Second, we have proposed a technique that allows for modeling these collections of tuples as context-free grammars. And third, we have introduced a technique that allows for representing model dependencies utilizing UML stereotypes. This dependency representation allows for denoting one-to-one, one-to-many, many-to-one, and many-to-many software model dependencies. In this respect model dependencies are encoded in the form of MOF models and can be programmatically processed.

4. To enable fine-grained model synchronization by utilizing identified triple graph grammar rules.

   For this challenge, we have proposed a technique for representing model dependencies encoded in the form of MOF stereotypes as triple graph grammars and ultimately as QVT rules. The objective here is to allow for extracted dependencies to be modeled in a way that can be used from within a programmatic model transformation environment. The selection of triple graph grammars is based on the strength of this formalism to represent contextual information that can be taken into consideration during the mapping or synchronization process. For this thesis, we have experimented on mapping and synchronizing IBM WebSphere Commerce business process models with the underlying run time Java code that enacts these business process models. In this respect, when a business process is altered, the corresponding Java code is tagged for editing, and similarly, when the Java code is evolved the business process that is likely to be affected is also tagged for editing.

7.2 Future Research

In the context of this thesis, there are several avenues for future research. These include:

1. Extend the synchronization framework by integrating software quality metrics, thereby enabling synchronization of quality-specific aspects, such as performance or modifiability.

   The objective here is to quantify the effect of synchronization on the quality of the produced models. In this respect, when there are more that one alternative ways to

re-synchronize models, to be able to select the synchronization path the maximizes the quality of the produced models and minimizes model decay.

2. Expand the interoperability with other CASE tools that do not directly support the QVT framework.

   The objective here is to provide adapters, so that Triple Graph Grammar models can be mapped to other formalisms (*e.g.*, ATL [JK05]), and the required architecture to integrate the proposed approach to Integrated Development Environments (IDEs) such as the Rational Software Architect.

3. Further evaluate and refine the framework through additional case studies from domains not considered within the scope of the thesis.

   The objective of this avenue of research is to investigate how the proposed approach can be generalized to other domains beyond software models, and assess its applicability and generality so that models pertaining to the interest of other stakeholders can also be taken into account. Examples of such non-software models may be regulatory models and conformance constraint models.

Bibliography

[ABMM07] Y. An, A. Borgida, R. J. Miller, and J. Mylopoulos. A semantic approach to discovering schema mapping expressions. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 206–215, 2007.

[ADL96] P. Alencar, D.Cowan, and C. Lucena. A formal approach to architectural design patterns. In *Proceedings of the Industrial Benefit and Advances in Formal Methods (FME)*, volume 1051 of *Lecture Notes in Computer Science*, pages 576–594. 1996.

[Ake00] D. H. Akehurst. *Model Translation: A UML-based specification technique and active implementation approach*. PhD thesis, University of Kent at Canterbury, Dec 2000.

[All70] F. E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5:1–19, July 1970.

[AP03] M. Alanen and I. Porres. A relation between context-free grammars and meta object facility metamodels. TUCS Technical Report No 606, Turku Center for Computer Science, bo Akademi University, Turku, Finland, 2003.

[ASU86] A. V. Aho, R. Sethi, and J.D. Ullman. *Compiler Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BCK98] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, Reading, MA, 1998.

[BHB99] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 555–563, Los Angeles, CA, May 1999.

[Bre98] D. Bredemeyer. How to lead, how to follow, and how to get out of the way. Comdex 98 Presentation, 1998.

[Bro87] F. P. Brooks, Jr. No silver bullet; essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, Apr 1987.

[CH03] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, Anaheim, CA, Oct 2003.

[CK06] K. Czarnecki and C. H. P. Kim. Feature models are views on ontologies. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 21–24, 2006.

[CKK02] P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, Indianapolis, IN, 2002.

[CNYM00] L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Publishing, 2000.

[Cor03] IBM Corporation. Rational extended development environment (xde). Online, 2003. http://www.rational.com/products/xde/index.jsp.

[DDF+90] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[Dev10] DevTopics.com. 20 famous software disasters. Online, 2010. http://www.devtopics.com/20-famous-software-disasters/.

[DG08] D. Dang and M. Gogolla. On integrating ocl and triple graph grammars. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS) Workshops*, pages 124–137, 2008.

[Dis09] Z. Diskin. Model synchronization: Mappings, tiles, and categories. In *Proceedings of the Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 92–165, 2009.

[DT10] A. Dahanayake and B. Thalheim. Co-evolution of (information) system models. In *Proceedings of the BMMDS/EMMSAD*, pages 314–326, 2010.

[EHSW99] G. Engels, R. Huecking, S. Sauer, and A. Wagner. Uml collaboration diagrams and their transformation to java. In *Proceedings of the International Conference on The Unified Modeling Language (UML)*, pages 750–, Fort Collins, CO, Oct 1999.

[FMG99] M. Faid, R. Missaoui, and R. Godin. Knowledge discovery in complex objects. *Computational Intelligence*, 15(1):28–49, Jan 1999.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In *Proceedings of the International Workshop on Theory and Application of Graph Transformation (TAGT)*, pages 157–167, Paderborn, Germany, Nov 1998.

[Fou10] The Eclipse Foundation. Eclipse documentation. Online by The Eclipse Foundation, 2010. http://www.eclipse.org/documentation/.

[Fuj05] Fujaba. Fujaba tool suite. Online by Fujaba Tool Suite Developer Team, 2005. http://www.fujaba.de/.

[GH08] H. Giese and S. Hildebrandt. Incremental model synchronization for multiple updates. In *Proceedings of the International Workshop on Graph and Model Transformations*, GRaMoT, pages 1–8, 2008.

[GK07] J. Greenyer and E. Kindler. Reconciling tggs with qvt. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 16–30, 2007.

[GK10] J. Greenyer and E. Kindler. Comparing relational model transformation technologies: implementing query/view/transformation with triple graph grammars. *Software and System Modeling*, 9(1):21–46, 2010.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.

[GW06] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 543–557. Springer Verlag, 2006.

[GW09] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software and System Modeling*, 8(1):21–43, 2009.

[HIK+08] J. L. Hawkins, I. Ivkovic, K. Kontogiannis, R. Mckegney, W. Ng, and T. Tong. Incremental model refinement and transformation in generating commerce applications using a model driven architecture. US Patent 20080276229, 2008. http://www.freepatentsonline.com/y2008/0276229.html.

[Hol00] G. K. Holman. What is xslt? *O'Reilly's XML.com*, August 2000. http://www.xml.com/pub/a/2000/08/holman/.

[IBM04] IBM. Rational unified process (rup). Online by IBM Corporation, 2004. http://www.ibm.com/software/awdtools/rup/.

[IK04a] I. Ivkovic and K. Kontogiannis. Model synchronization as a problem of maximizing model dependencies. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 222–223, Vancouver, BC, Oct 2004.

[IK04b] I. Ivkovic and K. Kontogiannis. Tracing evolution changes through model synchronization. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 252–261, Chicago, IL, Sep 2004.

[IK05] I. Ivkovic and K. Kontogiannis. Using formal concept analysis to estalish model dependencies. In *Proceedings of the IEEE International Conference on Information Technology Coding and Computing*, pages 365–372, Las Vegas, NV, Apr 2005.

[IK06] I. Ivkovic and K. Kontogiannis. Towards automatic establishment of model dependencies. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 16(4):499–522, 2006.

[Jez03] J. Jezequel. Model-driven engineering with contracts, patterns, and aspects. Tutorial Program of AOSD 2003: 2nd International Conference on Aspect-Oriented Software Development, March 2003.

[JK05] F. Jouault and I. Kurtev. Transforming models with atl. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS) Satellite Events*, pages 128–138, 2005.

[Kea97] L. Kean. Feature-oriented domain analysis. Software technology review, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1997.

[KLKS10] F. Klar, M. Lauder, A. Königs, and A. Schürr. Extended triple graph grammars with efficient and compatible graph translators. In *Graph Transformations and Model-Driven Engineering*, pages 141–174. 2010.

[Kön10] P. Könemann. Capturing the intention of model changes. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 108–122, 2010.

[Kru95] P. B. Kruchten. The 4+1 view model of architecture. In *IEEE Software*, volume 12(6), pages 42–50. IEEE, 1995.

[KWB03] A. Kleppe, J. Warmer, and W. Bast. *The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[LLM09] T. Levendovszky, L. Lengyel, and T. Mészáros. Supporting domain-specific model patterns with metamodeling. *Software and System Modeling*, 8(4):501–520, 2009.

[Lud03] J. Ludewig. Models in software engineering: An introduction. *Springer Software and Systems Modeling*, 2:5–14, 2003.

[Mal90] T. W. Malone. What is coordination theory and how can it help design cooperative work systems. In *Proceedings of the Computer Support Cooperative Work (CSCW)*, pages 357–370, Los Angeles, CA, Oct 1990.

[Mal10] I. Malavolta. A model-driven approach for managing software architectures with multiple evolving concerns. In *European Conference on Software Architecture (ECSA) Companion Volume*, pages 4–8, 2010.

[Mam00] E. Mamas. Design and implementation of an integrated software maintenance environment. Master's thesis, University of Waterloo, 2000.

[Mar97] J. C. Martin. *Introduction to Languages and the Theory of Computation*. WCB/McGraw-Hill, 1997.

[MBZR03] T. Mens, J. Buckley, M. Zenger, and A. Rashid. Towards a taxonomy of software evolution. In *Proceedings of the International Workshop on Unanticipated Software Evolution (USE)*, Warsaw, Poland, Apr 2003.

[MCG05] T. Mens, K. Czarnecki, and P. V. Gorp. A taxonomy of model transformation. In *Proceedings of the Dagstuhl Seminar on Language Engineering for Model-Driven Software Development*, Schloss Dagstuhl, Germany, 2005. Electronic.

[Men] Tom Mens. Graph-transformation based support for model evolution. Online.

[Men05] T. Mens. On the use of graph transformations for model refactoring. In *Proceedings of the Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 219–257, 2005.

[MES02] M. A. D. Miguel, D. Exertier, and S. Salicki. Specification of model transformations based on meta templates. In *Workshop in Software Model Engineering*, 2002.

[Met96] D. L. Metayer. Software architecture styles as graph grammars. In *Proceedings of the ACM SIGSOFT symposium on Foundations of Software Engineering*, pages 15–23, San Francisco, CA, Oct 1996.

[Mic10] Microsoft. Inside the .net framework. Online by Microsoft Corporation, 2010. http://msdn.microsoft.com/en-us/library/a4t23ktk.aspx.

[Mil02] D. Milicev. Automatic model transformations using extended uml object diagrams in modeling environments. *IEEE Transaction on Software Engineering*, 28(4):413–431, Apr 2002.

[OAS07] OASIS. Web services business process execution language (wsbpel). Technical report, Organization for the Advancement of Structured Information Standards, April 2007. docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[Ois02] R. Oiseth. Model driven software development part2. Online; Rational Corporation Webinar, October 2002. http://www.rational.com/products/xde/index.jsp.

[OMG01] OMG. Model driven architecture - a technical perspective. Object Management Group's (OMG's) Architecture Board ORMSC Document ORMSC/01-07-01, Object Management Group, July 2001.

[OMG02] OMG. Common object request broker architecture: Core specification. Technical report, Object Management Group, December 2002. http://www.omg.org/docs/formal/02-12-02.pdf.

[OMG06] OMG. Meta object facility (mof) core specification version 2.0. Technical report, Object Management Group, Jan 2006. http://www.omg.org/spec/MOF/2.0/.

[OMG10] OMG. Unified modelling language (uml) infrastructure specification version 2.4. Technical report, Object Management Group, November 2010. http://www.omg.org/spec/UML/2.4/.

[OMG11a] OMG. Introduction to omg's unified modeling language (uml. Technical report, Object Management Group, Feb 2011. http://www.uml.org/.

[OMG11b] OMG. Meta object facility (mof) 2.0 query/view/transformation version 1.1. Technical report, Object Management Group, 2011. http://www.omg.org/spec/QVT/1.1/PDF/.

[Ora08] Oracle. Enterprise javabeans 3.0 specification. Online by Oracle Corporation, Jan 2008. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html.

[Ora10] Oracle. Java ee 6 documentation. Online by Oracle Corporation, Aug 2010. http://download.oracle.com/javaee/.

[PZB00] M. Peltier, F. Ziserman, and J. Bezivin. On levels of model transformation. In *Proceedings of the XML Europe Conference*, Paris, France, June 2000.

[RGM01] P. Rodriguez-Gianolli and J. Mylopoulos. A semantic approach to xml-based data integration. In *Proceedings of the International Conference on Conceptual Modeling*, pages 117–132, 2001.

[RMJ06] D. Rayside, L. Mendel, and D. Jackson. A dynamic analysis for revealing object ownership and sharing. In *Workshop on Dynamic Systems Analysis*, WODA, pages 57–64, 2006.

[RÖV10] I. Ráth, A. Ökrös, and D. Varró. Synchronization of abstract and concrete syntax in domain-specific modeling languages - by mapping models and live transformations. *Software and System Modeling*, 9(4):453–471, 2010.

[RW90] C. Rich and L. M. Willis. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, pages 196–215, Jan 1990.

[SC94] G. Spanoudakis and P. Constantopoulos. Measuring similarity between software artifacts. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 387–394, Jurmala, Latvia, Jun 1994.

[Sev87] R. E. Seviora. Knowledge-based program debugging systems. *IEEE Software*, 4:20–32, May 1987.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Upper Saddle River, NJ, 1996.

[SK08] A. Schürr and F. Klar. 15 years of triple graph grammars. In *Proceedings of the International Conference on Graph Transformation (ICGT)*, pages 411–425, 2008.

[SME08] R. Salay, J. Mylopoulos, and S. M. Easterbrook. Managing models through macromodeling. In *Proceedings of the IEEE/ACM International Conference Automated Software Engineering (ASE)*, pages 447–450, 2008.

[SME09] R. Salay, J. Mylopoulos, and S. M. Easterbrook. Using macromodels to manage collections of related models. In *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE)*, pages 141–155, 2009.

[SNEC06] M. Sabetzadeh, S. Nejati, S. M. Easterbrook, and M. Chechik. A relationship-driven approach to view merging. *ACM SIGSOFT Software Engineering Notes*, 31(6):1–2, 2006.

[SNPM90] S. M. Shieber, G. V. Noord, F. C. N. Pereira, and R. C. Moore. Semantic-head-driven generation. *Computational Linguistics*, 16:30–42, 1990.

[Tae03] G. Taentzer. Agg: A graph transformation environment for modeling and validation of software. In *Proceedings of the Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, pages 446–453, 2003.

[Tah04] Ladan Tahvildari. Quality-driven object-oriented re-engineering framework. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 479–483, 2004.

[Tea10a] The AndroMDA Core Team. Andromda documentation. Online by The AndroMDA Core Team, 2010. http://www.andromda.org/index.php.

[Tea10b] The ATL Team. Atl documentation. Online by The ATL Team, 2010. http://atlanmod.emn.fr/.

[TML+04] T. Tong, R. McKegney, T. Lau, K. Kontogiannis, I. Ivkovic, P. Liew, Y. Zou, Q. Zhang, and M. Hung. Model synchronization for efficient software application maintenance. In *Proceedings of the International Conference/Workshop on Program Comprehension*, page 499, Bari, Italy, Jun 2004.

[TSP96] S. R. Tilley, D. B. Smith, and S. Paul. Towards a framework for program understanding. In *Proceedings of the International Conference/Workshop on Program Comprehension*, pages 19–28, 1996.

[W3C00] W3C. Extensible markup language (xml) specification. Technical report, W3C XML Core Working Group, October 2000. http://www.w3.org/TR/2000/REC-xml-20001006.pdf.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[XLH+07] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Proceedings of the IEEE/ACM International Conference Automated Software Engineering (ASE)*, pages 164–173, 2007.

[ZLK+04] Y. Zou, T. Lau, K. Kontogiannis, T. Tong, and R. McKegney. Model driven business process recovery. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, page 224, Amsterdam, The Netherlands, Nov 2004.